

R3AL.ai — Technische Handleiding

R3AL Quant & r3alai SDK's

Juli 2026

Inleiding

Dit document is de volledige technische handleiding voor de twee open-source Python SDK's in deze monorepo:

1. **R3AL Quant** (`r3al-quant`) — Post-Training Quantization voor LLM's en Vision-modellen, ontworpen om te draaien op RunPod GPU's.
2. **r3alai** (`r3alai`) — Random Set Neural Networks (RSNN) voor onzekerheidsschatting, conformal prediction en active learning.

Beide SDK's zijn onafhankelijk installeerbaar en hebben geen runtime- afhankelijkheid van elkaar. Deze handleiding legt exact uit **hoe elk onderdeel werkt**: de architectuur, de onderliggende wiskunde, de volledige API, en hoe je alles in productie (o.a. op RunPod) draait.

Deel 1 — R3AL Quant SDK

1.1 Wat is Post-Training Quantization (PTQ)?

Een neurale netwerk bestaat uit gewichten (weights) en activaties, standaard opgeslagen als 32-bit floating point (FP32) getallen. Quantization zet deze getallen om naar een kleiner formaat (bijv. INT8 of INT4) om:

- **Geheugen te besparen** (een INT4-model is ~8x kleiner dan FP32),
- **Sneller te rekenen** (integer-arithmetiek is goedkoper dan float-arithmetiek op veel hardware),
- met een acceptabel verlies aan nauwkeurigheid.

Post-Training betekent dat dit gebeurt **na** het trainen, op een reeds getraind model, zonder het volledige trainingsproces te herhalen. Dit staat tegenover *Quantization-Aware Training (QAT)*, waarbij quantization al tijdens het trainen wordt gesimuleerd.

De SDK ondersteunt twee hoofdcategorieën van methodes:

Categorie	Betekenis	Calibratiedata nodig?
Dynamic quantization	Alleen de gewichten worden vooraf gekwantiseerd; activaties worden tijdens inference "on the fly" gekwantiseerd.	Nee
Static quantization	Zowel gewichten als activaties worden vooraf gekwantiseerd, gebaseerd op statistieken verzameld met een kleine representatieve dataset.	Ja (calibratie)

1.2 Architectuur

De SDK is opgebouwd rond vier kernconcepten:

```
QuantConfig ---> Quantizer ---> Backend (per methode) ---> QuantizedModel
                    |
                    +--> Benchmark
```

- **QuantConfig** (Pydantic-model) — beschrijft *wat* je wilt kwantiseren (bits, methode, model-type) en valideert dit meteen bij constructie.
- **Quantizer** — de hoofdklasse die de juiste backend opzoekt op basis van de config en de gebruikersvriendelijke `quantize()` / `benchmark()` / `load()` methodes aanbiedt.
- **Backend** — één klasse per combinatie van `model_type` + `method` (bijv. `AWQBackend`, `ONNXBackend`, `APoTBackend`), die de daadwerkelijke kwantisatie-logica implementeert via de onderliggende bibliotheek (AutoAWQ, AutoGPTQ, bitsandbytes, ONNX Runtime, PyTorch) of, voor APoT, eigen

wiskunde.

- **QuantizedModel** — een handle naar het resultaat op schijf, met `.save()` en `.load()`.

Dit ontwerp betekent dat het toevoegen van een nieuwe methode neerkomt op het schrijven van één nieuwe backend-klasse en het registreren ervan in `backends/__init__.py` — de rest van de SDK (config-validatie, benchmarking, foutafhandeling) werkt automatisch mee.

Bestandsstructuur

```
src/r3al_quant/
■■■ config.py          # QuantConfig (Pydantic) + validatie
■■■ types.py           # Enums: ModelType, LLMMethod, VisionMethod, QuantBackend
■■■ quantizer.py        # Quantizer + QuantizedModel (hoofd-API)
■■■ benchmark.py        # Latency / VRAM / kwaliteitsvergelijking
■■■ calibration.py      # Hulpfuncties voor calibratiedata
■■■ errors.py           # Typed exceptions
■■■ algorithms/
■   ■■■ apot.py         # APoT / APoT-R3AL niveauconstructie & fake-quant wiskunde
■■■ backends/
■   ■■■ base.py         # BaseBackend abstracte klasse
■   ■■■ llm/            # awq.py, gptq.py, bnb.py, apot.py
■   ■■■ vision/         # onnx.py, pytorch.py, apot.py, _common.py

handler.py             # RunPod serverless entrypoint
test_input.json        # Lokaal testbestand voor handler.py
Dockerfile             # RunPod GPU image
```

1.3 Installatie

```
# Alleen de kern (config, quantizer, benchmark-logica)
pip install -e .

# LLM-methodes (AWQ + GPTQ + bitsandbytes)
pip install -e ".[llm,awq,gptq,bnb]"

# Vision-methodes (ONNX Runtime + PyTorch/torchvision)
pip install -e ".[vision]"

# RunPod serverless handler
pip install -e ".[serverless]"

# Alles in één keer
pip install -e ".[all,dev]"
```

De optionele dependencies zijn bewust gescheiden: als je alleen LLM's kwantiseert met bitsandbytes, hoef je geen ONNX Runtime of torchvision te installeren.

1.4 QuantConfig — alle velden uitgelegd

`QuantConfig` is een Pydantic-model. Elke instantie wordt **automatisch gevalideerd** bij aanmaak (`model_validator(mode="after")`): een ongeldige combinatie van `bits` + `method` gooit direct een duidelijke fout, in plaats van pas te falen tijdens een dure quantization-run.

Veld	Type	Default	Betekenis
<code>model_type</code>	<code>"llm" "vision"</code>	<code>"llm"</code>	Bepaalt welke methode-set en backend-registry gebruikt wordt.
<code>bits</code>	<code>int</code> (4–8)	<code>4</code>	Bit-breedte. LLM: 4 of 8. Vision: alleen 8.
<code>method</code>	<code>str</code>	<code>"awq"</code>	De kwantisatie-methode (zie tabel in §1.5/1.6).
<code>backend</code>	<code>"cuda" "cpu" "auto"</code>	<code>"auto"</code>	Gewenst rekenapparaat; <code>"auto"</code> kiest GPU als beschikbaar.
<code>target_device</code>	<code>str</code>	<code>"auto"</code>	Specifiek device-string (bijv. <code>"cuda:1"</code>), overschrijft <code>backend</code> .
<code>group_size</code>	<code>int</code>	<code>128</code>	Groepsgrootte voor gewicht-packing bij AWQ/GPTQ.
<code>zero_point</code>	<code>bool</code>	<code>True</code>	Of een zero-point wordt gebruikt (asymmetrische i.p.v. symmetrische quantization) bij AWQ.
<code>dtype</code>	<code>"int4" "int8" "uint8"</code>	<code>"int4"</code>	Wordt automatisch afgeleid van <code>bits/method</code> tijdens validatie.
<code>apot_smoothing</code>	<code>float</code> (0, 1]	<code>0.6</code>	Alleen voor <code>method="apot_r3a1"</code> : de power-law-exponent (zie §1.7).

<code>apot_per_channel</code>	<code>bool</code>	<code>True</code>	Of APoT een aparte schaal per output-kanaal gebruikt i.p.v. één schaal per tensor.
-------------------------------	-------------------	-------------------	--

Twee handige, afgeleide properties:

- `config.requires_calibration` — `True` voor `awq`, `gptq`, `onnx_static`, `pytorch_static`. Als je deze methodes gebruikt zonder `calibration_data` mee te geven, gooit de backend een `CalibrationRequiredError` — dit gebeurt dus vóórdat er kostbare GPU-tijd wordt verspild.
- `config.is_apot` — `True` voor `apot` en `apot_r3al`.

1.5 LLM-methodes

Methode	Bits	Calibratie	Onderliggende bibliotheek
<code>awq</code>	4	Verplicht	AutoAWQ
<code>gptq</code>	4	Verplicht	AutoGPTQ
<code>bnb</code>	4, 8	Niet nodig (dynamic)	bitsandbytes
<code>apot</code>	4, 8	Niet nodig	Eigen implementatie (zie §1.7)
<code>apot_r3al</code>	4, 8	Niet nodig	Eigen implementatie (zie §1.7)

AWQ (Activation-aware Weight Quantization)

AWQ observeert tijdens calibratie welke gewichten het meest "belangrijk" zijn (gebaseerd op de grootte van de bijbehorende activaties) en beschermt die gewichten extra tijdens het kwantiseren, terwijl de rest agressief naar 4-bit gaat. Dit levert doorgaans een betere nauwkeurigheid op dan naïeve 4-bit quantization bij dezelfde compressie.

```
from r3al_quant import Quantizer, QuantConfig

config = QuantConfig(model_type="llm", bits=4, method="awq")
quantizer = Quantizer(config)

calib_texts = ["Voorbeeldtekst voor calibratie..."] * 128
result = quantizer.quantize(
    model="meta-llama/Llama-3.2-3B",
    calibration_data=calib_texts,
    output_dir="./quantized_llm",
```

```
)
```

Intern (`backends/llm/awq.py`) laadt de backend het model via `AutoAWQForCausalLM`, tokenizeert de `calibratiedata`, roept `awq_model.quantize(...)` aan met een `quant_config` (`w_bit`, `q_group_size`, `zero_point`), en slaat het resultaat + een `r3al_quant_meta.json` metadatabestand op.

GPTQ

GPTQ kwantiseert laag-voor-laag en lost na elke laag een least-squares probleem op om het kwantisatie-residu te compenseren in de nog niet verwerkte gewichten, wat de cumulatieve fout beperkt. Ook dit vereist `calibratiedata` (tekstvoorbeelden die door het model gestuurd worden).

BitsAndBytes (bnb)

Dynamische quantization: er is **geen `calibratiedata` nodig**. Het model wordt geladen met `BitsAndBytesConfig` (`load_in_4bit` met NF4-quantization + double quantization, of `load_in_8bit`). Dit is de snelste manier om een model te comprimeren, met iets minder controle over de nauwkeurigheid dan AWQ/GPTQ.

1.6 Vision-methodes

Methode	Bits	Calibratie	Onderliggende bibliotheek
<code>onnx_dynamic</code>	8	Niet nodig	ONNX Runtime
<code>onnx_static</code>	8	Verplicht	ONNX Runtime
<code>pytorch_dynamic</code>	8	Niet nodig	PyTorch (<code>torch.quantization</code>)
<code>pytorch_static</code>	8	Verplicht	PyTorch (<code>torch.quantization</code>)
<code>apot</code>	8	Niet nodig	Eigen implementatie (zie §1.7)
<code>apot_r3al</code>	8	Niet nodig	Eigen implementatie (zie §1.7)

Voor **static** methodes stuurt de SDK calibratie-afbeeldingen door het model (via `CalibrationDataReader` voor ONNX, of een `prepare()/observer-` forward-pass voor PyTorch) om de min/max-waarden van de activaties te meten, zodat de FP32-bereiken correct naar INT8 worden afgebeeld.

```

from r3al_quant import Quantizer, QuantConfig

config = QuantConfig(model_type="vision", method="onnx_static", bits=8)
quantizer = Quantizer(config)
result = quantizer.quantize(
    model="path/naar/model.onnx",
    calibration_data=calibration_images,
    output_dir="./quantized_onnx",
)

```

Voor **PyTorch-modellen** (`.pt/.pth` bestand, of een map met `model_config.json` die een torchvision-architectuurnaam specificeert) laadt de gedeelde loader (`backends/vision/_common.py`) het model consistent voor zowel de gewone PyTorch-backend als de APoT-backend.

1.7 APoT & APoT-R3AL — de wiskunde in detail

Dit is de kwantisatiemethode die R3AL.ai zelf heeft ontwikkeld, gebaseerd op het whitepaper "*APoT Quantization Whitepaper*" (R3AL.AI, juni 2026) en het onderliggende onderzoek van Li, Dong & Wang (2020).

Het probleem met uniforme quantization

Bij **uniforme quantization** worden de kwantisatieniveaus lineair verdeeld over een interval $[-\beta, \beta]$: $Q = \{-\beta, -\beta+\Delta, \dots, \beta\}$ voor een vaste stapgrootte Δ . Gewichten in een neurale netwerk zijn echter typisch **klokvormig verdeeld met zware staarten** (heavy-tailed): de meeste gewichten liggen dicht bij nul, met een paar uitschieters die het interval $[-\beta, \beta]$ oprekken. Het gevolg: een uniform rooster verspilt resolutie waar de meeste gewichten zitten (rond nul) en is te grof in de staarten.

Het idee achter APoT

Additive Powers-of-Two (APoT) bouwt elk kwantisatieniveau op als een **som van een paar machten van twee**, bijvoorbeeld $2^{-1} + 2^{-3}$, in plaats van een lineair stapje. Dit concentreert de niveaus rond nul (waar de meeste gewichten zitten) terwijl de staarten toch bereikbaar blijven — en, cruciaal, **vermenigvuldigen met zo'n niveau kan nog steeds via snelle bit-shifts** i.p.v. een volledige vermenigvuldiging, wat de hardware-efficiëntie van uniforme quantization behoudt.

Constructie van de niveaus (`build_apot_levels`)

Voor een bit-breedte b (moet even zijn, bijv. 4 of 8) wordt b verdeeld over $n = b / 2$ "additieve termen". Elke term t (van 0 tot $n-1$) krijgt een verzameling van 4 mogelijke waarden: $\{0\}$ plus 3 machten van twee met exponent $-(n \cdot i + t + 1)$ voor $i = 0, 1, 2$:

```

term_0: {0, 2-1, 2-3, 2-5, ...}    (bij n=2)
term_1: {0, 2-2, 2-4, 2-6, ...}

```

Een kwantisatieniveau is de som van **één gekozen waarde per term**. Met n termen en 4 keuzes per term geeft dit $4^n = 2^b$ unieke niveaus — exact het aantal dat je verwacht bij een b -bit quantizer. Voor $b=4$ ($n=2$) zijn dat 16 niveaus in de positieve helft; voor $b=8$ ($n=4$) zijn dat er 256.

Deze niveaus worden genormaliseerd zodat het grootste niveau exact `1.0` is, en vervolgens gespiegeld (`symmetric_levels`) om een symmetrische set rond nul te krijgen (bijv. 31 niveaus totaal voor 4-bit, met 0 als gedeeld middelpunt).

R3AL's aanpassing: `apot_r3a1`

Het whitepaper laat zien (Figuur 2 vs. Figuur 3) dat de standaard APoT- niveaus weliswaar beter zijn dan uniforme quantization, maar dat de **staarten nog steeds relatief schaars bevolkt** zijn — er zitten grote gaten tussen de hogere niveaus. R3AL past hierop een **concave machtstransformatie** toe:

$$\text{niveau}' = \text{niveau} \wedge \text{smoothing} \quad (\text{smoothing} \in (0, 1])$$

met `smoothing=1.0` die exact de standaard APoT-niveaus reproduceert. Omdat deze transformatie **monotoon** is, blijft het aantal niveaus, de volgorde en de symmetrie behouden — alleen de **afstand tussen niveaus** verandert:

- Kleine waarden (al dicht opeen in standaard APoT) worden verder uit elkaar getrokken.
- Grote waarden (schaars in standaard APoT) worden dichterbij elkaar gebracht — de staart wordt "evenrediger bevolkt".

Belangrijke afweging: na deze transformatie zijn de niveaus geen exacte sommen van machten van twee meer, dus gaat de bit-shift-snelheidswinst van standaard APoT verloren voor de `apot_r3a1`-variant. Gebruik `apot` als je die hardware-eigenschap nodig hebt; gebruik `apot_r3a1` als een betere fit op de gewichtsverdeling zwaarder weegt dan pure multiply-free arithmetiek.

Toepassen op een tensor (`apot_quantize_tensor`)

Voor een gewichtstensor wordt eerst een schaal γ bepaald:

- **Per-tensor:** $\gamma = \max(|w|)$ over de hele tensor.
- **Per-kanaal** (`apot_per_channel=True`, de default): een aparte γ per output-kanaal — nauwkeuriger, omdat verschillende kanalen vaak een andere waardebereik hebben.

Daarna wordt elk gewicht genormaliseerd (w / γ , geclipt naar `[-1, 1]`) en afgerond naar het **dichtstbijzijnde APoT-niveau** — dit gebeurt via `torch.searchsorted` (binaire zoektocht, $O(N \log L)$) in plaats van een volle afstandsmatrix, wat essentieel is bij tensoren met miljoenen gewichten. Het resultaat wordt terug geschaald ($\text{niveau} \times \gamma$) en teruggeschreven in het model: dit is **fake-quantization** (quantize → dequantize), de standaard manier om de nauwkeurigheidssimpact van een quantization-schema te evalueren zonder een aangepaste low-bit opslagkernel te hoeven bouwen.

Let op: de huidige implementatie bewaart het gekwantiseerde model nog als float16 (voor evaluatiedoeleinden). Een echte bit-packed export (waarbij de opslag daadwerkelijk kleiner wordt) staat op de roadmap.

```
from r3al_quant import Quantizer, QuantConfig

# Standaard APoT
config = QuantConfig(model_type="llm", bits=4, method="apot")
Quantizer(config).quantize("meta-llama/Llama-3.2-3B", output_dir="./apot_model")
```

```
# R3AL's variant, met aangepaste smoothing
r3al_config = QuantConfig(
    model_type="llm", bits=4, method="apot_r3al",
    apot_smoothing=0.6, apot_per_channel=True,
)
Quantizer(r3al_config).quantize("meta-llama/Llama-3.2-3B", output_dir="./apot_r3al_model")
```

Werkt identiek voor vision-modellen, waar zowel `nn.Linear` als `nn.Conv2d` gewichten worden gekwantiseerd.

1.8 Calibratie in detail

Voor methodes met `requires_calibration=True` verwacht de SDK `calibration_data`: een kleine, representatieve steekproef van echte data (tekstprompt of afbeeldingen), typisch 100–128 samples. De module `calibration.py` biedt:

- `validate_calibration(...)` — gooit direct een `CalibrationRequiredError` als calibratiedata ontbreekt voor een methode die het nodig heeft.
- `llm_calibration_generator(...)` — tokenizeert tekstsamples in batches.
- `vision_calibration_generator(...)` — past een optionele transform toe op afbeeldingsamples.

Deze calibratiedata wordt **nooit** gebruikt om het model verder te trainen — alleen om de min/max-statistieken van activaties te meten (static methods) of om AWQ/GPTQ te laten bepalen welke gewichten belangrijk zijn.

1.9 Benchmarking

`Quantizer.benchmark(...)` vergelijkt het originele en het gekwantiseerde model op drie assen, en retourneert een `BenchmarkResult` (Pydantic-model):

Metriek	Velden	Betekenis
Snelheid	<code>latency_ms_original</code> , <code>latency_ms_quantized</code> , <code>latency_speedup</code>	Gemiddelde inferentietijd per forward-pass (na warmup-runs, <code>torch.cuda.synchronize()</code> voor eerlijke GPU-timing).
Geheugen	<code>vram_mb_original</code> , <code>vram_mb_quantized</code> , <code>vram_reduction_pct</code>	Piek-VRAM-gebruik via <code>torch.cuda.max_memory_allocated()</code> .
Kwaliteit	<code>quality_original</code> , <code>quality_quantized</code> , <code>quality_delta</code>	Optioneel — je geeft zelf een <code>quality_fn(model, data) -> float</code> mee (bijv. perplexity voor LLM's, F1-score voor classificatie).

```

bench = quantizer.benchmark(
    original_model="meta-llama/Llama-3.2-3B",
    quantized_model=result,          # of een pad-string
    eval_data=["The capital of France is"],
    quality_metric="perplexity",
)
print(f"Snelheid: {bench.latency_speedup:.2f}x sneller")
print(f"VRAM: {bench.vram_mb_original:.0f}MB -> {bench.vram_mb_quantized:.0f}MB")

```

Voor vision-ONNX-modellen wordt alleen de gekwantiseerde latency gemeten (een aparte ONNX-sessie voor het origineel is aan de gebruiker); voor PyTorch-vision en LLM's worden beide modellen in hetzelfde proces geladen en vergeleken.

1.10 Foutafhandeling

Alle SDK-specifieke fouten erven van `R3ALQuantError`, zodat een aanroeper ze met één `except` kan opvangen zonder generieke Python-excepties per ongeluk mee te vangen:

```

R3ALQuantError
├── UnsupportedConfigError
├── ┌── UnsupportedBitsError          (bijv. 3-bit gevraagd, maar alleen 4/8 ondersteund)
│   ├── BackendNotInstalledError    (optionele dependency ontbreekt)
│   ├── CalibrationRequiredError     (static methode zonder calibration_data)
│   ├── ModelLoadError              (model kan niet geladen worden)
│   ├── QuantizationError           (fout tijdens het kwantiseren zelf)
│   └── BenchmarkError              (fout tijdens benchmarken)

```

Deze hiërarchie wordt direct gebruikt in `handler.py` (zie §1.11) om gestructureerde JSON-foutmeldingen terug te geven in plaats van de RunPod- worker te laten crashen.

1.11 RunPod Deployment

Serverless — `handler.py`

`handler.py` (repo-root) is een kant-en-klare RunPod Serverless-entrypoint met twee acties:

"quantize" — voert `Quantizer.quantize(...)` uit:

```

{
  "input": {
    "action": "quantize",
    "config": { "model_type": "llm", "bits": 4, "method": "apot_r3al", "apot_smoothing": 0.6 },
    "model": "meta-llama/Llama-3.2-3B",
    "output_dir": "/runpod-volume/quantized_output",
    "calibration_data": ["voorbeeldtekst", "..."],
    "max_calib_samples": 128
  }
}

```

"benchmark" — voert `Quantizer.benchmark(...)` uit:

```
{
  "input": {
    "action": "benchmark",
    "config": { "model_type": "llm", "bits": 4, "method": "apot_r3al" },
    "original_model": "meta-llama/Llama-3.2-3B",
    "quantized_model": "/runpod-volume/quantized_output",
    "eval_data": ["The capital of France is"],
    "quality_metric": "perplexity"
  }
}
```

Foutafhandeling: elke `R3ALQuantError`-subklasse wordt opgevangen en teruggegeven als `{"status": "error", "error": "...", "type": "UnsupportedBitsError"}` in plaats van de worker te laten crashen — de RunPod-job faalt netjes met een leesbare reden. Na elke job wordt `torch.cuda.empty_cache()` + `gc.collect()` aangeroepen, omdat serverless workers "warm" blijven tussen jobs en anders GPU-geheugen kunnen opstapelen tussen modellen van verschillende groottes.

Lokaal testen (zonder een RunPod-endpoint):

```
pip install -e ".[all,serverless]"
python handler.py # leest automatisch test_input.json
```

Docker-image

De `Dockerfile` gebruikt `runpod/pytorch:2.1.0-py3.10-cuda11.8.0-devel` als basis, installeert de SDK met alle extras (`pip install -e ".[all]"`), kopieert `handler.py` + `test_input.json`, en start de handler als `CMD`:

```
docker build -t jouw-registry/r3al-quant:latest .
docker push jouw-registry/r3al-quant:latest
```

Koppel deze image vervolgens aan een RunPod Serverless-endpoint. Jobs worden aangeroepen via de RunPod API met exact dezelfde `{"input": {...}}`-vorm als `test_input.json`.

Pod (interactief / lang-lopend)

Voor niet-serverless gebruik (een losse GPU-pod) kun je dezelfde image gebruiken en een eigen Python-script draaien dat `Quantizer` direct aanroept.

1.12 Testen

```
pip install -e ".[dev]"
pytest # 33 tests: config-validatie, APoT-wiskunde, backend-routing
ruff check src tests
```

Deel 2 — r3alai SDK

2.1 Wat is een Random Set Neural Network (RSNN)?

Een gewone classifier voorspelt één kansverdeling over de klassen (via softmax). Een RSNN voorspelt in plaats daarvan een **massafunctie** (afkomstig uit de Dempster-Shafer / random-set-theorie), die expliciet onderscheid maakt tussen:

- **Aleatorische onzekerheid** — de klassen zelf zijn inherent ambigu (het plaatje toont écht iets tussen een kat en een hond in).
- **Epistemische onzekerheid** — het model heeft simpelweg **niet genoeg bewijs** gezien om zeker te zijn (bijv. een out-of-distribution sample).

Een standaard softmax-verdeling kan deze twee vormen van onzekerheid niet uit elkaar houden: een uniforme verdeling over 3 klassen kan zowel "de klassen zijn echt ambigu" als "ik heb geen idee" betekenen. RSNN's lossen dit op door onzekerheid een **eigen plek in de output** te geven.

De massafunctie

De volledige random-set-theorie staat massa toe op **elke deelverzameling** van de klassenruimte (de machtsverzameling), wat exponentieel is in het aantal klassen en dus onbruikbaar om direct te voorspellen. Deze SDK gebruikt daarom de gangbare **tractabele restrictie**: massa op elke singleton-klasse, plus één **extra focal element** — de volledige klassenverzameling **Theta** ("ik weet het niet"):

```
m = (m_1, ..., m_n, m_Theta)          met som(m_i) + m_Theta = 1,   alle termen >= 0
```

Dit wordt gegarandeerd door de belief-laag een **softmax** te laten toepassen op **n_classes + 1** outputs (**logits_to_mass** in **models/belief.py**) — een geldige massafunctie per constructie, geen extra constraints nodig.

Belief, Plausibility en de pignistic transformatie

Omdat **Theta** de enige niet-singleton focal element is (en een superset van elke klasse), vereenvoudigen de klassieke Dempster-Shafer-formules tot gesloten vorm:

Grootheid	Formule	Betekenis
Belief Bel(c_i)	= m_i	Ondergrens: bewijs dat <i>specifiek</i> voor klasse i pleit.
Plausibility Pl(c_i)	= m_i + m_Theta	Bovengrens: bewijs dat klasse i <i>niet uitsluit</i> (want Theta overlapt met alles).

Pignistic <code>BetP(c_i)</code>	<code>= m_i + m_Theta/n</code>	Puntschatting voor beslissingen: onwetendheid gelijk verdeeld over alle klassen.
---	--------------------------------	--

Het interval `[Bel(c_i), Pl(c_i)]` heet het **credal interval**: de breedte ervan is exact `m_Theta` voor elke klasse — dus de gemiddelde credal-breedte is simpelweg de onwetendheidsmassa zelf. Dit is precies de **epistemische onzekerheid**, losgekoppeld van klasse-ambigüiteit.

De twee onzekerheidsmaten

- **pignistic_entropy(mass)** — Shannon-entropie van de pignistic- verdeling `BetP`. Hoog wanneer de verdeling bijna uniform is — dit kan komen door échte klasse-ambigüiteit **of** door hoge onwetendheid (beide mengen hier samen).
- **credal_width(mass)** — simpelweg `m_Theta`. Isoleert **puur** de "ik heb niet genoeg bewijs gezien"-component, onafhankelijk van of de klassen onderling ambigu zijn.

Praktisch verschil: een sample dat volledig buiten de trainingsdistributie valt (out-of-distribution) zal typisch hoge `credal_width` hebben (het model heeft er niets mee gedaan), terwijl een sample dat *wél* bekend is maar inherent dubbelzinnig is (bijv. een kat die op een hond lijkt) hoge `pignistic_entropy` maar lage `credal_width` kan hebben.

De trainingsdoelfunctie (`rsnn_loss`)

```
loss = cross_entropy(BetP, y) + alpha * gemiddelde(correct * m_Theta) + beta * gemiddelde(foute_kl
```

- **Cross-entropy-term:** standaard classificatieverlies, maar berekend op de *pignistic* verdeling (niet direct op de massa).
- **alpha (ignorance-decay):** bestraft resterende onwetendheidsmassa `m_Theta` op samples die het model al **correct** classificeert — dit dwingt het model om te "committeren" zodra er genoeg bewijs is, in plaats van altijd een beetje te hedgen. Verhoog α als je model te besluiteloos blijft.
- **β (wrong-evidence-penalty):** bestraft massa die op een **foute** singleton-klasse wordt gelegd. Dit is wat onzekerheid naar `m_Theta` duwt in plaats van naar een verkeerde klasse. Verhoog β als het model te overtuigd is van foute antwoorden.

Beide zijn instelbaar via `RSNNClassifier(alpha=..., beta=...)`.

2.2 `RSNNClassifier` — architectuur en training

Constructie

```
from r3alai import RSNNClassifier

model = RSNNClassifier(
    backbone="resnet50", # of "mobilenet_v2", "efficientnet_b0", "custom"
    n_classes=10,
    alpha=0.01,
    beta=0.01,
    pretrained=True,
```

```
device="auto",          # "auto" | "cuda" | "cpu" | "cuda:0" ...
)
```

Bij een torchvision-backbone (`resnet50`, `mobilenet_v2`, `efficientnet_b0`) wordt de **originele classificatiekop automatisch verwijderd** (`backbones.py` vervangt `.fc/.classifier` door `nn.Identity()`) zodat de backbone een platte featurevector teruggeeft, en wordt daar direct een `nn.Linear(feature_dim, n_classes + 1)` belief-laag op gezet.

Bij `backbone="custom"` moet je zelf twee dingen doen:

```
model = RSNNClassifier(backbone="custom", n_classes=5)
model.base_model = mijn_eigen_feature_extractor      # elke nn.Module -> platte vector
model.set_belief_layer(feature_dim=512)
```

Hugging Face-modellen integreer je op dezelfde manier, door een kleine adapter te schrijven die de HF-output (bijv. `pooler_output`) naar een vaste dimensie afbeeldt:

```
class HFBackbone(nn.Module):
    def __init__(self, hf_model, hidden_size, out_dim=512):
        super().__init__()
        self.hf_model = hf_model
        self.adapter = nn.Linear(hidden_size, out_dim)

    def forward(self, x):
        pooled = self.hf_model(x).pooler_output
        return self.adapter(pooled)

model.base_model = HFBackbone(hf_model, hidden_size=768)
model.set_belief_layer(feature_dim=512)
```

Training — `fit()`

```
history = model.fit(
    X_train, y_train_one_hot,      # tensors of ndarrays; y is one-hot
    epochs=50,
    batch_size=32,
    lr=1e-3,
    val_split=0.1,                 # interne validatiesplit voor early stopping
    patience=5,
    verbose=True,
)
```

Intern gebeurt het volgende (`models/rsnn.py`):

1. Input wordt geconverteerd naar tensors (`_to_tensor`, ondersteunt zowel `torch.Tensor` als `numpy.ndarray`).
2. Een willekeurige train/val-split wordt gemaakt (`torch.randperm`).
3. **Adam-optimizer** + **ReduceLRonPlateau**-scheduler (halveert de learning rate als de validatieloss stagneert).
4. Per epoch: train op mini-batches met `rsnn_loss`, evalueer op de validatieset.
5. **Early stopping**: als de validatieloss `patience` epochs niet verbetert, stopt het trainen en wordt het **beste** modelgewicht (laagste val-loss, via `copy.deepcopy(state_dict())`) teruggezet — niet het

laatste.

6. Als `val_split=0`, wordt er getraind zonder validatie/early stopping (en is `history["val_loss"]` een lege lijst).

Retourwaarde: `{"train_loss": [...], "val_loss": [...]}` — handig voor het plotten van leercurves.

Voorspellen

```
predictions = model.predict(X_test) # alleen labels
predictions, entropy, width = model.predict(X_test, return_uncertainty=True)
mass = model.predict_mass(X_test) # ruwe massafunctie
```

`predict()` neemt de `argmax` van de **pignistic** verdeling (niet van de ruwe massa) — dit is de correcte manier om van een massafunctie naar een concrete klassenbeslissing te gaan.

2.3 ConformalPredictor — gegarandeerde dekking

Het probleem dat conformal prediction oplost

Een model dat zegt "90% zeker" is niet per se ook echt 90% van de tijd correct — modellen zijn vaak *overconfident* of *underconfident*. **Split conformal prediction** lost dit op met een wiskundig bewijsbare garantie: mits de calibratie- en testdata *exchangeable* zijn (uit dezelfde verdeling komen), heeft de voorspelde **confidence set** een **marginale dekking** $\geq$ `confidence_level` — ongeacht welk onderliggend model je gebruikt.

Dit is de reden dat `ConformalPredictor` niet gebonden is aan RSNN: elk model met een `predict_mass(X)` (RSNN-stijl) of `predict_proba(X)` (sklearn-stijl) interface werkt.

De LAC-methode (Least Ambiguous set-valued Classifier)

Dit is de methode die geïmplementeerd is (Sadinle, Lei & Wasserman, 2019):

Stap 1 — Calibreren op een aparte, niet voor training gebruikte split:

```
from r3alai.conformal import ConformalPredictor

conformal = ConformalPredictor(model, confidence_level=0.9)
qhat = conformal.calibrate(X_cal, y_cal_one_hot)
```

Voor elk calibratiesample wordt een **non-conformity score** berekend: `score = 1 - P(ware_klasse)` (met `P` de pignistic-kans). Vervolgens wordt `qhat` bepaald als het **empirische kwantiel** van deze scores, met de eindige-steekproefcorrectie uit Angelopoulos & Bates (2021):

```
q_level = min(1, ceil((n + 1) * confidence_level) / n)
qhat = kwantiel(scores, q_level)
```

Deze **+1**-correctie is precies wat de **eindige-steekproef**-garantie oplevert (in plaats van alleen een asymptotische garantie).

Stap 2 — Voorspellen op nieuwe data:

```
confidence_sets = conformal.predict(X_test)          # bool-masker (n_samples, n_classes)
sets_als_lijs = conformal.predict_sets(X_test)      # [[0, 2], [1], ...]
```

Een klasse `c` zit in de confidence set van sample `i` als $P(c \mid x_i) \geq 1 - \hat{q}$. Bij hogere `confidence_level` wordt `qhat` groter, dus de drempel $1 - \hat{q}$ lager, dus de sets **groter** (meer klassen inbegrepen) — een fundamentele afweging tussen zekerheid en informativiteit.

Stap 3 — Dekking meten:

```
coverage = conformal.get_coverage(X_test, y_test_one_hot)
```

Dit telt simpelweg hoe vaak de ware klasse daadwerkelijk in de voorspelde set zat.

2.4 **ActiveLearner** — welke samples moet je labelen?

Bij active learning kies je, in plaats van willekeurige samples te labelen, de samples waar het model **het meest onzeker** over is — dit levert doorgaans een nauwkeuriger model op met minder labels (sample-efficiëntie).

```
from r3alai.active_learning import ActiveLearner

learner = ActiveLearner(model, uncertainty_measure="entropy") # of "credal", "disagreement"
selected_samples, indices = learner.query(unlabeled_pool, n_samples=20)
```

Drie strategieën (`active_learning/learner.py`):

Strategie	Score	Wanneer gebruiken
"entropy"	Pignistic Shannon-entropie	Standaardkeuze; vangt zowel klasse-ambigüiteit als onwetendheid.
"credal"	<code>m_Theta</code> (credal-breedte)	Als je specifiek out-of-distribution / "nooit eerder gezien"-samples wilt vinden, los van klasse-ambigüiteit.
"disagreement"	<code>-(top1_kans - top2_kans)</code>	Eenvoudig, modelagnostisch alternatief — werkt goed wanneer entropy/credal verzadigen op een moeilijke pool.

`query()` sorteert de pool op aflopende onzekerheidsscore en retourneert de `top-n_samples` — zowel de samples zelf als hun **indices** (handig om de bijbehorende ground-truth labels op te zoeken voor annotatie).

2.5 YOLO-integratie (optioneel)

`pip install r3alai[yolo]` geeft toegang tot `RSNNYOLOWrapper`, die een RSNN belief-hoofd bovenop een YOLOv8-backbone plaatst:

```
from r3alai.utils import RSNNYOLOWrapper

wrapper = RSNNYOLOWrapper("yolov8n.pt", n_classes=80, freeze_backbone=True)
wrapper.fit(train_loader, epochs=10)
wrapper.calibrate_conformal(cal_loader, confidence_level=0.9)

preds, entropy, width = wrapper.predict(images, return_uncertainty=True)
confidence_sets = wrapper.predict_confidence_sets(images)
```

Hoe het werkt: `YOLOFeatureExtractor` registreert een **forward hook** op de voorlaatste laag van het YOLO-model (typisch de laatste backbone/neck-laag, vóór de detectiekop), vangt daar de feature-map op, en past er een `adaptive_avg_pool2d` op toe om een platte vector te krijgen — exact het "custom backbone"-contract dat `RSNNClassifier` verwacht. Met `freeze_backbone=True` (default) worden alleen de gewichten van de RSNN-belief-laag getraind, niet die van YOLO zelf.

Scope-opmerking: dit voegt gekalibreerde klasse-onzekerheid toe naast YOLO's eigen detecties (per-crop/per-afbeelding classificatievertrouwen); het verandert niets aan YOLO's eigen box-regressie of objectness-scores.

2.6 Bestandsstructuur

```
r3alai/
├── pyproject.toml
├── README.md
├── src/r3alai/
│   ├── errors.py                # Typed exceptions
│   ├── models/
│   │   ├── belief.py            # Massa / belief / plausibility / pignistic / rsnn_loss
│   │   ├── backbones.py         # resnet50, mobilenet_v2, efficientnet_b0
│   │   ├── rsnn.py              # RSNNClassifier
│   │   ├── conformal/
│   │   │   ├── predictor.py     # ConformalPredictor (LAC-methode)
│   │   │   └── active_learning/
│   │   │       ├── learner.py   # ActiveLearner
│   │   │       └── utils/
│   │   │           └── yolo.py  # RSNNYOLOWrapper, YOLOFeatureExtractor
```

2.7 Testen

```
cd r3alai
pip install -e ".[vision,dev]"
pytest                # 46 tests: wiskundige invarianten, conformal-formule, active learning-ranking
ruff check src tests
```

Appendix — Snelstart-cheatsheet

R3AL Quant

```
from r3al_quant import Quantizer, QuantConfig

config = Quantizer(QuantConfig(model_type="llm", bits=4, method="apot_r3al"))
result = config.quantize("meta-llama/Llama-3.2-3B", output_dir="./out")
bench = config.benchmark("meta-llama/Llama-3.2-3B", result, eval_data=["Hallo wereld"])
```

r3alai

```
from r3alai import RSNNClassifier
from r3alai.conformal import ConformalPredictor
from r3alai.active_learning import ActiveLearner

model = RSNNClassifier(backbone="resnet50", n_classes=10)
model.fit(X_train, y_train_one_hot, epochs=50)

preds, entropy, width = model.predict(X_test, return_uncertainty=True)

conformal = ConformalPredictor(model, confidence_level=0.9)
conformal.calibrate(X_cal, y_cal_one_hot)
sets = conformal.predict(X_test)

learner = ActiveLearner(model, uncertainty_measure="entropy")
samples, indices = learner.query(unlabeled_pool, n_samples=20)
```

Beide SDK's samen — één omgeving

```
pip install -e ".[all,dev]" # r3al_quant, vanuit repo-root
pip install -e "./r3alai[all,dev]" # r3alai, als sub-package
```

Einde van de handleiding. Zie de docstrings in de broncode en de afzonderlijke README's ([README.md](#), [r3alai/README.md](#)) voor de meest actuele referentie.