Keras Core

Keras for TensorFlow, JAX, and PyTorch

About Me

RED DRAGON AI

- Google Developer Expert for Machine Learning and Deep Learning (2017-)

- Deep Learning R&D :
  - Language & Dialogue systems
  - Generative Models
  - Text-to-Speech

- MeetUp Co-organiser:
  - "Machine Learning Singapore"

Martin Andrews

Google Developers Experts

K

About Red Dragon AI

RED DRAGON AI

- Founded 2017
- Google Partner
- Consulting, Prototyping & Building
- Research - NeurIPS, EMNLP, COLING, NAACL
- Interactive Digital Personas

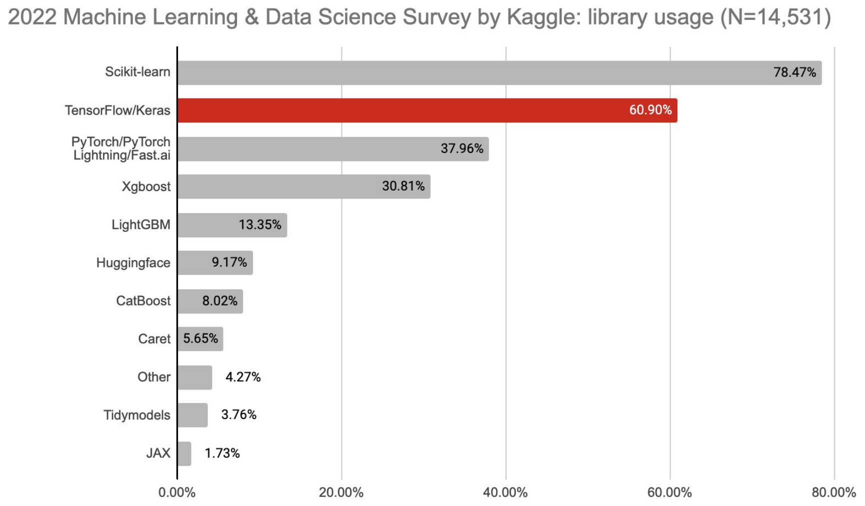
Why Keras?

Why do we need a framework?

- Neural Networks consist of Layers
  - Need a library of different layer types
  - Need a way of connecting them together
- But also need lots of other machinery:
  - Handle training phase, including metrics and visualisation
  - Interface with Accelerators (GPUs, TPUs)
  - Data loaders that move the data efficiently
  - Handle inference (production) phase
- Two main camps : Google (TensorFlow, JAX, Keras) & PyTorch
  - The 'race' is becoming more interesting...

Why Keras?

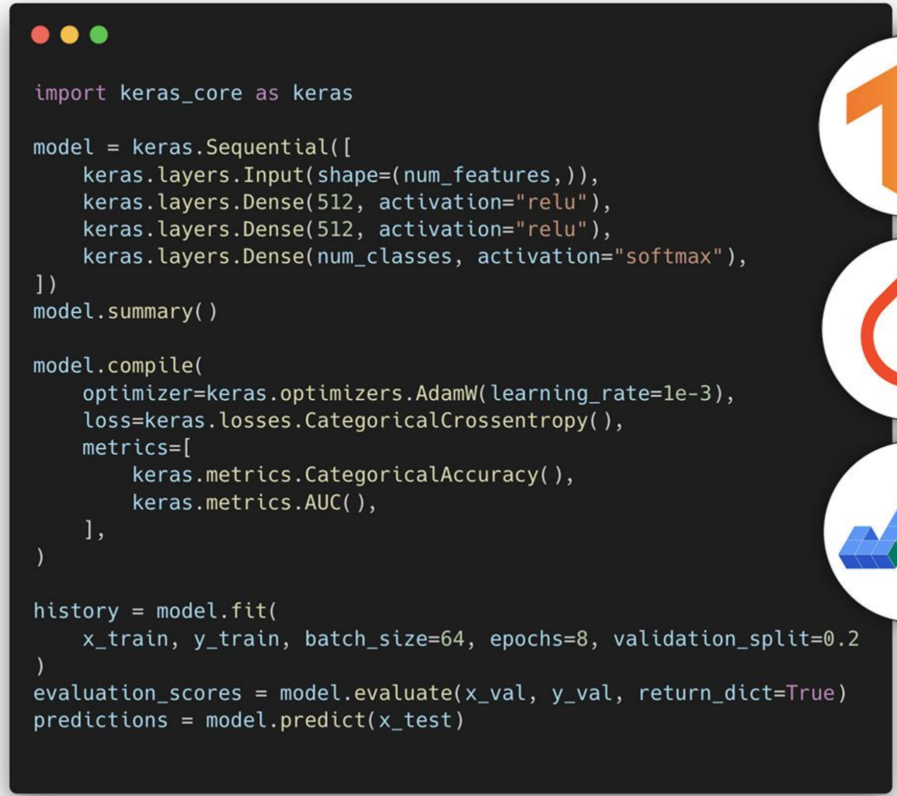
- Great developer experience
  - Consistent and simple APIs
  - Used by 2.5+ million developers
- Large ecosystem
  - KerasNLP, KerasCV, KerasTuner
  - TensorFlow Recommenders
  - etc
- Easy to turn models ↣ products
  - Can deploy across a greater range of platforms than other deep learning frameworks
  - TF Serving, tf.js, TFlite



What's new in Keras Core?

Multi-backend Keras is back

- Full rewrite of Keras
    - Now only 45k loc instead of 135k
- Support for TensorFlow, JAX, PyTorch, NumPy backends
    - NumPy backend is inference-only
- Drop-in replacement for **`tf.keras`** when using TensorFlow backend
    - Just change your imports!



```python
import keras_core as keras

model = keras.Sequential([
    keras.layers.Input(shape=(num_features,)),
    keras.layers.Dense(512, activation="relu"),
    keras.layers.Dense(512, activation="relu"),
    keras.layers.Dense(num_classes, activation="softmax"),
])
model.summary()

model.compile(
    optimizer=keras.optimizers.AdamW(learning_rate=1e-3),
    loss=keras.losses.CategoricalCrossentropy(),
    metrics=[
        keras.metrics.CategoricalAccuracy(),
        keras.metrics.AUC(),
    ],
)

history = model.fit(
    x_train, y_train, batch_size=64, epochs=8, validation_split=0.2
)
evaluation_scores = model.evaluate(x_val, y_val, return_dict=True)
predictions = model.predict(x_test)
```

```
$ python example.py
Using TensorFlow backend
```

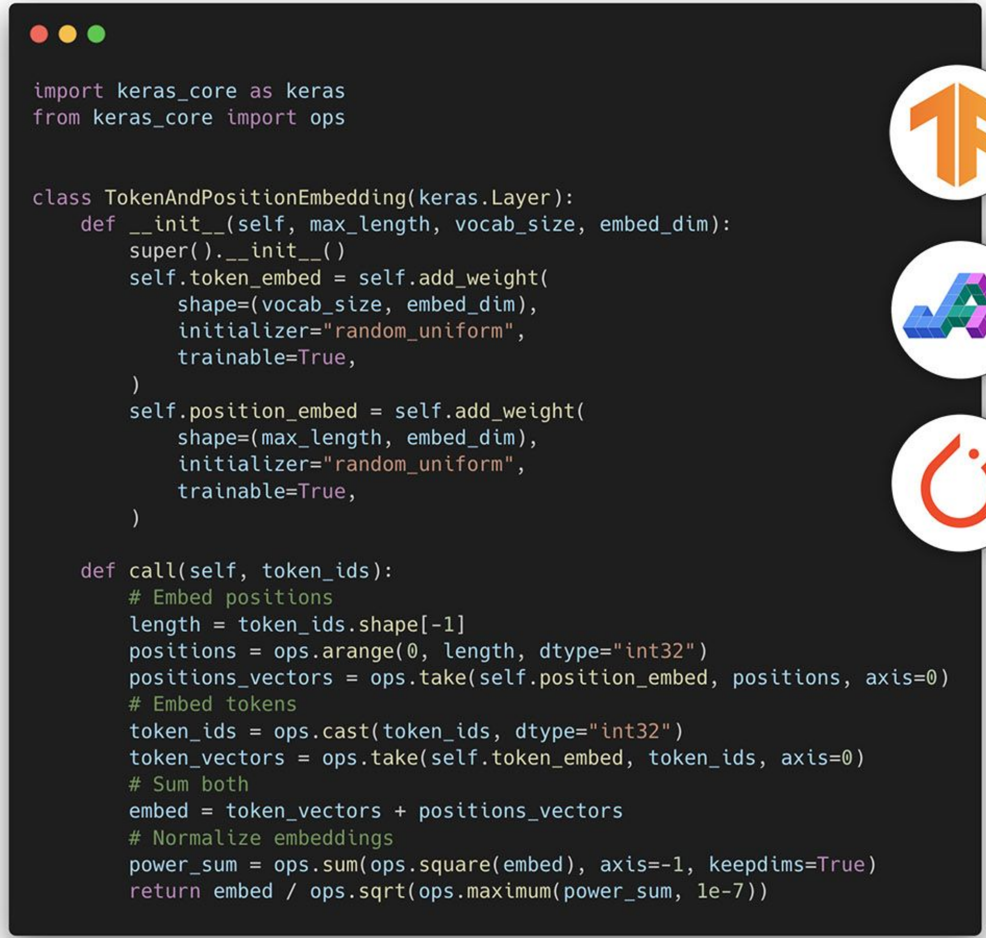
```
$ python example.py
Using PyTorch backend
```

```
$ python example.py
Using JAX backend
```

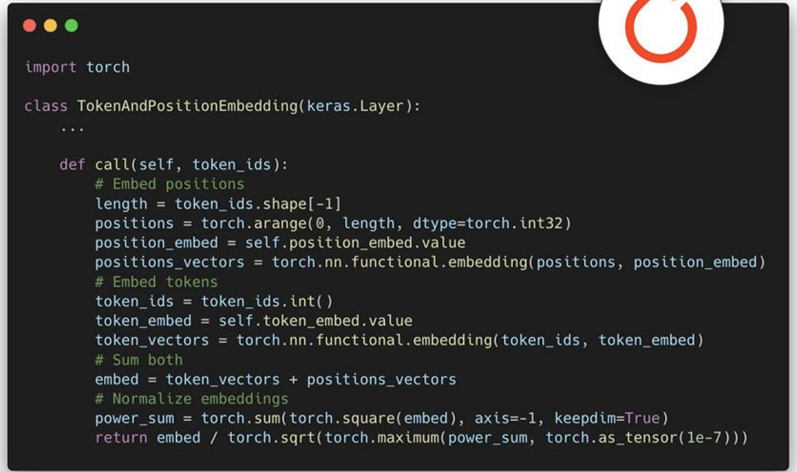
Develop cross-framework components with `keras.ops`

- Includes the **NumPy API** – same functions, same arguments.
  - `ops.matmul`, `ops.sum`, `ops.stack`, `ops.einsum`, etc.
- Plus neural network-specific functions absent from NumPy
  - `ops.softmax`, `ops.binary_crossentropy`, `ops.conv`, etc.
- Models / layers / losses / metrics / optimizers written with Keras APIs **work the same with any framework**
  - They can even be used outside of Keras workflows!

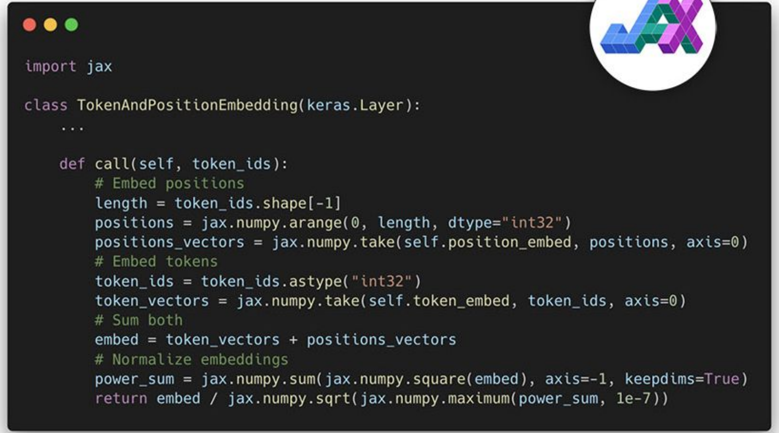
Develop
custom components
that work with
any framework
using `keras.ops`
(which includes the
NumPy API)
...



```python
import keras_core as keras
from keras_core import ops


class TokenAndPositionEmbedding(keras.Layer):
    def __init__(self, max_length, vocab_size, embed_dim):
        super().__init__()
        self.token_embed = self.add_weight(
            shape=(vocab_size, embed_dim),
            initializer="random_uniform",
            trainable=True,
        )
        self.position_embed = self.add_weight(
            shape=(max_length, embed_dim),
            initializer="random_uniform",
            trainable=True,
        )

    def call(self, token_ids):
        # Embed positions
        length = token_ids.shape[-1]
        positions = ops.arange(0, length, dtype="int32")
        positions_vectors = ops.take(self.position_embed, positions, axis=0)
        # Embed tokens
        token_ids = ops.cast(token_ids, dtype="int32")
        token_vectors = ops.take(self.token_embed, token_ids, axis=0)
        # Sum both
        embed = token_vectors + positions_vectors
        # Normalize embeddings
        power_sum = ops.sum(ops.square(embed), axis=-1, keepdims=True)
        return embed / ops.sqrt(ops.maximum(power_sum, 1e-7))
```

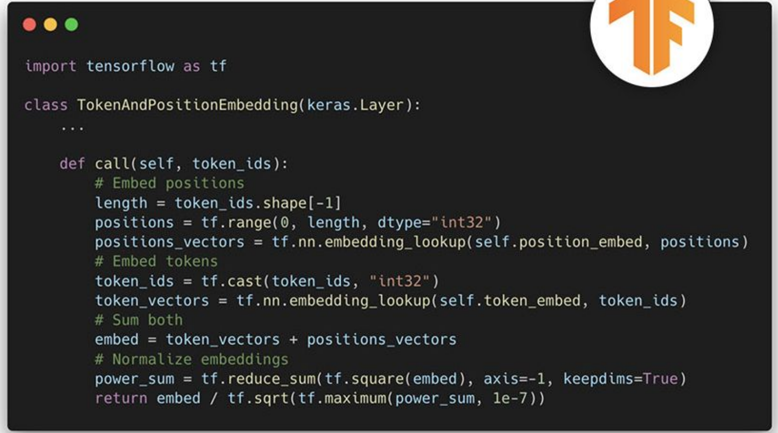


```python
import torch

class TokenAndPositionEmbedding(keras.Layer):
    ...

    def call(self, token_ids):
        # Embed positions
        length = token_ids.shape[-1]
        positions = torch.arange(0, length, dtype=torch.int32)
        position_embed = self.position_embed.value
        positions_vectors = torch.nn.functional.embedding(positions, position_embed)
        # Embed tokens
        token_ids = token_ids.int()
        token_embed = self.token_embed.value
        token_vectors = torch.nn.functional.embedding(token_ids, token_embed)
        # Sum both
        embed = token_vectors + positions_vectors
        # Normalize embeddings
        power_sum = torch.sum(torch.square(embed), axis=-1, keepdim=True)
        return embed / torch.sqrt(torch.maximum(power_sum, torch.as_tensor(1e-7)))
```



```python
import jax

class TokenAndPositionEmbedding(keras.Layer):
    ...

    def call(self, token_ids):
        # Embed positions
        length = token_ids.shape[-1]
        positions = jax.numpy.arange(0, length, dtype="int32")
        positions_vectors = jax.numpy.take(self.position_embed, positions, axis=0)
        # Embed tokens
        token_ids = token_ids.astype("int32")
        token_vectors = jax.numpy.take(self.token_embed, token_ids, axis=0)
        # Sum both
        embed = token_vectors + positions_vectors
        # Normalize embeddings
        power_sum = jax.numpy.sum(jax.numpy.square(embed), axis=-1, keepdims=True)
        return embed / jax.numpy.sqrt(jax.numpy.maximum(power_sum, 1e-7))
```

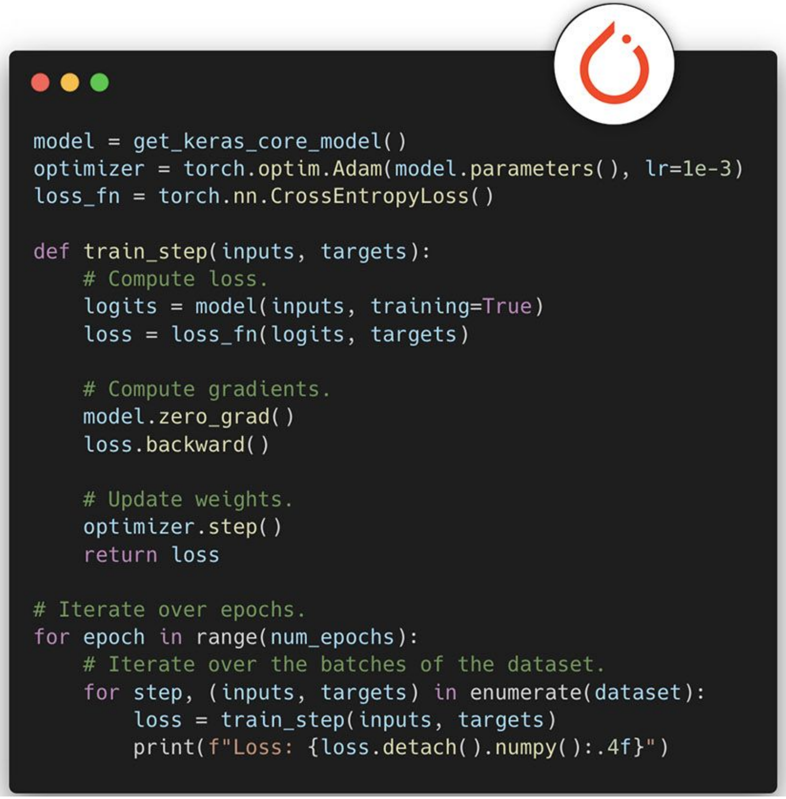
...
or use
your framework of choice
for backend-specific
components



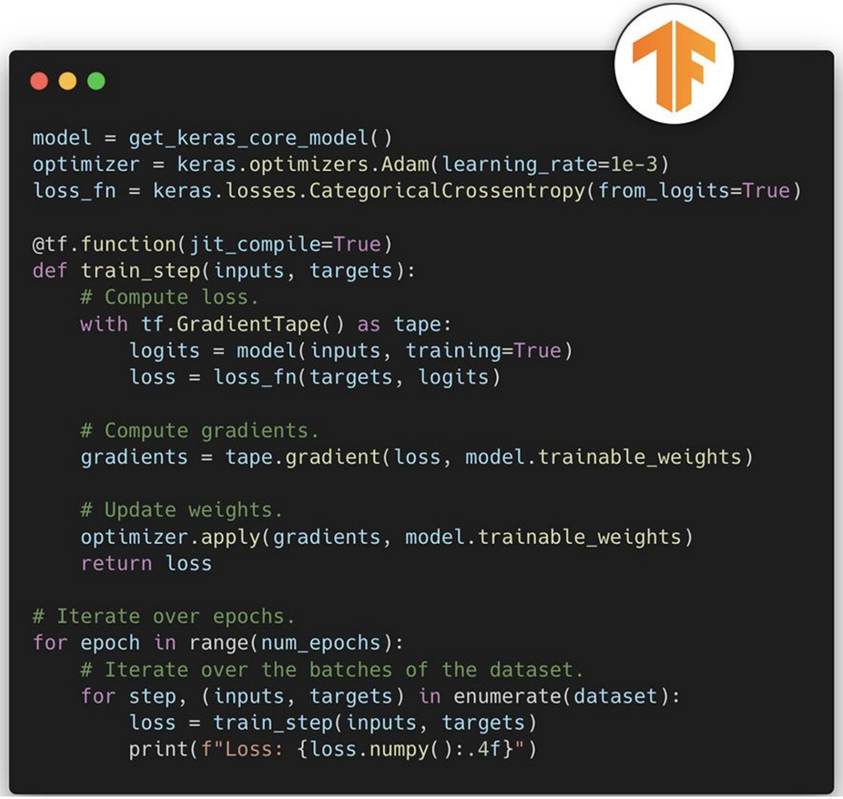
```python
import tensorflow as tf

class TokenAndPositionEmbedding(keras.Layer):
    ...

    def call(self, token_ids):
        # Embed positions
        length = token_ids.shape[-1]
        positions = tf.range(0, length, dtype="int32")
        positions_vectors = tf.nn.embedding_lookup(self.position_embed, positions)
        # Embed tokens
        token_ids = tf.cast(token_ids, "int32")
        token_vectors = tf.nn.embedding_lookup(self.token_embed, token_ids)
        # Sum both
        embed = token_vectors + positions_vectors
        # Normalize embeddings
        power_sum = tf.reduce_sum(tf.square(embed), axis=-1, keepdims=True)
        return embed / tf.sqrt(tf.maximum(power_sum, 1e-7))
```

Seamless integration with backend-native workflows

- Write a low-level JAX training loop to train a Keras model
    - e.g. optax optimizer, `jax.grad`, `jax.jit`, `jax.pmap`...
- Write a low-level TensorFlow training loop to train a Keras model
    - `e.g. tf.GradientTape` & `tf.distribute`.
- Write a low-level PyTorch training loop to train a Keras model
    - e.g. `torch.optim optimizer`, torch loss function, `torch.nn.parallel.DistributedDataParallel`
- Use a Keras layer or model as part of a `torch.nn.Module`.
    - PyTorch users can start leveraging Keras models whether or not they use Keras APIs! You can treat a Keras model just like any other PyTorch Module.
- etc.



```
model = get_keras_core_model()
optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)
loss_fn = torch.nn.CrossEntropyLoss()

def train_step(inputs, targets):
    # Compute loss.
    logits = model(inputs, training=True)
    loss = loss_fn(logits, targets)

    # Compute gradients.
    model.zero_grad()
    loss.backward()

    # Update weights.
    optimizer.step()
    return loss

# Iterate over epochs.
for epoch in range(num_epochs):
    # Iterate over the batches of the dataset.
    for step, (inputs, targets) in enumerate(dataset):
        loss = train_step(inputs, targets)
        print(f"Loss: {loss.detach().numpy():.4f}")
```



```
model = get_keras_core_model()
optimizer = keras.optimizers.Adam(learning_rate=1e-3)
loss_fn = keras.losses.CategoricalCrossentropy(from_logits=True)

@tf.function(jit_compile=True)
def train_step(inputs, targets):
    # Compute loss.
    with tf.GradientTape() as tape:
        logits = model(inputs, training=True)
        loss = loss_fn(targets, logits)

    # Compute gradients.
    gradients = tape.gradient(loss, model.trainable_weights)

    # Update weights.
    optimizer.apply(gradients, model.trainable_weights)
    return loss

# Iterate over epochs.
for epoch in range(num_epochs):
    # Iterate over the batches of the dataset.
    for step, (inputs, targets) in enumerate(dataset):
        loss = train_step(inputs, targets)
        print(f"Loss: {loss.numpy():.4f}")
```

Writing a custom training loop for a Keras model

Advantages of Keras Core

Advantages of Keras Core?

- Support for cross-framework data pipelines
- Pretrained models
- Progressive disclosure of complexity
- Introduces new stateless API for pure functional programming
- Distributed training as easy as non-distributed training

Advantages of Keras Core?

- Support for cross-framework data pipelines
  - tf.data.Dataset
  - torch.utils.data.DataLoader
  - Numpy arrays
  - pandas dataframes
  - PyDatasets
- Pretrained models
- Progressive disclosure of complexity
- Introduces new stateless API for pure functional programming
- Distributed training as easy as non-distributed training

Advantages of Keras Core?

- Support for cross-framework data pipelines
- Pretrained models
  - Out of the box integration with KerasNLP and KerasCV
- Progressive disclosure of complexity
- Introduces new stateless API for pure functional programming
- Distributed training as easy as non-distributed training

Pretrained models

Keras Core includes all Keras Applications (popular image classifiers)

KerasCV and KerasNLP work out of the box with Keras Core and all backends as of the latest releases

- YOLOv8
- Whisper
- BERT
- OPT
- etc.

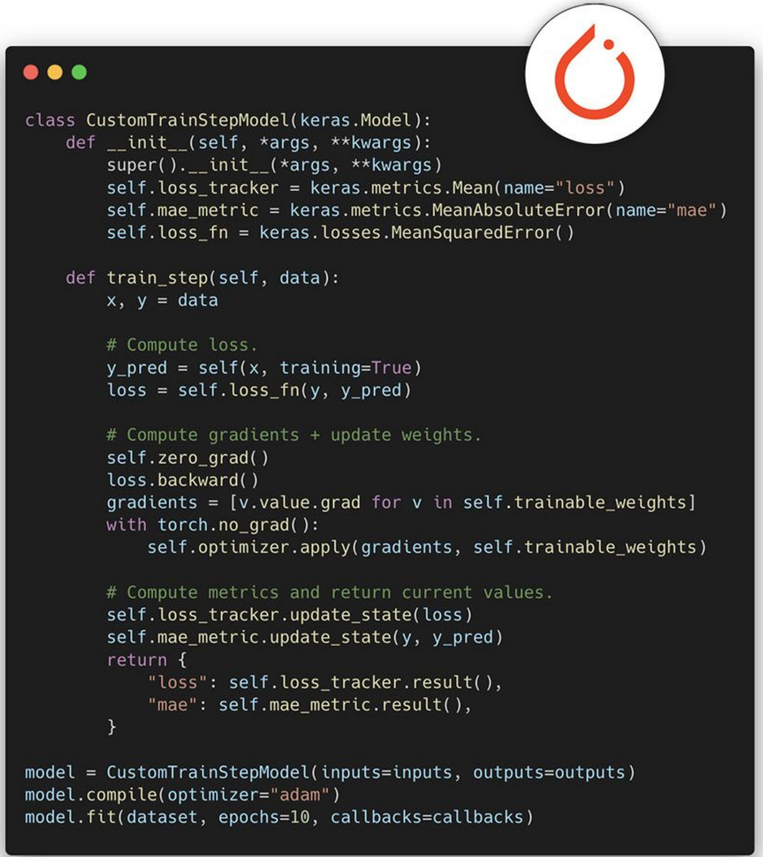
Advantages of Keras Core?

- Support for cross-framework data pipelines
- Pretrained models
- Progressive disclosure of complexity
  - Start simple
  - Customize as per your needs
  - Go from Sequential/Functional to custom train_step to custom loops in no time
- Introduces new stateless API for pure functional programming
- Distributed training as easy as non-distributed training

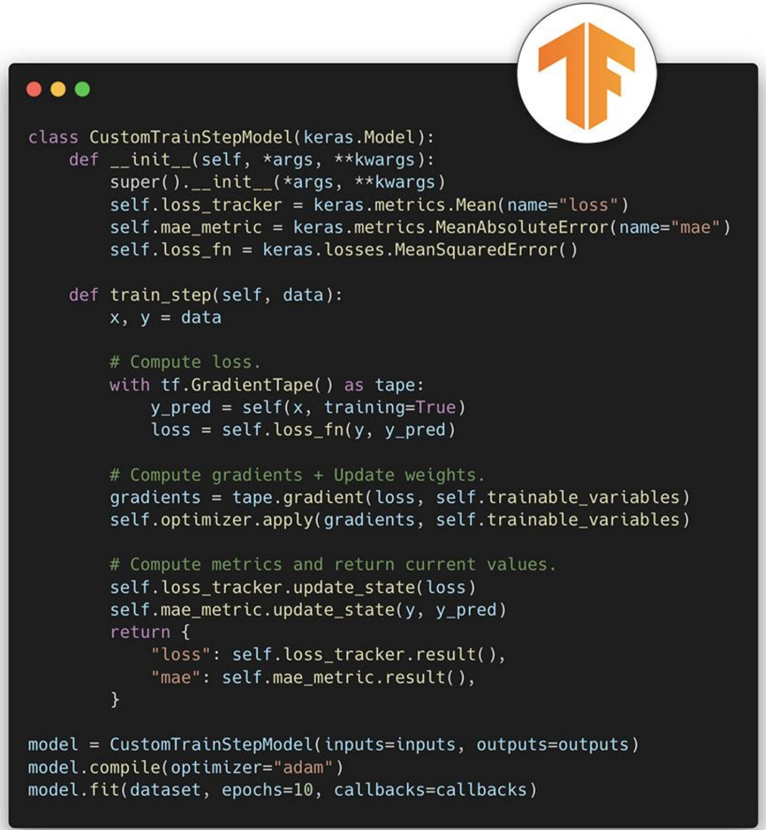
Progressive disclosure of complexity

- Start simple, then gradually gain arbitrary flexibility ...
  - ... by "opening up the box"
- Example: model training
  - fit → callbacks → custom train_step → custom training loop
- Example: model building
  - Sequential → Functional → Functional with custom layers → subclassed model

- Makes Keras suitable for students AND for Waymo engineers



```python
class CustomTrainStepModel(keras.Model):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.loss_tracker = keras.metrics.Mean(name="loss")
        self.mae_metric = keras.metrics.MeanAbsoluteError(name="mae")
        self.loss_fn = keras.losses.MeanSquaredError()

    def train_step(self, data):
        x, y = data

        # Compute loss.
        y_pred = self(x, training=True)
        loss = self.loss_fn(y, y_pred)

        # Compute gradients + update weights.
        self.zero_grad()
        loss.backward()
        gradients = [v.value.grad for v in self.trainable_weights]
        with torch.no_grad():
            self.optimizer.apply(gradients, self.trainable_weights)

        # Compute metrics and return current values.
        self.loss_tracker.update_state(loss)
        self.mae_metric.update_state(y, y_pred)
        return {
            "loss": self.loss_tracker.result(),
            "mae": self.mae_metric.result(),
        }

model = CustomTrainStepModel(inputs=inputs, outputs=outputs)
model.compile(optimizer="adam")
model.fit(dataset, epochs=10, callbacks=callbacks)
```



```python
class CustomTrainStepModel(keras.Model):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.loss_tracker = keras.metrics.Mean(name="loss")
        self.mae_metric = keras.metrics.MeanAbsoluteError(name="mae")
        self.loss_fn = keras.losses.MeanSquaredError()

    def train_step(self, data):
        x, y = data

        # Compute loss.
        with tf.GradientTape() as tape:
            y_pred = self(x, training=True)
            loss = self.loss_fn(y, y_pred)

        # Compute gradients + Update weights.
        gradients = tape.gradient(loss, self.trainable_variables)
        self.optimizer.apply(gradients, self.trainable_variables)

        # Compute metrics and return current values.
        self.loss_tracker.update_state(loss)
        self.mae_metric.update_state(y, y_pred)
        return {
            "loss": self.loss_tracker.result(),
            "mae": self.mae_metric.result(),
        }

model = CustomTrainStepModel(inputs=inputs, outputs=outputs)
model.compile(optimizer="adam")
model.fit(dataset, epochs=10, callbacks=callbacks)
```

Customizing model.fit(): PyTorch, TensorFlow

Advantages of Keras Core?

- Support for cross-framework data pipelines
- Pretrained models
- Progressive disclosure of complexity
- Introduces new stateless API for pure functional programming
- Distributed training as easy as non-distributed training

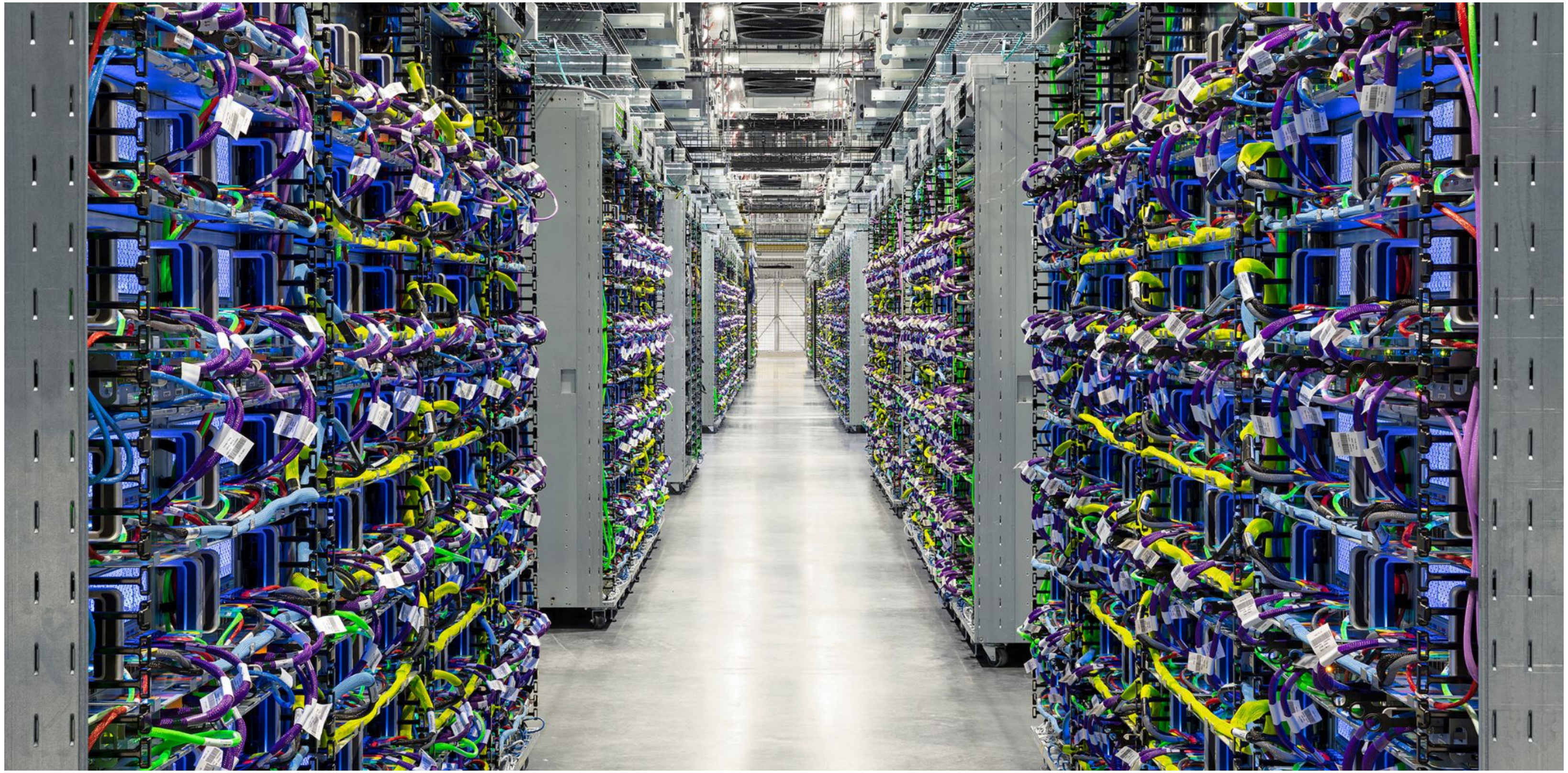
Stateless API (advanced)

- All Keras Core objects that have 'state' ...
  - ... also have a stateless API
  - So : Can use them in JAX functions
- Example :
  - outputs, updated_non_trainable_variables = layer.stateless_call(
  -     trainable_variables,
  -     non_trainable_variables,
  -     inputs,
  -)
- This interface is created automatically from stateful version
- Enables use as a high-level JAX interface

Advantages of Keras Core?

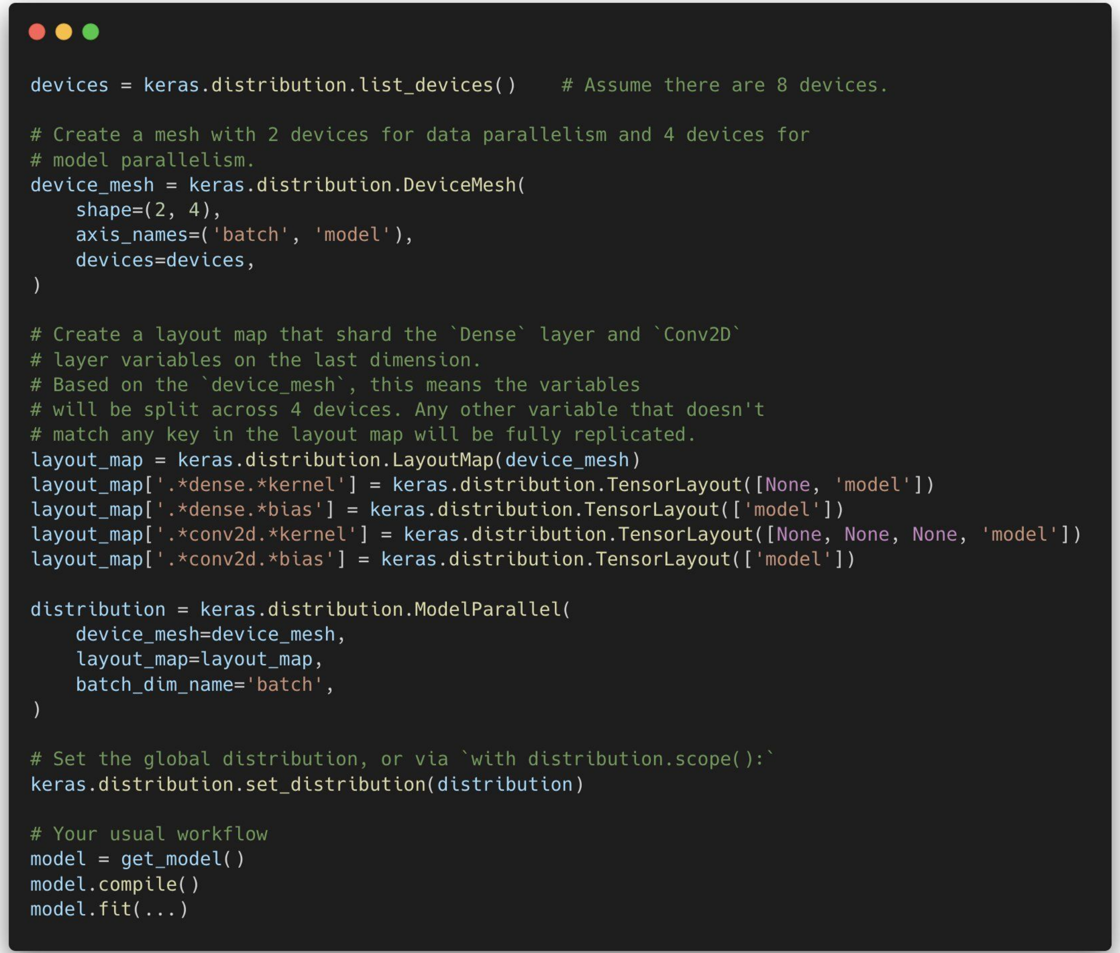
- Support for cross-framework data pipelines
- Pretrained models
- Progressive disclosure of complexity
- Introduces new stateless API for pure functional programming
- Distributed training as easy as non-distributed training
  - Ehhh... Distributed training is always painful
  - Keras Core now integrates JAX distributed training

Cloud TPU - v5 with optical interconnects



JAX distributed computation in Keras Core

- Can specify :
  - Data parallelism
  - Model parallelism
- Make most of TPU infrastructure



```python
devices = keras.distribution.list_devices()    # Assume there are 8 devices.

# Create a mesh with 2 devices for data parallelism and 4 devices for
# model parallelism.
device_mesh = keras.distribution.DeviceMesh(
    shape=(2, 4),
    axis_names=('batch', 'model'),
    devices=devices,
)

# Create a layout map that shard the `Dense` layer and `Conv2D`
# layer variables on the last dimension.
# Based on the `device_mesh`, this means the variables
# will be split across 4 devices. Any other variable that doesn't
# match any key in the layout map will be fully replicated.
layout_map = keras.distribution.LayoutMap(device_mesh)
layout_map['.*dense.*kernel'] = keras.distribution.TensorLayout([None, 'model'])
layout_map['.*dense.*bias'] = keras.distribution.TensorLayout(['model'])
layout_map['.*conv2d.*kernel'] = keras.distribution.TensorLayout([None, None, None, 'model'])
layout_map['.*conv2d.*bias'] = keras.distribution.TensorLayout(['model'])

distribution = keras.distribution.ModelParallel(
    device_mesh=device_mesh,
    layout_map=layout_map,
    batch_dim_name='batch',
)

# Set the global distribution, or via `with distribution.scope():`
keras.distribution.set_distribution(distribution)

# Your usual workflow
model = get_model()
model.compile()
model.fit(...)
```

François Chollet
@fchollet

2:37 AM · Sep 5, 2023 · **87.6K** Views

Wrapping up

Keras = future-proof stability

If you were a **Theano** user in **2016**, you had to migrate to **TF 1**...

... but if you were a Keras user on top of Theano, **you got TF 1 nearly for free**

If you were a **TF 1** user in **2019**, you had to migrate to **TF 2**...

... but if you were a Keras user on top of TF 1, **you got TF 2 nearly for free**

If you are using Keras on top of TF 2 in **2023**...

... **you get JAX and PyTorch support nearly for free**

And so on going forward (Mojo next?)

Frameworks are transient, Keras is your rock.

Why Keras Core?

- **Maximize performance**
  - Pick the backend that's the fastest for your particular model
  - Typically, PyTorch < TensorFlow < JAX (by 10-20% jumps between frameworks)
- **Maximize available ecosystem surface**
  - Export your model to TF SavedModel (TFLite, TF.js, TF Serving, TF-MOT, etc.)
  - Instantiate your model as a PyTorch Module and use it with the PyTorch ecosystem
  - Call your model as a stateless JAX function and use it with JAX transforms
- **Maximize addressable market for your OSS model releases**
  - PyTorch, TF have only 40-60% of the market each
  - Keras models are usable by **anyone** with no framework lock-in
- **Maximize data source availability**
  - Use tf.data, PyTorch DataLoader, NumPy, Pandas, etc. – with any backend

Resources

- Keras Core Announcement
- Introduction to Keras Core with Francois Chollet | PyImageSearch | LiveStream
- Keras Core developer guides
- PyImageSearch annotated example

Questions

Slides included contributions from :

Francois Chollet
Keras Founder
@fchollet

Aakash Kumar Nain
ML GDE
@A_K_Nain

Aritra Roy Gosthipaty
ML GDE
@ariG23498