Journal of Machine Learning Research 25 (2024) 1-10 Submitted 3/24; Revised 9/24; Published 11/24

# KerasCV and KerasNLP: Multi-framework Models

Lead Authors
**Matthew Watson, Divyashree Shivakumar Sreepathihalli, François Chollet**
**Martin Görner, Kiranbir Sodhia, Ramesh Sampath, Tirth Patel**
**Haifeng Jin, Neel Kovelamudi, Gabriel Rasskin, Samaneh Saadat**
**Luke Wood, Chen Qian, Jonathan Bischof, Ian Stenbit**

{mattdangerw, divyasreepat, fchollet, mgorner, ksodhia}@google.com
{rameshsampath, tirthp, haifengj, nkovela, grasskin, ssaadat}@google.com
{lukewoodcs, qianchen94era, jbischof1}@gmail.com, ian@stenbit.com

*Keras Team, Google, USA*

Community Contributors
**Abheesht Sharma, Anshuman Mishra**

**Editor:** Joaquin Vanschoren

## Abstract

We present the Keras domain packages KerasCV and KerasNLP, extensions of the Keras API for Computer Vision and Natural Language Processing workflows, capable of running on either JAX, TensorFlow, or PyTorch. These domain packages are designed to enable fast experimentation, with a focus on ease-of-use and performance. We adopt a modular, layered design: at the library's lowest level of abstraction, we provide building blocks for creating models and data preprocessing pipelines, and at the library's highest level of abstraction, we provide pretrained "task" models for popular architectures such as Stable Diffusion, YOLOv8, GPT2, BERT, Mistral, CLIP, Gemma, T5, etc. Task models have built-in preprocessing, pretrained weights, and can be fine-tuned on raw inputs. To enable efficient training, we support XLA compilation for all models, and run all preprocessing via a compiled graph of TensorFlow operations using the `tf.data` API. The libraries are fully open-source (Apache 2.0 license) and available on GitHub.

**Keywords:** KerasCV, KerasNLP, Keras multi-backend, Deep learning, Generative AI

## 1. Introduction

Keras (Chollet et al., 2015) is among the most widely used tools for machine learning today[1]. The Keras library acts as a high-level abstraction for machine learning models and layers, and seeks to be accessible to a broad group of machine learning researchers and practitioners by focusing on rapid experimentation and progressive disclosure of complexity.

Notably, recent developments in Computer Vision (CV) and Natural Language Processing (NLP) have created new challenges for practitioners. The most obvious is the shift towards larger and larger models trained on self-supervised tasks. Pretraining a state of

1. `https://survey.stackoverflow.co/2022/`

the art model is now cost-prohibitive for many researchers and practitioners, in particular in NLP. Access to open-source model architectures with pretrained weights is imperative in a large amount of CV and NLP.

Additionally, pairing efficient preprocessing and metrics computation for modern models has become more difficult, with a proliferation of disparate techniques, backends, and licenses. An ML researcher or practitioner today must select among a range of auto differentiation frameworks such as JAX, TensorFlow, and PyTorch, and even within each framework, they are often forced to stay within a specific modeling library for cross-compatibility of components. Further, improving the train-time performance of models on NLP problems presents additional hurdles. The XLA compiler (Sabne, 2020) offers dramatic speedups for many model architectures, but adds complex restrictions on the shape and flow of tensor operations. The TensorFlow-based `tf.data` (Murray et al., 2021) and `tf.text` APIs provide a scalable, dynamic, and multi-process approach for preprocessing, but many common text operations do not easily compile to a TensorFlow graph.

Aiming to reduce these framework barriers for both practitioners and researchers, we present KerasCV and KerasNLP, extensions of the Keras API for CV and NLP workflows. These packages expand upon the modular approach of Keras, adding pretrained backbone models, easy-to-use domain-specific losses and metrics, and out-of-the-box support for XLA (Sabne, 2020) compilation and data and model parallelism. Because these domain packages are written on top of Keras 3, all of their modeling components natively support JAX (Bradbury et al., 2018), TensorFlow (Abadi et al., 2015), and PyTorch (Paszke et al., 2019), and can be freely used in framework-native workflows that do not otherwise involve any Keras components.

## 2. The Keras Domain Packages API

We adopt a layered approach to API design. Our library has three levels of abstraction:

- **Foundational Components**: A collection of composable modules for building and training preprocessing pipelines, models, and evaluation logic. These are pure Keras 3 components which can be used outside of the Keras Domain Packages ecosystem.
- **Pretrained Backbones**: We extend the common CV concept of a *backbone*, and use it as a general term for a pretrained model without a task specific head. We provide a collection of pretrained model backbones for fine-tuning. For NLP models, matching tokenizers can be created alongside backbones.
- **Task Models**: A collection of end-to-end models specialized for a specific task, e.g. text generation in NLP or object detection in CV. These task models combine the preprocessing and modeling modules from the lower API levels to create a unified training and inference interface that can operate directly on plain text or image input. Task models aim to allow fine-tuning with zero configuration for common use cases.

Each additional API layer is built on top of the previous one. Modules from each level can be mixed and matched in usage, for example, extending a pretrained backbone with foundational preprocessing modules to pack input sequences or perform data augmentation.

Any KerasCV and KerasNLP model can be instantiated as a PyTorch `torch.nn.Module`, a TensorFlow `tf.Module`, or as a stateless JAX function. This means that the models can be used with PyTorch ecosystem packages, with the full range of TensorFlow deployment and production tools (such as TF-Serving, TF.js and TFLite), and with JAX large-scale TPU training infrastructure.

## 3. Training, Serving, and Deployment

KerasCV and KerasNLP offer large vision and language models. State of the art models are expected to continually increase in size in the future. To address these problems, KerasCV and KerasNLP are compatible with the Keras Unified Distribution API (Qianli and others., 2023). This API enables both model parallelism and data parallelism across all Keras backends. The API maintains a clear separation between the model definition, training logic, and sharding configuration. As a result, models within KerasCV and KerasNLP can be written as if they were intended to run on a single device. Later, specific sharding configurations can be added to these models when it's time to train them.

## 4. Pretrained models on Kaggle Models

All pretrained models of KerasCV and KerasNLP are published on Kaggle Models `https://www.kaggle.com/organizations/keras/models`. Importantly, these models are also available on Kaggle competition notebooks in internet-off mode.

| | SegmentAnything | | Gemma | | BERT | | Mistral | |
|---|---|---|---|---|---|---|---|---|
| | train | predict | train | predict | train | predict | train | predict |
| Batch Size | 1 | 7 | 8 | 32 | 54 | 531 | 8 | 32 |
| Keras 2 (TF) | 386.93 | 3,187.09 | NA | NA | 841.84 | 965.21 | NA | NA |
| Keras 3 (TF) | **355.25** | 762.67 | **232.52** | 1,134.91 | **404.17** | 962.11 | **185.92** | 966.06 |
| Keras 3 (JAX) | 361.69 | **660.16** | 273.67 | **1,128.21** | 414.26 | **865.29** | 213.22 | **957.25** |
| Keras 3 (PT) | 1,388.87 | 2,973.64 | 525.15 | 7,952.67* | 1320.441 | 3869.72 | 452.12 | 10932.59* |
| Keras 3 (best) | 355.25 | 660.16 | 232.52 | 1,128.21 | 404.17 | 865.29 | 185.92 | 957.25 |

Table 1: Average time taken (in ms/step) per training or inference step across different models, namely Segment Anything (Kirillov et al., 2023), Gemma (Team et al., 2024), BERT (Devlin et al., 2019) and Mistral (Jiang et al., 2023). * LLM inference with the PyTorch backend is abnormally slow at this time because KerasNLP uses static sequence padding. This will be addressed soon.

## 5. Performance

Framework performance depends on the specific model. Keras 3 offers flexibility by letting users select the fastest framework for their task. Picking the fastest backend for a given model consistently outperforms Keras 2 as seen in Table 1. All benchmarks are done with a single NVIDIA A100 GPU with 40GB of GPU memory on a Google Cloud Compute Engine of machine type a2-highgpu-1g with 12 vCPUs and 85GB host memory.

For fair comparison, we use the same batch size across frameworks if it is the same model and task (fit or predict). However, for different models and tasks, due to their

different sizes and architectures, we use different batch sizes to avoid either running out of memory (too large) or under GPU utilization (too small). We also used the same batch size for Gemma and Mistral since they are the same model type with similar number of parameters. (see Table 1). XLA-compiled Keras models in JAX and TensorFlow exhibit no overhead compared to equivalent code written without Keras. The resulting XLA graphs are virtually identical, ensuring identical performance. However, Keras 3 with Pytorch shows lower performance because writing performant Pytorch requires heavy manual optimization on the part of the end user. Do note that Keras models running on top of the JAX or TensorFlow backends are nearly always significantly faster than the same models written in native PyTorch. The benchmarks will continue to be updated here `https://keras.io/getting_started/benchmarks/`.

## 6. Related Work

A library with clear parallels to KerasNLP and KerasCV is the HuggingFace Transformers library (Wolf et al., 2020). Both libraries offer access to pretrained model checkpoints for a number of widely-used transformer architectures.

The Transformers library is built with a "repeat yourself" approach. KerasNLP, in contrast, is built with a layered approach, with an explicit goal of allowing the re-implementation of any large language model in a relatively small amount of code. We believe there are strengths and weaknesses to both of these approaches.

## 7. Future Work

Future efforts are directed towards consolidating KerasNLP and KerasCV into a unified repository, KerasHub (Watson et al., 2024), to simplify the development and maintenance of multimodal models. This initiative is already underway, and KerasHub has now been officially released. Moving forward, we will expand the repository by integrating additional multimodal models and enhancing fine-tuning capabilities.

## 8. Conclusions

KerasCV and KerasNLP are new toolboxes offering both modular components for rapid prototyping of new models, as well as standard pretrained backbones and task models for many computer vision and natural language processing workflows. They can be leveraged by users of either JAX (Bradbury et al., 2018), TensorFlow (Abadi et al., 2015), or PyTorch (Paszke et al., 2019). Thanks to backend optionality and XLA (Sabne, 2020) compilation, KerasCV and KerasNLP deliver state-of-the-art training and inference performance. KerasCV and KerasNLP offer extensive user guides, available at keras.io.

## Acknowledgments

We thank all contributors to Keras (Chollet et al., 2015), KerasCV, KerasNLP, TensorFlow (Abadi et al., 2015), TensorFlow Text, TensorFlow Data, and the XLA (Sabne, 2020) compiler, all of which are crucial to the functionality provided in KerasCV and KerasNLP.

## Appendix A. Preprocessing Layers

KerasNLP offers a comprehensive suite of preprocessing layers that enable users to build state-of-the-art, industry-grade data augmentation pipelines for tasks such as text classification, text generation, language translation, and text feature extraction. These include tokenizers, samplers, and other data preprocessing layers. Below is an example demonstrating how to use KerasNLP's preprocessing layers.

```
# Apply RandomSwap preprocessing layer on input data
augmenter = keras_nlp.layers.RandomSwap(rate=0.4, seed=42)
augmented_data = augmenter(input_data)

# Example to demonstrate how to use a tokenizer
vocab = ["[UNK]", "the", "qu", "##ick", "br", "##own", "fox", "."]
inputs = ["The quick brown fox."]
tokenizer = keras_nlp.tokenizers.WordPieceTokenizer(
    vocabulary=vocab,
    sequence_length=10,
    lowercase=True,
)
tokenized_outputs = tokenizer(inputs)
```

KerasCV provides a comprehensive suite of preprocessing layers that empower users to construct state-of-the-art, industry-grade data augmentation pipelines for image classification, object detection, image segmentation and image generation tasks. These layers implement a wide range of commonly used data augmentation techniques, enabling users to effortlessly enhance the robustness and generalizability of their models. By using preprocessing layers, users can ensure that their models are trained on data that is representative of the data that they will encounter at inference time. KerasCV offers 38 data augmentation layers. These layers implement a wide range of commonly used data augmentation techniques, enabling users to effortlessly manipulate image data in a variety of ways and handle all types of labels out-of-the-box (e.g. class labels, box labels, mask labels).

TF Data is a TensorFlow API for building input pipelines. Input pipelines are responsible for loading data from disk, preprocessing it, and batching it. TF Data provides a number of features that make it a powerful tool for preprocessing data for machine learning, such as:

- **Dataset APIs**: for loading data from a variety of sources, such as CSV files, `TFRecords`, and images.
- **Preprocessing functions**: for performing common preprocessing tasks, such as decoding images, resizing images, and normalizing images.
- **Batching functions**: for grouping data into batches.
- **Prefetching and caching**: for improving the performance of input pipelines.

```
# Applies grayscale preprocessing to input images.
(images, labels), _ = keras.datasets.cifar10.load_data()
to_grayscale = keras_cv.layers.preprocessing.Grayscale()
augmented_images = to_grayscale(images)
```

## Appendix B. Preset API

The presets API provides a convenient way to create state-of-the-art CV and NLP models. Presets are pre-configured models that have been trained on a specific dataset and can be used for a specific task.

To use the presets API, one simply needs to import the `keras_cv.models` or `keras_nlp.models` module and then call the `from_preset()` method on the desired model class. The presets API provides a number of advantages over creating models from scratch. First, presets are pretrained on a large dataset, which means that they can achieve high accuracy on a variety of tasks. Second, presets are pre-configured, which means that users do not need to worry about setting hyperparameters. Third, presets are easy to use, which means that users can get started with them quickly.

```
# Load architecture and weights from preset
model = keras_nlp.models.RetinaNet.from_preset(
    "resnet50_imagenet",
)

# Load randomly initialized model from the preset architecture
model = keras_cv.models.RetinaNet.from_preset(
    "resnet50_imagenet",
    load_weights=False,
)
```

## Appendix C. Backbone API

Both KerasCV and KerasNLP offer a Backbone API. Backbones can be thought of as the central architecture of a model, without the final output layer. This allows users to leverage powerful pretrained backbones (often trained on vast datasets) as the starting point for their own customized models. The pretrained backbones within KerasCV and KerasNLP offer more than just a starting point, they are also finetunable. Several examples of how to do this can be seen on the Keras.io webpage (Chollet et al., 2015).

```
# Load backbone and weights from preset
model = keras_cv.models.ResNetBackbone.from_preset(
    "resnet50_imagenet",
)

# Randomly initialized backbone with a custom config
model = keras_cv.models.ResNetBackbone(
```

```
    stackwise_filters=[64, 128, 256, 512],
    stackwise_blocks=[2, 2, 2, 2],
    stackwise_strides=[1, 2, 2, 2],
    include_rescaling=False,
)
```

## Appendix D. Task Models

KerasCV and KerasNLP provide a number of task models that are designed for specific tasks. These task models are built on top of the KerasCV and KerasNLP modeling layers and provide a high level of performance. These models are ready for use in applications, but can be further fine-tuned if desired. Some examples of available task models include image classification, object detection, semantic segmentation, image generation, text generation, text classification, and question answering.

**Pretrained Task Models.** Pretrained task models can be used by using presets trained on different datasets. This allows users to quickly and easily get started with deep learning without having to train a model from scratch. For example, KerasCV provides a number of presets for image classification models that have been trained on different datasets, such as ImageNet, COCO, and Pascal VOC. These presets can be used to create models that can achieve state-of-the-art results on a variety of image classification tasks. To use a pretrained task model with a preset, one simply needs to import the `keras_cv.models` or `keras_nlp.models` module and then call the `from_preset()` method on the desired model class.

**Specifying a Backbone in a Task Model.** It is possible to specify a backbone for task models. This is done by passing the backbone argument to the task class constructor. By specifying a different backbone, users can change the features that are extracted. This can be useful if one wants to improve the performance of the model on a specific task. Users can also specify their own custom backbones. To do this, one simply need to create a subclass of the `models.Backbone` class.

**Fine-Tuning a Task Model.** Fine-tuning a task model is the process of adapting a pretrained model to a specific task. This is done by training the model on a dataset of labeled data for the specific task.

```
# Example of a BERT classifier task model
features = ["The quick brown fox jumped.", "I forgot my homework."]
labels = [0, 3]

# Pretrained classifier task model.
classifier = keras_nlp.models.BertClassifier.from_preset(
    "bert_base_en",
    num_classes=4,
)
classifier.fit(x=features, y=labels, batch_size=2)
classifier.predict(x=features, batch_size=2)
```

```
# Completely customize the task model
features = ["The quick brown fox jumped.", "I forgot my homework."]
labels = [0, 3]

vocab = ["[UNK]", "[CLS]", "[SEP]", "[PAD]", "[MASK]"]
vocab += ["The", "quick", "brown", "fox", "jumped", "."]
# Custom tokenizer
tokenizer = keras_nlp.models.BertTokenizer(
    vocabulary=vocab,
)
# Custom preprocessor
preprocessor = keras_nlp.models.BertClassifierPreprocessor(
    tokenizer=tokenizer,
    sequence_length=128,
)
# Custom backbone
backbone = keras_nlp.models.BertBackbone(
    vocabulary_size=30552,
    num_layers=4,
    num_heads=4,
    hidden_dim=256,
    intermediate_dim=512,
    max_sequence_length=128,
)
# Custom task model
classifier = keras_nlp.models.BertClassifier(
    backbone=backbone,
    preprocessor=preprocessor,
    num_classes=4,
)
classifier.fit(x=features, y=labels, batch_size=2)
```

An extensive list of models offered by KerasCV and KerasNLP can be found at `https://keras.io/api/keras_cv/models/` and `https://keras.io/api/keras_nlp/models/`. Pretrained weights are available on Kaggle at `https://www.kaggle.com/organizations/keras/models`.

## Appendix E. Resources

To facilitate users in exploring the full potential of our libraries and tools through hands-on experimentation, we provide a comprehensive collection of guides and illustrative examples. KerasCV guides are accessible at `https://keras.io/guides/keras_cv/`, while KerasNLP guides can be found at `https://keras.io/guides/keras_nlp/`. For practical demonstrations, KerasCV example guides are located at `https://keras.io/examples/vision/`, and KerasNLP example guides can be found at `https://keras.io/examples/nlp/`.

Furthermore, our pretrained models are hosted on Kaggle at `https://www.kaggle.com/organizations/keras/models`. Each model is accompanied by a detailed model card that provides a comprehensive description and illustrative examples of how to effectively utilize them.

## References

Martín Abadi, Ashish Agarwal, Paul Barham, and others. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL `https://www.tensorflow.org/`. Software available from tensorflow.org.

James Bradbury, Roy Frostig, and others. JAX: Composable transformations of Python+NumPy programs, 2018. URL `http://github.com/google/jax`.

François Chollet et al. Keras, 2015. URL `https://keras.io`.

Jacob Devlin, Ming-Wei Chang, and others. BERT: Pre-training of deep bidirectional transformers for language understanding. In Jill Burstein, Christy Doran, and Thamar Solorio, editors, *Proceedings of the NAACL-HLT 2019, Volume 1 (Long and Short Papers)*, pages 4171–4186, Minneapolis, Minnesota, June 2019. Association for Computational Linguistics. doi: 10.18653/v1/N19-1423. URL `https://aclanthology.org/N19-1423`.

Albert Q. Jiang, Alexandre Sablayrolles, and others. Mistral 7b, 2023.

Alexander Kirillov, Eric Mintun, and others. Segment anything. In *Proceedings of the IEEE/CVF ICCV*, pages 4015–4026, October 2023.

Derek Gordon Murray, Jiri Simsa, Ana Klimovic, and Ihor Indyk. tf.data: A machine learning data processing framework. *CoRR*, abs/2101.12127, 2021. URL `https://arxiv.org/abs/2101.12127`.

Adam Paszke, Sam Gross, and others. PyTorch: An imperative style, high-performance deep learning library. In *NeurIPS 2019*, pages 8024–8035. Curran Associates, Inc., 2019. URL `http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf`.

Zhu Qianli and others. Keras distribution, 2023. URL `https://keras.io/api/distribution/`.

Amit Sabne. XLA : Compiling machine learning for peak performance, 2020.

Gemma Team, Thomas Mesnard, and others. Gemma: Open models based on Gemini research and technology, 2024.

Matthew Watson, François Chollet, Divyashree Sreepathihalli, Samaneh Saadat, Ramesh Sampath, Gabriel Rasskin, , Scott Zhu, Varun Singh, Luke Wood, Zhenyu Tan, Ian Stenbit, Chen Qian, Jonathan Bischof, et al. Kerashub. `https://github.com/keras-team/keras-hub`, 2024.

Thomas Wolf, Lysandre Debut, and others. Transformers: State-of-the-art natural language processing. In *EMNLP 2020*, pages 38–45, 2020.