De PyTorch a Triton

Para llevar un modelo entrenado en PyTorch a un servicio rápido y medido con la documentación abierta, y para diagnosticar por qué va lento, predice distinto o no carga. El modelo se exporta a ONNX, se compara con el original, se compila con TensorRT para la GPU donde va a correr y se sirve con Triton, que junta los requests en batches; al final se mide. Cada sección trae la sintaxis como plantilla con variables, las piezas, un ejemplo real del curso, una figura, un truco para recordarla, la trampa típica y, cuando se puede sin GPU, un laboratorio opcional. En la versión web, al final hay un diagnosticador de síntomas, un revisor de config.pbtxt y un quiz de errores.

Cómo recordarla · Trampa típica · *variable* la cambias tú · Ejemplos en verde

EN LAS FIGURAS: CPU · la GPU y su trabajo · espera y latencia · precisión: FP32 · FP16 · INT8 · falla, se rechaza o no carga · se ignora o es un ejemplo

0 Números de bolsillo: bytes, VRAM, la T4 y las rutas de Triton

¿Qué números, puertos y rutas conviene saberse de memoria para servir un modelo, medirlo y leer sus métricas sin abrir la documentación?

SINTAXIS CON VARIABLES

`curl -i localhost:`*puerto_http*`/v2/health/ready`

¿Triton puede recibir tráfico? Si sí, la primera línea dice `HTTP/1.1 200 OK`; si no, `400 Bad Request`. Con `--strict-readiness=true` (el default, que el C01 igual escribe explícito), solo da 200 si el servidor responde **y** todos sus modelos cargaron. `/v2/health/live` pregunta algo más simple: si el proceso responde.

**`curl -s localhost:`*puerto_http*`/v2/models/`*modelo*

La metadata del modelo según el protocolo KServe v2 (el contrato estándar que habla Triton por HTTP y por gRPC, un protocolo binario sobre HTTP/2): nombre, versiones, plataforma y, por cada tensor, `name`, `datatype` (como `FP32`) y `shape`. El eje de batch sale como `-1`. Con `/config` al final ves la configuración que Triton usa de verdad, en JSON.

`curl -s localhost:`*puerto_metricas*`/metrics | grep` *metrica*

Las métricas de Triton, en el formato de texto de Prometheus (el sistema que las junta y las guarda). Las de requests y tiempos (`nv_inference_request_success`, `nv_inference_queue_duration_us` …) son contadores: acumulan desde que arrancó Triton, como un odómetro, y lo útil es cuánto suben por segundo, que en Prometheus se calcula con `rate()`. Las de la GPU (`nv_gpu_utilization`, `nv_gpu_memory_used_bytes`) y `nv_inference_pending_request_count` son gauges: marcan el valor de ahora, como un velocímetro.

`nvidia-smi --query-gpu=name,compute_cap,memory.used,memory.total,utilization.gpu --format=csv`

En la máquina con GPU, no en tu Mac: qué GPU es, su compute capability (la generación de su arquitectura: 7.5 en la T4, y el engine queda atado a ella), cuánta memoria de GPU usa y qué fracción del tiempo tuvo algún kernel (una función que corre en la GPU) en marcha. Ojo: 100 % no quiere decir que esté llena.

VARIABLES

VARIABLE	QUÉ ES	EJEMPLO
puerto_http	puerto HTTP de Triton (protocolo KServe v2)	8000
puerto_metricas	puerto de métricas para Prometheus	8002
modelo	nombre del modelo: su carpeta en el model repository	classifier
metrica	nombre de una métrica de Triton	nv_inference_queue_duration_us
P	parámetros del modelo: los números que aprendió	24,532,532 ≈ 24.5 M (ResNet-50 con su capa final, fc, de 500 clases)
b	bytes por valor según la precisión	FP32 4 · FP16 2 · INT8 1
B	batch: cuántas imágenes van juntas en una ejecución	1 a 32 (max_batch_size: 32)
$t_{\text{exec}}(B)$	lo que tarda una ejecución del modelo con ese batch	≈ 3–6 ms (p50) con batch 1–8 en la T4 (presupuesto del curso)
X, $X_{\max}$	throughput: imágenes por segundo; X_max es el máximo medido	≈ 1,200 img/s por T4 (supuesto del C01: se mide con perf_analyzer)
λ	tasa de llegada: requests por segundo	250 req/s por GPU en el pico de diseño (500 entre 2 GPUs)

VARIABLE	QUÉ ES	EJEMPLO
ρ	utilización: qué fracción de la capacidad usas	≈ 21 % por T4 en el pico; se planifica hasta ≈ 50 %
W, L	W: tiempo que pasa un request adentro (cola + cómputo). L: cuántos requests hay adentro a la vez	W ≈ 5–10 ms dentro de Triton (estimación) → L ≈ 1–3
Δt, $\bar{B}$	Δt: la ventana del dynamic batching, lo que Triton espera para juntar requests en un batch (max_queue_delay_microseconds). $\bar{B}$: el batch promedio real	Δt = 2,000 µs = 2 ms; $\bar{B}$ ≈ 1–3 en el pico y ≈ 2–4 si una GPU absorbe todo (failover)

FÓRMULAS

$$\text{VRAM} \approx \underbrace{P \times b}_{\text{pesos}} + \text{activaciones} + \text{contexto CUDA} + \text{margen} \qquad P \times b = 24.5\text{M} \times 2\text{B} \approx 49\text{MB (FP16)}$$

$$\text{tensor} = 3 \times 224 \times 224 \times 4\text{B} = 602{,}112\text{B} \approx 602\text{KB} \qquad \text{logits} = 500 \times 4\text{B} = 2\text{KB}$$

$$X = \frac{B}{t_{\text{exec}}(B)} \qquad \rho = \frac{\lambda}{X_{\max}} = \frac{250}{1{,}200} \approx 21\%$$

$$L = \lambda W \qquad 250\ \frac{\text{req}}{\text{s}} \times (0.005\text{–}0.010)\ \text{s} \approx 1\text{–}3\ \text{requests dentro de cada Triton}$$

$$\bar{B} = \frac{\Delta \text{nv_inference_count}}{\Delta \text{nv_inference_exec_count}} \qquad \bar{B} \gtrsim 1 + \lambda \Delta t = 1 + 250 \times 0.002 = 1.5$$

LAS PIEZAS

PIEZA	QUÉ HACE	EJEMPLO
FP32 · 4 B	Precisión completa. Es la referencia: el checkpoint guarda los pesos en FP32 (aunque el C01 entrena con AMP, en precisión mixta), y así se evalúa y se exporta.	98 MB de pesos; en la T4 corre en los CUDA cores: 8.1 TFLOPS
FP16 · 2 B	Media precisión: la mitad de bytes y acceso a los tensor cores (las unidades que multiplican matrices). Tope 65,504; cerca de 10, los valores van de a 0.008.	el engine del C01: 49 MB de pesos; 65 TFLOPS en la T4
INT8 · 1 B	Enteros de 8 bits con una escala. Pide calibración (fijar las escalas con imágenes reales) y un gate por slices: medir la accuracy en cada subgrupo de datos antes de aprobarlo.	25 MB; 130 TOPS en la T4; el C01 no lo usa porque la GPU le sobra
BF16 · FP8	Formatos de 2 y 1 byte, comunes en LLMs (modelos de lenguaje grandes). La T4 no los tiene: BF16 pide una GPU Ampere o más nueva, y FP8, una Ada, Hopper o más nueva.	en una T4, un modelo entrenado en BF16 se sirve en FP32, o en FP16 si pasa la paridad: FP16 tiene menos rango (tope 65,504)
NVIDIA T4	La GPU del C01, de la generación Turing (2018): 16 GB de memoria GDDR6 que se lee a ≈ 320 GB/s, 70 W, conectada por PCIe Gen3 x16 (el bus del servidor) y con compute capability 7.5 (sm_75).	g4dn.xlarge (la máquina de AWS con una T4) ≈ US$0.526/h; nvidia-smi muestra 15,360 MiB porque el ECC (corrección de errores) reserva 1/16 de la memoria
Tensores de entrada y salida	Lo que viaja entre el gateway (el servicio en CPU que recibe la foto y llama a Triton) y Triton: la imagen preprocesada, y de vuelta un logit (puntaje antes del softmax) por clase.	entrada: 602 KB en FP32 (150 KB en UINT8; un batch de 32 dentro de Triton, 19 MB); salida: 2 KB
VRAM ocupada	La memoria de la GPU en uso. Por proceso, el contexto CUDA (lo que reservan el driver y las librerías de NVIDIA al arrancar); por cada instancia del modelo, su copia del engine con los pesos y sus activaciones (los resultados intermedios de cada capa).	≈ 1–3 GB de 16 con 2 instancias (estimación del curso)

PIEZA	QUÉ HACE	EJEMPLO
8000 · 8001 · 8002	Puertos por defecto de Triton: HTTP/REST, gRPC y métricas.	el gateway llama al 8000; Prometheus lee el 8002
/v2/health/live · /v2/health/ready	live: ¿el proceso responde? Sirve para la liveness (¿lo reinicio?). ready: ¿puede recibir tráfico? Sirve para la readiness (¿le mando requests?).	livenessProbe en /live; startupProbe y readinessProbe en /ready
/v2/models/ *modelo* **/ready**	¿Ese modelo está listo? Mira uno solo, no el servidor entero.	lo consulta el /readyz del gateway
/v2/models/ *modelo* **· /v2/models/** *modelo* **/config**	La metadata (nombres, datatypes, shapes) y la configuración efectiva del modelo, en JSON.	input FP32 [−1, 3, 224, 224] → logits FP32 [−1, 500]
/v2/models/ *modelo* **/infer**	La inferencia por HTTP (POST). Con la extensión binaria, los tensores viajan como bytes crudos detrás de un JSON corto.	el cliente async (tritonclient.http.aio) del gateway manda 602 KB binarios por request
nv_inference_queue_duration_us · nv_inference_compute_infer_duration_us	Microsegundos acumulados en cola y en cómputo, por modelo. Divide cuánto suben por cuánto sube nv_inference_request_success y tienes el promedio por request.	la espera en cola es la señal con la que KEDA (el autoscaler por métricas) escala Triton
2 · 100 · 150 · 200 ms	La ventana del batching, el rechazo en la cola, el timeout del gateway hacia Triton y el SLO (el objetivo de latencia) del servicio.	max_queue_delay 2000 µs · REJECT a los 100 ms · INFER_TIMEOUT_S=0.15 · p99 ≤ 200 ms

ResNet-50 del C01 (capa final de 500 clases) · 24,532,532 parámetros

PRECISIÓN	BYTES POR PARÁMETRO	PESOS DEL MODELO (MB, A ESCALA)		PARA QUÉ · EN LA T4
FP32	4 B		98.1 MB	referencia: checkpoint y ONNX CUDA cores · 8.1 TFLOPS
FP16	2 B		49.1 MB	el engine TensorRT del C01 tensor cores · 65 TFLOPS
INT8	1 B		24.5 MB	opcional: pide calibración tensor cores · 130 TOPS

NVIDIA T4 · 16 GB de VRAM (15,360 MiB en nvidia-smi)



0 4 8 12 16 GB

en uso con 2 instancias: ≈ 1–3 GB (rango estimado) · los pesos FP16: 49.1 MB, el 0.3 % de la tarjeta · libre

DE QUÉ ESTÁ HECHO LO QUE SE USA (RANGO ESTIMADO, A ESCALA DE 0 A 1 GB)

engine (pesos FP16 + kernels elegidos)	0.05–0.15 GB
activaciones a batch 32	0.2–0.5 GB
contexto CUDA + librerías	0.3–0.8 GB

Con 2 instancias del engine (instance_group count 2), ≈ 1–3 GB en total. Es una estimación del curso: se confirma con nv_gpu_memory_used_bytes.

Bytes por parámetro y peso de los pesos del modelo del C01 (24,532,532 parámetros), a escala. Al lado, la T4: los pesos son una rebanada mínima de sus 16 GB; lo que ocupa memoria es el runtime y las activaciones.

EJEMPLO REAL · EL CONTRATO DE ENTRADA DEL C01: LOS NÚMEROS DEL PREPROCESAMIENTO COMPARTIDO

PYTHON reference/c01/gateway/app/preprocess.py (extracto)

```python
RESIZE_SHORT = 256  # lado corto tras el resize: el mismo número que Resize(256) en training
CROP = 224  # tamaño del center crop: el input del ResNet-50 (y del engine TensorRT)
# mean/std de ImageNet con forma (3,1,1) para hacer broadcasting sobre CHW sin copiar
MEAN = np.array([0.485, 0.456, 0.406], dtype=np.float32).reshape(3, 1, 1)
STD = np.array([0.229, 0.224, 0.225], dtype=np.float32).reshape(3, 1, 1)
```

- `CROP = 224` y los 3 canales RGB dan la forma de cada imagen: `[3, 224, 224]`. Es la misma de `dims: [ 3, 224, 224 ]` en el `config.pbtxt` y la de `--minShapes=input:1x3x224x224` en trtexec (la herramienta que compila el engine).
- En FP32 son 3 × 224 × 224 × 4 B = 602,112 B: los ≈ 602 KB que el gateway manda a Triton en cada request (≈ 2.4 Gb/s a 500 req/s).
- `MEAN` y `STD` son las estadísticas de ImageNet por canal. Si un lado normaliza y el otro no, el modelo recibe otra distribución y pierde accuracy sin lanzar ningún error.
- `RESIZE_SHORT = 256` copia el `Resize(256)` del training. El training, la evaluación y el gateway importan este mismo archivo: es la única fuente de estos números.
- Del lado del modelo, `export_onnx.py` fija el resto del contrato: `NUM_CLASSES = 500` (la salida `[500]`), los nombres `input` y `logits`, y `MAX_BATCH = 32`, que es el `max_batch_size` de Triton y el `--maxShapes` de trtexec.

CÓMO RECORDARLA

4 · 2 · 1 bytes por valor en FP32, FP16 e INT8: el modelo del C01 pesa **98 · 49 · 25 MB**. Puertos **8000 · 8001 · 8002**: HTTP, gRPC y métricas. **live** = ¿lo reinicio?; **ready** = ¿le mando tráfico?

TRAMPA

Creer que la VRAM la llenan los pesos: en el C01 son 49 MB de 16 GB, y lo que ocupa es el contexto CUDA, las activaciones y cada instancia extra (≈ 1–3 GB en total). Y poner `/v2/health/ready` en la livenessProbe (el chequeo de ¿lo reinicio?): si un modelo deja de estar listo, por ejemplo porque se descargó, el kubelet (el agente de Kubernetes en cada nodo) reinicia Triton en bucle, y reiniciar no lo arregla. Liveness va con `/live`; readiness, con `/ready`.

En el curso: Números · Bytes, dtypes y tamaños típicos · M4 · T4, L4, A10G, A100, H100: cuándo usar cada una · M4 · Cálculo de VRAM paso a paso · M3 · Little's law: la cuenta que ordena todo · M7 · NVIDIA Triton Inference Server · Hoja · Formulario de estimaciones

La cadena: del checkpoint a una respuesta medida

¿Qué pasa, en orden, entre el checkpoint que evaluaste y una respuesta de Triton con su latencia medida, y qué revisa cada paso?

SINTAXIS CON VARIABLES

`python model/export_onnx.py --checkpoint` *checkpoint* `--out` *onnx*

Exporta a ONNX (un formato de archivo estándar para redes, que no depende de PyTorch) y lo verifica: `onnx.checker`, el opset real (la versión de los operadores que quedó escrita) y ONNX Runtime (un motor que ejecuta ONNX en CPU) con batch 1 y 32 contra PyTorch. Exit 0 = la cadena sigue.

`python model/parity_check.py --checkpoint` *checkpoint* `--onnx` *onnx* `--images` *muestra* `--n 2000`

Primer gate de paridad (¿predice lo mismo que PyTorch?), en CPU con ONNX Runtime. Sin `--checkpoint`, la referencia de PyTorch tendría pesos aleatorios y compararías contra otro modelo.

`NGC_RELEASE=`*release* `EXPECTED_GPU=`*gpu* `bash model/build_engine.sh`

En la máquina con la GPU de producción: TensorRT (el compilador de NVIDIA) convierte el ONNX en `model.plan`, el engine (el modelo compilado para un tipo de GPU y una versión de TensorRT). Las dos variables valen `24.08` y `T4` por defecto, y el script aborta si la GPU no coincide.

`python model/parity_check.py --backend triton --triton-url` *host* `:8000 --checkpoint` *checkpoint* `--images` *muestra* `--n 2000`

Segundo gate: el engine FP16 servido por Triton contra PyTorch FP32, de a una imagen como en producción. Después viene el gate de latencia con perf_analyzer (la herramienta de Triton que genera carga y mide la latencia).

VARIABLES

VARIABLE	QUÉ ES	EJEMPLO
checkpoint	el state_dict entrenado y aprobado (.pt)	`resnet50_500.pt`
onnx	el archivo ONNX que sale del export	`model/classifier.onnx`
muestra	carpeta con imágenes reales del set de evaluación	`≥ 2,000 imágenes (lo que recomienda parity_check.py)`
release	release de NGC (las imágenes de contenedor de NVIDIA): la misma de la imagen de Triton	`24.08 (TensorRT 10.3)`
gpu	la GPU donde correrá el engine	`T4`
host	dónde corre Triton	`localhost, en el runner de CI con la T4`
top1, top5	acuerdo del 1.º lugar y solapamiento del top-5 contra PyTorch FP32	`≥ 99.5 % y ≥ 99 % (defaults de parity_check.py)`
max|Δlogit|	la mayor diferencia entre los logits de los dos modelos	`≤ 0.5: atrapa exports rotos`
p99	la latencia que solo el 1 % de los requests supera	`perf_analyzer a 250 y 500 req/s contra el champion (el modelo en producción); el challenger es el candidato`

FÓRMULAS

$$\text{registrar como challenger} \iff \underbrace{\text{top1} \geq 0.995 \wedge \text{top5} \geq 0.99 \wedge \max|\Delta\text{logit}| \leq 0.5}_{\text{paridad, en CPU y en la T4}} \wedge \underbrace{p99_{\text{nuevo}} \leq 1.1 \cdot p99_{\text{champion}}}_{\text{latencia}}$$

LAS PIEZAS

PIEZA	QUÉ HACE	EJEMPLO
`checkpoint (.pt)`	Los pesos entrenados, guardados con torch.save como un state_dict: un diccionario con el nombre de cada tensor de pesos (como fc.weight) y sus valores.	`resnet50_500.pt: 24.5 M parámetros en FP32`
`build_model() + eval()`	Rearma la arquitectura del training, carga los pesos y deja el modelo en modo inferencia.	`export_onnx.py y parity_check.py usan la misma función`
`torch.onnx.export`	Recorre el modelo con una entrada de ejemplo y escribe el grafo ONNX con sus pesos.	`exportador TorchScript, opset 17, batch dinámico`
`classifier.onnx`	El punto de entrega entre training y serving: grafo + pesos + firma de entradas y salidas.	`≈ 98 MB; input [batch, 3, 224, 224] → logits [batch, 500]`
`onnx.checker`	Revisa que el archivo sea ONNX válido; con full_check también deduce la forma de cada tensor.	`dentro de verify() de export_onnx.py`
`ONNX Runtime (ORT)`	Un runtime (el motor que ejecuta el grafo). Aquí corre el ONNX en CPU, sin GPU.	`verify() con batch 1 y 32, y el gate de paridad en CPU`

PIEZA	QUÉ HACE	EJEMPLO
parity_check.py	El gate de paridad: compara top-1, top-5 y |Δlogit| contra PyTorch FP32. Exit 1 frena todo.	`--backend ort en CPU; --backend triton en la T4`
trtexec	La herramienta de línea de comandos de TensorRT: compila el ONNX a un engine y hace un benchmark rápido.	`--fp16 con perfil 1/16/32, dentro de tensorrt:24.08-py3 (TensorRT 10.3; TensorRT 11 ya no tiene --fp16)`
model.plan (engine)	El modelo compilado para un tipo de GPU y una versión de TensorRT: corre rápido, pero solo ahí.	`T4 (sm_75) + TensorRT 10.3; ≈ 50–150 MB (estimación del curso)`
model repository	La carpeta que lee Triton: una subcarpeta por modelo con su config.pbtxt, y dentro una por versión.	`classifier/config.pbtxt + classifier/1/model.plan`
imagen de Triton	El servidor con el model repository horneado adentro: el tag de la imagen es la versión del modelo.	`tritonserver:24.08-py3 + engine; tag 2026-09-01-resnet50-fp16`
Triton (tritonserver)	El servidor de modelos: carga el engine, encola los requests, arma batches y responde por HTTP y por gRPC (otro protocolo, binario).	`READY recién después del warmup: 3 ejecuciones de prueba a batch 32`
gateway (el cliente)	El servicio FastAPI en CPU que preprocesa la imagen y llama a Triton con tritonclient.	`POST /v1/classify → infer en :8000 con 150 ms de timeout`
perf_analyzer	Genera carga contra Triton y mide throughput y latencias por percentil.	`gate: p99 a 250 y 500 req/s ≤ champion + 10 %`

1 Checkpoint

`resnet50_500.pt · ≈ 98 MB en FP32`

registry · cualquier máquina

```
torch.load("resnet50_500.pt", map_location="cpu", weights_only=True)
```

✓ **REVISA:** load_state_dict estricto: las llaves y shapes calzan con la arquitectura

2 ONNX

`classifier.onnx · ≈ 98 MB · opset 17`

CI · CPU

```
python model/export_onnx.py --checkpoint resnet50_500.pt --out model/classifier.onnx
```

✓ **REVISA:** onnx.checker, opset real = 17, ORT con batch 1 y 32 ≈ PyTorch; después, paridad en CPU

▼ desde aquí, en una T4 con NGC 24.08

3 Engine

`model.plan · FP16 · batch 1 a 32`

runner con T4 · tensorrt:24.08-py3

🔒 Solo carga en `T4 · sm_75` + `TensorRT 10.3`

```
bash model/build_engine.sh
```

✓ **REVISA:** la GPU es una T4; imprime GPU, NGC_RELEASE y sha256 para el registry

4 Repository

`classifier/config.pbtxt + 1/model.plan, horneados en la imagen`

docker build · tritonserver:24.08-py3

```
docker build -t classifier-triton:2026-09-01-resnet50-fp16 triton/
```

✓ **REVISA:** RUN test -f /models/classifier/1/model.plan

5 Servidor

`Triton con el modelo READY`

la misma T4

```
curl -i localhost:8000/v2/health/ready
```

✓ **REVISA:** carga el engine, corre el warmup y recién ahí responde 200

6 Cliente

`logits [500] por request y el p99 medido`

la misma T4 en CI; en producción, el gateway

```
python model/parity_check.py --backend triton --triton-url localhost:8000 --checkpoint resnet50_500.pt --images /data/eval_sample --n 2000
```

✓ **REVISA:** paridad FP16 vs FP32 (top-1 ≥ 99.5 %) y perf_analyzer: p99 ≤ champion + 10 %

corre en cualquier CPU · corre en la GPU · precisión de lo que sale

Seis eslabones del checkpoint a una respuesta medida. Los dos primeros corren en cualquier CPU; desde el engine, todo corre en una T4 con la release 24.08 de NGC. Cada eslabón tiene su chequeo, y un exit code distinto de 0 frena la cadena.

EJEMPLO REAL · LA CADENA TAL COMO LA DIBUJA EL README DEL C01

TEXTO **reference/c01/README.md (Flujo: del checkpoint a producción)**

```
checkpoint .pt ──export_onnx.py──► classifier.onnx ──parity_check.py (ORT, CPU)──► ¿pasa?
      └─ en la máquina con T4: build_engine.sh ──► model.plan ──docker build triton/──► imagen
         └─ Triton arriba ──parity_check.py --backend triton (FP16 vs FP32)──► ¿pasa? ──► registry / canary
```

- La primera línea corre en cualquier máquina, también en tu Mac: export y paridad en CPU con ONNX Runtime. Valida el export sin tocar una GPU.
- La segunda corre en la máquina con la T4 (una g4dn.xlarge): el engine se compila donde va a correr, con la misma release de NGC (24.08) que la imagen de Triton.
- `docker build triton/` hornea el engine en la imagen. Su `RUN test -f /models/classifier/1/model.plan` hace fallar el build si falta el engine: mejor ahí que un pod reiniciándose en bucle (CrashLoopBackOff).
- La tercera es el gate final: Triton arriba con ese engine, y `parity_check.py --backend triton` compara FP16 contra FP32. Si pasa, y la latencia también, va al registry (el catálogo de versiones de modelos) como challenger, el candidato, y después a un canary: una parte chica del tráfico real.
- Cada `¿pasa?` es un exit code: 0 sigue, 1 frena. El CI no lee los números; lee ese código.

CÓMO RECORDARLA

Exportas una vez, compilas donde corre y comparas dos veces: el ONNX en CPU (paridad con ONNX Runtime), el engine en la T4, y otra paridad ya servido por Triton antes de medir la latencia.

TRAMPA

Correr `parity_check.py` sin `--checkpoint`: la referencia de PyTorch nace con pesos aleatorios. Contra el ONNX del checkpoint real, el gate falla siempre; si el ONNX también salió sin checkpoint, pasa y no prueba nada. Y compilar el engine en otra máquina por comodidad: por defecto, solo carga en el tipo de GPU y la versión de TensorRT con que se construyó.

En el curso: C01 · Archivos de referencia · M10 · CI/CD para modelos: la pirámide de tests · M8 · Cómo validar una optimización · M7 · Qué hace un model server (y qué no)

1 Qué es servir un modelo (y por qué model(x) dentro de FastAPI no basta)

¿Qué tiene que hacer un servicio alrededor de model(x) para aguantar tráfico real, y qué te da un servidor de modelos como Triton?

SINTAXIS CON VARIABLES

`uvicorn` *modulo* `:app --host 0.0.0.0 --port` *puerto* `--workers` *n*

Levanta una app FastAPI (un framework web de Python) con uvicorn (el servidor que atiende las conexiones). Con `--workers` mayor que 1 son varios procesos, cada uno con **su copia** del modelo y sin batches entre ellos. El C01 usa 1 proceso por contenedor y agrega pods.

resultado **`= await run_in_threadpool(`** ***funcion*** **`,`** ***datos*** **`)`**

Dentro de un `async def`, el trabajo de CPU (decodificar el JPEG, correr el modelo) va a un hilo aparte. Si no, bloquea el event loop (el bucle que atiende todas las requests del proceso): nadie más avanza, ni los health checks.

`result = await client.infer("` ***modelo*** **`", [`** ***entrada*** **`],`** **`outputs=[`** ***salida*** **`])`**

Con Triton, el gateway (el servicio delante del modelo) ya no corre el modelo: manda el tensor por HTTP con el cliente async `tritonclient.http.aio` y espera. El `await` libera el loop mientras la GPU trabaja.

`tritonserver --model-repository=` ***repo***

Arranca el servidor de modelos: lee la carpeta, carga cada modelo con su backend (el motor que lo ejecuta: TensorRT, ONNX Runtime, Python…) y abre los puertos 8000 (HTTP), 8001 (gRPC, un protocolo binario) y 8002 (métricas).

VARIABLES

VARIABLE	QUÉ ES	EJEMPLO
modulo	módulo de Python con la app FastAPI	`app.main (el gateway del C01)`
puerto	puerto donde escucha uvicorn	`8080`
n	procesos de uvicorn; cada uno carga su propia copia del modelo	`1 en el C01: se escala con pods`
resultado, *funcion*, *datos*	lo que devuelve, la función bloqueante y sus argumentos	`batch, digest = … (_decode_and_hash, data)`
modelo	nombre del modelo en Triton	`classifier`
entrada, *salida*	el InferInput (nombre, shape, dtype y datos) y el InferRequestedOutput	`input FP32 [1, 3, 224, 224] y logits`
repo	carpeta del model repository	`/models (horneada en la imagen)`
n	procesos con su copia del modelo en la misma GPU; cada uno paga además su contexto CUDA (la memoria fija que CUDA reserva por proceso)	`4 workers en una T4`
$\lambda_{\text{diseño}}$	el tráfico a aguantar en el pico	`500 req/s`
X	capacidad planificada por GPU	`eager a batch 1 ≈ 100 img/s; Triton + TensorRT FP16 ≈ 600 (el 50 % de 1,200)`

FÓRMULAS

$$\text{VRAM}_{\text{workers}} \approx n \times (\text{pesos} + \text{contexto CUDA}) = 4 \times (0.1 + 0.3\text{–}1\ \text{GB}) \approx 1.6\text{–}4.4\ \text{GB}$$

$$N_{\text{GPU}} = \left\lceil \frac{\lambda_{\text{diseño}}}{X} \right\rceil \qquad \text{eager, batch 1: } \left\lceil \frac{500}{100} \right\rceil = 5 \qquad \text{Triton + TensorRT: } \left\lceil \frac{500}{600} \right\rceil = 1\ (+1 \text{ de repuesto: N+1})$$

LAS PIEZAS

PIEZA	QUÉ HACE	EJEMPLO
`servir (serving)`	Poner el modelo detrás de una API para que otros sistemas le pidan predicciones, con latencia, errores y costo bajo control.	`POST /v1/classify con p99 ≤ 200 ms`
`online síncrono`	Quien llama espera la respuesta en la misma conexión. Lo opuesto: batch (offline) o una cola asíncrona.	`la persona sigue en el formulario de publicación`
`latencia p50 / p99`	La mediana y el valor que solo el 1 % de los requests supera. En el C01, el SLO (el objetivo pactado) se fija en el p99.	`SLO: p99 ≤ 200 ms medido en el servidor`

PIEZA	QUÉ HACE	EJEMPLO
throughput	Cuántas predicciones por segundo completa el servicio.	**pico de diseño: 500 req/s**
event loop	El bucle de asyncio que atiende todas las requests de un proceso; entre un await y el siguiente, solo corre una cosa.	**decodificar una foto de 12 MP sin draft mode (el decode reducido de JPEG), ≈ 40–80 ms de CPU, dentro de un async def lo frena todo ese tiempo**
threadpool	Hilos aparte para el trabajo de CPU. PyTorch, Pillow y NumPy sueltan el GIL (el candado que deja correr un solo hilo de Python a la vez) en su parte pesada.	**run_in_threadpool, con 2 decodes a la vez por proceso**
batch 1	Cada request corre sola. En GPU deja el acelerador casi ocioso; en CPU casi da lo mismo.	**≈ 100 img/s por proceso con PyTorch eager en una T4 (estimación del C01)**
dynamic batching	El servidor junta requests de clientes distintos durante una ventana corta y las corre juntas. Como un bus que espera unos milisegundos a llenarse antes de salir.	**ventana de 2 ms, hasta 32 imágenes**
instancia del modelo	Una copia del modelo en ejecución dentro del servidor. Dos en la misma GPU se turnan: mientras una copia datos, la otra calcula.	**instance_group count 2 en la T4**
servidor de modelos (model server)	Un proceso dedicado a convertir tensores en tensores: carga modelos y versiones, encola, arma batches y expone salud y métricas.	**Triton (NVIDIA hoy lo llama Dynamo-Triton); no es el paquete triton de pip, el lenguaje de kernels que usa torch.compile**
backend	El motor que ejecuta cada modelo dentro de Triton.	**backend tensorrt (platform: tensorrt_plan) para el engine; onnxruntime (platform: onnxruntime_onnx) para un .onnx**
gateway	El servicio delante del servidor de modelos. Sabe del producto: quién llama, qué bytes acepta, el top-k y qué error devolver.	**FastAPI en CPU: valida, preprocesa, llama a Triton y arma el top-5**
readiness / liveness	¿Le mando tráfico? / ¿Lo reinicio? Son dos preguntas distintas con dos endpoints distintos.	**/readyz del gateway depende de Triton; /healthz no**
timeout y load shedding	Cortar lo que ya no llega a tiempo y rechazar rápido lo que no cabe, en vez de encolarlo.	**150 ms a Triton → 504; más de 16 en vuelo → 503 con Retry-After**

NECESIDAD	MODEL(X) EN FASTAPI	TRITON	EN EL C01
Batching	batch 1, o una cola propia (asyncio.Queue, futures, timeouts, límites)	dynamic_batching: dos líneas de config	ventana de 2 ms, hasta 32
Varias copias en la GPU	un proceso por copia (--workers), sin batches entre ellas	instance_group	2 instancias en la T4
Cola con plazo	lo que escribas	default_queue_policy (rechaza lo que vence)	REJECT a los 100 ms en cola
Versiones y recarga	reiniciar el proceso, o una recarga en caliente propia	version_policy y modos de control del repository	latest 1; una versión por imagen
Salud por modelo	un /readyz propio	/v2/health/ready y /v2/models/{modelo}/ready	readinessProbe en /v2/health/ready
Cola vs cómputo	histogramas propios	métricas por modelo en el puerto 8002	KEDA (el autoscaler) escala con la espera en cola
Protocolo de tensores	el JSON que diseñes	KServe v2 por HTTP y gRPC, con tensores binarios	602 KB binarios por request
Engine optimizado	torch.compile, o TensorRT integrado a mano	backends tensorrt, onnxruntime, python…	TensorRT FP16 en la T4

Contra el MVP: ≈ 100 img/s por proceso con PyTorch eager a batch 1, contra ≈ 1,200 img/s por T4 con TensorRT FP16 y dynamic batching. Son estimaciones del C01, a medir con perf_analyzer; el salto mezcla el engine y el batching.

Lo que tienes que construir tú si sirves con model(x) dentro de FastAPI, contra lo que Triton ya trae, y cómo lo usa el C01.

EJEMPLO REAL · EL CORTE DEL C01: EL GATEWAY PREPARA UNA IMAGEN Y TRITON HACE EL RESTO

PYTHON `reference/c01/gateway/app/main.py (extractos; # … marca líneas omitidas)`

```python
"""Gateway HTTP del clasificador de categorías (caso c01).

Por qué existe un gateway en CPU delante de Triton: validar, decodificar y redimensionar imágenes es
trabajo de CPU; si lo hiciera Triton (python backend o DALI) competiría con la GPU cara. Así cada
capa escala por separado: gateway por CPU (HPA) y Triton por cola de GPU (KEDA).

Endpoints: POST /v1/classify?top_k=5 · GET /healthz (liveness) · GET /readyz (readiness) · GET /metrics
"""
# …
def _decode_and_hash(data: bytes) -> tuple[np.ndarray, str]:
    """CPU pura (bloqueante): corre en el threadpool. Pillow y numpy sueltan el GIL en decode y resize."""
    digest = hashlib.sha256(data).hexdigest()  # huella de la imagen: deduplicar y cruzar con labels
    tensor = preprocess(data, max_pixels=MAX_PIXELS)  # float32 [3,224,224], idéntico a training/eval
    return tensor[np.newaxis], digest  # batch de 1: Triton junta requests con dynamic batching
# …
        batch, digest = await run_in_threadpool(_decode_and_hash, data)
```

- El gateway (FastAPI, en nodos de CPU) hace lo que depende del producto y de la imagen; Triton (en la T4) solo convierte tensores en tensores. Cada uno escala con su propia señal: CPU para el gateway, tiempo en cola para Triton.
- `_decode_and_hash` es CPU pura y bloqueante: por eso se llama con `run_in_threadpool`. Llamada directo dentro de un `async def`, congelaría el event loop mientras dura (el lab lo mide).
- `tensor[np.newaxis]` agrega el eje de batch: `[3, 224, 224]` → `[1, 3, 224, 224]`. El gateway manda **una** imagen por request; juntar requests es trabajo de Triton (dynamic batching), no del cliente.
- En el MVP del C01 (una VM g4dn y un solo proceso con PyTorch eager), todo esto vive en el mismo proceso. Se rompe primero por throughput (cada request es un batch de 1) y después por los deploys con reinicio: ahí la inferencia pasa a Triton y FastAPI se queda como gateway.
- Pillow y NumPy sueltan el GIL en el decode y el resize: por eso los 2 hilos de decode pueden usar 2 cores a la vez (el pod pide 1 core y no tiene límite de CPU, así que aprovecha ráfagas).

CÓMO RECORDARLA

El gateway sabe del producto; el servidor de modelos, solo de tensores. FastAPI con model(x) alcanza para un MVP; se rompe primero por el batch 1 (throughput) y después por los reinicios en cada deploy.

TRAMPA

Subir `--workers` para ganar throughput en GPU: cada worker es otro proceso con su copia de los pesos y su contexto CUDA, y siguen corriendo batches de 1 que se turnan la GPU. Y llamar a `model(x)` dentro de un `async def`: el p99 sube con la concurrencia sin que la CPU esté llena, y hasta `/healthz` tarda.

En el curso: M7 · Qué hace un model server (y qué no) · M7 · FastAPI + PyTorch: el punto de partida · M7 · NVIDIA Triton Inference Server · C01 · Serving

2 Preparar el modelo en PyTorch: eval, inference_mode, preprocesamiento idéntico y determinismo

¿Qué tienes que dejar fijo en PyTorch antes de exportar, para que el modelo servido se comporte como el que evaluaste?

SINTAXIS CON VARIABLES

`model.load_state_dict(torch.load("` *checkpoint* **`", map_location="cpu", weights_only=True))`

Carga solo tensores y tipos simples: un .pt es un pickle (el formato de Python para guardar objetos) y podría ejecutar código al cargarse (`weights_only=True` ya es el default en torch 2.14, y el C01 lo escribe explícito). `load_state_dict` es estricto por defecto: si una llave o una shape no calza, falla aquí y no en producción.

`model.eval()`

Modo evaluación: BatchNorm usa sus estadísticas guardadas (running stats) y Dropout se apaga. Equivale a `model.train(False)`. No apaga los gradientes: eso es otro interruptor.

`with torch.inference_mode():`
`    ` *salida* **`= model(`** *entrada* **`)`**

Sin autograd (el registro de operaciones para calcular gradientes): no guarda activaciones (los resultados intermedios de cada capa) para un backward que nunca llega, y es algo más rápido que `torch.no_grad()`. Vale solo en el hilo donde lo activas: en un servidor, va dentro de la función que corre el modelo.

`torch.manual_seed(` *semilla* **`)`**

Fija los números aleatorios de CPU y GPU. El C01 la usa para que el export y la paridad creen el mismo modelo cuando no hay checkpoint. No garantiza resultados idénticos entre versiones de PyTorch, plataformas, ni entre CPU y GPU.

VARIABLES

VARIABLE	QUÉ ES	EJEMPLO
checkpoint	archivo con el state_dict entrenado	`resnet50_500.pt`
entrada, *salida*	el tensor de entrada y los logits	`x [N, 3, 224, 224] FP32 → [N, 500]`
semilla	el número que fija lo aleatorio	`0 en export_onnx.py y parity_check.py`
γ, β	escala y corrimiento que BatchNorm aprende, uno por canal	`53 BatchNorm2d en el ResNet-50`
$\hat{\mu}$, $\hat{\sigma}^2$	running stats: media y varianza que BatchNorm guardó durante el training	`las usa eval()`
μ_B, σ_B^2	media y varianza del batch actual	`las usa train(): la salida depende de las otras imágenes`
m, ε	momentum de la actualización y un épsilon numérico	`0.1 y 1e-5 (defaults de BatchNorm2d)`
p, c	valor de un píxel (0–255) y canal de color RGB	`mean y std de ImageNet, en preprocess.py`

FÓRMULAS

$$\text{eval: } y = \gamma \frac{x - \hat{\mu}}{\sqrt{\hat{\sigma}^2 + \varepsilon}} + \beta \qquad \text{train: } y = \gamma \frac{x - \mu_B}{\sqrt{\sigma_B^2 + \varepsilon}} + \beta, \quad \hat{\mu} \leftarrow (1 - m)\hat{\mu} + m\mu_B$$

$$x_c = \frac{p/255 - \text{mean}_c}{\text{std}_c} \qquad \text{mean} = (0.485,\ 0.456,\ 0.406) \qquad \text{std} = (0.229,\ 0.224,\ 0.225)$$

LAS PIEZAS

PIEZA	QUÉ HACE	EJEMPLO
`model.eval()`	Pone el modelo en modo inferencia. Solo afecta capas como BatchNorm y Dropout; equivale a train(False).	`build_model() termina en return model.eval()`
`model.train()`	Modo entrenamiento: BatchNorm normaliza con el batch y actualiza sus running stats; Dropout apaga neuronas al azar.	`un forward en train() cambia el modelo aunque estés en no_grad`
`torch.no_grad()`	No registra operaciones para el backward. Los tensores que salen se pueden usar después con autograd.	`el loop de validación dentro del training`
`torch.inference_mode()`	Como no_grad pero más estricto: tampoco lleva la cuenta de las vistas (tensores que comparten memoria) ni de las versiones (cuántas veces se modificó cada tensor). Lo que sale no puede entrar después a autograd.	`run_torch() y verify() del C01`
`el modo vale por hilo`	inference_mode (y no_grad) solo valen en el hilo donde los activaste.	`alrededor de un await run_in_threadpool no hacen nada en el hilo del pool`

PIEZA	QUÉ HACE	EJEMPLO
torch.load(…, weights_only=True)	Carga solo tensores y tipos simples; rechaza objetos arbitrarios del pickle.	**default en torch 2.14; el C01 lo escribe explícito**
load_state_dict (estricto)	Exige que las llaves y las shapes calcen exacto con la arquitectura.	**size mismatch for fc.weight: copying a param with shape torch.Size([1000, 2048]) from checkpoint, … torch.Size([500, 2048])**
prefijo module.	Un checkpoint guardado desde DataParallel o DDP (entrenamiento en varias GPUs) trae module. delante de cada llave.	**Missing key(s) in state_dict: "conv1.weight"…: quita el prefijo antes de cargar**
preprocess.py compartido	Una sola función de preprocesamiento que importan el training, la evaluación y el serving.	**bytes → float32 [3, 224, 224], RGB, normalizado**
Resize(256) + CenterCrop(224)	Lado corto a 256 con interpolación bilinear, con antialias al reducir (un suavizado que evita bordes dentados), y recorte central de 224 × 224.	**medido contra torchvision 0.29: diferencia 0.0 sin draft mode (el decode reducido de JPEG)**
RGB, no BGR	El orden de canales del training. OpenCV lee en BGR; Pillow, en RGB.	**convert("RGB") en decode()**
EXIF	Metadatos de la foto; las de celular guardan ahí la rotación.	**ImageOps.exif_transpose: sin esto el modelo ve la foto de lado**
semillas	Fijan lo aleatorio: pesos iniciales, datos de ejemplo, augmentations.	**torch.manual_seed(0): export y paridad crean el mismo modelo sin checkpoint**
determinismo	Mismos números en cada corrida. Otro batch, otra versión u otro hardware cambian el orden de las sumas: diferencias de ≈ 1e-4 en los logits.	**batch 8 contra batch 1 en CPU: 1.5e-4 con logits de hasta 85 (medido)**

MODO	QUÉ CAMBIA	LA SALIDA	ÚSALO PARA
model.train()	BatchNorm usa la media y la varianza del batch y actualiza sus running stats; Dropout activo	depende de las otras imágenes del batch	entrenar
model.eval()	BatchNorm usa sus running stats; Dropout apagado. No toca los gradientes	la misma para cada imagen, con cualquier batch	evaluar, exportar y servir
torch.no_grad()	no registra operaciones para el backward	requires_grad=False; se puede usar después con autograd	la validación dentro del training
torch.inference_mode()	lo de no_grad, y además sin seguimiento de vistas ni de versiones	requires_grad=False e is_inference()=True; no entra a autograd	inferencia pura: servir y la referencia de la paridad
eval() + inference_mode()	las dos cosas: no se pisan	la misma por imagen y sin memoria de autograd	lo que hacen verify() y run_torch() en el C01

La documentación de PyTorch lo dice así: el modo de evaluación no es un mecanismo para apagar gradientes; es ortogonal a no_grad y a inference_mode.

Los dos interruptores de PyTorch: el modo del modelo (train/eval) y el de los gradientes (no_grad/inference_mode). Son independientes, y para servir van los dos.

EJEMPLO REAL · BUILD_MODEL(): EL MODELO QUE TODO EL PIPELINE USA COMO REFERENCIA

PYTHON **reference/c01/model/export_onnx.py (build_model completo)**

```python
def build_model(checkpoint: str | None) -> torch.nn.Module:
    """Reconstruye la arquitectura del training y carga los pesos."""
    if not checkpoint:
        print("[model] AVISO: sin --checkpoint → pesos aleatorios (sirve para probar el pipeline, no para servir)")
        torch.manual_seed(0)  # aleatorios pero reproducibles: export y parity_check crean el MISMO modelo
    # weights=None: nada de descargar pesos de ImageNet; num_classes=500 crea la fc correcta
    model = torchvision.models.resnet50(weights=None, num_classes=NUM_CLASSES)
    if checkpoint:
        # weights_only=True: carga solo tensores, no ejecuta pickle arbitrario (un .pt es código)
        state = torch.load(checkpoint, map_location="cpu", weights_only=True)
```

```
        # strict (por defecto): si una llave no coincide, falla AQUÍ y no en producción con pesos al azar
        model.load_state_dict(state)
    # eval(): BatchNorm usa sus running stats y Dropout se apaga. torch.onnx.export (2.14) ya exporta
    # en eval aunque lo olvides, pero este modelo es también la referencia PyTorch de verify() y del
    # gate de paridad: en modo train daría otros logits y el gate compararía contra una referencia rota.
    return model.eval()
```

- `weights=None` evita bajar pesos de ImageNet y `num_classes=500` crea la capa final con 500 salidas. Tiene que ser la misma arquitectura del training, o las llaves no calzan.
- `weights_only=True`: un .pt es un pickle, y un pickle puede ejecutar código al cargarse. Así solo entran tensores.
- `load_state_dict` es estricto por defecto: un checkpoint de 1000 clases falla con `size mismatch for fc.weight`. Mejor ese error aquí que un modelo a medio cargar en producción.
- Sin checkpoint, `torch.manual_seed(0)` hace que el export y la paridad creen el mismo modelo aleatorio: sirve para probar el pipeline, no para servir.
- `return model.eval()`, y el comentario dice por qué importa aunque el exportador TorchScript ya exporte en eval: este mismo objeto es la referencia de PyTorch en `verify()` y en el gate de paridad. En modo train daría otros logits, y el gate compararía contra una referencia rota. El exportador dynamo (el default) ni siquiera cambia el modo: exporta el modelo como esté y solo avisa con un `UserWarning`.

CÓMO RECORDARLA

eval() cambia las capas; inference_mode() apaga el autograd. Son dos interruptores distintos y para servir van los dos prendidos. Y un solo `preprocess.py` para training, evaluación y serving.

TRAMPA

Un forward en `model.train()`, aunque sea dentro de `torch.no_grad()`, actualiza las running stats del BatchNorm: el modelo cambia en silencio y deja de ser el que evaluaste. Y reescribir el preprocesamiento "parecido" en el servicio (BGR, otro resize, sin normalizar): no da ningún error, solo menos accuracy.

En el curso: M7 · FastAPI + PyTorch: el punto de partida · M9 · Training-serving skew: causas, detección y prevención · M10 · Reproducibilidad: los cinco ingredientes · C01 · Archivos de referencia

3 Exportar a ONNX: dynamo, opset, ejes dinámicos, checker y ONNX Runtime

¿Cómo pasas el modelo de PyTorch a un archivo ONNX que TensorRT (el compilador de NVIDIA) y ONNX Runtime entiendan, y cómo sabes que salió bien?

SINTAXIS CON VARIABLES

`torch.onnx.export(model, (`*ejemplo*`,), "`*archivo*`", dynamo=False, opset_version=`*opset*`, input_names=["`*entrada*`"], output_names=["`*salida*`"], dynamic_axes={"`*entrada*`": {0: "batch"}, "`*salida*`": {0: "batch"}})`

El exportador TorchScript, el que usa el C01: ejecuta el modelo con la entrada de ejemplo y registra lo que pasó (tracing). En torch 2.14 hay que pedirlo con `dynamo=False`, avisa con un `DeprecationWarning` y necesita el paquete `onnx`.

`torch.onnx.export(model, (`*ejemplo*`,), "`*archivo*`", dynamo=True, opset_version=18, input_names=["`*entrada*`"], output_names=["`*salida*`"], dynamic_shapes={"`*argumento*`": {0: torch.export.Dim("batch", min=1, max=`*max_batch*`)}}, external_data=False)`

El exportador nuevo, default desde torch 2.9: captura el programa entero con `torch.export`. No pone el modelo en eval (solo avisa): llama tú a `model.eval()` antes. El eje dinámico va en `dynamic_shapes`, con el nombre del argumento de `forward`. Necesita `onnx` y `onnxscript`. Sin `external_data=False`, los pesos salen en un `.onnx.data` aparte.

`onnx.checker.check_model(onnx.load("`*archivo*`"), full_check=True)`

Revisa que el archivo sea ONNX válido; con `full_check=True` además infiere las shapes (deduce la forma de cada tensor). Si todo está bien no imprime nada; si algo falla, lanza una excepción.

`ort.InferenceSession("`*archivo*`", providers=["CPUExecutionProvider"]).run(["`*salida*`"], {"`*entrada*`": `*x_float32*`})`

ONNX Runtime (un motor que ejecuta grafos ONNX) en CPU. Si corre con batch 1 y con el máximo, el eje dinámico funciona de verdad. La entrada tiene que ser un array de numpy `float32`: con `float64` falla.

VARIABLES

VARIABLE	QUÉ ES	EJEMPLO
ejemplo	tensor de entrada de ejemplo, con la forma real	`torch.randn(2, 3, 224, 224)`: batch 2, no 1
archivo	el archivo ONNX	`classifier.onnx`
opset	versión del conjunto de operadores ONNX	`17` con el exportador TorchScript; `18` con dynamo
entrada, *salida*	nombres de los tensores: el contrato con Triton y el gateway	`input` y `logits`
argumento	nombre del argumento de forward() que lleva el eje dinámico	`x`, porque `ResNet.forward(self, x)`
max_batch	batch máximo del eje dinámico	`32` = `max_batch_size` de Triton
x_float32	array de numpy float32 con la forma de la entrada	`torch.randn(7, 3, 224, 224).numpy()`
W, b	pesos y bias de una convolución	las convoluciones de ResNet no tienen bias: `b = 0`
γ, β, $\hat{\mu}$, $\hat{\sigma}^2$, ε	lo que aprendió y guardó el BatchNorm que sigue a esa convolución	53 pares Conv + BatchNorm en el ResNet-50
$P_{\max}$	parámetros que caben en un .onnx de un solo archivo (el límite de protobuf, el formato de serialización)	≈ 500 M en FP32; el C01 usa 24.5 M

FÓRMULAS

$$W' = W \cdot \frac{\gamma}{\sqrt{\hat{\sigma}^2 + \varepsilon}} \qquad b' = \beta + (b - \hat{\mu})\frac{\gamma}{\sqrt{\hat{\sigma}^2 + \varepsilon}} \qquad (\text{Conv} + \text{BatchNorm} \to \text{un Conv})$$

$$\text{un solo .onnx} \le 2\ \text{GB} \Rightarrow P_{\max} \approx \frac{2\ \text{GB}}{4\ \text{B}} \approx 500\text{M parámetros en FP32} \qquad \text{C01: } 24.5\text{M} \times 4\text{B} \approx 98\text{MB}$$

LAS PIEZAS

PIEZA	QUÉ HACE	EJEMPLO
ONNX	Un formato de archivo estándar para redes: grafo de operadores + pesos + firma de entradas y salidas, sin depender de PyTorch.	`classifier.onnx` ≈ 98 MB
opset	La versión del conjunto de operadores. "Opset 17" = cada operador con la semántica de esa versión del estándar; quien lo lee tiene que conocerla.	el C01 usa 17; torch 2.14 escribe el 20 si no pides otro

PIEZA	QUÉ HACE	EJEMPLO
dynamo=	Qué exportador usar. True (default desde 2.9): torch.export. False: TorchScript, deprecado.	**export_onnx.py usa False; con --dynamo, True**
opset_version=	El opset que se escribe. Con dynamo, si pides menos de 18 intenta convertir hacia abajo y puede fallar: el archivo queda en 18.	**el C01 pide 17 con TorchScript, y 18 con --dynamo**
input_names= / output_names=	Los nombres de los tensores dentro del grafo. Triton y el gateway los usan tal cual.	**["input"] y ["logits"]**
dynamic_axes=	Qué ejes quedan variables, en el exportador TorchScript. Sin esto, el batch queda fijo en el del ejemplo.	**{"input": {0: "batch"}, "logits": {0: "batch"}}**
dynamic_shapes=	Lo mismo para dynamo, con torch.export.Dim; la llave es el nombre del argumento de forward.	**{"x": {0: Dim("batch", min=1, max=32)}}**
external_data=	Solo dynamo; default True: los pesos van a un .onnx.data aparte. Hace falta por encima de 2 GB.	**el C01 pasa False: un solo archivo para trtexec y el registry**
do_constant_folding=	Precalcula lo que no depende de la entrada. Es del exportador TorchScript (True por defecto); la documentación de torch 2.14 lo marca como deprecado.	**el C01 lo escribe explícito**
optimize= / verify=	Solo dynamo. optimize (True por defecto) pliega constantes y quita redundancias; verify (False por defecto) corre el modelo en ONNX Runtime al exportar.	**el C01 verifica aparte, en verify()**
la entrada de ejemplo	El tensor con que se recorre el modelo. Con batch 1, algunos tracers (lo que registra cada operación) vuelven constante ese eje.	**torch.randn(2, 3, 224, 224)**
onnx.checker.check_model	Valida la estructura, la versión del formato (IR) y el opset; full_check=True agrega la inferencia de shapes.	**el primer chequeo de verify()**
ONNX Runtime (ORT)	Un runtime de Microsoft que ejecuta el grafo con execution providers (bibliotecas por hardware). Fuera de CPU, los providers se piden explícitos.	**onnxruntime 1.30.0 en el C01; la más nueva es 1.31.0 (octubre de 2026)**
optimización de grafo de ORT	Al crear la sesión, ORT pliega constantes, quita Dropout y fusiona Conv + BatchNorm. Todo viene activado por defecto (ORT_ENABLE_ALL).	**por eso el ONNX corre bien en CPU sin tocarlo**

PYTORCH: RESNET-50 CON CAPA FINAL (FC) DE 500 CLASES		ONNX: 122 NODOS
Conv2d ×53	→	**Conv** ×53 con el BatchNorm plegado adentro
BatchNorm2d ×53	↗	*plegado en* `Conv` *: no queda como nodo*
ReLU ×49 17 módulos, 49 llamadas	→	**Relu** ×49
suma residual (out += identity) ×16 en forward, no es un módulo	→	**Add** ×16
MaxPool2d ×1	→	**MaxPool** ×1
AdaptiveAvgPool2d ×1	→	**GlobalAveragePool** ×1
torch.flatten ×1	→	**Flatten** ×1
Linear ×1 la capa fc	→	**Gemm** ×1 la capa fc

FUSIÓN
`Conv2d` + `BatchNorm2d` → `Conv` : export en eval (TorchScript, plegado de constantes)

ENTRADA `input · FP32 · [batch, 3, 224, 224]` SALIDA `logits · FP32 · [batch, 500]` PESOS ≈ 98 MB

Con pesos aleatorios aparecen además nodos Identity: muchos bias plegados salen iguales y el exportador los comparte. Con un checkpoint entrenado no aparecen.

Qué hay dentro del classifier.onnx del C01, contado en un export real con torch 2.14 (exportador TorchScript, opset 17, en eval, con estadísticas de BatchNorm como las de un modelo entrenado). Los 53 BatchNorm desaparecen: el export los plegó en cada Conv. TensorRT fusiona más todavía (sección de TensorRT).

EJEMPLO REAL · EXPORT(): LOS DOS EXPORTADORES, CON SUS TRAMPAS ANOTADAS

PYTHON **reference/c01/model/export_onnx.py (export completo)**

```python
def export(model: torch.nn.Module, out: str, opset: int, dynamo: bool) -> None:
    """PyTorch → ONNX con eje de batch dinámico (Triton junta requests en batches de 1 a 32)."""
    # Ejemplo con batch 2 y no 1: con 1 algunos tracers especializan el eje y lo vuelven constante
    example = torch.randn(2, 3, 224, 224)
    if dynamo:
        # Exportador nuevo (torch.export). Verificado con torch 2.14: el eje dinámico se declara con
        # dynamic_shapes (la clave "x" es el nombre del argumento de ResNet.forward). OJO: si le
        # pides opset 17, intenta convertir hacia abajo, esa conversión falla con ReduceMean y el
        # archivo queda en opset 18 (solo imprime el error). Úsalo con --opset 18.
        batch = torch.export.Dim("batch", min=1, max=MAX_BATCH)
        torch.onnx.export(
            model, (example,), out, dynamo=True, opset_version=opset,
            input_names=[INPUT_NAME], output_names=[OUTPUT_NAME],
            dynamic_shapes={"x": {0: batch}},
            external_data=False,  # un solo archivo (~98 MB): trtexec y el registry no pierden el .data aparte
        )
    else:
        # Exportador TorchScript (legacy). torch ≥ 2.9 usa dynamo=True POR DEFECTO, así que hay que
        # pedirlo explícito: es el que produce opset 17 limpio (GlobalAveragePool) y respeta
        # dynamic_axes tal cual. Está deprecado ("will be removed"): fija la versión de torch del
        # job de export y planea migrar a --dynamo --opset 18.
        torch.onnx.export(
            model, (example,), out, dynamo=False, opset_version=opset,
            input_names=[INPUT_NAME], output_names=[OUTPUT_NAME],
            dynamic_axes={INPUT_NAME: {0: "batch"}, OUTPUT_NAME: {0: "batch"}},  # eje 0 = batch variable
            do_constant_folding=True,  # precalcula lo constante: grafo más simple para TensorRT
        )
```

- El ejemplo es de batch 2 a propósito: con batch 1, algunos tracers (lo que registra cada operación) fijan el eje como constante.

- Rama dynamo: el eje se declara con `torch.export.Dim`, y la llave `"x"` es el nombre del argumento de `ResNet.forward`. Si le pides opset 17, intenta convertir hacia abajo, falla con ReduceMean y el archivo queda en 18: por eso el script dice que se use con `--opset 18`.
- `external_data=False` deja un solo archivo de ≈ 98 MB. Con el default (`True`), dynamo escribe los pesos en un `.onnx.data` aparte; si alguien copia solo el `.onnx`, trtexec no encuentra los pesos.
- Rama TorchScript, la que corre el script por defecto: `dynamo=False` explícito, opset 17 y `dynamic_axes` por nombre de tensor, para la entrada y para la salida.
- Exportado con torch 2.14 y los pesos aleatorios del script (sin `--checkpoint`): ≈ 98 MB (97,959,677 bytes), opset 17, `input['batch', 3, 224, 224]` → `logits['batch', 500]`, y ningún nodo BatchNormalization: el export los plegó dentro de cada Conv.

CÓMO RECORDARLA

Modelo en eval, nombres fijos, eje de batch dinámico, opset fijo, y verificar el archivo, no la llamada: lee el opset real y las dims del .onnx, y córrelo en ONNX Runtime con batch 1 y con el máximo.

TRAMPA

Confiar en que el export salió como lo pediste. Con dynamo y opset 17, el archivo queda en 18 (solo imprime el error, no falla); y sin `dynamic_axes`, el batch queda fijo en el del ejemplo y ONNX Runtime responde `Got invalid dimensions for input: input … index: 0 Got: 7 Expected: 2`. Las dos cosas se atrapan leyendo el archivo exportado, como hace `verify()`.

En el curso: M8 · ONNX y ONNX Runtime · C01 · Archivos de referencia · M8 · TensorRT

4 Paridad: el modelo servido no es el evaluado (top-1, Δlogit, FP16)

¿Cómo pruebas que el ONNX y el engine FP16 predicen lo mismo que el PyTorch FP32 que evaluaste, y qué diferencia es normal?

SINTAXIS CON VARIABLES

`python model/parity_check.py --checkpoint` *checkpoint* **`--onnx`** *onnx* **`--images`** *muestra* **`--n`** *n*

Gate 1 (una prueba que frena la cadena si falla), en CPU: ONNX Runtime contra PyTorch FP32, las dos ramas con el mismo `preprocess.py` del gateway (el servicio que prepara las imágenes en producción). Atrapa exports rotos sin tocar una GPU.

`python model/parity_check.py --backend triton --triton-url` *host* **`:8000 --checkpoint`** *checkpoint* **`--images`** *muestra* **`--n`** *n*

Gate 2, en la T4: el engine TensorRT FP16 (el modelo ya compilado para esa GPU) servido por Triton, llamado de a una imagen como en producción. Exit 1 = no se registra.

`python model/parity_check.py --checkpoint` *checkpoint* **`--onnx`** *onnx* **`--min-top1 0.995 --min-top5 0.99 --max-abs-diff 0.5`**

Los tres umbrales, escritos con sus defaults. Se ajustan con datos cuando cambia el backend; nunca se bajan para que un candidato pase.

`polygraphy run` *onnx* **`--trt --fp16 --onnxrt`**

Sin Triton, en la máquina con GPU: Polygraphy (la herramienta de NVIDIA para depurar TensorRT) construye un engine FP16 y compara sus salidas con ONNX Runtime; también puede comparar capa por capa. Sus tolerancias por defecto (1e-5) son para FP32: con FP16, sube `--atol` y `--rtol`, o fallará aunque el engine esté bien. Es la alternativa que da el propio `parity_check.py` para el TensorRT 10 del C01.

VARIABLES

VARIABLE	QUÉ ES	EJEMPLO
checkpoint	el state_dict evaluado: la referencia	`resnet50_500.pt`
onnx	el ONNX candidato	`model/classifier.onnx`
muestra	carpeta con imágenes reales del set de evaluación	`una muestra del golden set (el set fijo de evaluación)`
n	máximo de imágenes a comparar	`default 256; el script recomienda ≥ 2,000`
host	dónde corre Triton	`localhost, en el runner con la T4`
N	imágenes comparadas	`2,000 → el gate tolera 10 cambios de top-1`
ref, got	logits de PyTorch FP32 y del candidato, de forma [N, 500]	`got: ONNX Runtime en CPU, o el engine vía Triton`
T_5	las 5 clases con el logit más alto: lo que ve quien publica	`el orden dentro del top-5 no importa`
ε	épsilon de máquina: la distancia de 1 al siguiente número que el formato puede guardar	`FP16 2⁻¹⁰; FP32 2⁻²³`

FÓRMULAS

$$\text{top1} = \frac{1}{N}\sum_{i=1}^{N} 1\left[\text{argmaxref}_i = \text{argmaxgot}_i\right] \qquad \text{top5} = \frac{1}{N}\sum_{i=1}^{N} \frac{|T_5(\text{ref}_i) \cap T_5(\text{got}_i)|}{5}$$

$$\max|\Delta| = \max_{i,k}|\text{ref}_{ik} - \text{got}_{ik}| \qquad \text{FP32: } \approx 10^{-4} \qquad \text{FP16: } 10^{-2}\text{–}10^{-1} \qquad \text{bug: } \gtrsim 1$$

$$\text{cambios de top-1 tolerados} = \lfloor (1 - 0.995)N \rfloor \qquad N = 256 \Rightarrow 1 \qquad N = 2{,}000 \Rightarrow 10$$

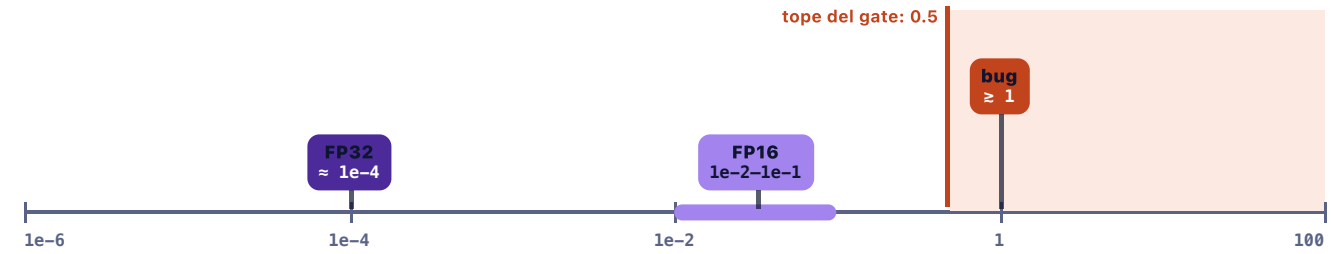
$$\varepsilon_{\text{FP16}} = 2^{-10} \approx 9.8 \times 10^{-4} \qquad \varepsilon_{\text{FP32}} = 2^{-23} \approx 1.2 \times 10^{-7} \qquad \text{paso de FP16 entre 8 y 16} = 2^{-7} \approx 0.008$$

LAS PIEZAS

PIEZA	QUÉ HACE	EJEMPLO
paridad (parity)	Probar que el artefacto servido predice lo mismo que el modelo que evaluaste.	`PyTorch FP32 contra ONNX Runtime, y contra el engine FP16 en Triton`
la referencia	PyTorch FP32 en eval() y con inference_mode(): el modelo tal como se evaluó.	`run_torch(), con batches de 32`
acuerdo top-1	La fracción de imágenes donde el 1.° lugar coincide: ¿cambia la sugerencia principal?	`≥ 99.5 % (--min-top1 0.995)`
solapamiento top-5	Cuántas de las 5 sugerencias coinciden, en promedio, sin importar el orden.	`≥ 99 % (--min-top5 0.99)`
max|Δlogit|	La mayor diferencia entre logits. Un salto de órdenes de magnitud es un bug, no redondeo.	`≤ 0.5 (--max-abs-diff): atrapa exports rotos`
ruido de FP32	Cambiar el orden de las sumas (otro runtime, otro batch) mueve los logits ≈ 1e-4.	`8.0e-05 en la paridad con ORT del README (512 imágenes)`

PIEZA	QUÉ HACE	EJEMPLO
redondeo de FP16	FP16 guarda 10 bits de mantisa: cerca de 10, los valores van de a 0.008.	**FP16 mueve los logits ≈ 1e-2 en promedio (1.3e-2 en el lab)**
overflow de FP16	Por encima de 65,504, FP16 da inf, y después aparecen NaN.	**raro en una CNN; más común en transformers**
margen del top-1	La distancia entre el 1.º y el 2.º logit. Si es menor que el ruido, ese top-1 puede cambiar.	**por eso el umbral es 99.5 % y no 100 %**
el mismo tensor	Las dos ramas reciben exactamente la misma entrada, hecha con el preprocess.py del gateway.	**load_images() lo importa desde gateway/app**
imágenes reales	Una muestra del set de evaluación. Las sintéticas solo prueban que el script corre.	**sin --images, gradientes + ruido codificados como JPEG**
exit code	0 = pasa; 1 = no pasa. Es lo que lee el CI para frenar la cadena.	**[parity] NO PASA: no promuevas este artefacto**
paridad de punta a punta	Imágenes de referencia por el gateway real, contra las predicciones del pipeline de evaluación.	**atrapa el skew de preprocesamiento (preparar la imagen distinto que en training) y los labels en otro orden**
Polygraphy	Herramienta de NVIDIA que compara TensorRT contra ONNX Runtime, también capa por capa.	**para encontrar la capa que diverge y dejarla en FP32**

LA ESCALA DE MAX|ΔLOGIT| (LOGARÍTMICA): QUÉ DIFERENCIA ES NORMAL



A la izquierda del tope el gate pasa; a la derecha, frena.

1 **ONNX Runtime FP32 contra PyTorch FP32** CPU

```
python model/parity_check.py --checkpoint resnet50_500.pt --onnx model/classifier.onnx --images /data/eval_sample --n 2000
```

En CPU, en cualquier máquina. Esperas top-1 ≈ 100 % y max|Δlogit| ≈ 1e-4.

ATRAPA: un export roto, un opset mal convertido, un eje de batch fijo.

2 **TensorRT FP16 en Triton contra PyTorch FP32** GPU

```
python model/parity_check.py --backend triton --triton-url localhost:8000 --checkpoint resnet50_500.pt --images /data/eval_sample --n 2000
```

En la T4, de a una imagen. Esperas top-1 ≥ 99.5 %, un |Δlogit| medio ≈ 1e-2 y un máximo muy por debajo de 0.5.

ATRAPA: una capa que diverge o desborda en FP16, un engine construido mal.

3 **De punta a punta por el gateway** SERVICIO

```
curl -s -H "Content-Type: image/jpeg" --data-binary @foto.jpg "localhost:8080/v1/classify?top_k=5"
```

En staging: imágenes de referencia por el gateway real, contra las predicciones del pipeline de evaluación.

ATRAPA: skew de preprocesamiento, labels en otro orden, otro dtype.

en CPU · en la GPU · por el servicio entero · lo que atrapa

Tres pruebas en orden, de la más barata a la más completa; cada una atrapa algo que la anterior no ve. La escala de |Δlogit| dice qué pasó: ≈ 1e-4 es aritmética, ≈ 1e-2 es FP16 y ≥ 1 es un bug.

EJEMPLO REAL · COMPARE() Y EL GATE: LAS MÉTRICAS, LOS UMBRALES Y EL EXIT CODE

PYTHON reference/c01/model/parity_check.py (extractos; # … marca líneas omitidas)

```python
def compare(ref: np.ndarray, got: np.ndarray) -> dict:
    """Métricas de paridad que importan para el producto (top-k), no solo diferencias numéricas."""
```

```
    top1 = float((ref.argmax(1) == got.argmax(1)).mean())  # ¿cambia la sugerencia principal?
    ref5 = np.argsort(-ref, 1)[:, :5]  # las 5 sugerencias que ve el vendedor
    got5 = np.argsort(-got, 1)[:, :5]
    overlap5 = float(np.mean([len(set(a) & set(b)) / 5 for a, b in zip(ref5, got5)]))  # orden dentro del top-5 no
importa
    diff = np.abs(ref - got)
    return {"n": len(ref), "top1_agreement": top1, "top5_overlap": overlap5,
            "max_abs_diff": float(diff.max()), "mean_abs_diff": float(diff.mean())}
# …
    p.add_argument("--n", type=int, default=256, help="máximo de imágenes a comparar")
    p.add_argument("--min-top1", type=float, default=0.995, help="umbral del gate: acuerdo top-1 ≥ 99.5 %%")
    p.add_argument("--min-top5", type=float, default=0.99, help="solapamiento medio del top-5")
    # FP32 vs FP32 da ~1e-4; TensorRT FP16 da ~1e-2: el umbral depende del backend (ajústalo con datos)
    p.add_argument("--max-abs-diff", type=float, default=0.5, help="tope de |Δlogit| (atrapa exports rotos)")
# …
    ok = (m["top1_agreement"] >= args.min_top1 and m["top5_overlap"] >= args.min_top5
          and m["max_abs_diff"] <= args.max_abs_diff)
    print("[parity] PASA" if ok else "[parity] NO PASA: no promuevas este artefacto")
    return 0 if ok else 1  # CI lee el exit code
```

- El acuerdo top-1 y el solapamiento del top-5 miden lo que ve el producto: la sugerencia principal y las 5 que se muestran. El top-5 se compara como conjunto: el orden de adentro no importa.
- `max_abs_diff` se reporta y tiene tope. FP32 contra FP32 da ≈ 1e-4 y TensorRT FP16 ≈ 1e-2; el tope de 0.5 queda muy arriba del redondeo y muy abajo de un export roto (los logits de otro modelo difieren en decenas).
- Un mismo script para los dos gates: `--backend ort` compara el ONNX en CPU; `--backend triton`, el engine servido. Si un backend cambia la escala del ruido, los umbrales se ajustan con datos.
- El default de `--n` es 256: con 99.5 % eso tolera un solo cambio de top-1. El script recomienda ≥ 2,000 imágenes (10 cambios): más resolución para decidir.
- El exit code es la interfaz con el CI: 1 frena la cadena antes del registry. Un 97 % en FP16 no se arregla bajando el umbral: en una CNN es un bug, no redondeo.

CÓMO RECORDARLA

1e-4 · 1e-2 · ≥ 1: aritmética, FP16, bug. La paridad dice si es **el mismo modelo**; el golden set por slices dice si es **un buen modelo**. Son dos preguntas distintas.

TRAMPA

Bajar el umbral cuando el engine FP16 da 97 % de acuerdo. En una CNN, FP16 suele pasar el 99.5 % con holgura: 97 % es un bug, no redondeo. Corre primero `--backend ort`: si el ONNX en FP32 ya falla, el problema está en el export; si pasa, busca la capa que diverge en el engine. Y no tomes como gate una corrida sin `--checkpoint` o con imágenes sintéticas: solo prueba que el script corre.

En el curso: M8 · Cómo validar una optimización · C01 · Evaluación offline y online · M10 · El validation gate: cuándo un modelo puede reemplazar al actual · M9 · Training-serving skew: causas, detección y prevención

5 TensorRT: qué es, trtexec, precisión, perfiles de optimización, portabilidad del engine

¿Cómo convierte TensorRT tu ONNX en un engine rápido para la T4, y por qué ese engine no sirve en otro tipo de GPU ni con otra versión de TensorRT?

SINTAXIS CON VARIABLES

```
trtexec --onnx=modelo.onnx
--saveEngine=engine.plan --fp16 \
  --minShapes=entrada:min x3x224x224 \
  --optShapes=entrada:opt x3x224x224 \
  --maxShapes=entrada:max x3x224x224
```

TensorRT 10 (NGC 24.08: la release de agosto de 2024 de los contenedores de NVIDIA, la de C01). Corre en la GPU de destino, dentro de la imagen `nvcr.io/nvidia/tensorrt`, y deja un **engine**: el modelo compilado para ese tipo de GPU y esa versión de TensorRT. `--fp16` **permite** FP16: TensorRT elige capa por capa y deja en FP32 lo que no conviene. Min, opt y max forman el **perfil de optimización** (el rango de shapes que acepta el engine).

```
python -m modelopt.onnx.autocast --onnx_path
modelo.onnx \
  --output_path modelo_fp16.onnx --keep_io_types
trtexec --onnx=modelo_fp16.onnx
--saveEngine=engine.plan \
  --minShapes=entrada:min x3x224x224 \
  --optShapes=entrada:opt x3x224x224 \
  --maxShapes=entrada:max x3x224x224
```

TensorRT 11 (NGC 26.06 en adelante; la 26.09 trae la 11.3). Las redes son **strongly typed** (de tipos fijos): el tipo de cada tensor sale del ONNX y TensorRT no lo cambia. Si le pasas `--fp16`, `--int8` o `--best`, trtexec sale con error. FP16 se pide antes, con ModelOpt AutoCast (una herramienta de NVIDIA que reescribe el ONNX en FP16). `--keep_io_types` deja la entrada y la salida en FP32; sin él también pasan a FP16 y ya no calzan con el `TYPE_FP32` del `config.pbtxt`. INT8 y FP8 van con nodos Q/DQ (marcas de cuantización en el ONNX) que agrega ModelOpt.

```
trtexec --loadEngine=engine.plan
--shapes=entrada:batch x3x224x224
```

Mide el engine solo: sin cola, sin red y sin Triton. Usa el batch que llega online (1–3 en C01), no solo el de opt. En TensorRT 10 agrega `--dumpProfile --separateProfileRun` para ver el tiempo de cada capa en una corrida aparte, sin frenar la medición principal; en 11 basta `--dumpProfile`, porque esa corrida aparte ya va siempre.

```
--timingCacheFile=cache                # guarda y
reusa las mediciones de tácticas
--builderOptimizationLevel=nivel      # 0 a 5,
default 3: más alto = build más largo
--hardwareCompatibilityLevel=ampere+   # un engine
para Ampere y posteriores
--versionCompatible                    # corre
también con TensorRT más nuevos
```

Flags de build opcionales (trtexec de TensorRT 10.3). Los dos modos de compatibilidad pueden ser más lentos que un engine normal. `ampere+` exige compute capability (el número de generación de la GPU) ≥ 8.0 y la T4 es 7.5: en C01 no sirve.

VARIABLES

VARIABLE	QUÉ ES	EJEMPLO
modelo.onnx	el ONNX exportado: la entrada del build	`model/classifier.onnx (≈ 98 MB, opset 17)`
engine.plan	dónde queda el engine serializado (el archivo .plan)	`triton/model_repository/classifier/1/model.plan`
entrada	nombre del tensor de entrada en el ONNX	`input`
min, *opt*, *max*	batch mínimo, el batch para el que TensorRT elige los kernels más rápidos (los programas que corren en la GPU) y el máximo	`1, 16, 32`
modelo_fp16.onnx	el ONNX que deja AutoCast, ya en precisión mixta (TensorRT 11)	`no existe en C01: C01 usa TensorRT 10.3`
batch	batch con que mides el engine	`1–3 (lo que llega online); 16 en el chequeo final de build_engine.sh`
cache	archivo donde se guardan las mediciones del build (timing cache)	`t4.cache (C01 no lo usa)`
nivel	cuánto tiempo se da el builder (el compilador de TensorRT) para probar opciones	`3, el default (C01 no lo cambia)`
qps	inferencias por segundo que reporta trtexec en Throughput	`se mide en la T4`
b	batch de la medición	`16 → imágenes/s = qps × 16`
H2D, D2H	copia de la entrada a la GPU y de la salida de vuelta (el trtexec de TensorRT 11 no las mide por defecto; --includeDataTransfers las suma)	`602,112 B de entrada por imagen`
GPU Compute	tiempo de los kernels en la GPU	`≈ 3–6 ms p50 con batch 1–8 (presupuesto de C01)`
max_batch_size	el techo de batch de config.pbtxt	`32`
$\max_{\text{perfil}}$	el batch máximo del perfil del engine	`32 (--maxShapes)`

FÓRMULAS

$\text{imágenes/s} = \text{qps} \times b$ (trtexec cuenta inferencias: cada una es un batch de b)

$\text{Latency} = \text{H2D} + \text{GPU Compute} + \text{D2H}$

$\text{min} \le \text{opt} \le \text{max}$ $\text{max_batch_size} \le \text{max}_{\text{perfil}}$ $(1 \le 16 \le 32,\ 32 \le 32)$

LAS PIEZAS

PIEZA	QUÉ HACE	EJEMPLO
builder	La parte de TensorRT que compila: lee el ONNX, optimiza el grafo y mide kernels en tu GPU.	`lo corre trtexec dentro de tensorrt:24.08`
engine (.plan)	El modelo ya compilado y guardado en un archivo: corre rápido, pero solo en ese tipo de GPU y con esa versión de TensorRT.	`model.plan, ≈ 50–150 MB para ResNet-50`
parser de ONNX	Traduce cada nodo del ONNX a una capa de TensorRT. Si un operador no tiene traducción, el build falla.	`el opset 17 de export_onnx.py`
fusión de capas (layer fusion)	Junta capas seguidas en un solo kernel: el resultado intermedio no vuelve a la memoria de la GPU.	`Conv + ReLU; Conv + suma del atajo + ReLU`
BatchNorm plegado	En inferencia, BatchNorm es una escala y un corrimiento fijos: se mete en los pesos de la conv anterior.	`el export de C01 ya lo hizo: el ONNX no trae nodos BatchNormalization`
precisión	Cuántos bits usa cada número: FP32, FP16, INT8, FP8. Menos bits = más rápido y algo de error numérico.	`FP16 en C01: los logits se mueven ≈ 1e-2`
tácticas (tactics)	Las implementaciones posibles de cada capa. El builder las cronometra en tu GPU y se queda con la más rápida.	`por eso el build tarda minutos y el resultado es de esa GPU`
perfil de optimización (optimization profile)	El rango de shapes que acepta una entrada dinámica: min, opt y max. Fuera del rango, la inferencia falla.	`1 / 16 / 32`
timing cache	Archivo con las mediciones de tácticas: el siguiente build en la misma GPU tarda menos.	`--timingCacheFile`
strongly typed	Red donde los tipos salen del modelo y TensorRT no elige la precisión. Obligatorio desde TensorRT 11.	`en 11, --fp16 hace fallar a trtexec`
ModelOpt AutoCast	Herramienta de NVIDIA que convierte el ONNX a precisión mixta FP16 antes del build. Con --keep_io_types, la entrada y la salida siguen en FP32.	`python -m modelopt.onnx.autocast --onnx_path … --keep_io_types`
nodos Q/DQ	Pares QuantizeLinear/DequantizeLinear en el ONNX: dicen dónde cuantizar a INT8 o FP8 y con qué escala.	`los genera ModelOpt; C01 no usa INT8`
compute capability (sm_XX)	El número de arquitectura de la GPU. Los kernels del engine se eligen para ese número.	`T4 = 7.5 (sm_75); L4 = 8.9`
hardware compatibility	Un engine para varias arquitecturas: ampere+ (compute capability ≥ 8.0) o, desde TensorRT 10.9, cualquier GPU con la misma compute capability. Puede ser más lento.	`ampere+ no cubre la T4`
version compatibility	El engine lleva adentro un runtime chico (el lean runtime) y corre con TensorRT más nuevos. Puede ser más lento; Triton lo carga solo con --backend-config=tensorrt,version-compatible=true.	`un engine 10 compatible corre con TensorRT 11`
trtexec	La herramienta de línea de comandos de TensorRT: construye engines y los mide solos.	`build_engine.sh la corre en la T4`

10 → 7 → 3 operaciones por bloque, de PyTorch al engine



El atajo: la entrada del bloque salta las tres convs y entra a la suma (Add); en 4 de los 16 bloques, los que cambian de tamaño, el atajo pasa antes por su propia Conv 1×1

- **PYTORCH EN EVAL():** 10 operaciones
- **ONNX DE EXPORT_ONNX.PY (OPSET 17):** 7 nodos: el export plegó cada BatchNorm en los pesos de su Conv
- **ENGINE TENSORRT EN LA T4:** 3 capas: cada conv lleva su activación y la última también la suma del atajo, en un solo kernel

la entrada y la salida del engine siguen en **FP32**: TensorRT agrega conversiones en los bordes

MÍRALO EN TU ENGINE: `trtexec --loadEngine=model.plan --dumpLayerInfo` — los nombres de las capas fusionadas se unen con " + "

¿Dónde carga este engine?

	T4 (compute capability 7.5)	otra GPU (otra compute capability)
TensorRT 10.3 (NGC 24.08)	✓ **carga: es la del build**	✕ no carga: `generated on an incompatible device`
otro TensorRT (otra release de NGC)	✕ no carga: `not compatible with this version of TensorRT`	✕ no carga

Salvo que lo construyas con un modo de compatibilidad (`--hardwareCompatibilityLevel` o `--versionCompatible`): opcional y más lento.

una operación · junta varias: plegadas en el export o fusionadas en un kernel · capa del engine, en la GPU

Un bloque bottleneck de ResNet-50 (hay 16) en PyTorch, en el ONNX de C01 y en el engine. Ilustrativo: qué fusiona exactamente depende de la versión de TensorRT y de la GPU; lo confirmas con trtexec --dumpLayerInfo.

EJEMPLO REAL · BUILD_ENGINE.SH: EL ENGINE FP16 SE COMPILA EN LA T4, CON LA MISMA RELEASE QUE TRITON

BASH · ZSH **reference/c01/model/build_engine.sh (líneas 27–52)**

```
# trtexec viene dentro de la imagen oficial de TensorRT: no instalas nada en el host.
#   --gpus all            expone la GPU al contenedor (requiere NVIDIA Container Toolkit)
#   --user                el .plan queda con tu usuario, no como root
#   --fp16                habilita kernels FP16 (Tensor Cores de la T4); TensorRT deja en FP32 lo que no convenga.
#                         Es TensorRT 10 (NGC 24.08). En TensorRT 11 (NGC 26.06+) ya no existe: conviertes el
#                         ONNX a FP16 con ModelOpt AutoCast y construyes el engine sin flags de precisión.
#   --minShapes/opt/max   perfil de optimización del batch dinámico: 1 ≤ batch ≤ 32, afinado para 16
#                         (max = max_batch_size de config.pbtxt). Ojo: online los batches son de ≈ 1–4
#                         (ver config.pbtxt y c01 · Serving); opt 16 es un punto de partida pensado para
#                         picos y batches grandes, no "el batch típico". Compara con perf_analyzer contra
#                         un engine con --optShapes=input:4x3x224x224 y quédate con el de mejor p99 a
#                         250–500 req/s por GPU.
#   --shapes              batch con el que trtexec hace su benchmark rápido al terminar (sanity check;
#                         la medición seria es perf_analyzer contra Triton)
docker run --rm --gpus all \
  --user "$(id -u):$(id -g)" \
  -v "$ROOT:/work" -w /work \
  "nvcr.io/nvidia/tensorrt:${NGC_RELEASE}-py3" \
  trtexec \
    --onnx="$ONNX" \
```

```
    --saveEngine="$PLAN_DIR/model.plan" \
    --fp16 \
    --minShapes=input:1x3x224x224 \
    --optShapes=input:16x3x224x224 \
    --maxShapes=input:32x3x224x224 \
    --shapes=input:16x3x224x224
```

- `--fp16` (TensorRT 10) deja que el builder use FP16 donde rinde. La entrada y la salida del engine siguen en FP32: por eso `config.pbtxt` declara `TYPE_FP32` y el gateway manda FP32. Los logits cambian ≈ 1e-2 frente a PyTorch FP32; el gate de paridad (sección 4) lo controla.
- El perfil 1 / 16 / 32: el engine acepta batches de 1 a 32 y afina los kernels para 16. El máximo coincide con `max_batch_size: 32` de Triton. Online llegan batches de ≈ 1–3, así que el script propone comparar con un engine de `--optShapes=input:4x3x224x224` usando perf_analyzer.
- `--shapes=input:16x3x224x224` solo fija el batch del benchmark corto que trtexec corre al terminar. Su Throughput viene en qps: con batch 16, imágenes/s = qps × 16. La medición que decide es perf_analyzer contra Triton (sección 9).
- La imagen `nvcr.io/nvidia/tensorrt:${NGC_RELEASE}-py3` con `NGC_RELEASE=24.08` trae TensorRT 10.3, la misma versión que `tritonserver:24.08` (Triton 2.49.0 con TensorRT 10.3.0.26). `--gpus all` le da la GPU al contenedor y `--user` deja el .plan a tu nombre.
- El comentario de TensorRT 11 es la migración pendiente: con NGC 26.06 o más nueva, este comando falla en `--fp16`. El ONNX pasa antes por AutoCast con `--keep_io_types` (así la entrada y la salida siguen en FP32, como pide `config.pbtxt`), el engine se construye sin flags de precisión y la paridad FP16 se repite.

CÓMO RECORDARLA

TensorRT es un compilador: el ONNX es el código fuente y el engine, el binario para una GPU (T4, sm_75) y una versión de TensorRT (10.3). Se compila donde va a correr, con el rango de batch que va a ver, y se mide con el batch real.

TRAMPA

subir solo la imagen de Triton (por ejemplo, de 24.08 a 26.09 por parches de seguridad) sin reconstruir el engine. La 26.09 trae TensorRT 11.3 y, por defecto, un engine solo carga con la versión de TensorRT que lo construyó: el modelo no carga y, con `--exit-on-error=true`, Triton se cierra y el pod entra en CrashLoopBackOff (se reinicia una y otra vez). Y en TensorRT 11 el mismo `build_engine.sh` falla, porque `--fp16` ya no existe. Las imágenes `tensorrt` y `tritonserver` suben juntas, con el mismo YY.MM.

En el curso: M8 · TensorRT · M8 · Cómo validar una optimización · M4 · T4, L4, A10G, A100, H100: cuándo usar cada una · C01 · Archivos de referencia · Números · Optimización y serving: rangos y umbrales

6 El model repository y config.pbtxt

¿Qué carpetas y qué configuración necesita Triton para cargar tu modelo, y qué pasa cuando algo no coincide?

SINTAXIS CON VARIABLES

```
repositorio/
└── modelo/
    ├── config.pbtxt
    └── versión/
        └── archivo
```

El **model repository** es una carpeta (local o en la nube: `s3://`, `gs://` o `as://`). Cada modelo tiene su carpeta y cada versión, una subcarpeta con número. Triton ignora las subcarpetas que no son números o que empiezan con 0. Nombres por defecto: `model.plan` (TensorRT), `model.onnx` (ONNX Runtime), `model.pt` (PyTorch), `model.py` (Python).

```
name: "modelo"
platform: "plataforma"
max_batch_size: max_batch
input [ { name: "entrada", data_type: tipo, dims: [ dims ] } ]
output [ { name: "salida", data_type: tipo, dims: [ dims ] } ]
```

Lo mínimo: quién lo ejecuta (`platform` o `backend`) y el contrato de tensores (nombre, tipo y shape de cada entrada y salida), con los nombres del ONNX o del engine. Con `max_batch_size` mayor que 0, `dims` va sin el eje de batch: Triton antepone -1. Con 0 no hay batching y `dims` es la shape completa.

```
tritonserver --model-repository=ruta \
  --model-control-mode=modo \
  --disable-auto-complete-config \
  --strict-readiness=true --exit-on-error=true
```

Modos: `none` (default) carga todo al arrancar y no vuelve a mirar la carpeta; `poll` recarga lo que cambia (la doc no lo recomienda en producción); `explicit` carga solo lo que pidas: con `--load-model` al arrancar o por la API de control de modelos. `--disable-auto-complete-config`: Triton usa tu config tal cual, sin completarlo leyendo el modelo.

```
curl -s host:8000/v2/models/modelo/config
curl -s -X POST host:8000/v2/repository/index
```

`/config` devuelve, en JSON, el config que Triton cargó de verdad, con los defaults que agregó (por ejemplo, `instance_group`). El índice lista cada modelo con su versión, su estado (READY o UNAVAILABLE) y el motivo.

VARIABLES

VARIABLE	QUÉ ES	EJEMPLO
repositorio	carpeta raíz que recibe --model-repository	`/models dentro de la imagen`
modelo	nombre del modelo = nombre de su carpeta	`classifier`
versión	subcarpeta numérica de la versión	`1`
archivo	el modelo, con su nombre por defecto	`model.plan (lo genera build_engine.sh; no va en git)`
plataforma	quién lo ejecuta: platform (o backend)	`tensorrt_plan`
max_batch	techo del batch; 0 = sin batching	`32 (= max del perfil del engine)`
entrada, *salida*	nombres de los tensores del modelo	`input, logits`
tipo	tipo de dato del config	`TYPE_FP32 (en el cliente se escribe FP32)`
dims	shape de una muestra, sin el batch	`3, 224, 224 y 500`
ruta	ruta o URL del repository	`/models (o s3://bucket/models)`
modo	modo de control de modelos	`none`
host	dónde escucha Triton	`localhost con kubectl port-forward`

FÓRMULAS

$$\text{shape completa} = \begin{cases} [-1] \| \text{dims} & \text{max_batch_size} > 0 \\ \text{dims} & \text{max_batch_size} = 0 \end{cases} \qquad [-1] \| [3, 224, 224] = [-1, 3, 224, 224]$$

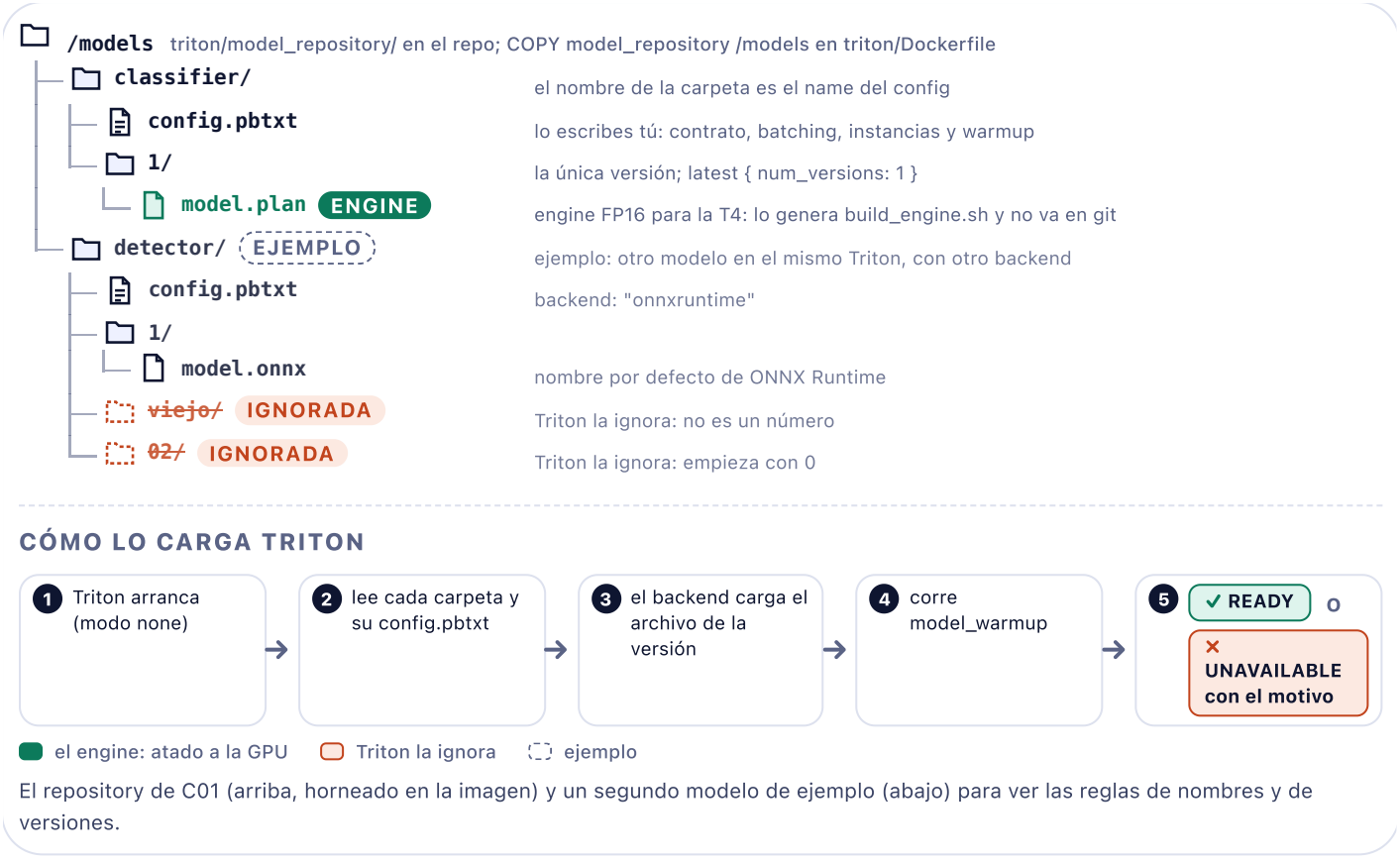
$$\text{bytes por imagen} = 3 \times 224 \times 224 \times 4\ \text{B} = 602{,}112\ \text{B} \approx 602\ \text{KB}$$

$$\text{entrada a batch máximo} = 32 \times 602\ \text{KB} \approx 19\ \text{MB}$$

LAS PIEZAS

PIEZA	QUÉ HACE	EJEMPLO
model repository	La carpeta de modelos que lee Triton, con un modelo por subcarpeta.	`triton/model_repository/, copiada a /models`
carpeta de versión	Subcarpeta con número. Las que no son números o empiezan con 0 se ignoran.	`classifier/1/`

PIEZA	QUÉ HACE	EJEMPLO
config.pbtxt	La configuración del modelo, en el formato de texto de protobuf (campos con dos puntos, llaves y listas) que define el esquema ModelConfig: contrato, batching, instancias y warmup.	**el de C01, 77 líneas comentadas**
name	Opcional. Si falta, Triton usa el nombre de la carpeta; si está, tiene que ser igual a la carpeta.	**classifier**
platform / backend	Quién ejecuta el modelo. TensorRT: platform tensorrt_plan o backend tensorrt. ONNX: onnxruntime_onnx u onnxruntime. Python: backend python.	**platform: "tensorrt_plan"**
default_model_filename	Cambia el nombre del archivo que busca el backend.	**C01 usa el default, model.plan**
max_batch_size	El batch más grande que acepta el modelo. Mayor que 0: Triton antepone el eje de batch. 0: sin batching.	**32**
input / output	El contrato de cada tensor: name, data_type y dims. Tiene que coincidir con el modelo y con el cliente.	**input FP32 [3, 224, 224] → logits FP32 [500]**
data_type	El tipo en el config: TYPE_FP32, TYPE_INT64, TYPE_STRING… El cliente usa el nombre del protocolo: FP32, INT64, BYTES.	**TYPE_FP32**
reshape	Para cuando la shape que manda el cliente y la que espera el modelo difieren.	**C01 no lo necesita**
version_policy	Qué versiones de la carpeta se cargan: latest (default, la más alta), all o specific.	**latest { num_versions: 1 }**
model_warmup	Requests de prueba que Triton corre al cargar. El modelo no queda listo hasta que terminan bien.	**batch 32, 3 veces, con ceros**
--disable-auto-complete-config	Usa el config tal cual, sin leer el modelo para completarlo. Igual agrega instance_group si falta.	**en el CMD de triton/Dockerfile**
--model-control-mode	Cuándo lee Triton el repository: none (al arrancar), poll (cada --repository-poll-secs) o explicit (con --load-model o por la API).	**none: un modelo nuevo llega en una imagen nueva**
--strict-readiness / --exit-on-error	/v2/health/ready da 200 solo con todos los modelos cargados / el proceso termina si algo falla al arrancar. Los dos son true por defecto.	**C01 los escribe igual, para que se lean**
UNAVAILABLE	Estado de un modelo que no cargó. El motivo sale en el log y en el índice del repository.	**un engine de otra versión de TensorRT**



El repository de C01 (arriba, horneado en la imagen) y un segundo modelo de ejemplo (abajo) para ver las reglas de nombres y de versiones.

EJEMPLO REAL · CONFIG.PBTXT DE C01: EL CONTRATO DEL MODELO

PROTOBUF (TEXTO) `reference/c01/triton/model_repository/classifier/config.pbtxt (líneas 1–29)`

```
# Configuración del modelo "classifier" en Triton Inference Server (caso c01).
# Formato: protobuf en texto (mensaje ModelConfig). Triton arranca con
# --disable-auto-complete-config: no "adivina" nada del engine; todo lo que el gateway asume
# (nombres, dtypes, shapes) está escrito y revisado aquí.
# El engine (1/model.plan) lo genera model/build_engine.sh y NO se versiona en git.

name: "classifier"  # = nombre del directorio; el gateway lo manda en MODEL_NAME
platform: "tensorrt_plan"  # backend TensorRT: carga 1/model.plan (nombre por defecto del archivo)
max_batch_size: 32  # hasta 32 requests en un batch; debe ser ≤ --maxShapes del engine (32)

# Solo la versión más nueva del directorio. Con la imagen "horneada" hay una sola versión; el
# rollback es volver a desplegar la imagen anterior, no cargar dos versiones en la misma GPU.
version_policy: { latest: { num_versions: 1 } }

input [
  {
    name: "input"  # = input_names del export ONNX y InferInput("input") del gateway
    data_type: TYPE_FP32  # el gateway manda FP32; TensorRT convierte a FP16 dentro del engine
    dims: [ 3, 224, 224 ]  # SIN el eje de batch: con max_batch_size > 0 Triton antepone [-1]
  }
]

output [
  {
    name: "logits"  # logits crudos: softmax y top-k los hace el gateway (así se puede calibrar)
    data_type: TYPE_FP32
    dims: [ 500 ]  # una salida por categoría hoja
  }
]
```

- Triton arranca con `--disable-auto-complete-config` (CMD de `triton/Dockerfile`): no completa nada leyendo el engine. Cada nombre, tipo y shape que asume el gateway está escrito aquí, a la vista.
- `platform: "tensorrt_plan"` sin `default_model_filename`: el backend busca `1/model.plan`. El Dockerfile hace `RUN test -f /models/classifier/1/model.plan` para que el build de la imagen falle si el engine no está.

- `max_batch_size: 32` con `dims: [ 3, 224, 224 ]`: Triton ve la entrada como [-1, 3, 224, 224], igual que el engine. Si escribieras el -1 en `dims`, serían 5 dimensiones y la carga falla con the model expects 4 dimensions … but the model configuration specifies 5 dimensions.
- `version_policy` con `latest` 1: una sola versión en la GPU. El rollback es volver a la imagen anterior, no tener dos engines cargados, cada uno con su VRAM.
- Los nombres `input` y `logits` salen de `export_onnx.py` y los usa el gateway: si cambias uno, tienes que cambiarlo en los tres archivos. Las líneas 60–77 agregan el warmup (batch 32, 3 veces): de paso prueban al cargar que el engine acepta el batch máximo.

CÓMO RECORDARLA

carpeta = nombre, subcarpeta = versión, config.pbtxt = contrato. Nombres, tipos, shapes y batch están tres veces: en el engine, en el config y en el cliente. Las tres copias tienen que coincidir.

TRAMPA

copiar la shape del ONNX tal cual, `dims: [ -1, 3, 224, 224 ]`, con `max_batch_size: 32`. Triton antepone otro -1 y le pide al engine 5 dimensiones: el modelo queda UNAVAILABLE y el error habla de an initial batch dimension because max_batch_size > 0. Con batching, `dims` describe una sola muestra.

En el curso: M7 · NVIDIA Triton Inference Server · C01 · Archivos de referencia · C01 · Serving · Docker y kubectl · GPUs en Docker

7 Dynamic batching, instancias y concurrencia

¿Cómo junta Triton requests sueltos en un batch, cuántas copias del modelo conviene correr y cuánto cuesta eso en latencia?

SINTAXIS CON VARIABLES

```
max_batch_size: max_batch
dynamic_batching {
  max_queue_delay_microseconds: espera_us
  preferred_batch_size: [ tamaños ]
}
```

El **dynamic batcher** junta requests de clientes distintos del lado del servidor. Un batch sale cuando llega a `max_batch_size` (o a un tamaño preferido) o cuando el primero lleva `max_queue_delay_microseconds` esperando; default de la espera: 0. `preferred_batch_size` casi nunca: la doc lo reserva para tamaños que rinden mucho mejor (varios perfiles de TensorRT).

```
dynamic_batching {
  default_queue_policy {
    timeout_action: REJECT
    default_timeout_microseconds: timeout_us
    max_queue_size: cola_max
  }
  priority_levels: niveles
  default_priority_level: nivel_default
}
```

La política de la cola. `default_timeout_microseconds` 0 = sin timeout; `max_queue_size` 0 = cola sin tope. Al vencer, REJECT (default) devuelve error y DELAY lo atiende después de los que siguen a tiempo. Con prioridades, 1 es la más alta y el default tiene que estar entre 1 y `priority_levels`.

```
instance_group [
  { count: n, kind: KIND_GPU, gpus: [ ids ] }
]
```

`count` copias en cada GPU de `gpus` (o en todas, si no lo pones). Sin `instance_group`: 1 instancia por GPU. `KIND_CPU` corre en CPU (en ONNX Runtime, 2 instancias si no pones `count`); TensorRT solo acepta GPU.

```
sum(rate(nv_inference_count{model="modelo"}[5m]))
  /
sum(rate(nv_inference_exec_count{model="modelo"}
[5m]))
```

El batch promedio que de verdad se forma: inferencias ÷ ejecuciones del modelo. Es la consulta del serving de C01; la corres en Prometheus (el sistema que junta las métricas), que las lee del puerto 8002.

VARIABLES

VARIABLE	QUÉ ES	EJEMPLO
max_batch	techo del batch; no puede pasar el máximo del perfil del engine	32
espera_us	espera máxima para juntar requests, en µs	2000 (2 ms)
tamaños	tamaños de batch preferidos	no se usan: el engine tiene un solo perfil, 1–32
timeout_us	cuánto puede esperar un request en la cola, en µs	100000 (100 ms, menos que los 150 ms del gateway)
cola_max	requests máximos en la cola; 0 = sin tope	no se fija
niveles	cuántas prioridades hay; 1 es la más alta	2, en la propuesta del serving de C01 (el config.pbtxt no las trae)
nivel_default	prioridad de los requests que no mandan una	2: la app manda priority=1 y los scripts quedan detrás
n	instancias por GPU	2
ids	GPUs donde van las copias; si falta, todas	no se pone: el pod ve una sola T4
modelo	label model de las métricas	classifier
λ	requests que llegan por segundo a cada GPU	250 en el pico de diseño; 500 si cae una GPU
W	tiempo promedio dentro de Triton: cola + copia + cómputo	≈ 8 ms (supuesto de M03)
L	requests dentro de Triton en promedio	≈ 2: dos instancias alcanzan
t_{ventana}	max_queue_delay_microseconds, en segundos	0.002
t_{llenar}	tiempo para juntar un batch lleno	128 ms, más de la mitad del p99 de 200 ms: por eso el batch no espera a llenarse y sale a los 2 ms
$\bar{b}$	batch promedio real	≈ 1–3 a 250 req/s; ≈ 2–4 en failover

FÓRMULAS

$L = \lambda \cdot W$ $\quad 250 \text{ req/s} \times 0.008 \text{ s} \approx 2 \text{ requests dentro de Triton}$

$$\text{llegadas en la ventana} = \lambda \cdot t_{\text{ventana}} = 250 \times 0.002 = 0.5$$

$$t_{\text{llenar}} = \frac{\text{max_batch_size}}{\lambda} = \frac{32}{250\ \text{req/s}} = 128\ \text{ms}$$

$$\bar{b} = \frac{\Delta \text{nv_inference_count}}{\Delta \text{nv_inference_exec_count}}$$

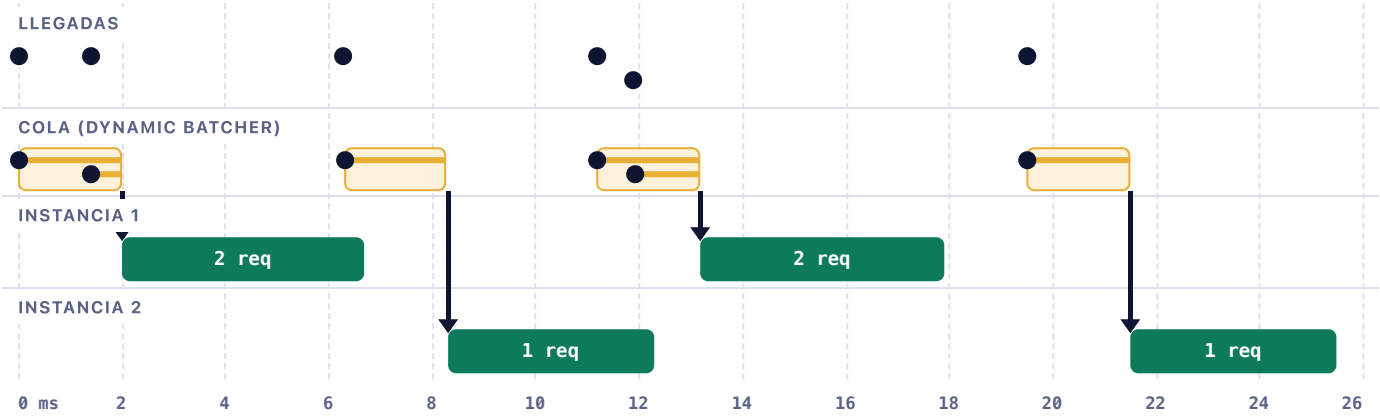
LAS PIEZAS

PIEZA	QUÉ HACE	EJEMPLO
dynamic batcher	El scheduler (la parte de Triton que decide qué request corre y cuándo) que junta requests sueltos de varios clientes en un batch, del lado del servidor.	`dynamic_batching { … } en config.pbtxt`
scheduler por defecto	Si no hay dynamic_batching ni otro scheduler, cada request se ejecuta sola, aunque max_batch_size sea 32.	`con --disable-auto-complete-config nadie lo agrega por ti`
max_queue_delay_microseconds	La espera máxima para juntar compañía. Con 0 no espera: igual arma el batch con lo que se acumuló mientras las instancias estaban ocupadas.	`2000 en C01`
batch real vs max_batch_size	max_batch_size es un techo. El batch que se forma depende de cuántos requests llegan.	`techo 32, real ≈ 1–3`
preferred_batch_size	Tamaños con los que el batch sale de inmediato. La doc lo desaconseja para la mayoría de los modelos.	`C01 no los pone`
default_queue_policy	Qué hace la cola: timeout, acción al vencer y tope de tamaño.	`REJECT a los 100 ms`
timeout_action	REJECT (default): devuelve error. DELAY: lo atiende después de los que siguen a tiempo.	`REJECT en C01`
allow_timeout_override	Deja que cada request pida un timeout más corto (el parámetro timeout de tritonclient, en µs).	`no se usa en C01`
priority_levels	Varias colas por prioridad: la 1 se atiende primero.	`propuesta del serving de C01: app en 1, scripts en 2`
preserve_ordering	Devuelve las respuestas en el orden de llegada. No se permite con más de un nivel de prioridad.	`no se usa`
instance_group	Cuántas copias del modelo corren y dónde. Cada copia es un contexto de ejecución con su propia memoria.	`count: 2, kind: KIND_GPU`
kind	KIND_GPU, KIND_CPU, KIND_MODEL o KIND_AUTO (el default: GPU si hay).	`KIND_GPU`
rate limiter	Opcional (--rate-limit=execution_count): reparte la ejecución entre instancias según recursos y prioridad.	`C01 no lo usa`
sequence batcher	Para modelos con estado: manda los requests de una misma secuencia a la misma instancia.	`no aplica a un clasificador`
client-side batching	El cliente junta imágenes antes de mandarlas. Online casi nunca conviene: suma latencia y Triton ya junta.	`C01: una imagen por request`

CUÁNDO SALE UN BATCH

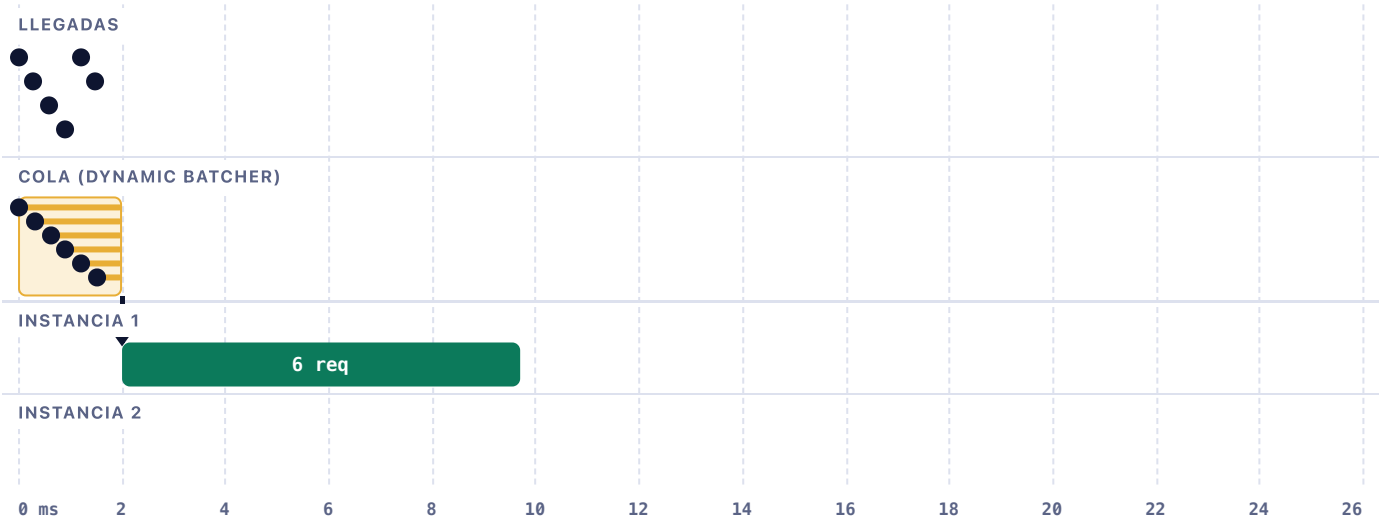
1. el batch sale cuando junta max_batch_size (32) o cuando el primero lleva 2 ms esperando
2. lo toma una instancia libre; si las dos están ocupadas, sigue juntando
3. con poco tráfico el costo es la espera: hasta 2 ms por request

1 **Carga normal de C01: ≈ 250 req/s por GPU**



6 requests en 4 ejecuciones: batch promedio 1.5; cada request espera 2 ms o menos en la cola

2 **Ráfaga: 6 requests en 1.5 ms (un pico o un failover)**



una ejecución de ≈ 7.7 ms atiende a los 6: el batch crece solo cuando hay tráfico

● llega un request — espera en la cola (hasta 2 ms) ■ el batch corre en la GPU: cuántos requests van juntos eje en ms, a escala

El bus que espera 2 ms: llegadas, ventana y batch despachado a la instancia libre, a escala. Ilustrativo, con los números de C01: un request cada ≈ 4 ms y un cómputo de ≈ 3.3 + 0.74 × b ms (≈ la tabla de M03).

EJEMPLO REAL · CONFIG.PBTXT DE C01: BATCHING, COLA E INSTANCIAS

PROTOBUF (TEXTO) `reference/c01/triton/model_repository/classifier/config.pbtxt (líneas 31–58)`

```
# Dynamic batching: junta requests individuales en un batch para usar bien la GPU (1 imagen por
# llamada deja la T4 casi ociosa). Espera como máximo 2 ms a que lleguen más requests. Con 2 GPUs,
# cada Triton recibe ≈ 175–250 req/s en el pico (350–500 req/s en total): en 2 ms llegan ≈ 0.35–0.5
# requests más y el batch promedio queda en ≈ 1–3 (≈ 2–4 si una GPU absorbe los 500 req/s del
# failover). Cuesta como mucho 2 ms del p99 de 200 ms; el batch crece solo en picos y failover.
# Mídelo, no lo supongas: rate(nv_inference_count) / rate(nv_inference_exec_count) = batch promedio.
dynamic_batching {
  max_queue_delay_microseconds: 2000
  # Sin preferred_batch_size a propósito: con un solo perfil de optimización (1–32) Triton arma el
  # batch más grande posible dentro de la espera; fijar tamaños solo tiene sentido con varios
  # perfiles/engines por tamaño, y mal elegidos agregan espera.
  default_queue_policy {
    # Protección ante sobrecarga: un request que lleva 100 ms en cola ya no llega a tiempo al
    # timeout del gateway (150 ms). Mejor rechazarlo y liberar GPU para requests que siguen vivos
```

```
    # que calcular una respuesta que nadie espera.
    timeout_action: REJECT
    default_timeout_microseconds: 100000
  }
}

# 2 instancias del engine en la MISMA GPU: mientras una copia datos o espera, la otra computa.
# Cada instancia reserva su propia memoria de activaciones; con ResNet-50 FP16 sobra en 16 GB.
instance_group [
  {
    count: 2
    kind: KIND_GPU
  }
]
```

- La cuenta de las líneas 32–35: con ≈ 250 req/s por GPU, en 2 ms llegan 250 × 0.002 = 0.5 requests más. El batch promedio queda en ≈ 1–3 (≈ 2–4 en failover) y la ventana cuesta como mucho 2 ms del p99 de 200 ms.
- Sin `preferred_batch_size`: con un solo perfil (1–32), lo mejor es el batch más grande que se forme dentro de la espera. Fijar tamaños tiene sentido con varios perfiles o engines por tamaño.
- `default_queue_policy` con REJECT a los 100 ms: un request que ya esperó eso no llega al timeout de 150 ms del gateway. Triton lo rechaza (se cuenta en `nv_inference_request_failure` con `reason="REJECTED"`) y el gateway responde 503 con Retry-After.
- `count: 2` en la misma T4: mientras una instancia copia datos o espera, la otra calcula. Cada una reserva su propia memoria de activaciones; con ResNet-50 FP16 son unos pocos GB en total, de 16.
- Little lo confirma: a 250 req/s y ≈ 8 ms dentro de Triton hay ≈ 2 requests adentro. Más instancias que requests no ayudan; el número final se elige midiendo con model_analyzer.

CÓMO RECORDARLA

el dynamic batcher es un bus que espera como mucho 2 ms a que suban más pasajeros. Si se llena (32 asientos), sale antes; si no, sale cuando vence la espera. Las instancias son buses en paralelo. Con poco tráfico el bus sale casi vacío y la espera es el costo; en un pico se llena solo.

TRAMPA

poner `max_batch_size: 32` y creer que ya hay batching. Sin el bloque `dynamic_batching`, Triton usa el scheduler por defecto y cada request va sola a la GPU. Con el autocompletado activo, el backend de TensorRT agrega el bloque si el batch máximo es mayor que 1; el de ONNX Runtime, solo si además tiene que completar `input` y `output`. Con `--disable-auto-complete-config`, como C01, nadie lo agrega. Escríbelo siempre y mídelo con inferencias ÷ ejecuciones.

En el curso: M3 · Batching: throughput a cambio de latencia · M3 · Little's law: la cuenta que ordena todo · C01 · Serving · M7 · NVIDIA Triton Inference Server · M4 · Cálculo de VRAM paso a paso

8 Clientes: HTTP, gRPC, protocolo v2, health y metadata, tritonclient

¿Cómo le habla tu servicio a Triton: qué endpoints usa, qué viaja por la red y qué cliente de Python eliges?

SINTAXIS CON VARIABLES

```
GET  /v2/health/live
GET  /v2/health/ready
GET  /v2/models/ modelo /ready
GET  /v2/models/ modelo
GET  /v2/models/ modelo /config
POST /v2/models/ modelo [/versions/ v ]/infer
```

El protocolo **KServe v2** (también llamado Open Inference Protocol: las rutas y el JSON estándar para hablarle a un servidor de modelos) por HTTP, en el puerto 8000. Health: 200 = sí, 400 = no, sin body. Dos casos más: mientras los modelos cargan al arrancar (con `--model-control-mode=none`) el servidor HTTP todavía no acepta conexiones, y al apagarse responde 503. `/v2/models/` *modelo* es la metadata (nombres, tipos y shapes); `/config` es una extensión de Triton.

```
import tritonclient.http as httpclient

client =
httpclient.InferenceServerClient(url=" host :8000")
inp = httpclient.InferInput(" entrada ", [ shape ],
" tipo ")
inp.set_data_from_numpy( arreglo , binary_data=True)
out = httpclient.InferRequestedOutput(" salida ",
binary_data=True)
res = client.infer(" modelo ", [inp], outputs=[out])
y = res.as_numpy(" salida ")
```

La URL va sin `http://`: si lo pones, el cliente falla con url should not include the scheme. El tipo es el nombre del protocolo (`FP32`, no `TYPE_FP32`) y la shape incluye el batch. La versión async es `tritonclient.http.aio`, la de C01.

```
import tritonclient.grpc as grpcclient

client =
grpcclient.InferenceServerClient(url=" host :8001")
inp = grpcclient.InferInput(" entrada ", [ shape ],
" tipo ")
inp.set_data_from_numpy( arreglo )
res = client.infer(" modelo ", [inp],
client_timeout= segundos )
```

gRPC (llamadas a funciones remotas sobre HTTP/2), en el puerto 8001: el tensor viaja como bytes crudos dentro de un mensaje protobuf (el formato binario de Google). `client_timeout` sí es un timeout del cliente, en segundos: si vence, lanza InferenceServerException con Deadline Exceeded. En los dos clientes, `timeout` es otra cosa: µs que Triton aplica en su cola.

```
curl -i  host :8000/v2/health/ready
curl -s  host :8000/v2/models/ modelo
```

Para mirar a mano. La primera muestra HTTP/1.1 200 OK o 400 Bad Request. La metadata es un JSON con `name`, `versions`, `platform`, `inputs` y `outputs` (nombre, datatype y shape, con -1 en el eje de batch).

VARIABLES

VARIABLE	QUÉ ES	EJEMPLO
modelo	nombre del modelo en el repository	`classifier`
v	versión explícita (opcional)	`no se usa: hay una sola`
host	dónde escucha Triton, sin http://	`triton.classifier.svc.cluster.local en EKS; triton en compose`
entrada, *salida*	nombres de los tensores, los del config	`input, logits`
shape	shape del request, con el batch	`1, 3, 224, 224`
tipo	datatype del protocolo	`FP32`
arreglo	arreglo de NumPy con ese dtype y esa shape	`float32 [1, 3, 224, 224] de preprocess.py`
segundos	timeout del cliente gRPC	`C01 usa HTTP: su tope es asyncio.wait_for, 150 ms`
B	bytes del tensor en binario	`602,112`
n	cantidad de valores del tensor	`150,528`
B_{JSON}	el mismo tensor con los números escritos como texto (≈ 20 B por número; M07 midió 20.6 con json.dumps)	`≈ 3 MB (3.1 MB en M07)`
λ	requests por segundo	`500 en el pico de diseño`
deadline	el presupuesto total del request en el gateway	`250 ms desde que llegó`

FÓRMULAS

$$B = \prod_i \text{shape}_i \times \text{bytes del tipo} = 1 \times 3 \times 224 \times 224 \times 4 = 602{,}112 \text{ B}$$

$B_{\text{JSON}} \approx n \times 20\ \text{B} = 150{,}528 \times 20\ \text{B} \approx 3\ \text{MB} (\approx 5\times)$

$\text{red} = B \times \lambda \times 8 = 602\ \text{KB} \times 500\ \text{req/s} \times 8 \approx 2.4\ \text{Gb/s}$

$t_{\text{Triton}} = \min(150\ \text{ms},\ \text{deadline} - \text{ahora})$

LAS PIEZAS

PIEZA	QUÉ HACE	EJEMPLO
KServe v2 (Open Inference Protocol)	Un contrato estándar de endpoints y JSON para servidores de modelos: Triton, KServe, MLServer, OpenVINO Model Server.	el gateway habla v2 por HTTP
extensión binaria	Header JSON corto + los bytes crudos del tensor. El header Inference-Header-Content-Length dice dónde termina el JSON.	binary_data=True en tritonclient
datatype del protocolo	BOOL, UINT8..., FP16, FP32, FP64, BYTES, BF16. En el config se escriben TYPE_*, y el texto es TYPE_STRING allá y BYTES aquí.	FP32
/v2/health/live	¿El proceso responde? Para la livenessProbe (el chequeo con que Kubernetes decide si reiniciar el contenedor).	livenessProbe de k8s/triton.yaml
/v2/health/ready	¿Puede atender? Con --strict-readiness, solo si todos los modelos están cargados.	startupProbe y readinessProbe
/v2/models/ *modelo* **/ready**	¿Ese modelo está cargado? Si no, 400 con un error en JSON.	is_model_ready() en el /readyz del gateway
metadata y config	Metadata: nombres, tipos y shapes. Config: el config.pbtxt cargado, en JSON (extensión de Triton).	get_model_metadata() y get_model_config()
gRPC	El mismo modelo de datos sobre HTTP/2: el servicio inference.GRPCInferenceService, en el puerto 8001.	perf_analyzer -i grpc
tritonclient	La librería oficial de Python: tritonclient.http, .http.aio, .grpc y .grpc.aio. Se instala con pip install "tritonclient[http]" o "[grpc]".	tritonclient[http]==2.72.0 en C01
InferInput / InferRequestedOutput	El tensor que mandas (nombre, shape, datatype) y la salida que pides. El cliente revisa el dtype y la shape del arreglo antes de mandar.	InferInput("input", [1, 3, 224, 224], "FP32")
InferenceServerException	El error que devuelve Triton (modelo no cargado, request rechazado); .message() trae el texto.	el gateway lo traduce a 503
timeout de infer()	No es del cliente: son µs que Triton aplica en su cola, y solo si el modelo tiene dynamic batching.	por eso C01 usa asyncio.wait_for
memoria compartida	Si cliente y Triton están en la misma máquina, el tensor pasa por memoria (del sistema o CUDA), sin red.	perf_analyzer --shared-memory system
gRPC y el balanceo de Kubernetes	Una conexión HTTP/2 lleva todos los requests, y el Service de Kubernetes reparte conexiones, no requests (balanceo L4): todo termina en un solo pod.	por eso C01 usa HTTP con un pool de conexiones

gateway: tritonclient.http.aio, un cliente por proceso ⇄ Triton, en el pod con la T4

8000 HTTP/REST, KServe v2

gateway y probes del kubelet

SALUD

kubelet	**GET** **/v2/health/live** EN EL C01: livenessProbe cada 10 s	← 200 vivo · 400 no
kubelet	**GET** **/v2/health/ready** EN EL C01: startupProbe hasta 5 min; readinessProbe cada 5 s	← 200 si todos los modelos están listos · 400 si no
gateway (/readyz)	**GET** **/v2/models/classifier/ready** EN EL C01: is_model_ready(), tope de 0.5 s	← 200 · 400 {"error": "Model version not ready"}

METADATA

tú, al depurar	**GET** **/v2/models/classifier** EN EL C01: get_model_metadata()	← input FP32 [-1, 3, 224, 224] · logits FP32 [-1, 500]

INFERENCIA

gateway	**POST** **/v2/models/classifier/infer** EN EL C01: header JSON + 602,112 bytes del tensor	← header JSON + 2,000 bytes de logits (500 × 4 B)

↑ sube		**602,112 B**
↓ baja		**2,000 B**

8001 gRPC

perf_analyzer o clientes que lo prefieran; la NetworkPolicy de C01 no lo abre al gateway

8002 HTTP /metrics, formato Prometheus

Prometheus (cada 15 s) y, a través de él, KEDA

MÉTRICAS

Prometheus	**GET** **:8002/metrics** EN EL C01: scrape cada 15 s	← texto con nv_inference_* y nv_gpu_*

Quién llama a qué en C01: el kubelet (el agente de Kubernetes en cada nodo) pregunta por la salud, el gateway por el modelo y la inferencia, y Prometheus lee las métricas. gRPC (8001) ofrece las mismas operaciones como llamadas RPC.

EJEMPLO REAL · MAIN.PY: LA LLAMADA A TRITON, CON SU PRESUPUESTO Y SUS ERRORES

PYTHON **reference/c01/gateway/app/main.py (líneas 147–171)**

```python
async def _infer(client, batch: np.ndarray, request_id: str, timeout: float = INFER_TIMEOUT_S) -> np.ndarray:
    """Llama a Triton y traduce sus fallas a códigos HTTP útiles para el cliente."""
    inp = triton_http.InferInput("input", list(batch.shape), "FP32")  # nombre, shape y dtype = config.pbtxt
    # binary tensor extension: 602 KB crudos detrás del header JSON, en vez de 150k floats como texto
    inp.set_data_from_numpy(batch, binary_data=True)
    out = triton_http.InferRequestedOutput("logits", binary_data=True)  # la respuesta también en binario
    try:
        # Verificado en tritonclient 2.72: el parámetro `timeout` de infer() NO es un timeout del
        # cliente, es un parámetro en microsegundos que Triton aplica en SU cola (solo con dynamic
        # batching). El único timeout del cliente aio es conn_timeout, global para toda la sesión.
        # Por eso el presupuesto por request se impone con asyncio.wait_for, que además cancela la
        # petición HTTP en vuelo.
        result = await asyncio.wait_for(
            client.infer(MODEL_NAME, [inp], outputs=[out], request_id=request_id),
            timeout=timeout,  # ≤ INFER_TIMEOUT_S: lo que quede del presupuesto del request
        )
    except TimeoutError:  # asyncio.TimeoutError es alias de TimeoutError desde Python 3.11
        # 504: Triton está vivo pero lento (cola llena, GPU saturada); el cliente cae a su fallback
```

```
        raise HTTPException(504, "timeout de inferencia") from None
    except aiohttp.ClientError:  # conexión rechazada/reseteada, DNS: Triton caído o reiniciando
        raise HTTPException(503, "Triton no disponible", headers={"Retry-After": "1"}) from None
    except InferenceServerException as exc:  # Triton respondió error: modelo no cargado, request rechazado
        detail = f"Triton rechazó la inferencia: {exc.message()}"
        raise HTTPException(503, detail, headers={"Retry-After": "1"}) from None
    return result.as_numpy("logits")[0]  # [500] logits del único elemento del batch
```

- `InferInput("input", list(batch.shape), "FP32")`: nombre, shape y tipo calcan el config (`TYPE_FP32` allá, `FP32` aquí). La shape es la completa, [1, 3, 224, 224]: el batch de 1 lo agrega `_decode_and_hash`.
- `binary_data=True` en la entrada y en la salida: viajan 602,112 bytes crudos detrás del header, no 150,528 números como texto (≈ 3 MB).
- El `timeout` de `infer()` no corta nada en el cliente (comentario de las líneas 154–158). El presupuesto se impone con `asyncio.wait_for`, que además cancela la petición en vuelo; el `conn_timeout=1.0` del cliente (línea 108) queda de respaldo.
- Cada falla tiene su código: TimeoutError → 504 (Triton lento), aiohttp.ClientError → 503 (Triton caído o reiniciando), InferenceServerException → 503 con el mensaje de Triton, por ejemplo un rechazo de la cola.
- Un solo cliente por proceso, creado en el lifespan con `conn_limit=64` : reusa las conexiones keep-alive. El `/readyz` del gateway llama `is_model_ready(MODEL_NAME)` con un tope de 0.5 s.

CÓMO RECORDARLA

dos preguntas de salud y una de trabajo: ¿vive? (live), ¿puede atender? (ready) e infer. El tensor viaja en binario: un header JSON corto y 602,112 bytes. Y el tipo cambia de nombre: `TYPE_FP32` en el config, `FP32` en el cable.

TRAMPA

pasarle `timeout=` a `infer()` creyendo que el cliente dejará de esperar. Ese parámetro son microsegundos que Triton aplica en su cola, y solo si el modelo tiene dynamic batching; el cliente sigue esperando. Con HTTP, C01 usa `asyncio.wait_for`; con gRPC existe `client_timeout`, en segundos.

En el curso: M7 · Protocolos: REST, gRPC, KServe v2 y OpenAI-compatible · C01 · Serving · M3 · Timeouts, reintentos y hedging · C01 · Archivos de referencia

9 Medir: perf_analyzer, métricas de Triton, uso de GPU

¿Cuánto aguanta una GPU, dónde se va la latencia y cómo sabes si el batching de verdad está pasando?

SINTAXIS CON VARIABLES

```
perf_analyzer -m modelo -u host:8000 -i http \
  --concurrency-range inicio:fin:paso \
  --input-data datos.json --percentile=99 \
  --measurement-interval ms
```

Carga cerrada: cada cliente virtual espera su respuesta antes de mandar otra. Sirve para encontrar el throughput máximo. Sin `--input-data` manda datos al azar; `-b` es el batch de cada request (1, como el gateway). Viene en la imagen `nvcr.io/nvidia/tritonserver:release-py3-sdk` (o con `pip install perf-analyzer`, que solo trae versiones para Linux).

```
perf_analyzer -m modelo -u host:8000 \
  --request-rate-range tasa --request-distribution poisson \
  --input-data datos.json --percentile=99
```

Carga abierta: manda una tasa fija de requests aunque el servidor se frene, como el tráfico de verdad (siempre que no se le acaben los hilos: usa `--async` o sube `--max-threads`). Es la que dice el p99 a esa tasa. La distribución por defecto es `constant`; `poisson` se parece más al tráfico real.

```
curl -s host:8002/metrics | grep 'model="modelo"' \
  | grep -E 'nv_inference_(count|exec_count|request_success|queue_duration_us|compute_infer_duration_us)'
```

Estas cinco son contadores (counters) acumulados desde que arrancó Triton: lo que importa es cuánto suben en un intervalo (`rate` en Prometheus), no el valor suelto. Las de GPU (`nv_gpu_utilization`, `nv_gpu_memory_used_bytes`) y `nv_inference_pending_request_count` son gauges: valen lo que marcan en ese momento y se leen directo, sin `rate`.

```
sum(rate(nv_inference_queue_duration_us{model="modelo"}[5m]))
  /
sum(rate(nv_inference_request_success{model="modelo"}[5m]))
  / 1000
```

Espera media en la cola, en ms: la alerta TritonQueueTimeHigh de C01 salta a 20 ms y KEDA (el autoscaler que agrega pods de Triton) escala con un objetivo de 10 ms. Incluye la ventana de batching: el request ya está en la cola mientras espera compañía.

VARIABLES

VARIABLE	QUÉ ES	EJEMPLO
modelo	modelo a medir (-m)	`classifier`
host	Triton, sin http://	`localhost con kubectl port-forward, o triton en compose`
inicio:*fin*:*paso*	clientes concurrentes: desde, hasta, de a cuánto	`1:32:2 (M03)`
datos.json	entradas reales, en el formato JSON de perf_analyzer	`tensores_reales.json: fotos ya preprocesadas`
ms	largo de cada ventana de medición, en ms (default 5000)	`10000`
tasa	requests por segundo, en carga abierta	`250 (pico de diseño por GPU) y 500 (failover)`
release	release de NGC de la imagen SDK	`24.08, la misma que el servidor`
X	throughput: requests atendidos por segundo	`el máximo de una T4 en C01 es un supuesto: ≈ 1,200 img/s`
N	clientes concurrentes de perf_analyzer	`8`
W	latencia media de cada request	`10 ms`
$\bar{t}_{\text{cola}}$	espera media en la cola de Triton	`≈ 1–3 ms normal; más de 20 ms sostenidos dispara la alerta`
pico	carga de diseño	`500 req/s`
máx. medido	el codo de la rampa, medido con perf_analyzer	`≈ 1,200 img/s → 600 planeadas (50 %)`

FÓRMULAS

$$X = \frac{N}{W} \qquad 8 \text{ clientes}/10 \text{ ms} = 800 \text{ req/s} \quad \text{(carga cerrada, Little)}$$

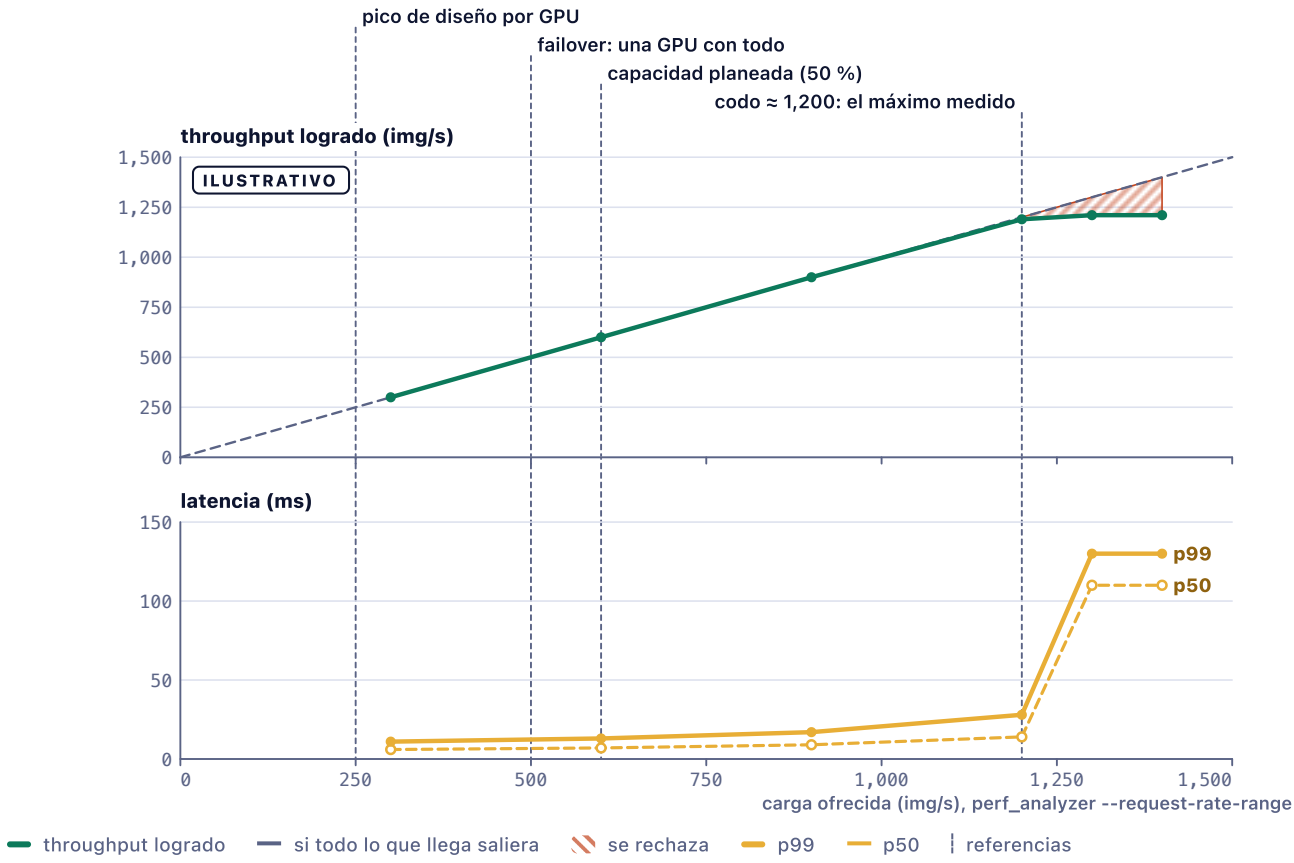
$$\bar{t}_{\text{cola}} = \frac{\Delta \text{nv_inference_queue_duration_us}}{\Delta \text{nv_inference_request_success}}$$

$$\text{GPUs} = \left\lceil \frac{\text{pico}}{0.5 \times \text{máx. medido}} \right\rceil + 1 = \left\lceil \frac{500}{600} \right\rceil + 1 = 2$$

LAS PIEZAS

PIEZA	QUÉ HACE	EJEMPLO
perf_analyzer	Genera carga contra Triton (HTTP o gRPC) y reporta throughput y latencias por percentil, del cliente y del servidor.	`la rampa contra una T4`

PIEZA	QUÉ HACE	EJEMPLO
`carga cerrada (--concurrency-range)`	N clientes que esperan su respuesta antes de mandar otra. Encuentra el máximo, pero esconde la cola.	`1:32:2`
`carga abierta (--request-rate-range)`	Tasa fija aunque el servidor se frene (con --async o hilos de sobra): mide la latencia a esa tasa sin el sesgo de la carga cerrada.	`250 y 500 req/s, el gate de latencia de C01`
`--input-data`	Qué datos manda: random (default), zero o un JSON con tensores reales.	`fotos reales ya preprocesadas`
`--percentile`	Usa ese percentil, en vez del promedio, para decidir que la medición ya es estable.	`--percentile=99`
`desglose del servidor`	perf_analyzer muestra cuánto se fue en overhead, queue, compute input, compute infer y compute output.	`queue alto con compute normal = falta capacidad`
`model_analyzer`	Barre configuraciones (instancias, max batch, dynamic batching) usando perf_analyzer y elige la mejor bajo una restricción de latencia.	`model-analyzer profile --model-repository … --profile-models classifier`
`trtexec, perf_analyzer, Locust o k6`	El motor solo; el servidor; el servicio completo con el gateway.	`cada uno responde una pregunta distinta`
`nv_inference_count / nv_inference_exec_count`	Inferencias y ejecuciones del modelo: su cociente es el batch promedio.	`≈ 1–3 en C01`
`nv_inference_queue_duration_us`	µs acumulados en la cola, incluida la ventana de batching.	`la señal de escalado de KEDA`
`nv_inference_compute_infer_duration_us`	µs acumulados de cómputo, contados por request. Dividido por los requests exitosos da el cómputo medio: si ese promedio sube sin que suba la cola, cambió el modelo, el batch o la GPU.	`tablero de Triton`
`nv_inference_pending_request_count`	Requests que esperan a un backend en este momento: es un gauge, sube y baja.	`alternativa para escalar`
`nv_inference_request_failure`	Fallas por motivo, en el label reason. REJECTED = venció el timeout de la cola.	`los rechazos a los 100 ms`
`nv_gpu_utilization`	Fracción del tiempo con al menos un kernel corriendo (de 0.0 a 1.0), no la capacidad usada. Es un gauge.	`alerta GPUSaturated: más de 0.85 durante 15 min`
`nv_gpu_memory_used_bytes`	VRAM usada en cada GPU, en bytes (un gauge).	`unos pocos GB de 16 en C01`
`/v2/models/` *modelo* `/stats`	Las mismas estadísticas en JSON: cola y cómputo acumulados, y ejecuciones por tamaño de batch (batch_stats).	`curl localhost:8000/v2/models/classifier/stats`



FUENTE: M03, load testing: rampa ilustrativa

pasado el codo, la latencia se estanca en ≈ 130 ms porque la cola rechaza a los 100 ms; lo que sobra se rechaza (≈ 7 % a 1,300 y ≈ 14 % a 1,400)

Rampa contra una T4 con perf_analyzer (números ILUSTRATIVOS de M03): hasta el codo el throughput sigue a la carga y la latencia casi no cambia; después, el throughput se aplana y la cola de C01 corta a los 100 ms. La forma es la que vas a ver; tus números se miden.

EJEMPLO REAL · ALERTS.YML: LA COLA DE TRITON COMO SEÑAL DE SATURACIÓN

YAML `reference/c01/monitoring/alerts.yml (líneas 78–93)`

```yaml
      # Tiempo medio en la cola de Triton (µs acumulados / requests exitosos → ms).
      # Con max_queue_delay de 2 ms, > 20 ms significa GPU saturada: KEDA debería estar escalando.
      - alert: TritonQueueTimeHigh
        expr: |
          (
            sum(rate(nv_inference_queue_duration_us{model="classifier"}[5m]))
              /
            sum(rate(nv_inference_request_success{model="classifier"}[5m]))
          ) / 1000 > 20
        for: 10m
        labels:
          severity: warning
        annotations:
          summary: "Cola de Triton sobre 20 ms en promedio"
          description: "Espera media en cola = {{ $value | printf \"%.1f\" }} ms. Si KEDA ya está en maxReplicaCount
(4), falta capacidad de GPU."
          runbook_url: "https://runbooks.example.com/classifier/triton-queue-time-high"
```

- Espera media = µs acumulados en la cola ÷ requests exitosos (los dos como `rate`), y ÷ 1000 para pasar a ms. Es la misma cuenta que usa KEDA en `k8s/keda-triton.yaml`, con un objetivo de 10 ms.
- Con `max_queue_delay` de 2 ms, lo normal es ≈ 1–3 ms en la cola, porque la ventana cuenta como espera. Más de 20 ms durante 10 minutos (`for: 10m`) es una GPU saturada.
- La descripción dice qué mirar: si KEDA ya está en `maxReplicaCount` (4), falta capacidad de GPU y hay que subir el techo o mejorar el engine.
- La regla vecina, GPUSaturated (`nv_gpu_utilization > 0.85` por 15 min), mide tiempo con kernels corriendo, no capacidad: con batches de 1 la utilización sube aunque el throughput esté lejos del máximo. Por eso C01 escala por la cola, no por la utilización.

- Lo que no está aquí: el p99 del servicio se mide en el gateway (`classifier_request_latency_seconds`). Y antes de producción, perf_analyzer mide el p99 de una GPU a 250 y 500 req/s (el gate de latencia de C01) y su capacidad máxima con la rampa.

CÓMO RECORDARLA

trtexec mide el motor, perf_analyzer el servidor, Locust o k6 el servicio. En producción, dos cocientes: inferencias ÷ ejecuciones = batch promedio; µs en cola ÷ requests = espera media.

TRAMPA

medir con carga cerrada y quedarte con su p99. Cuando el servidor se frena, los clientes virtuales también: los peores momentos casi no se miden y el p99 sale mejor que el real (coordinated omission). El máximo se busca con `--concurrency-range`; la latencia a una tasa, con `--request-rate-range`.

En el curso: M3 · Load testing: cómo se mide de verdad · M3 · Utilización y colas: por qué no se planea al 100 % · C01 · Monitoreo, alertas y reentrenamiento · M12 · Prometheus, Grafana y alertas por SLO · Números · Throughput: qué medir y con qué · Hoja · Formulario de estimaciones

10 Pre y postprocesamiento: dónde va (gateway, ensemble, backend de Python) y el riesgo de skew

¿Dónde conviertes los bytes de la foto en el tensor que espera el modelo, y los logits (los puntajes crudos del modelo) en la respuesta, sin que el servicio vea algo distinto de lo que vio el entrenamiento?

SINTAXIS CON VARIABLES

```
from app.preprocess import preprocess
# el MISMO archivo en entrenamiento, eval y servicio
batch = preprocess( bytes , max_pixels= tope )
[np.newaxis]   # float32 [1, 3, 224, 224]
inp = InferInput(" entrada ", list(batch.shape),
"FP32")
inp.set_data_from_numpy(batch, binary_data=True)
# bytes crudos, no floats en JSON
```

Preprocesamiento (preprocessing: convertir los bytes de la foto en el tensor que espera el modelo). En C01 vive en un solo archivo, `preprocess.py`, que importan el entrenamiento, la evaluación y el gateway. Nombre, shape y dtype (el tipo de número) tienen que coincidir con `config.pbtxt`.

```
logits = result.as_numpy(" salida ")[0]      # [500]:
logits crudos, sin softmax
probs = _softmax(logits)                      # en
float64 y restando el máximo: no desborda
top = np.argsort(probs)[::-1][: k ]           # los k
índices más probables
preds = [{"label": labels[i], "score":
round(float(probs[i]), 4)} for i in top]
```

Postprocesamiento (postprocessing: convertir la salida del modelo en la respuesta del producto). C01 devuelve logits y hace en el gateway el softmax (logits → probabilidades), el top-k (las k más altas) y los nombres: así puede calibrar scores o cambiar el top-k sin tocar el engine.

VARIABLES

VARIABLE	QUÉ ES	EJEMPLO
bytes	los bytes crudos de la foto, tal como llegan en el body	≈ 150–250 KB desde el app móvil
tope	máximo de píxeles que aceptas, leído del header antes de decodificar	40_000_000 (40 MP)
entrada · *salida*	nombres de los tensores: los fija el export y los repite config.pbtxt	input · logits
k	cuántas sugerencias devuelves	5 (máximo 20 en C01)
pipeline · *preproceso* · *modelo*	nombres de modelos (carpetas) del model repository	pipeline · preprocess · classifier
n	max_batch_size del ensemble	32
clases	largo del vector de salida	500
x	valor de un píxel, de 0 a 255	200
μ_c, σ_c	media y desvío de ImageNet del canal c, en orden RGB	R: 0.485 y 0.229
λ	requests por segundo	500 en el pico de diseño
t_{CPU}	CPU por imagen: decode, resize y normalizar	≈ 6 ms con la mezcla de fotos de C01
B	bytes del tensor que viaja del gateway a Triton	602 KB en FP32
z_i, p_i	logit y probabilidad de la clase i	restar el máximo evita que exp() desborde

```
name: "pipeline"
platform: "ensemble"
max_batch_size: n
input [ { name: "IMAGE", data_type: TYPE_UINT8,
dims: [ -1 ] } ]     # los bytes del JPEG
output [ { name: "LOGITS", data_type: TYPE_FP32,
dims: [ clases ] } ]
ensemble_scheduling {
  step [
    { model_name: "preproceso"
      model_version: -1
      input_map { key: "RAW" value: "IMAGE" }
      output_map { key: "TENSOR" value:
"preprocessed" } },
    { model_name: "modelo"
      model_version: -1
      input_map { key: "input" value: "preprocessed"
}
      output_map { key: "logits" value: "LOGITS" } }
  ]
}
```

Un **ensemble** (un modelo de Triton que no ejecuta nada: encadena otros modelos) recibe el JPEG y devuelve logits en una sola llamada. En cada `input_map` / `output_map`, la key es el tensor del paso y el value, el tensor dentro del ensemble. No lleva `instance_group`: cada paso escala con el suyo y conserva su propia cola.

```
import numpy as np
import triton_python_backend_utils as pb_utils
from app.preprocess import preprocess          # el
mismo archivo que usa el entrenamiento

class TritonPythonModel:
    def execute(self, requests):
        responses = []
        for request in requests:
            raw =
pb_utils.get_input_tensor_by_name(request,
"RAW").as_numpy()   # [1, n] UINT8
            x = preprocess(raw[0].tobytes())
[np.newaxis]                        # [1, 3, 224,
224] FP32

responses.append(pb_utils.InferenceResponse(output_t
ensors=[pb_utils.Tensor("TENSOR", x)]))
        return responses
```

El **backend de Python** (Triton corre tu `model.py`; cada instancia corre en un proceso aparte y le pasa los tensores a Triton por memoria compartida, `/dev/shm`) sería el paso *preproceso* del ensemble, con `backend: "python"` en su config. Corre en la CPU del nodo con GPU, y lo que importe (Pillow, tu paquete) tiene que existir en su entorno.

FÓRMULAS

$$x' = \frac{x/255 - \mu_c}{\sigma_c} \qquad \mu = (0.485,\ 0.456,\ 0.406) \qquad \sigma = (0.229,\ 0.224,\ 0.225)$$

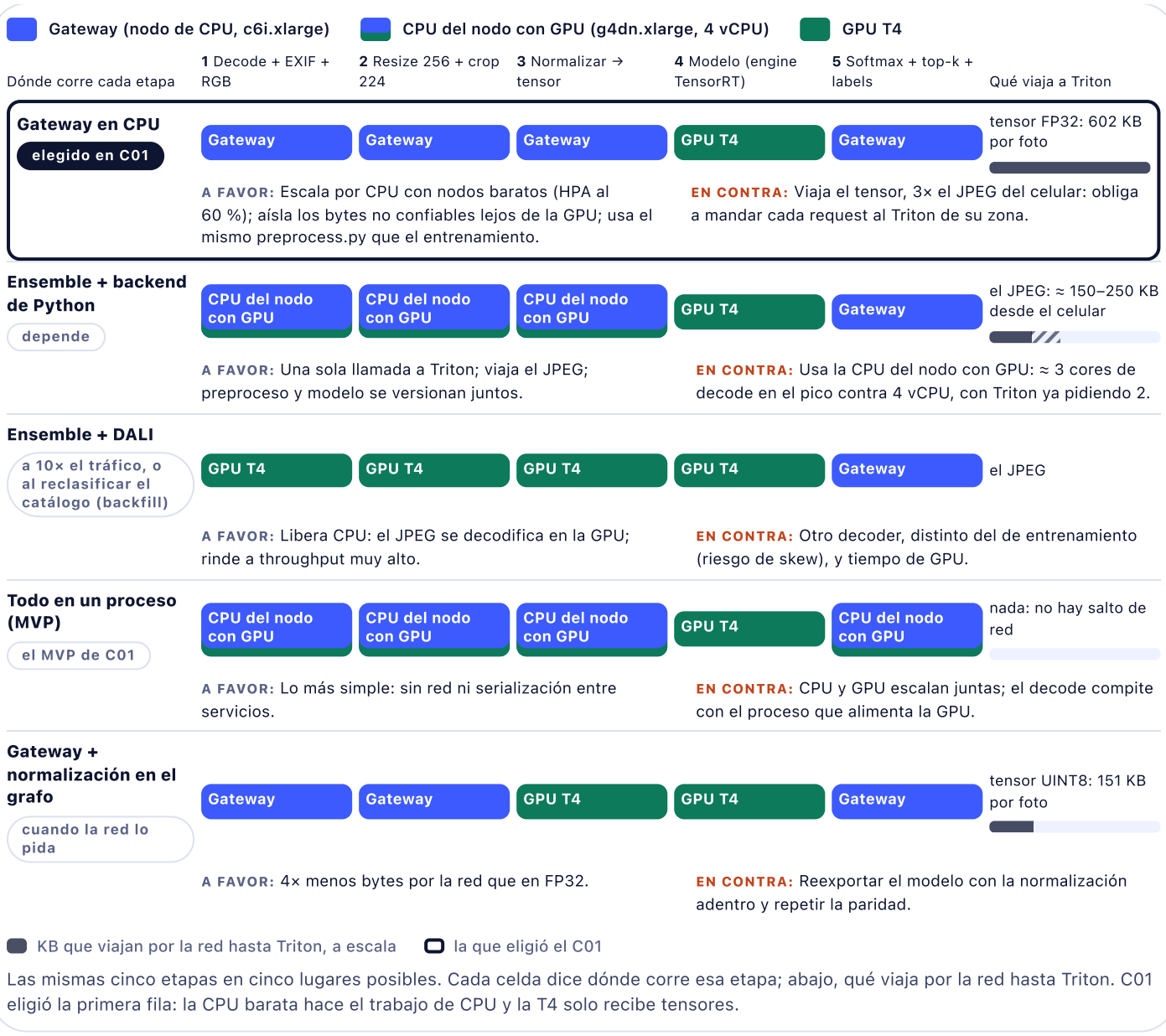
$$B = 3 \times 224 \times 224 \times 4\ \text{B} = 602{,}112\ \text{B} \approx 602\ \text{KB} \qquad \text{en UINT8: } 150{,}528\ \text{B} \approx 151\ \text{KB}$$

$$\text{cores} = \lambda \cdot t_{\text{CPU}} = 500\ \frac{\text{req}}{\text{s}} \times 6\ \text{ms} = 3 \qquad \text{red} = \lambda \cdot B \approx 2.4\ \text{Gb/s}$$

$$p_i = \frac{e^{z_i - \max_j z_j}}{\sum_k e^{z_k - \max_j z_j}}$$

LAS PIEZAS

PIEZA	QUÉ HACE	EJEMPLO
Gateway en CPU (C01)	Un servicio aparte (el gateway: la API en FastAPI que va delante de Triton), en nodos baratos, valida y decodifica la foto y arma el tensor; Triton solo ve tensores. Escala por CPU con su HPA (el autoscaler de pods de Kubernetes).	`FastAPI + preprocess.py, HPA al 60 % de CPU`
Ensemble + backend de Python	Triton recibe el JPEG y lo convierte con tu model.py: una sola llamada, pero el decode usa la CPU del nodo con GPU.	`g4dn.xlarge: 4 vCPU, y Triton ya pide 2`
Ensemble + DALI	DALI (la librería de NVIDIA para cargar imágenes en la GPU) decodifica y hace el resize en la GPU (backend: "dali", archivo model.dali). Libera CPU, pero es otro decoder.	`decoder con device="mixed"`
Todo en un proceso	El MVP (la primera versión, la más simple): decode y modelo en el mismo proceso de FastAPI + PyTorch. CPU y GPU escalan juntas.	`el MVP de C01, una VM g4dn`
Normalización dentro del grafo	El modelo recibe UINT8 de 0 a 255 y normaliza adentro: 4× menos bytes por la red. Obliga a reexportar y repetir la paridad.	`602 KB → 151 KB por foto`
Training-serving skew	El servicio prepara la entrada distinto que el entrenamiento: el modelo ve otra distribución y pierde accuracy, sin ningún error.	`accuracy online menor que la offline`
RGB vs BGR	OpenCV lee en BGR; Pillow y torchvision, en RGB. Los canales al revés no dan error.	`cv2.imread entrega BGR`
Resize + center crop	Resize(256) lleva el lado corto a 256 y CenterCrop(224) recorta el centro; un resize directo a 224 × 224 deforma la foto.	`Resize((224, 224)) ≠ Resize(256) + CenterCrop(224)`
Antialias	Al achicar, Pillow suaviza siempre; F.interpolate no, salvo antialias=True. En torchvision 0.29, Resize trae antialias=True por defecto.	`F.interpolate(..., mode="bilinear")`
EXIF	Las fotos de celular guardan la rotación aparte, en sus metadatos EXIF: sin exif_transpose, el modelo las ve de lado.	`ImageOps.exif_transpose`
HWC vs CHW	Una imagen de Pillow u OpenCV pasada a NumPy queda en alto × ancho × canales (HWC); el modelo espera los canales primero (CHW). Este sí falla con un error.	`x.transpose(2, 0, 1)`
Draft mode de JPEG	El decoder reduce la resolución mientras decodifica: mucho menos CPU y píxeles casi iguales.	`solo si el lado corto > 1024 px, sin bajar de 512`
Softmax y top-k	El postproceso: logits → probabilidades → las k más altas, con su nombre.	`labels.json: 500 nombres, índice = id de clase`
labels.json desalineado	Si el orden de los nombres no es el del entrenamiento, todo responde 200 con la categoría equivocada.	`una falla silenciosa de C01`



Las mismas cinco etapas en cinco lugares posibles. Cada celda dice dónde corre esa etapa; abajo, qué viaja por la red hasta Triton. C01 eligió la primera fila: la CPU barata hace el trabajo de CPU y la T4 solo recibe tensores.

EJEMPLO REAL · C01: EL PREPROCESAMIENTO COMPARTIDO Y LA LLAMADA QUE LO USA

PYTHON `reference/c01/gateway/app/preprocess.py y main.py (extractos; # … marca líneas omitidas)`

```python
# gateway/app/preprocess.py (extracto)
RESIZE_SHORT = 256  # lado corto tras el resize: el mismo número que Resize(256) en training
CROP = 224  # tamaño del center crop: el input del ResNet-50 (y del engine TensorRT)
# mean/std de ImageNet con forma (3,1,1) para hacer broadcasting sobre CHW sin copiar
MEAN = np.array([0.485, 0.456, 0.406], dtype=np.float32).reshape(3, 1, 1)
STD = np.array([0.229, 0.224, 0.225], dtype=np.float32).reshape(3, 1, 1)
# …
def to_chw_normalized(img: Image.Image) -> np.ndarray:
    """Imagen RGB 224×224 → float32 [3,224,224] normalizado (ToTensor + Normalize)."""
    x = np.asarray(img, dtype=np.float32) / 255.0  # HWC en [0, 1], igual que ToTensor
    x = x.transpose(2, 0, 1)  # HWC → CHW: el layout que espera el modelo de PyTorch/TensorRT
    x = (x - MEAN) / STD  # normalización por canal con las estadísticas de ImageNet
    # contiguous: el cliente de Triton serializa el buffer tal cual; un view transpuesto se copiaría
    return np.ascontiguousarray(x, dtype=np.float32)

# gateway/app/main.py (extracto)
def _decode_and_hash(data: bytes) -> tuple[np.ndarray, str]:
    """CPU pura (bloqueante): corre en el threadpool. Pillow y numpy sueltan el GIL en decode y resize."""
    digest = hashlib.sha256(data).hexdigest()  # huella de la imagen: deduplicar y cruzar con labels
    tensor = preprocess(data, max_pixels=MAX_PIXELS)  # float32 [3,224,224], idéntico a training/eval
    return tensor[np.newaxis], digest  # batch de 1: Triton junta requests con dynamic batching
```

- Un solo `preprocess.py` para entrenamiento, evaluación y gateway: el skew no se evita con cuidado, se evita con un solo archivo. Contra torchvision 0.29 con Pillow 12.3 da una diferencia de 0.0 cuando no se activa el draft mode (lo miden el docstring del archivo y el lab de abajo).
- La salida es exactamente el contrato de `config.pbtxt`: `input`, FP32, [3, 224, 224]. El gateway agrega el eje de batch (`[np.newaxis]`) y manda una foto por request: el batching lo hace Triton.
- `np.ascontiguousarray`: el `transpose` no mueve datos; deja una vista (view), los mismos números leídos en otro orden. tritonclient la copiaría igual al serializar (`tobytes()`); hacer la copia aquí, contigua y en FP32 explícito, deja el tensor listo y con el tipo del contrato.
- El decode corre en un threadpool (hilos aparte), y un semáforo (un contador de turnos) deja pasar 2 decodes a la vez por proceso (`PREPROCESS_CONCURRENCY`). Es CPU pura: dentro del event loop (el hilo que atiende todos los requests del proceso) bloquearía a los demás.
- ¿Por qué en el gateway y no en Triton? A 500 req/s son ≈ 3 cores de decode; la g4dn.xlarge tiene 4 vCPU y el pod de Triton ya pide 2. En un ensemble con backend de Python, para tener más CPU tendrías que comprar más GPU.
- El precio: viaja el tensor (602 KB) y no el JPEG del celular (≈ 200 KB), 3× más red. Por eso C01 manda cada request al Triton de su misma zona (cruzar de zona se cobra por GB) y, al crecer, pasaría a entrada UINT8 con la normalización dentro del grafo.

CÓMO RECORDARLA

un solo preprocess.py, y lo de CPU en CPU barata. El preproceso se escribe una vez y lo importan entrenamiento, evaluación y servicio; vive donde sobra CPU (el gateway), y Triton solo ve tensores.

TRAMPA

reescribir el preproceso en el servicio «igual que en el notebook». BGR en vez de RGB, un resize directo a 224 × 224, `F.interpolate` sin antialias o la normalización aplicada dos veces no lanzan ningún error: todo responde 200 y la accuracy online cae. La paridad del modelo (PyTorch contra el engine, con el mismo tensor) tampoco lo ve: para eso está la paridad **end-to-end**, con fotos de referencia por el gateway real.

En el curso: C01 · Serving · C01 · Estimaciones back-of-the-envelope · M7 · Protocolos: REST, gRPC, KServe v2 y OpenAI-compatible · M7 · NVIDIA Triton Inference Server · M9 · Training-serving skew: causas, detección y prevención · C01 · Modos de falla y degradación

11 Optimizaciones en orden de impacto, y la alternativa sin GPU (ONNX Runtime en CPU)

Tu engine ya sirve: ¿qué optimizas primero, cuánto compra cada paso y cuándo conviene no usar una GPU?

SINTAXIS CON VARIABLES

```
histogram_quantile(0.99, sum by (le)
(rate(classifier_preprocess_seconds_bucket[5m])))
histogram_quantile(0.99, sum by (le)
(rate(classifier_inference_seconds_bucket[5m])))
sum(rate(nv_inference_queue_duration_us{model="modelo"}[5m])) /
sum(rate(nv_inference_request_success{model="modelo"}[5m]))
sum(rate(nv_inference_compute_infer_duration_us{model="modelo"}[5m])) /
sum(rate(nv_inference_request_success{model="modelo"}[5m]))
```

Mide antes de optimizar. Las dos primeras son el p99 (la latencia que solo supera el 1 % de los requests) del preproceso y de la llamada a Triton, medidos en el gateway; las dos últimas, la espera en cola y el cómputo promedio por request dentro de Triton, en µs. Optimizar el modelo solo mueve el último número.

```
# TensorRT 10 (NGC 24.08, la de C01)
trtexec --onnx=onnx --saveEngine=model.plan --fp16 \
  --minShapes=input:1x3x224x224
--optShapes=input:opt x3x224x224
--maxShapes=input:32x3x224x224
# TensorRT 11 (NGC 26.06 en adelante): la precisión va en el ONNX
python -m modelopt.onnx.autocast --onnx_path onnx
--output_path onnx_fp16 --keep_io_types
```

El paso de más impacto en C01: TensorRT (un compilador que convierte el ONNX en un engine para una GPU concreta) en FP16 da varias veces el throughput de PyTorch eager (el modo normal, operación por operación) en FP32 en la T4, casi sin tocar la calidad. Cada release de NGC (las imágenes de NVIDIA, YY.MM) trae su TensorRT. En TensorRT 11 ya no existe `--fp16`: AutoCast, de NVIDIA ModelOpt, convierte el ONNX a precisión mixta (FP16 donde se puede) y `--keep_io_types` deja la entrada y la salida en FP32, como pide `config.pbtxt`.

```
quant_pre_process("onnx", "onnx_pre")
quantize_static("onnx_pre", "onnx_int8", lector,
                quant_format=QuantFormat.QDQ,
activation_type=QuantType.QInt8,
                weight_type=QuantType.QInt8,
per_channel=True,

calibrate_method=CalibrationMethod.Percentile)
```

INT8 (enteros de 8 bits con una escala por tensor o por canal) con ONNX Runtime: ≈ 4× menos bytes que FP32; el ONNX de C01 baja de 98 MB a ≈ 25 MB. El lector de calibración (calibration reader) entrega fotos reales pasadas por el mismo `preprocess.py`. TensorRT solo acepta INT8 simétrico (el 0 real se guarda como 0): con ONNX Runtime agrega `extra_options={"ActivationSymmetric": True}`, o cuantiza con ModelOpt.

VARIABLES

VARIABLE	QUÉ ES	EJEMPLO
modelo	nombre del modelo en Triton (el label model de las métricas)	`classifier`
onnx · *onnx_fp16* · *onnx_pre* · *onnx_int8*	el ONNX FP32 y sus variantes	`model/classifier.onnx` · `model/classifier_int8.onnx`
opt	batch para el que TensorRT afina el engine	`16 en C01; compáralo con 4`
lector	tu CalibrationDataReader: entrega una foto preprocesada por llamada	`≈ 500–1,000 fotos reales`
vcpus	vCPU del pod (su límite), no los cores del nodo	`4`
p	fracción del tiempo total que es cómputo del modelo	`≈ 0.05–0.10 en el p99 de C01`
s	cuántas veces más rápido queda el modelo	`2`
S_{total}	cuántas veces más rápido queda el request completo	`≈ 1.03–1.05: el p99 baja ≈ 3–5 %`
λ	requests por segundo a atender	`500 en el pico de diseño`
t_{img}	CPU por imagen, medida en la instancia real	`INT8 ≈ 8–25 ms; FP32 ≈ 40–80 ms (estimaciones de C01)`
U_{obj}	utilización objetivo del HPA (el autoscaler de pods)	`0.6`
X_{max}	máximo medido por GPU con perf_analyzer (la prueba de carga contra Triton)	`≈ 1,200 img/s por T4 (supuesto de C01, a medir)`

```
opciones = ort.SessionOptions()
opciones.intra_op_num_threads = vcpus      # los
vCPU del pod, no los cores del nodo
opciones.inter_op_num_threads = 1
sesion = ort.InferenceSession("onnx_int8",
opciones, providers=["CPUExecutionProvider"])
logits = sesion.run(["logits"], {"input": batch})[0]
```

La alternativa sin GPU: el mismo ONNX en ONNX Runtime (ORT), en CPU. Fija los hilos con el límite de CPU del pod: si no, ORT usa tantos hilos como cores físicos ve en el nodo (no lee el límite del contenedor) y pelea por tu cuota.

FÓRMULAS

$$S_{\text{total}} = \frac{1}{(1-p) + p/s} \qquad p = \frac{10}{150} \approx 0.067,\ s = 2 \Rightarrow S \approx 1.03$$

$$\text{vCPU} = \frac{\lambda \cdot t_{\text{img}}}{U_{\text{obj}}} = \frac{500 \times 0.015\ \text{s}}{0.6} = 12.5$$

$$N_{\text{GPU}} = \left\lceil \frac{\lambda}{0.5 X_{\max}} \right\rceil + 1 = \left\lceil \frac{500}{600} \right\rceil + 1 = 2$$

LAS PIEZAS

PIEZA	QUÉ HACE	EJEMPLO
TensorRT FP16	Compila el ONNX para la T4 en FP16, que usa sus tensor cores (las unidades que multiplican matrices en baja precisión): varias veces el throughput de PyTorch eager FP32. Riesgo: algún top-1 cambia en casos límite.	sí en C01, con paridad (top-1 ≥ 99.5 %)
INT8	≈ 1.5–2× más sobre FP16 en GPU. Necesita calibración con datos representativos y pierde más en el long tail (las clases con pocas fotos).	no en C01: la GPU trabaja al ≈ 5–20 %; revisar a 10×
Engine construido en CI sobre T4	Construirlo en el pipeline de CI (el que corre en cada cambio), en una T4, no acelera nada: evita que Triton no pueda cargar el modelo en producción.	misma GPU (sm_75: la arquitectura de la T4) y misma release de NGC (24.08)
Downscale en el cliente (≤ 1024 px)	El celular achica la foto antes de subirla: menos bytes y menos CPU de decode; sin riesgo si el lado corto sigue ≥ 256.	el app móvil manda ≈ 150–250 KB
JPEG draft mode conservador	El decoder reduce la foto mientras decodifica: un JPEG de 12 MP baja de ≈ 40 a ≈ 18 ms de CPU (medido en una laptop).	solo si el lado corto > 1024 px, sin bajar de 512
Threadpool para el trabajo de CPU	El event loop (el hilo que atiende todos los requests) sigue con la red mientras otro hilo decodifica. Es corrección, no optimización.	run_in_threadpool + 2 decodes por proceso
Entrada FP16 o UINT8 con normalización en el grafo	2× o 4× menos bytes entre el gateway y Triton. Obliga a reexportar y a repetir la paridad.	después, si la red o el costo entre zonas lo pide
ONNX Runtime en CPU (FP32)	El mismo ONNX sin GPU: un servicio menos, sin engine atado a la T4 ni la espera de levantar un nodo GPU (cold start).	≈ 40–80 ms por imagen por vCPU: ≈ 20–40 vCPU ocupados en el pico
INT8 en CPU	Pesos y activaciones en INT8 con ONNX Runtime u OpenVINO: el candidato serio de C01 para quitar la GPU.	≈ 8–25 ms por imagen por vCPU, a medir en la semana 1
VNNI · AMX	Instrucciones de CPU que aceleran INT8: VNNI desde Cascade Lake (la c6i, Ice Lake, lo trae) y AMX desde Sapphire Rapids (c7i). Sin ellas, INT8 puede ser más lento que FP32.	en Arm, las de producto punto (Graviton2 en adelante)
intra_op_num_threads	Hilos de ORT dentro de cada operador. Con 0, ORT usa uno por core físico que ve: los del nodo, no los de tu límite.	fíjalo al límite de CPU del pod

PIEZA	QUÉ HACE	EJEMPLO
Procesos × hilos	Más procesos dan más throughput total; más hilos por proceso, menos latencia por request.	`FP32 en la Mac M4 del curso: 4 × 1 hilo ≈ 86 img/s; 1 × 4 hilos ≈ 61 img/s`

MÁS BARATO Y SEGURO

MÁS CARO Y ARRIESGADO

1. **Medir dónde se va el tiempo**
   `histogram_quantile(0.99, sum by (le) (rate(classifier_preprocess_seconds_bucket[5m])))`
   Si el cómputo es el 5–10 % del p99, un modelo 2× más rápido baja el p99 ≈ 3–5 %.
2. **Menos trabajo por request**
   `img.draft("RGB", (DRAFT_MIN_SHORT, DRAFT_MIN_SHORT))`
   Caché, cascadas (un modelo barato primero y el caro solo si hace falta), menos píxeles. En C01, el draft mode ahorra en una foto web tanto como todo el cómputo de la GPU.
3. **Batching e instancias**
   `dynamic_batching { max_queue_delay_microseconds: 2000 }`
   Sin tocar el modelo ni la calidad. Rinde cuando hay tráfico para llenar los batches; en C01 llegan ≈ 1–3 por batch.
4. **Runtime compilado, misma precisión**
   `trtexec --onnx=model/classifier.onnx --saveEngine=model.plan`
   Fusiona capas y elige kernels (las rutinas que corren en la GPU) para esa GPU. El riesgo es un bug de export: lo atrapa la paridad.
5. **Menos precisión**
   `trtexec … --fp16` (TensorRT 10) ·
   `python -m modelopt.onnx.autocast --onnx_path model/classifier.onnx --keep_io_types` (TensorRT 11)
   FP16 casi nunca cambia una CNN. INT8 necesita calibración y un gate por slices.
6. **Un modelo más chico**
   `torchvision.models.mobilenet_v3_large(num_classes=500)`
   Destilación (un modelo chico que aprende a imitar al grande) o pruning (quitar pesos): la ganancia más grande y el costo más alto (entrenar de nuevo y mantenerlo).

Aparte: sin GPU (un servicio menos)
`ort.InferenceSession("classifier_int8.onnx", opciones, providers=["CPUExecutionProvider"])`
Si INT8 en CPU da ≈ 15 ms por imagen por vCPU sin perder más de 0.5 puntos de accuracy, son ≈ 13 vCPU en el pico y desaparece el servicio con GPU.

La escalera, de lo más barato y seguro a lo más caro (M08). Cada peldaño se mide antes de subir al siguiente. C01 usa los cinco primeros (con FP16, sin INT8), deja el sexto al equipo de modelado y mide el último en la semana 1.

EJEMPLO REAL · C01: ¿QUÉ COMPRA CADA OPTIMIZACIÓN? (LAS CUENTAS DEL CURSO)

TEXTO

Cuentas del curso: M08 (por qué optimizar) y C01 (estimaciones). Son estimaciones con supuestos, no benchmarks: el máximo por T4 y los ms por vCPU se miden.

```
Cómputo GPU en el p99              8–15 ms de ≈ 150 ms (5–10 %)
Modelo 2× más rápido               p99 ≈ 142–146 ms: ≈ 3–5 % menos
GPUs hoy (FP16, 600 img/s plan)    ⌈500 / 600⌉ + 1 = 2 T4
GPUs con INT8 (900–1,200 plan)     ⌈500 / 900⌉ + 1 = 2 T4 → ahorro US$0
A 10× (5,000 req/s), FP16          ⌈5,000 / 600⌉ + 1 = 10 T4
A 10×, INT8                        6–7 T4 → ≈ US$1,150–1,540/mes menos
Sin GPU: INT8 en CPU (HPA al 60 %) ≈ 7–21 vCPU en el pico; ≈ US$125–250/mes contra ≈ US$768 de 2 T4
```

- Ley de Amdahl (lo que no aceleras pone el techo) con los números de C01: si el cómputo es el 5–10 % del p99, ningún truco del modelo mueve el p99 más que eso. Primero el presupuesto por tramo; después, la técnica que mueve el número que importa.
- Las 2 T4 están por disponibilidad (N+1: una de sobra, una por zona), no por throughput: hoy INT8 no ahorra ni una GPU y agrega riesgo en el long tail.
- TensorRT FP16 igual se usa: es casi gratis en calidad (la paridad lo confirma) y da margen para picos y failover sin escalar.
- A 10× el tráfico, INT8 sí paga 3–4 T4, si pasa el gate por slices (grupos como móvil, web o poca luz); y queda por debajo de pasar a 5 L4 en FP16 (≈ US$2,938/mes).
- Sin GPU decide el benchmark de la semana 1. En x86 con VNNI o AMX se espera INT8 ≈ 3–5× sobre FP32, pero no está medido para este stack; en la Mac M4 del curso dio ≈ 1–2×. Si en la instancia real da ≈ 1.5–2×, el ahorro contra las dos T4 deja de ser claro.

CÓMO RECORDARLA

mide, después lo barato, al final lo arriesgado. Presupuesto por tramo → menos trabajo → batching → runtime compilado → menos precisión → modelo más chico. Optimizas el modelo cuando cambia el número de máquinas, elimina un servicio o hace que algo quepa.

TRAMPA

«le pongo INT8 para bajar la latencia» en un sistema donde el modelo es el 10 % del p99 y las GPUs están al 5 %. No baja el p99, no ahorra una GPU y agrega riesgo en el long tail. Y no midas la ganancia con `trtexec` a batch 32 si online llegan batches de 1–3: el número que decide sale de `perf_analyzer` contra Triton, con la carga real.

En el curso: C01 · Serving · C01 · Estimaciones back-of-the-envelope · M8 · Por qué y cuándo optimizar · M8 · El mapa de técnicas · M8 · Cuantización: FP16, INT8, FP8 e INT4 · M8 · Inferencia en CPU: ONNX Runtime y OpenVINO · M8 · Cómo validar una optimización · M4 · CPU vs GPU vs TPU vs aceleradores

12 Diagnóstico: el método y la tabla de síntomas

Triton no carga, responde con error, va lento o predice distinto: ¿qué miras primero, qué después, y cómo partes el problema en dos?

SINTAXIS CON VARIABLES

```
curl -fsS host:8000/v2/health/live && echo vivo
curl -fsS host:8000/v2/health/ready && echo listo
curl -fsS host:8000/v2/models/modelo/ready &&
echo modelo listo
```

¿Vive? ¿Está listo? Triton responde 200 si sí y 400 si no (503 mientras se apaga); con `-f`, curl falla con el error 22 en vez de imprimir el echo. Si ni siquiera conecta (error 7), Triton todavía no escucha: con `--model-control-mode=none` (C01) abre los puertos recién después de cargar los modelos. Con `--strict-readiness=true` (C01), el ready del servidor exige que todos los modelos estén listos; el del modelo dice si esa versión cargó y terminó su warmup (las inferencias de prueba).

```
kubectl logs pod -n ns --previous | grep -E
"failed to load|UNAVAILABLE|error: creating server"
curl -s -X POST host:8000/v2/repository/index
curl -s -d '{"log_verbose_level":1}' -X POST
host:8000/v2/logging
```

¿Qué no cargó y por qué? Con `--exit-on-error=true` (C01), Triton sale si un modelo no carga (queda UNAVAILABLE): el motivo queda en el log del intento anterior. Si Triton sigue arriba, el índice del repository (`POST /v2/repository/index`) da `state` y `reason` por modelo, y la extensión de logging sube el detalle en caliente (o arranca con `--log-verbose=1`).

```
curl -s host:8000/v2/models/modelo          #
nombres, datatype y shape de entradas y salidas
curl -s host:8000/v2/models/modelo/config   #
el config.pbtxt que cargó, en JSON
```

¿El contrato coincide? Compara lo que manda el cliente (nombre, shape con el eje de batch, dtype: el tipo de número) con la metadata (la descripción de entradas y salidas que publica Triton). Casi todos los 400 de inferencia salen de aquí.

```
curl -s host:8000/v2/models/modelo/stats    #
cola, cómputo y ejecuciones por tamaño de batch
curl -s host:8002/metrics | grep -E
"^nv_inference_(queue|compute_infer)_duration_us"
nvidia-smi
--query-gpu=utilization.gpu,memory.used,memory.total
--format=csv -l 1
```

¿Dónde se va el tiempo? Cola contra cómputo en Triton, preproceso contra inferencia en el gateway, y la GPU. Si el problema son las predicciones y no la velocidad, el paso es la paridad: ONNX Runtime contra Triton con `parity_check.py`.

VARIABLES

VARIABLE	QUÉ ES	EJEMPLO
host	dónde escucha Triton; en el cluster, a través de un port-forward (un túnel de kubectl a tu máquina)	`localhost`
modelo	nombre del modelo (la carpeta del repository)	`classifier`
pod · *ns*	pod de Triton y su namespace de Kubernetes	`triton-5f6b…-7tq2m` · `classifier`
t_{pre}, t_{post}	preproceso y postproceso en el gateway	`preprocess_ms` en el log JSON de C01
t_{red}	ida y vuelta de la red y la serialización	≈ 2–4 ms en p50 (la mediana) para 602 KB binarios
t_{cola}, $t_{cómputo}$	espera en la cola de Triton y ejecución del modelo	≤ 2 ms de ventana · 3–6 ms en p50
ΔQ	aumento de nv_inference_queue_duration_us (µs acumulados en cola)	`rate(…[5m])`
ΔN_{ok}	aumento de nv_inference_request_success	`rate(…[5m])`
ΔN_{inf}, ΔN_{exec}	aumento de nv_inference_count (inferencias) y nv_inference_exec_count (ejecuciones)	su cociente es el batch promedio
U_{GPU}	lo que nvidia-smi llama utilización: tiempo con kernels (programas de la GPU) corriendo, no capacidad usada	con batches de 1 marca mucho más que la carga: C01 espera 10–35 % con ≈ 5 % de carga

FÓRMULAS

$$t_{total} \approx t_{pre} + t_{red} + t_{cola} + t_{cómputo} + t_{post}$$

$$\bar{t}_{cola} = \frac{\Delta Q}{\Delta N_{ok}} \qquad \bar{b} = \frac{\Delta N_{inf}}{\Delta N_{exec}}$$

$$U_{GPU} = \frac{t_{\text{con al menos un kernel}}}{t_{muestra}}$$

LAS PIEZAS

PIEZA	QUÉ HACE	EJEMPLO
CrashLoopBackOff al arrancar Triton	Un modelo no cargó (engine de otra versión o de otra GPU, config que no coincide, ninguna GPU visible) y --exit-on-error=true cerró el servidor con exit 1. Kubernetes lo reinicia una y otra vez, cada vez esperando más.	`error: creating server: Internal – failed to load all models`
Running pero READY 0/1	Todavía carga el engine (el modelo compilado) o hace el warmup, y Triton aún no abre el puerto; o, con --exit-on-error=false, un modelo quedó UNAVAILABLE y el ready da 400.	`Startup probe failed: … connection refused`
400 en /infer	El request no coincide con el contrato: nombre, shape, dtype o batch.	`unexpected shape for input 'input'`
503 de Triton	La cola rechazó por tiempo o por tamaño, el modelo no está listo, o el servidor se está apagando.	`Request timeout expired`
504 del gateway	Triton tardó más que el timeout del cliente (150 ms en C01): cola llena o GPU saturada.	`timeout de inferencia`
p99 alto con cómputo estable	El p99 (la latencia del 1 % más lento) sube, pero el tiempo está fuera del modelo: preproceso, red u overhead de Python.	`classifier_preprocess_seconds sube`
La cola de Triton crece	La GPU está cerca de saturarse: faltan réplicas o un modelo más barato.	`alerta TritonQueueTimeHigh`
GPU al 100 % con poco throughput	nvidia-smi cuenta tiempo con kernels, no capacidad: batches de 1, o la T4 bajando su frecuencia (clocks).	`batch promedio ≈ 1`
Predicciones distintas a PyTorch	Bug de export, del engine o del preproceso: falla sin errores.	`[parity] NO PASA`
Primer request lento	Inicialización pendiente: sin warmup, la paga el primer request real.	`model_warmup ausente`
CUDA out of memory	VRAM llena: instancias, batch máximo, versiones cargadas o procesos de más.	`nvidia-smi --query-compute-apps`
«Ayer funcionaba»	Cambió algo debajo: la release de NGC (la imagen de NVIDIA, con otro TensorRT), el tipo de GPU o el driver.	`The engine plan file is not compatible…`
Batch promedio ≈ 1	No hay dynamic_batching, o con la carga actual no llega nadie más dentro de la ventana.	`rate(nv_inference_count) / rate(nv_inference_exec_count)`

1 ¿Vive? → 2 ¿Está listo? → 3 ¿Qué no cargó y por qué? → 4 ¿El contrato coincide? →
5 ¿Una inferencia sola funciona? → 6 ¿Predice lo mismo? → 7 ¿Dónde se va el tiempo? → 8 ¿Y la GPU?

1 ¿Vive?

`curl -fsS localhost:8000/v2/health/live && echo vivo`

Si curl no conecta: el proceso no corre, o todavía carga los modelos (con --model-control-mode=none abre el puerto al final). Mira el pod y el log del intento anterior.

2 ¿Está listo?

`curl -fsS localhost:8000/v2/models/classifier/ready && echo listo`

400: Triton escucha, pero el modelo no está listo (no cargó con --exit-on-error=false, o se está recargando). 503: Triton se está apagando.

3 ¿Qué no cargó y por qué?

`kubectl logs POD -n classifier --previous | grep -E "failed to load|UNAVAILABLE"`

La tabla de modelos y la línea failed to load traen el motivo; la línea de TensorRT, el porqué.

4 ¿El contrato coincide?

`curl -s localhost:8000/v2/models/classifier`

Nombre, datatype y shape (con el -1 del batch) contra lo que manda el cliente.

5 ¿Una inferencia sola funciona?

`perf_analyzer -m classifier -u localhost:8000 --concurrency-range 1 --input-data zero`

Si el request falla, ves el error que devolvió Triton; si pasa, ya tienes una latencia base.

6 ¿Predice lo mismo?

```
python model/parity_check.py --backend triton --triton-url localhost:8000 --checkpoint resnet50_500.pt --images /ruta/muestra_eval
```

Si ONNX Runtime pasa y Triton no, es el engine; si ONNX Runtime también falla, es el export.

7 ¿Dónde se va el tiempo?

`curl -s localhost:8000/v2/models/classifier/stats`

Cola contra cómputo y ejecuciones por tamaño de batch; en el gateway, preproceso contra inferencia.

8 ¿Y la GPU?

`nvidia-smi -q -d PERFORMANCE,CLOCK`

Clocks y motivos de baja (potencia, temperatura): la T4 es una tarjeta pasiva de 70 W y baja clocks bajo carga sostenida.

De afuera hacia adentro y de lo barato a lo caro: cada paso descarta una capa. Si un paso falla, ahí está el problema; si pasa, sigues con el siguiente.

EJEMPLO REAL · C01: ACTUALIZAN LA IMAGEN DE TRITON Y EL POD NUEVO NO LEVANTA

Forma típica del log de Triton (prefijos de cada línea omitidos). Textos del core y del backend de TensorRT de Triton (24.08 y la rama actual), y del manual de errores de TensorRT; --getPlanVersionOnly existe en trtexec desde 8.6 (también en 10.x y 11.x)

```
$ kubectl logs -n classifier triton-7c4d…-p2x8k --previous | grep -E "TensorRT|failed to load|UNAVAILABLE|error:"
… The engine plan file is not compatible with this version of TensorRT, expecting library version … got …
… failed to load 'classifier' version 1: Internal: unable to create TensorRT engine: …
| classifier | 1       | UNAVAILABLE: Internal: unable to create TensorRT engine: … |
error: creating server: Internal - failed to load all models

# En el contenedor de TensorRT con el que se construyó el engine:
$ trtexec --getPlanVersionOnly --loadEngine=model.plan
```

- Léelo de abajo hacia arriba: `error: creating server` solo dice que algo falló; la tabla dice qué modelo; `failed to load`, en qué paso (crear el engine); la línea de TensorRT, por qué.
- La causa: la imagen nueva trae otra release de NGC (otro TensorRT) y el `model.plan` se construyó con la anterior. En C01, `build_engine.sh` y `triton/Dockerfile` fijan el mismo YY.MM (24.08, TensorRT 10.3).
- `trtexec --getPlanVersionOnly` imprime con qué versión de TensorRT se creó el plan, sin cargarlo (deserializarlo). Según la versión de TensorRT, el mismo problema también sale como `Serialization assertion … Version tag does not match`; si el plan es de otra GPU, dice `generated on an incompatible device`.

- No es un problema del pod: con `maxUnavailable: 0`, los Triton viejos siguen sirviendo. Arreglo: reconstruir el engine con la misma release que la imagen (o volver al tag anterior) y repetir paridad y perf_analyzer (la prueba de carga contra Triton).
- Si la imagen saltó de TensorRT 10 a 11 (NGC 26.06 en adelante; 26.09 trae TensorRT 11.3), también cambia el build: `--fp16` ya no existe y la precisión va en el ONNX, con AutoCast (de NVIDIA ModelOpt).

CÓMO RECORDARLA

vive → listo → qué cargó → contrato → una inferencia → paridad → tiempo → GPU. Cada paso es un comando barato que parte el problema en dos; no saltes al último porque es el más interesante.

TRAMPA

reiniciar el pod o subir réplicas para «arreglar» un modelo que no carga. El pod nuevo trae el mismo engine y falla igual, y el reinicio se lleva el log del intento que murió: primero `kubectl logs --previous`. La otra: leer `nvidia-smi` al 100 % como «GPU saturada»; es tiempo con kernels corriendo, no capacidad usada.

En el curso: C01 · Modos de falla y degradación · Docker y kubectl · Depurar pods: síntoma → comandos → causa típica · M7 · NVIDIA Triton Inference Server · M8 · Cómo validar una optimización · M3 · Presupuesto de latencia end-to-end · M12 · Dashboards, runbooks y on-call

13 Errores que debes reconocer rápido (export, TensorRT, Triton, CUDA)

Lees el mensaje: ¿quién lo dijo (PyTorch, el exportador, ONNX Runtime, TensorRT, Triton, el cliente, CUDA), qué significa y qué corres primero?

SINTAXIS CON VARIABLES

failed to load ' *modelo* **' version** *v* **:** *estado* **:** *motivo*
error: creating server: Internal - failed to load all models

Lo dijo el **core de Triton** (el núcleo del servidor) al cargar: el modelo queda UNAVAILABLE (no disponible) con ese motivo y, con `--exit-on-error=true`, el proceso sale con exit 1. El estado (`Internal`, `Invalid argument`, `Not found` …) y el motivo dicen qué paso falló.

[*http* **] [request id:** *id* **]** *mensaje*
[StatusCode. *grpc* **]** *mensaje*

Un request que Triton rechazó, visto desde **tritonclient**: por HTTP, el código entre corchetes (400 contrato, 404 o 503 estado del modelo, 503 cola); por gRPC, el código de gRPC. Muchos mensajes traen el `request id` que mandaste (`<id_unknown>` si no mandaste uno); otros, como `Request timeout expired`, no.

unable to create TensorRT engine: *motivo_trt*

Lo dijo el **backend de TensorRT** de Triton al cargar (deserializar) el plan; el motivo es el último error de TensorRT: casi siempre versión, tipo de GPU o memoria.

excepcion **:** *mensaje*
[ONNXRuntimeError] : *n* **:** *codigo* **:** *mensaje*

Lo dijo **Python**: el nombre de la excepción delata la librería (`torch`, `onnxruntime`, `PIL`, `tritonclient`) y la última línea del traceback dice el porqué. ONNX Runtime antepone su código: `2 : INVALID_ARGUMENT` es tu entrada.

VARIABLES

VARIABLE	QUÉ ES	EJEMPLO
modelo · *v*	modelo y versión que Triton intentó cargar	`classifier · 1`
estado	el código de estado de Triton, en palabras	`Internal, Invalid argument, Not found, Unavailable`
motivo	el porqué que agregó el backend o el core	`unable to create TensorRT engine: …`
http	el código HTTP que respondió Triton	`400, 404, 503`
id	el request_id que mandó el cliente	`el X-Request-ID del gateway de C01`
grpc	el código de gRPC	`UNAVAILABLE, INVALID_ARGUMENT`
motivo_trt	la última línea de error de TensorRT	`The engine plan file is not compatible…`
mensaje	el texto que explica el código	`unexpected shape for input 'input'…`
excepcion	el tipo de error de Python	`RuntimeError, TorchExportError, InferenceServerException`
n · *codigo*	el número y el nombre del código de ONNX Runtime	`2 · INVALID_ARGUMENT`

LAS PIEZAS

PIEZA	QUÉ HACE	EJEMPLO
failed to load '…' version …	El core de Triton no pudo cargar esa versión: queda UNAVAILABLE con el motivo.	`Internal: unable to create TensorRT engine: …`
error: creating server: …	Triton no arrancó; con --exit-on-error=true basta un modelo que falle. Exit 1.	`Internal - failed to load all models`
[request id: …]	Triton rechazó un request; el id es el que mandó el cliente (<id_unknown> si no hay).	`unexpected shape for input 'input'`
[400] … · [503] …	tritonclient por HTTP: el código que respondió Triton, entre corchetes.	`[400] … data-type is 'FP64'`
[StatusCode.…]	tritonclient por gRPC: el código de gRPC.	`[StatusCode.UNAVAILABLE] failed to connect to all addresses`
The engine plan file …	TensorRT, al cargar un plan: versión o tipo de GPU equivocados.	`… not compatible with this version of TensorRT`
[ONNXRuntimeError] : 2 : INVALID_ARGUMENT : …	ONNX Runtime rechazó la entrada: nombre, shape o dtype (tipo de número).	`Got invalid dimensions for input: input`
Failed to export the model with torch.export	El exportador dynamo (el default desde torch 2.9) no pudo capturar el modelo: paso 1 de 3.	`Could not guard on data-dependent expression`

PIEZA	QUÉ HACE	EJEMPLO
Exporting the operator '…' to ONNX opset version …	El exportador TorchScript (dynamo=False, el viejo, el de C01) no sabe convertir esa operación en ese opset (la versión del conjunto de operadores de ONNX).	`aten::…`
TracerWarning	Una advertencia del exportador TorchScript: congeló un valor que depende de los datos. No falla.	`Converting a tensor to a Python boolean might cause the trace to be incorrect`
CUDA out of memory	PyTorch no consiguió VRAM: torch.OutOfMemoryError (torch.cuda.OutOfMemoryError es el mismo tipo).	`Tried to allocate 2.00 GiB`
DecompressionBombError · UnidentifiedImageError	Pillow, en el preproceso: una imagen demasiado grande, o bytes que no son una imagen.	`413 y 400 en el gateway de C01`
[parity] NO PASA	El gate de C01: el artefacto no predice como PyTorch FP32. Exit 1: no se promueve.	`top1_agreement < 99.5 %`
Failed to obtain stable measurement	perf_analyzer (la herramienta de carga de Triton): la medición no se estabilizó en 10 ventanas; esos números no valen.	`--measurement-interval`

LOS 26 MENSAJES

Error(s) in loading state_dict for ResNet: size mismatch for fc.weight: copying a param with shape torch.Size([1000, 2048]) from checkpoint, the shape in current model is torch.Size([500, 2048]).

El checkpoint no es de esta arquitectura: aquí, una capa final de 1000 clases (ImageNet) contra el modelo de C01, de 500. load_state_dict es estricto por defecto y falla al cargar, no en producción con pesos al azar. Si dice Missing key(s) o Unexpected key(s), los nombres no coinciden (por ejemplo, el prefijo module. de DataParallel).

PRIMERO:

```
python -c "import torch; sd =
torch.load('resnet50_500.pt', map_location='cpu',
weights_only=True); print(sd['fc.weight'].shape)"
```

→ Sección 2 · Preparar el modelo en PyTorch: eval, inference_mode, preprocesamiento idéntico y determinismo

Weights only load failed. … In PyTorch 2.6, we changed the default value of the `weights_only` argument in `torch.load` from `False` to `True`.

Desde torch 2.6, torch.load solo acepta tensores y tipos permitidos, y el archivo trae otro objeto de Python (un modelo entero, un optimizador, una clase propia). Cargarlo con weights_only=False ejecuta pickle, es decir, código: solo si confías en el origen. Lo sano es guardar y cargar el state_dict, como C01.

PRIMERO:

```
python -c "import torch;
print(torch.serialization.get_unsafe_globals_in_checkpoi
nt('resnet50_500.pt'))"
```

→ Sección 2 · Preparar el modelo en PyTorch: eval, inference_mode, preprocesamiento idéntico y determinismo

Exporting the operator 'aten::…' to ONNX opset version 17 is not supported

El exportador TorchScript (dynamo=False, el que usa C01) no sabe convertir esa operación de PyTorch a ONNX en ese opset (la versión del conjunto de operadores). Si el mensaje agrega Support for this operator was added in version N, alcanza con subir el opset; si no, reescribe esa parte con operaciones estándar, déjala fuera del grafo (como el NMS de un detector) o registra un custom op.

PRIMERO:

```
python model/export_onnx.py --out model/classifier.onnx
--dynamo --opset 18
```

→ Sección 3 · Exportar a ONNX: dynamo, opset, ejes dinámicos, checker y ONNX Runtime

Failed to export the model with torch.export. This is step 1/3 of exporting the model to ONNX.

El exportador dynamo (el default desde torch 2.9) no pudo capturar el modelo con torch.export. El motivo real está abajo, en el resumen del traceback; el típico es Could not guard on data-dependent expression: un if que depende de los valores del tensor. TorchScript lo habría congelado sin avisar; dynamo falla explícito. Exprésalo con torch.cond o sácalo del modelo.

PRIMERO:

```
cd model && python -c "import torch; from export_onnx
import build_model;
torch.export.export(build_model(None), (torch.randn(2,
3, 224, 224),))"
```

→ Sección 3 · Exportar a ONNX: dynamo, opset, ejes dinámicos, checker y ONNX Runtime

Failed to convert the model to the target version 17 using the ONNX C API. The model was not modified

La trampa del exportador dynamo que documenta C01 (onnxscript 0.7.2): pediste opset 17, intentó convertir hacia abajo, la conversión falló (en ResNet-50, por ReduceMean) y solo lo avisó: el archivo quedó en opset 18. Lo atrapa el verify() de export_onnx.py, que lee el opset real del archivo.

PRIMERO:

```
python -c "import onnx;
print(onnx.load('model/classifier.onnx').opset_import)"
```

→ Sección 3 · Exportar a ONNX: dynamo, opset, ejes dinámicos, checker y ONNX Runtime

[ONNXRuntimeError] : 2 : INVALID_ARGUMENT : Got invalid dimensions for input: input for the following indices index: 0 Got: 7 Expected: 2 Please fix either the inputs/outputs or the model.

ONNX Runtime esperaba un tamaño fijo en ese eje: el eje de batch quedó constante (2, el del ejemplo del export) porque no se declaró dinámico. Funciona con batch 2 y falla con cualquier otro. Se arregla en el export: dynamic_axes (TorchScript) o dynamic_shapes (dynamo).

PRIMERO:

```
python -c "import onnx; m =
onnx.load('model/classifier.onnx'); print([d.dim_param
or d.dim_value for d in
m.graph.input[0].type.tensor_type.shape.dim])"
```

→ Sección 3 · Exportar a ONNX: dynamo, opset, ejes dinámicos, checker y ONNX Runtime

[parity] NO PASA: no promuevas este artefacto

El gate de C01 (parity_check.py) encontró que el artefacto no predice como PyTorch FP32: acuerdo top-1 bajo 99.5 %, solapamiento top-5 bajo 99 % o algún |Δlogit| sobre 0.5. Exit 1: el pipeline se detiene. Un FP16 bien hecho pasa con holgura (diferencia media ≈ 1e-2; 1.3e-2 en el lab de paridad); 97 % de acuerdo es un bug, no redondeo.

PRIMERO:

```
python model/parity_check.py --checkpoint
resnet50_500.pt --onnx model/classifier.onnx --images
/ruta/muestra_eval --n 2000
```

→ Sección 4 · Paridad: el modelo servido no es el evaluado (top-1, Δlogit, FP16)

FAILED | Output: 'logits' | Difference exceeds tolerance (rel=1e-05, abs=1e-05)

Polygraphy (la herramienta de NVIDIA para comparar runtimes) comparó TensorRT contra ONNX Runtime con sus tolerancias por defecto, 1e-5, pensadas para FP32. Con --fp16 los logits se mueven ≈ 1e-2: falla aunque el engine esté bien. Sube --atol y --rtol a lo que FP16 justifica y decide con métricas de producto, como el acuerdo top-1 de parity_check.py.

PRIMERO:

```
polygraphy run model/classifier.onnx --trt --fp16 --
onnxrt --trt-outputs mark all --onnx-outputs mark all
```

→ Sección 4 · Paridad: el modelo servido no es el evaluado (top-1, Δlogit, FP16)

The engine plan file is not compatible with this version of TensorRT, expecting library version … got …

El model.plan se construyó con otra versión de TensorRT que la que corre: típicamente, Triton se actualizó de release de NGC y el engine no. Un engine solo sirve en la versión con que se construyó, salvo que lo compiles en modo version-compatible (opcional, algo más lento, y Triton solo lo carga con --backend-config=tensorrt,version-compatible=true). En el log de Triton aparece junto a failed to load 'classifier' version 1: Internal: unable to create TensorRT engine. La puntuación cambia entre versiones, y algunas lo dicen como Serialization assertion … Version tag does not match.

PRIMERO:

```
trtexec --getPlanVersionOnly --loadEngine=model.plan
```

→ Sección 5 · TensorRT: qué es, trtexec, precisión, perfiles de optimización, portabilidad del engine

The engine plan file is generated on an incompatible device, expecting compute … got compute …

El engine se compiló para otra arquitectura de GPU (compute capability): por ejemplo, en una L4 (8.9) y se carga en la T4 de C01 (7.5), o al revés. TensorRT eligió sus kernels midiendo en esa GPU. Se reconstruye en el tipo de GPU de destino; el modo de compatibilidad ampere+ no cubre la T4, que es Turing.

PRIMERO:

```
nvidia-smi --query-gpu=name,driver_version --
format=csv,noheader
```

→ Sección 5 · TensorRT: qué es, trtexec, precisión, perfiles de optimización, portabilidad del engine

Plugin not found, are the plugin name, version, and namespace correct?

El parser de ONNX de TensorRT no sabe importar ese operador, lo busca como plugin (una capa propia registrada en TensorRT) y no hay ninguno con ese nombre. Más arriba, el log nombra el operador: No importer registered for op: X (No checker registered for op: X en TensorRT 10.3) y Cannot find plugin: X, version: 1. Típico de operaciones propias o de detección (NMS, deformable conv). Opciones: reescribir con operadores estándar, registrar un plugin, o dejar esa parte fuera del engine (en el gateway o en otro paso del ensemble).

PRIMERO: `trtexec --onnx=model/classifier.onnx --verbose`

→ Sección 5 · TensorRT: qué es, trtexec, precisión, perfiles de optimización, portabilidad del engine

model 'classifier', tensor 'input': the model expects 4 dimensions (shape [-1,3,224,224]) but the model configuration specifies 5 dimensions (an initial batch dimension because max_batch_size > 0 followed by the explicit tensor shape, making complete shape [-1,-1,3,224,224])

Pusiste el batch en dims: con max_batch_size > 0, Triton antepone el -1 del batch, así que dims: [-1, 3, 224, 224] se vuelve un tensor de 5 dimensiones que el engine no tiene. Va dims: [3, 224, 224], como en C01. El modelo no carga y, con --exit-on-error=true, Triton sale. Es el backend de TensorRT; el de ONNX Runtime avisa parecido.

PRIMERO:

```
grep -n "dims"
triton/model_repository/classifier/config.pbtxt
```

→ Sección 6 · El model repository y config.pbtxt

unable to find backend library for backend '…', try specifying runtime on the model configuration.

El backend que pide config.pbtxt no existe en la imagen: un typo (onnxruntime mal escrito) o un backend que esa imagen no trae (el de TensorFlow dejó de venir en los contenedores de NGC desde la 25.03). Para TensorRT y ONNX alcanza con platform (tensorrt_plan, como C01, u onnxruntime_onnx); si escribes backend, tiene que ser el nombre de una carpeta de /opt/tritonserver/backends.

PRIMERO:

```
docker run --rm --entrypoint ls
nvcr.io/nvidia/tritonserver:24.08-py3
/opt/tritonserver/backends
```

→ Sección 6 · El model repository y config.pbtxt

`instance group classifier_0 of model classifier has kind KIND_GPU but no GPUs are available`

El config pide KIND_GPU (como C01) y el proceso de Triton no ve ninguna GPU: el contenedor arrancó sin --gpus, el pod no pidió nvidia.com/gpu, o cayó en un nodo sin GPU o sin driver. Con KIND_GPU no hay plan B: el modelo no carga, y el backend de TensorRT solo corre en GPU. Es un problema del entorno, no del config: al arrancar, Triton suele avisar antes con pinned memory pool will not be available.

PRIMERO:

```
kubectl get pod POD -n classifier -o jsonpath='{.spec.containers[0].resources.limits}'
```

→ Sección 12 · Diagnóstico: el método y la tabla de síntomas

`error: creating server: Internal - failed to load all models`

Triton no arrancó: con --exit-on-error=true (C01) alcanza con que un modelo no cargue para que el proceso salga con exit 1, y el pod queda en CrashLoopBackOff. Este mensaje no trae la causa: búscala arriba, en la tabla de modelos y en la línea failed to load '...' version ....

PRIMERO:

```
kubectl logs POD -n classifier --previous | grep -E "failed to load|UNAVAILABLE"
```

→ Sección 12 · Diagnóstico: el método y la tabla de síntomas

`Exceeds maximum queue size`

La cola del dynamic batcher está llena: default_queue_policy tiene un max_queue_size y ya hay esa cantidad de requests esperando. Triton rechaza el nuevo al instante, sin encolarlo, y llega como 503. C01 no pone tope de tamaño (0 = cola sin tope) y corta por tiempo, a los 100 ms. Si aparece seguido, la GPU no da abasto en tus picos: faltan réplicas, o el tope es más chico que un pico normal.

PRIMERO:

```
curl -s localhost:8000/v2/models/classifier/config
```

→ Sección 7 · Dynamic batching, instancias y concurrencia

`Request timeout expired`

La cola del dynamic batcher rechazó un request que esperó más de default_timeout_microseconds (100 ms en C01, con timeout_action: REJECT). Llega al cliente como 503 y cuenta en nv_inference_request_failure con reason="REJECTED" (esa etiqueta existe desde Triton 24.07). Es a propósito: no gastar GPU en una respuesta que el gateway (150 ms) ya no va a esperar. Si sale seguido, la GPU está saturada: faltan réplicas.

PRIMERO:

```
curl -s localhost:8002/metrics | grep nv_inference_request_failure
```

→ Sección 7 · Dynamic batching, instancias y concurrencia

`unexpected shape for input 'input' for model 'classifier'. Expected [-1,3,224,224], got [3,224,224]. NOTE: Setting a non-zero max_batch_size in the model config requires a batch dimension to be prepended to each input shape.`

El cliente declaró una shape que no es la del modelo: aquí, sin el eje de batch. Con max_batch_size > 0, la shape completa es [-1] + dims, así que una foto va como [1, 3, 224, 224] (x[np.newaxis] en C01). Otro clásico: [1, 224, 224, 3], HWC en vez de CHW. Triton responde 400; el NOTE propone max_batch_size: 0, pero eso apaga el batching.

PRIMERO: `curl -s localhost:8000/v2/models/classifier`

→ Sección 8 · Clientes: HTTP, gRPC, protocolo v2, health y metadata, tritonclient

`inference input 'input' data-type is 'FP64', but model 'classifier' expects 'FP32'`

El request declara FP64: NumPy crea float64 por defecto (np.random.rand, np.zeros) y np_to_triton_dtype lo traduce a FP64. Convierte con astype(np.float32). Si en cambio declaraste "FP32" y pasaste un float64, el error sale antes, en tritonclient: got unexpected datatype FP64 from numpy array, expected FP32.

PRIMERO:

```
python -c "import numpy as np; print(np.zeros((1, 3, 224, 224)).dtype)"
```

→ Sección 8 · Clientes: HTTP, gRPC, protocolo v2, health y metadata, tritonclient

`Request for unknown model: 'classifier' version 1 is not at ready state`

El modelo existe, pero esa versión no está lista: falló al cargar (con --exit-on-error=false) o se está cargando o descargando (con --model-control-mode=explicit o poll). Llega como 503. Sale cuando pides una versión concreta; sin versión, el mismo caso dice ... has no available versions (404). Y ... 'classifier' is not found (404) es que ese nombre no existe: revisa MODEL_NAME.

PRIMERO:

```
curl -s -X POST localhost:8000/v2/repository/index
```

→ Sección 8 · Clientes: HTTP, gRPC, protocolo v2, health y metadata, tritonclient

`Failed to obtain stable measurement within 10 measurement windows for concurrency 1. Please try to increase the --measurement-interval.`

perf_analyzer considera estable una medición cuando las 3 últimas ventanas varían menos de --stability-percentage (10 %) en throughput y en latencia; no lo logró en --max-trials (10) ventanas de --measurement-interval (5000 ms). Pocos requests por ventana, una GPU que baja clocks u otro tráfico en el mismo Triton. Esos números no valen: alarga la ventana o aísla el servidor.

PRIMERO:

```
perf_analyzer -m classifier -u localhost:8000 --concurrency-range 1 --measurement-interval 10000
```

→ Sección 9 · Medir: perf_analyzer, métricas de Triton, uso de GPU

`[WARNING] Perf Analyzer was not able to keep up with the desired request rate. … % of the requests were delayed.`

En modo de tasa fija (--request-rate-range, como el gate de latencia de C01 a 250 y 500 req/s), perf_analyzer mandó tarde más del 1 % de los requests: la carga real fue menor que la pedida y ese p99 no sirve para el gate. Por defecto manda en modo síncrono con 4 hilos (--max-threads), y cada hilo espera su respuesta: si la latencia sube, no alcanzan. Sube --max-threads o usa --async, y mira con top que a la máquina que mide le sobre CPU.

PRIMERO:

```
perf_analyzer -m classifier -u localhost:8000 --request-rate-range 500 --max-threads 16
```

→ Sección 9 · Medir: perf_analyzer, métricas de Triton, uso de GPU

`Image size (… pixels) exceeds limit of … pixels, could be decompression bomb DOS attack.`

Pillow se niega a abrir una imagen con más del doble de Image.MAX_IMAGE_PIXELS (DecompressionBombError; entre una y dos veces el límite, solo avisa con un warning). Un PNG de 50 KB puede declarar 30,000 × 30,000 px. C01 fija MAX_IMAGE_PIXELS en 40 MP como segunda red y, antes, lee solo el header y rechaza más de 40 MP (413) o una relación de aspecto mayor que 1:8 (400).

PRIMERO:

```
python -c "from PIL import Image; Image.MAX_IMAGE_PIXELS = None; im = Image.open('foto.png'); print(im.format, im.size)"
```

→ Sección 10 · Pre y postprocesamiento: dónde va (gateway, ensemble, backend de Python) y el riesgo de skew

`Failed to increase the shared memory pool size … If you are running Triton inside docker, use '--shm-size' flag to control the shared memory region size.`

El backend de Python (por ejemplo, el paso de preproceso de un ensemble) pasa los tensores por memoria compartida (/dev/shm) y no pudo agrandar su región: Docker da 64 MB por defecto, y Kubernetes también, salvo que montes un emptyDir en memoria como C01 (1 GiB). En 24.08 el texto dice … pool size for key '…' to … bytes; desde la 25.07, sin for key. Si falla ya al arrancar, el mensaje es otro: Unable to initialize shared memory key … Each Python backend model instance requires at least 1 MB of shared memory.

PRIMERO:

```
kubectl exec -n classifier deploy/triton -- df -h /dev/shm
```

→ Sección 10 · Pre y postprocesamiento: dónde va (gateway, ensemble, backend de Python) y el riesgo de skew

`CUDA out of memory. Tried to allocate … GPU 0 has a total capacity of … of which … is free.`

PyTorch no consiguió VRAM (torch.OutOfMemoryError; torch.cuda.OutOfMemoryError es el mismo tipo). En serving: batch o resolución demasiado grandes, varios procesos con su copia del modelo (uvicorn --workers) o inferencia sin torch.inference_mode, que guarda activaciones. Si hay mucha memoria reservada sin usar, el mensaje sugiere PYTORCH_CUDA_ALLOC_CONF=expandable_segments:True (torch 2.14). No es un OOMKilled: subir limits.memory no ayuda.

PRIMERO:

```
nvidia-smi --query-compute-apps=pid,used_memory --format=csv
```

→ Sección 12 · Diagnóstico: el método y la tabla de síntomas

`Unable to allocate pinned system memory, pinned memory pool will not be available: CUDA driver version is insufficient for CUDA runtime version`

Una advertencia de Triton al arrancar: CUDA no funciona en ese contenedor. O no hay GPU ni driver (corriste la imagen sin --gpus, o en tu Mac), o el driver del host es más viejo que el CUDA de la imagen (24.08 trae CUDA 12.6: driver 560 o, en GPUs de data center como la T4, ramas compatibles desde 470.57). Triton sigue arrancando, y después falla el modelo con KIND_GPU.

PRIMERO: `nvidia-smi`

→ Sección 12 · Diagnóstico: el método y la tabla de síntomas

EJEMPLO REAL · CÓMO LEER UN ERROR DE INFERENCIA: EL CLIENTE SE OLVIDÓ DEL EJE DE BATCH

TEXTO Forma típica con Triton 24.08 y tritonclient 2.72: los textos salen del código del core de Triton (infer_request.cc, igual en 24.08 y en la rama actual) y del cliente; la metadata sigue el protocolo KServe v2

```
$ python cliente.py        # InferInput("input", [3, 224, 224], "FP32") con x de shape [3, 224, 224]
tritonclient.utils.InferenceServerException: [400] [request id: demo-1] unexpected shape for input 'input' for model 'classifier'. Expected [-1,3,224,224], got [3,224,224]. NOTE: Setting a non-zero max_batch_size in the model config requires a batch dimension to be prepended to each input shape. If you want to specify the full shape including the batch dim in your input dims config, try setting max_batch_size to zero. See the model configuration docs for more info on max_batch_size.

$ curl -s localhost:8000/v2/models/classifier
{"name":"classifier","versions":["1"],"platform":"tensorrt_plan","inputs":[{"name":"input","datatype":"FP32","shape":[-1,3,224,224]}],"outputs":[{"name":"logits","datatype":"FP32","shape":[-1,500]}]}
```

- Léelo por partes: `[400]` es el código HTTP (lo agrega tritonclient); `[request id: demo-1]`, el id que mandaste; después, el motivo. Un 400 es tu request, no el servidor: reintentar no sirve.
- `Expected [-1,3,224,224]`: con `max_batch_size` > 0, Triton antepone -1 (el batch) a los dims del config. `got [3,224,224]`: faltó el eje de batch. Arreglo: `x[np.newaxis]`, como el gateway de C01.

- El NOTE sugiere `max_batch_size: 0`. No en C01: apagaría el dynamic batching. Se arregla el cliente.
- Si el error sale antes de llegar a Triton (`got unexpected numpy array shape …` o `got unexpected datatype FP64 from numpy array, expected FP32`), lo dijo tritonclient al armar el request: el array no coincide con lo que declaraste en `InferInput`.
- Los 503 también vienen de Triton (`Request timeout expired`, `Exceeds maximum queue size`, `… is not at ready state`): son de capacidad o de estado, no de contrato. El gateway de C01 convierte cualquier error de Triton en 503 con `Retry-After` (el header que dice en cuántos segundos reintentar).

CÓMO RECORDARLA

el prefijo dice quién habla. `failed to load` y `error: creating server` son Triton cargando; `[request id: …]` es Triton rechazando un request; `The engine plan file` y `unable to create TensorRT engine`, TensorRT; `[ONNXRuntimeError]`, ONNX Runtime; el resto, una excepción de Python con su librería en el nombre.

TRAMPA

leer solo la última línea. `error: creating server: Internal - failed to load all models` no dice nada: el porqué está arriba, en `failed to load` y en la línea de TensorRT. Y los textos cambian entre versiones: TensorRT cambia la puntuación de sus mensajes y el backend de Python de Triton quitó `for key` de su mensaje de memoria compartida en la 25.07. Busca el núcleo estable del mensaje, no la línea exacta.

En el curso: C01 · Modos de falla y degradación · Docker y kubectl · Depurar pods: síntoma → comandos → causa típica · M8 · ONNX y ONNX Runtime · M8 · TensorRT · M7 · NVIDIA Triton Inference Server · M7 · Protocolos: REST, gRPC, KServe v2 y OpenAI-compatible

---

Comandos, campos, flags y mensajes revisados en octubre de 2026 contra la documentación oficial de PyTorch, ONNX, ONNX Runtime, NVIDIA TensorRT, Triton Inference Server y tritonclient, y contra los archivos de `reference/c01` del curso. Las versiones son las del C01: torch 2.14, NGC 24.08 (TensorRT 10.3 y Triton 2.49.0); cuando la versión actual cambia algo, como TensorRT 11, la sección lo dice. Ningún laboratorio necesita GPU: los marcados como probados se corrieron de verdad en un Mac; los demás se verificaron contra la documentación. Si tu terminal dice otra cosa, créele a tu terminal.

© 2026 Erik Palacios · Contenido bajo CC BY-NC-ND 4.0 · ML desde cero · Reportar un error