Community Meeting

December 5, 2025

Agenda

- KerasHub models – Divya
- Keras distillation API - Divya
- JaxLayer for TensorFlow- Fabien
- TPU test suite for Keras - Sachin
- Keras Advent of Code - Yufeng
- Community Feedback and Questions

New KerasHub models

Gemma Family

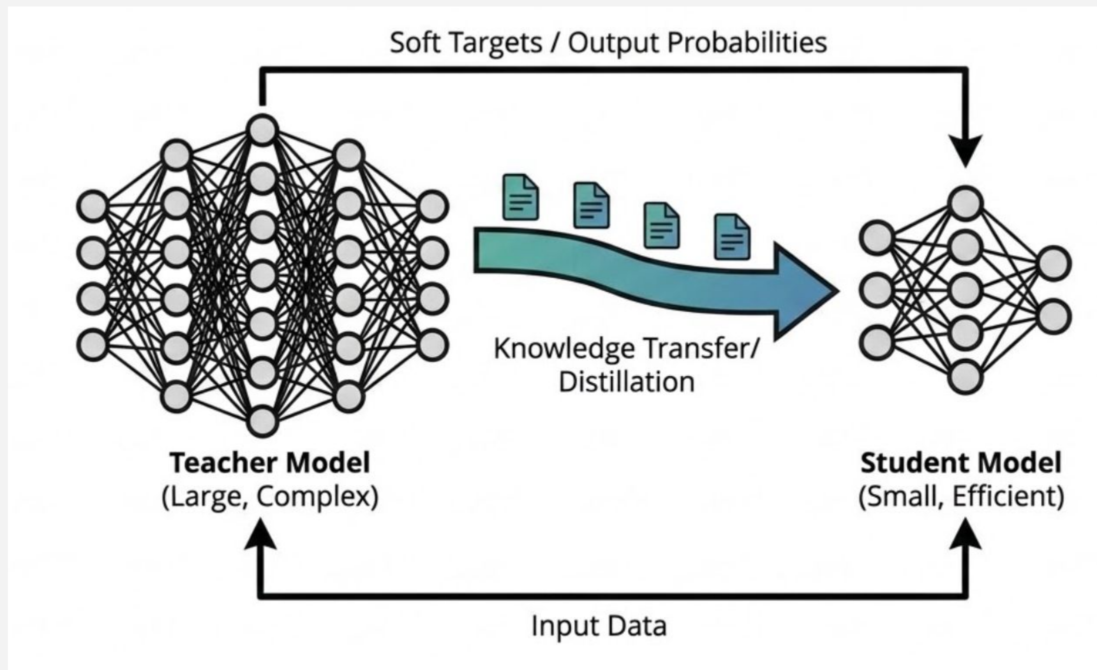
- Cell2Sentence
- MedGemma
- MedSigLip

Others

- MobileNetV5
- Qwen3 Embedding
- SmolLM3
- DinoV3

Thank you David Landup(SmolLM3), Laxma Reddy (Cell2Sentence,MedGemma, MedSigLip and Qwen 3 Embedding), Hongyu Chiu(DinoV3) and Harsha Janjani(MobileNetV5) for your impactful contributions!

Keras Distillation API



Keras.distillation API

Distiller

- A keras.Model wrapper that handles the training loop, freezing the teacher, and combining losses.

LogitsDistillation

- Matches the student's output probabilities to the teacher's (using temperature scaling).

FeatureDistillation

- Matches intermediate activations between teacher and student layers.

```python
import keras
from keras.distillation import Distiller, LogitsDistillation

# 1. Prepare your models
teacher = keras.models.load_model("large_teacher.keras")
student = create_smaller_student_model()

# 2. Wrap them in a Distiller
distiller = Distiller(
    teacher=teacher,
    student=student,
    # Use LogitsDistillation with temperature scaling
    distillation_losses=[
        LogitsDistillation(temperature=3.0)
    ],
    # Weighting: 50% from ground truth, 50% from distillation
    student_loss_weight=0.5,
)

# 3. Compile and Train (just like any Keras model)
distiller.compile(
    optimizer="adam",
    loss="sparse_categorical_crossentropy",  # Loss for ground truth
    metrics=["accuracy"]
)

# The teacher is automatically frozen during training
distiller.fit(x_train, y_train, epochs=10)

# 4. Use the trained student
inference_model = distiller.student
```

JAXLayer / FlaxLayer for TensorFlow

Keras 3 added interop building blocks:

- **keras.layers.FlaxLayer** to use **flax.linen.Module** instances in Keras
- **keras.layers.JaxLayer** to use JAX functions in Keras

Up until now, they were for the JAX backend only.

With Keras 3.13 these work with the TensorFlow backend:

- interop between JAX and TensorFlow
- migration tool
- Based on **jax2tf**

TPU test suite for Keras

Validates Multi-Backend Versatility: Ensures seamless model execution across TensorFlow, JAX, and PyTorch backends, guaranteeing hardware agnostic performance on CPUs, GPUs, and TPUs.

Enhances Framework Reliability: Significantly expands code coverage through dedicated TPU test pipelines, reducing the risk of regressions and hardware-specific failures.

Accelerates Bug Detection: Facilitates the early identification of accelerator-specific edge cases, ensuring a stable development experience for high-performance computing users.

Ensures Functional Consistency: Verifies that Keras layers and operations behave consistently across all supported hardware backends when deployed on specialized TPU infrastructure.

Helps maintain code health on TPUs: Establishes a robust testing foundation that supports the rapid evolution of AI hardware, allowing Keras to adapt quickly to new TPU generations.

Advent of Code

https://adventofcode.com/2025/day/1

Call for contributions

https://github.com/keras-team/keras-hub/issues/1836

https://github.com/keras-team/keras/issues/19519

https://github.com/keras-team/keras-rs/issues

The meeting will be rescheduled to 9 am PST.

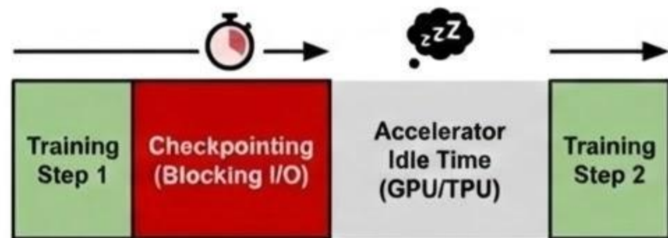
community feedback & questions

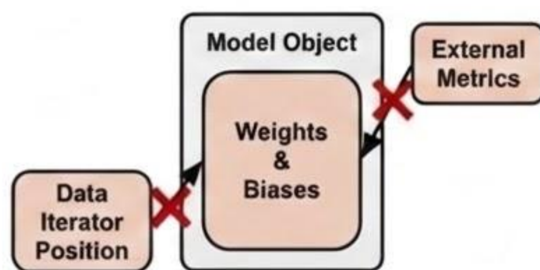
Keras 3.0 Checkpointing

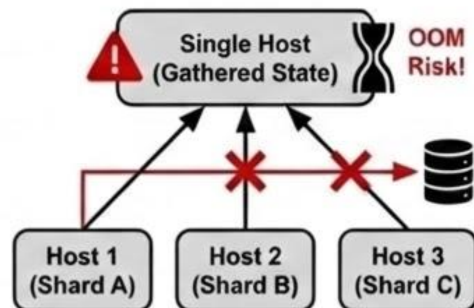
The Problem with Standard Keras Checkpointing

The standard Keras checkpointing mechanism suffers from critical limitations when applied to modern, large-scale training workloads:

- **Latency & Blocking I/O:** Checkpointing is typically **synchronous**, halting the entire training loop while writing multi-gigabyte states. This leads to **significant idle time** on expensive accelerators (GPUs/TPUs).
- **Rigid, Model-Centric Schema:** It is tightly coupled to the `Model` object. It is difficult to cleanly save **arbitrary, unrelated state** (e.g., complex data iterator positions, external metric history) alongside model weights, which is crucial for fully resumable training.
- **Lack of Native Distributed/Sharded Support:** For sharded models, it often requires state to be **gathered to a single host** (risking Out-of-Memory errors) rather than allowing all hosts to write distributed shards in parallel.





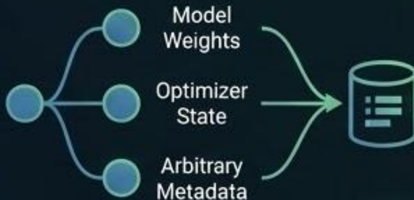


2. The Solution: Integrating Google's Orbax Library

Google's **Orbax** library is chosen as the backend for this new utility because it directly addresses the established limitations.

Native Asynchrony

Offloads writes to background threads, ensuring **minimal training pipeline stalls**.



Composite State (PyTrees)

Uses flexible dictionary structures to bundle **Model Weights + Optimizer State + Arbitrary Metadata** into a unified checkpoint.

JAX Distributed Primitives

Inherently understands **sharding** and coordinates **parallel multi-host writes**, avoiding the single-host gather bottleneck.

This integration results in a **Familiar API** (parity with `ModelCheckpoint`) with **Next-Gen Performance**.

Examples Showcase: Code in Action

Basic Best-Only Saving (Backend-Agnostic)

This example shows how to save **asynchronously**, monitor a **metric**, and manage **checkpoint retention**.

orbax_checkpoint.py

```
1  from keras.callbacks import OrbaxCheckpoint
2
3  orbax_cb = OrbaxCheckpoint(
4      directory='/tmp/orbax_ckpts',
5      monitor='val_accuracy',
6      mode='max',
7      save_best_only=True,
8      max_to_keep=5 # Retains only the 5 best performing checkpoints
9  )
10
11 model.fit(..., callbacks=[orbax_cb))
```