Unlocking the power of JAX with Keras Core

Duong Nguyen

The Keras Philosophy

"Simple things should be simple.
Complex things should be possible."
- Francois Chollet

Source

What is Keras?

Keras is an API designed for human beings, not machines.
Keras follows best practices for reducing cognitive load:
it offers consistent & simple APIs, it minimizes the number of
user actions required for common use cases, and it provides
clear & actionable error messages.

Source

Why Keras?

- The purpose of Keras is to give an unfair advantage to any developer looking to ship Machine Learning-powered apps.
    - Keras focuses on debugging speed, code elegance & conciseness, maintainability, and deployability.
    - When you choose Keras, your codebase is smaller, more readable, easier to iterate on.
    - Your models run faster thanks to XLA compilation and Autograph optimizations

Source

How Keras?

- Necessary imports
- Data pipeline
- Model construction
- Model compilation
- Model training
- Model inference

- Necessary imports

```python
# Import the necessary packages
import tensorflow as tf
import keras
```

- Data pipeline

```
# Create the data pipeline
dataset = tf.data.Dataset(...)
ds = (
    ds
    .shuffle(...)
    .batch()
    .map()
    ...
)
```

- Model construction

```
# Construct the model
model = keras.Sequential([
    keras.layers.Dense(...),
    ...
])
```

- Model compilation

```
# Compile the model
model.compile(
    optimizer="rmsprop",
    loss="mae",
    metrics=[..],
)
```

- Model training

```
# Train the model
model.fit(
    x=...,
    y=...,
    epochs=...,
    callbacks=[...],
)
```

- Model inference

```
# Infer on the model
model.predict(...)
```

Understanding the Keras Philosophy

Simple Flexible Powerful

The Sequential() API

```
# Adding layers to the Sequential model
model = keras.Sequential()
model.add(keras.layers.Dense(...))
model.add(keras.layers.Dense(...))
```

The Functional API

```python
# Create a model using the Functional API
inputs = keras.Input(...)
x = keras.layers.Dense(...)(inputs)
outputs = keras.layers.Dense(...)(x)

model = keras.Model(inputs=inputs, outputs=outputs)
```

The core data structures of Keras are `layers` and `models`.

Source

Layer Subclassing

(the combination of state (weights) and some computation)

$$y = Wx + B$$

```python
class Linear(keras.layers.Layer):
    def __init__(self, units, input_dim, **kwargs):
        super().__init__(**kwargs)
        # INSERT THE STATE

    def call(self, inputs):
        # INSERT THE COMPUTATION
```

```python
class Linear(keras.layers.Layer):
    def __init__(self, units, input_dim, **kwargs):
        super().__init__(**kwargs)
        self.w = self.add_weight(shape=(input_dim, units), trainable=True)
        self.b = self.add_weight(shape=(units,), trainable=True)

    def call(self, inputs):
        # y = Wx + B
        return tf.matmul(inputs, self.w) + self.b
```

Model Subclassing

```python
class CustomModel(keras.Model):
    def __init__(self, **kwargs):
        super().__init__(**kwargs)
        self.custom_layer1 = CustomLayer(...)
        self.custom_layer2 = CustomLayer(...)

    def call(self, inputs):
        x = self.custom_layer1(inputs)
        x = self.custom_layer2(x)
        return x
```

```python
class CustomModel(keras.Model):
    def __init__(self, **kwargs):
        # Define the state of the model

    def call(self, inputs):
        # Forward propagation of the model
        # This step is similar to the Functional API

    def train_step(self, inputs):
        # Use tf.GradientTape to have better access
        # to the training step

    def test_step(self, inputs):
        # Same as the train step but without GradientTape,
        # grad computation, and optimization

    def predict_step(self, inputs):
        # Code to evaluate the model

    def fit(self, inputs):
        # Have finer hold of the model.fit method
        # If we call the super().fit() here we invoke the train_step()
```

Simple

- Ideal for simple models, it provides an excellent starting point for *beginners*.

Flexible

- The Functional API, a step up in flexibility, caters to complex models with non-linear topology, shared layers, or multiple inputs/outputs, striking a balance between *simplicity* and *power*.

Powerful

- Enables dynamic, pythonic model building with conditionals, loops, and other structures, ideal for research and *complex scenarios*.

“Simple things should be simple.
Complex things should be possible.”

- Francois Chollet

Source

JAX

Google I/O Extended

"...single universal framework could not work for all scenarios...

...needs of production and cutting edge research are often in conflict."

TensorFlow Blog
Bringing Machine Learning to every developer's toolbox

JAX: Just After eXecution

- Minimalistic API for high-performance machine learning research

= NumPy + Autograd +

The beauty of JAX: Composable function transformations

- Function transformation
  - Input: a numerical function
  - Return: a **new** function that computes a related quantity
  - grad, vmap, pmap, jit

PyTorch
@PyTorch

Today we'd like to highlight features from functorch, a beta PyTorch library that provides JAX-inspired function transformations like vmap. (pytorch.org/functorch/)

grad: automatic backpropagation

```python
from jax import grad

def f(x):
    return x**2 - 5*x + 1

dfdx  = grad(f)             # 2x - 5
d2fdx = grad(grad(f))       # 2
d3fdx = grad(grad(grad(f))) # 0
```

```python
print(dfdx(1.))
```

```
-3.0
```

```python
print(d2fdx(1.))
```

```
2.0
```

```python
print(d3fdx(1.))
```

```
0.0
```

vmap: automatically batching

```python
x = np.arange(5)
w = np.array([2., 3., 4.])

def convolve(x, w):
    output = []
    for i in range(len(x)-len(w)+1):
        output.append(jnp.dot(x[i:i+len(w)], w))
    return jnp.array(output)

print(convolve(x, w))
```

```
[11. 20. 29.]
```

vmap: automatically batching

```
batch = np.stack([x for _ in range(2)])

batched_convolve = jax.vmap(convolve, in_axes=(0, None))
print(batched_convolve(batch, w))
```

```
[[11. 20. 29.]
 [11. 20. 29.]]
```

pmap: automatic parallelization

```
distributed_convolve = jax.pmap(convolve, in_axes=(0, None))
print(distributed_convolve(batch, w))
```

```
[[11. 20. 29.]
 [11. 20. 29.]]
```

jit: just-in-time compilation

```
%timeit convolve(x, w).block_until_ready()
```

```
2.46 ms ± 22.9 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

```
%timeit jax.jit(convolve)(x, w).block_until_ready()
```

```
702 µs ± 16.9 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

jit: just-in-time compilation

```
%timeit batched_convolve(batch, w).block_until_ready()
```

```
10.4 ms ± 108 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

```
%timeit jax.jit(batched_convolve)(batch, w).block_until_ready()
```

```
607 µs ± 10.7 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

```python
import numpy as np

def predict(params, inputs):
    for W, b in params:
        outputs = np.dot(inputs, W) + b
        inputs = np.tanh(outputs)
    return outputs
```

```python
import numpy as np

def predict(params, inputs):
    for W, b in params:
        outputs = np.dot(inputs, W) + b
        inputs = np.tanh(outputs)
    return outputs

def mse_loss(params, batch):
    inputs, targets = batch
    preds = predict(params, inputs)
    return np.mean((preds - targets) ** 2)
```

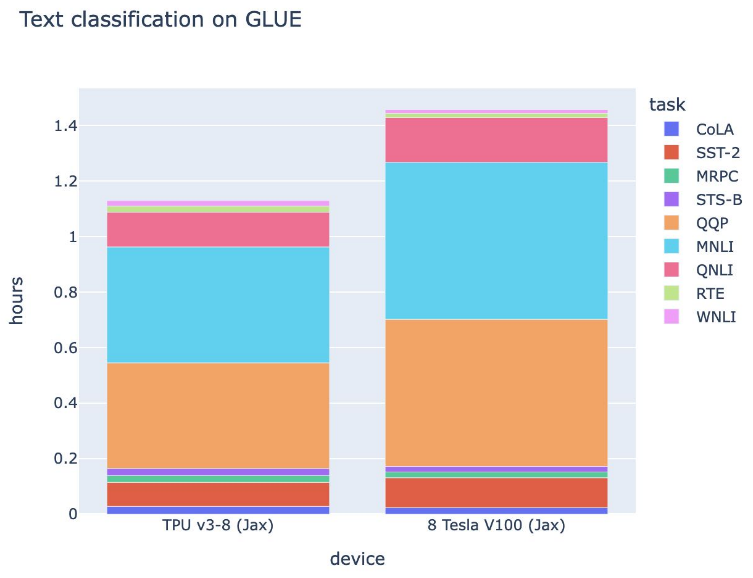
```python
import numpy as np
from jax import grad

def predict(params, inputs):
    for W, b in params:
        outputs = np.dot(inputs, W) + b
        inputs = np.tanh(outputs)
    return outputs

def mse_loss(params, batch):
    inputs, targets = batch
    preds = predict(params, inputs)
    return np.mean((preds - targets) ** 2)

grad_func = grad(mse_loss)
```

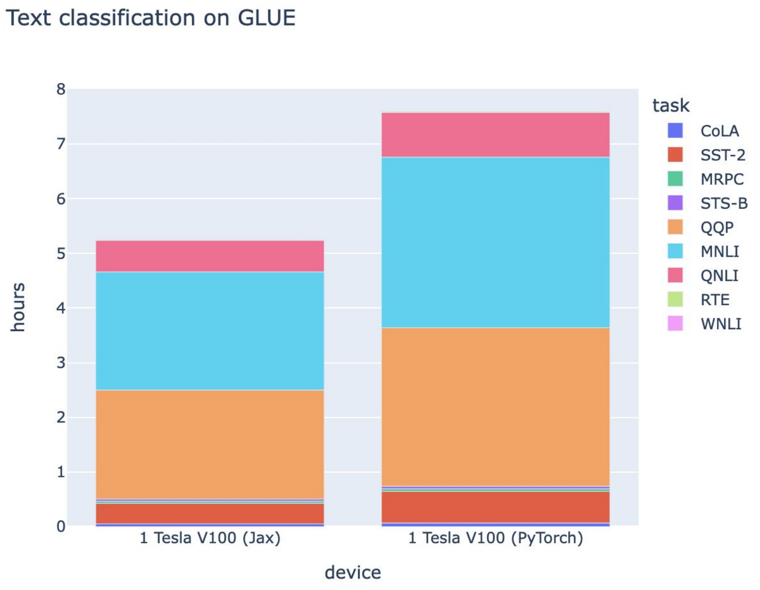
```python
import numpy as np
from jax import grad, jit

def predict(params, inputs):
    for W, b in params:
        outputs = np.dot(inputs, W) + b
        inputs = np.tanh(outputs)
    return outputs

def mse_loss(params, batch):
    inputs, targets = batch
    preds = predict(params, inputs)
    return np.mean((preds - targets) ** 2)

grad_func = jit(grad(mse_loss))
```

```python
import numpy as np
from jax import grad, jit, pmap

def predict(params, inputs):
    for W, b in params:
        outputs = np.dot(inputs, W) + b
        inputs = np.tanh(outputs)
    return outputs

def mse_loss(params, batch):
    inputs, targets = batch
    preds = predict(params, inputs)
    return np.mean((preds - targets) ** 2)

grad_func = pmap(grad(mse_loss), in_axes=(None, 0))
```

Who should care about JAX?

- Need for speed





Who should care about JAX?

- Need for flexibility
- Cool things can be efficiently done in JAX
  - Compute per-sample gradients
  - Ensemble training & inference
  - Compute (batched) Jacobians and Hessians
  - Differentiating through gradient updates (MAML)
  - ...

Keras Core

Keras for TensorFlow, JAX, and PyTorch

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.

```python
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}")
```

```python
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

K

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
- etc.
- 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

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

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**

Frameworks are transient, Keras is your rock.