TensorFlow 2.0 is here - what changes for R users?

TENSORFLOW/KERAS PACKAGES/RELEASES

TensorFlow 2.0 was finally released last week. As R users we have two kinds of questions. First, will my keras code still run? And second, what is it that changes? In this post, we answer both and, then, give a tour of exciting new developments in the r-tensorflow ecosystem.

AUTHOR
Sigrid Keydana

AFFILIATION
RStudio

PUBLISHED
Oct. 7, 2019

CITATION
Keydana, 2019

The wait is over – TensorFlow 2.0 (TF 2) is now officially here! What does this mean for us, users of R packages `keras` and/or `tensorflow`, which, as we know, rely on the Python TensorFlow backend?

Before we go into details and explanations, here is an *all-clear*, for the concerned user who fears their `keras` code might become obsolete (it won't).

Don't panic

- If you are using `keras` in standard ways, such as those depicted in most code examples and tutorials seen on the web, and things have been working fine for you in recent `keras` releases (>= 2.2.4.1), don't worry. Most everything should work without major changes.
- If you are using an older release of `keras` (< 2.2.4.1), syntactically things should work fine as well, but you will want to check for changes in behavior/performance.

And now for some news and background. This post aims to do three things:

- Explain the above *all-clear* statement. Is it really that simple – what exactly is going on?
- Characterize the changes brought about by TF 2, from the *point of view of the R user*.

- Characterize the changes brought about by TF 2, from the *point of view of the R user*.
- And, perhaps most interestingly: Take a look at what is going on, in the `r-tensorflow` ecosystem, around new functionality related to the advent of TF 2.

Some background

So if all still works fine (assuming standard usage), why so much ado about TF 2 in Python land?

The difference is that on the R side, for the vast majority of users, the framework you used to do deep learning was `keras`. `tensorflow` was needed just occasionally, or not at all.

Between `keras` and `tensorflow`, there was a clear separation of responsibilities: `keras` was the frontend, depending on TensorFlow as a low-level backend, just like the original Python Keras it was wrapping did. [1] . In some cases, this lead to people using the words `keras` and `tensorflow` almost synonymously: Maybe they said `tensorflow`, but the code they wrote was `keras`.

Things were different in Python land. There was original Python Keras, but TensorFlow had its own `layers` API, and there were a number of third-party high-level APIs built on TensorFlow. Keras, in contrast, was a separate library that just happened to rely on TensorFlow.

So in Python land, now we have a big change: *With TF 2, Keras (as incorporated in the TensorFlow codebase) is now the official high-level API for TensorFlow*. To bring this across has been a major point of Google's TF 2 information campaign since the early stages.

As R users, who have been focusing on `keras` all the time, we are essentially less affected. Like we said above, syntactically most everything stays the way it was. So why differentiate between different `keras` versions?

When `keras` was written, there was original Python Keras, and that was the library we were binding to. However, Google started to incorporate original Keras code into their TensorFlow codebase as a fork, to continue development independently. For a while there were two "Kerases": Original Keras and `tf.keras`. Our R `keras` offered to switch between implementations [2] , the default being original Keras.

In `keras` release 2.2.4.1, anticipating discontinuation of original Keras and wanting to get ready for TF 2, we switched to using `tf.keras` as the default. While in the beginning, the `tf.keras` fork and original Keras developed more or less in sync, the latest developments for TF 2 brought with them bigger changes in the `tf.keras` codebase, especially as regards optimizers. This is why, if you are using a `keras` version < 2.2.4.1, upgrading to TF 2 you will want to check for changes in behavior and/or performance. [3]

That's it for some background. In sum, we're happy most existing code will run just fine. But for us R users, *something* must be changing as well, right?

TF 2 in a nutshell, from an R perspective

In fact, the most evident-on-user-level change is something we wrote several posts about, more than a year ago [4]. By then, *eager execution* was a brand-new option that had to be turned on explicitly; TF 2 now makes it the default. Along with it came *custom models* (a.k.a. subclassed models, in Python land) and *custom training*, making use of `tf$GradientTape`. Let's talk about what those termini refer to, and how they are relevant to R users.

Eager Execution

In TF 1, it was all about the *graph* you built when defining your model. The graph, that was – and is – an *Abstract Syntax Tree* (AST), with operations as nodes and *tensors "flowing"* along the edges. Defining a graph and running it (on actual data) were different steps.

In contrast, with eager execution, operations are run directly when defined.

While this is a more-than-substantial change that must have required lots of resources to implement, if you use `keras` you won't notice. Just as previously, the typical `keras` workflow of `create model` -> `compile model` -> `train model` never made you think about there being two distinct phases (define and run), now again you don't have to do anything. Even though the overall execution mode is eager, Keras models are trained in graph mode, to maximize performance. We will talk about how this is done in part 3 when introducing the `tfautograph` package.

If `keras` runs in graph mode, how can you even see that eager execution is "on"? Well, in TF 1, when you ran a TensorFlow operation on a tensor [5], like so

```
library(tensorflow)
tf$math$cumprod(1:5)
```

this is what you saw:

```
Tensor("Cumprod:0", shape=(5,), dtype=int32)
```

To extract the actual values, you had to create a TensorFlow *Session* and `run` the tensor, or alternatively, use `keras::k_eval` that did this under the hood:

```
library(keras)
tf$math$cumprod(1:5) %>% k_eval()
```

```
[1]   1   2   6  24 120
```

With TF 2's execution mode defaulting to *eager*, we now automatically see the values contained in the tensor: [6]

```
tf$math$cumprod(1:5)
```

```
tf.Tensor([  1   2   6  24 120], shape=(5,), dtype=int32)
```

So that's eager execution. In our last year's *Eager*-category blog posts, it was always accompanied by custom models, so let's turn there next.

Custom models

As a `keras` user, probably you're familiar with the *sequential* and *functional* styles of building a model. Custom models allow for even greater flexibility than functional-style ones. Check out the documentation for how to create one.

Last year's series on eager execution has plenty of examples using custom models, featuring not just their flexibility, but another important aspect as well: the way they allow for modular, easily-intelligible code. [7]

Encoder-decoder scenarios are a natural match. If you have seen, or written, "old-style" code for a Generative Adversarial Network (GAN), imagine something like this instead:

```
# define the generator (simplified)
generator <-
  function(name = NULL) {
    keras_model_custom(name = name, function(self) {

      # define layers for the generator
      self$fc1 <- layer_dense(units = 7 * 7 * 64, use_bias = FALSE)
      self$batchnorm1 <- layer_batch_normalization()
      # more layers ...

      # define what should happen in the forward pass
      function(inputs, mask = NULL, training = TRUE) {
        self$fc1(inputs) %>%
          self$batchnorm1(training = training) %>%
          # call remaining layers ...
      }
    })
  }

# define the discriminator
discriminator <-
  function(name = NULL) {
    keras_model_custom(name = name, function(self) {

      self$conv1 <- layer_conv_2d(filters = 64, #...)
      self$leaky_relu1 <- layer_activation_leaky_relu()
      # more layers ...

      function(inputs, mask = NULL, training = TRUE) {
        inputs %>% self$conv1() %>%
          self$leaky_relu1() %>%
          # call remaining layers ...
      }
    })
```

```
}
```

Coded like this, picture the generator and the discriminator as agents, ready to engage in what is actually the opposite of a zero-sum game.

The game, then, can be nicely coded using *custom training*.

Custom training

Custom training, as opposed to using `keras` `fit`, allows to interleave the training of several models. Models are *called* on data, and all calls have to happen inside the context of a `GradientTape`. In eager mode, `GradientTapes` are used to keep track of operations such that during backprop, their gradients can be calculated.

The following code example shows how using `GradientTape`-style training, we can *see* our actors play against each other:

```
# zooming in on a single batch of a single epoch
with     (        tf       $        GradientTape(         )         %as%      gen_tape ,
{         with     (         tf       $        GradientTape(         )         %as%
disc_tape, {

  # first, it's the generator's call (yep pun intended)
  generated_images <-        generator(        noise    )
  # now the discriminator gives its verdict on the real images
  disc_real_output <-        discriminator(        batch    , training =         TRUE
)
  # as well as the fake ones
  disc_generated_output <-        discriminator(        generated_images, training
=         TRUE     )

  # depending on the discriminator's verdict we just got,
  # what's the generator's loss?
  gen_loss <-        generator_loss(        disc_generated_output)
  # and what's the loss for the discriminator?
  disc_loss <-        discriminator_loss(        disc_real_output, disc_generated_output
)
}        )         }        )

# now outside the tape's context compute the respective gradients
gradients_of_generator <-        gen_tape $        gradient (        gen_loss , generator
$        variables)
gradients_of_discriminator <-        disc_tape$        gradient (        disc_loss,
discriminator$        variables)

# and apply them!
generator_optimizer$        apply_gradients(
  purrr    ::       transpose(        list     (        gradients_of_generator, generator
$        variables)        )        )
```

```
discriminator_optimizer$       apply_gradients(
  purrr    ::      transpose(       list     (       gradients_of_discriminator,
discriminator$       variables)       )       )
```

Again, compare this with pre-TF 2 GAN training – it makes for a *lot* more readable code.

As an aside, last year's post series may have created the impression that with eager execution, you *have* to use custom (`GradientTape`) training instead of Keras-style `fit`. In fact, that was the case at the time those posts were written. Today, Keras-style code works just fine with eager execution.

So now with TF 2, we are in an optimal position. We *can* use custom training when we want to, but we don't have to if declarative `fit` is all we need.

That's it for a flashlight on what TF 2 means to R users. We now take a look around in the `r-tensorflow` ecosystem to see new developments – recent-past, present and future – in areas like data loading, preprocessing, and more.

New developments in the `r-tensorflow` ecosystem

These are what we'll cover:

- `tfdatasets`: Over the recent past, `tfdatasets` pipelines have become the preferred way for data loading and preprocessing.
- *feature columns* and *feature specs*: Specify your features `recipes`-style and have `keras` generate the adequate layers for them.
- Keras preprocessing layers: Keras preprocessing pipelines integrating functionality such as data augmentation (currently in planning).
- `tfhub`: Use pretrained models as `keras` layers, and/or as feature columns in a `keras` model.
- `tf_function` and `tfautograph`: Speed up training by running parts of your code in graph mode.

tfdatasets input pipelines

For 2 years now, the tfdatasets package has been available to load data for training Keras models in a streaming way.

Logically, there are three steps involved:

1. First, data has to be loaded from some place. This could be a csv file, a directory containing images, or other sources. In this recent example from Image segmentation with U-Net, information about file names was first stored into an R `tibble`, and then tensor_slices_dataset was used to create a `dataset` from it:

```
data      <-       tibble   (
  img =        list.files(       here     ::       here     (       "data-raw/train"
)       , full.names =        TRUE     )       ,
  mask =        list.files(       here     ::       here     (
"data-raw/train_masks"   )       , full.names =        TRUE     )
)

data      <-       initial_split(       data     , prop =        0.8      )

dataset   <-       training(       data     )       %>%
  tensor_slices_dataset(       )
```

2. Once we have a `dataset`, we perform any required transformations, *mapping* over the batch dimension. Continuing with the example from the U-Net post, here we use functions from the tf.image module to (1) load images according to their file type, (2) scale them to values between 0 and 1 (converting to `float32` at the same time), and (3) resize them to the desired format:

```
dataset   <-       dataset   %>%
  dataset_map(       ~       .x       %>%       list_modify(
    img =        tf       $       image    $       decode_jpeg(       tf       $
io       $       read_file(       .x       $       img      )       )       ,
    mask =        tf       $       image    $       decode_gif(       tf       $
io       $       read_file(       .x       $       mask     )       )       [
1       ,,,]       [       ,,1       ,drop=       FALSE    ]
  )       )       %>%
  dataset_map(       ~       .x       %>%       list_modify(
    img =        tf       $       image    $       convert_image_dtype(       .x
$       img      , dtype =        tf       $       float32  )       ,
    mask =        tf       $       image    $       convert_image_dtype(       .x
$       mask     , dtype =        tf       $       float32  )
  )       )       %>%
  dataset_map(       ~       .x       %>%       list_modify(
    img =        tf       $       image    $       resize   (       .x       $
img      , size =        shape    (       128      , 128      )       )       ,
    mask =        tf       $       image    $       resize   (       .x       $
mask     , size =        shape    (       128      , 128      )       )
  )       )
```

Note how once you know what these functions do, they free you of a lot of thinking (remember how in the "old" Keras approach to image preprocessing, you were doing things like dividing pixel values by 255 "by hand"?)

3. After transformation, a third conceptual step relates to item arrangement. You will often want to shuffle, and you certainly will want to batch the data:

```
 if       (       train    )       {
    dataset   <-       dataset   %>%
      dataset_shuffle(       buffer_size =        batch_size*       128      )
  }
```

```
dataset <- dataset %>% dataset_batch( batch_size)
```

Summing up, using `tfdatasets` you build a pipeline, from loading over transformations to batching, that can then be fed directly to a Keras model. From preprocessing, let's go a step further and look at a new, extremely convenient way to do feature engineering.

Feature columns and feature specs

Feature columns as such are a Python-TensorFlow feature, while feature specs are an R-only idiom modeled after the popular recipes package.

It all starts off with creating a feature spec object, using formula syntax to indicate what's predictor and what's target:

```
library ( tfdatasets)
hearts_dataset <- tensor_slices_dataset( hearts )
spec <- feature_spec( hearts_dataset, target ~ .
)
```

That specification is then refined by successive information about how we want to make use of the raw predictors. This is where feature columns come into play. Different column types exist, of which you can see a few in the following code snippet:

```
spec <- feature_spec( hearts , target ~ . )
%>%
  step_numeric_column(
    all_numeric( ) , - cp , - restecg , - exang ,
- sex , - fbs ,
    normalizer_fn = scaler_standard( )
  ) %>%
  step_categorical_column_with_vocabulary_list( thal ) %>%
  step_bucketized_column( age , boundaries = c ( 18 ,
25 , 30 , 35 , 40 , 45 , 50 , 55 , 60 ,
65 ) ) %>%
  step_indicator_column( thal ) %>%
  step_embedding_column( thal , dimension = 2 ) %>%
  step_crossed_column( c ( thal , bucketized_age) ,
hash_bucket_size = 10 ) %>%
  step_indicator_column( crossed_thal_bucketized_age)

spec %>% fit ( )
```

What happened here is that we told TensorFlow, please take all numeric columns (besides a few ones listed exprès) and scale them; take column `thal`, treat it as categorical and create an embedding for it; discretize `age` according to the given ranges; and finally, create a *crossed column* to capture interaction between `thal` and that discretized age-range column. [8]

This is nice, but when creating the model, we'll still have to define all those layers, right? (Which would be pretty cumbersome, having to figure out all the right dimensions.) Luckily, we don't have to. In sync with

pretty cumbersome, having to figure out all the right dimensions...) Luckily, we don't have to. In sync with `tfdatasets`, `keras` now provides layer_dense_features to create a layer tailor-made to accommodate the specification.

And we don't need to create separate input layers either, due to layer_input_from_dataset. Here we see both in action:

```
input <- layer_input_from_dataset(hearts %>% select(-target))

output <- input %>%
  layer_dense_features(feature_columns = dense_features(spec)) %>%
  layer_dense(units = 1, activation = "sigmoid")
```

From then on, it's just normal `keras` `compile` and `fit`. See the vignette for the complete example. There also is a post on feature columns explaining more of how this works, and illustrating the time-and-nerve-saving effect by comparing with the pre-feature-spec way of working with heterogeneous datasets.

As a last item on the topics of preprocessing and feature engineering, let's look at a promising thing to come in what we hope is the near future.

Keras preprocessing layers

Reading what we wrote above about using `tfdatasets` for building a input pipeline, and seeing how we gave an image loading example, you may have been wondering: What about data augmentation functionality available, historically, through `keras`? Like `image_data_generator`?

This functionality does not seem to fit. But a nice-looking solution is in preparation. In the Keras community, the recent RFC on preprocessing layers for Keras addresses this topic. The RFC is still under discussion, but as soon as it gets implemented in Python we'll follow up on the R side.

The idea is to provide (chainable) preprocessing layers to be used for data transformation and/or augmentation in areas such as image classification, image segmentation, object detection, text processing, and more. [9] The envisioned, in the RFC, pipeline of preprocessing layers should return a `dataset`, for compatibility with `tf.data` (our `tfdatasets`). We're definitely looking forward to having available this sort of workflow!

Let's move on to the next topic, the common denominator being convenience. But now convenience means not having to build billion-parameter models yourself!

Tensorflow Hub and the `tfhub` package

Tensorflow Hub is a library for publishing and using pretrained models. Existing models can be browsed on tfhub.dev.

As of this writing, the original Python library is still under development, so complete stability is not

guaranteed. That notwithstanding, the tfhub R package already allows for some instructive experimentation.

The traditional Keras idea of using pretrained models typically involved either (1) applying a model like *MobileNet* as a whole, including its output layer, or (2) chaining a "custom head" to its penultimate layer [10]. In contrast, the TF Hub idea is to use a pretrained model as a *module* in a larger setting.

There are two main ways to accomplish this, namely, integrating a module as a `keras` layer and using it as a feature column. The tfhub README shows the first option:

```
library ( tfhub )
library ( keras )

input <- layer_input( shape = c ( 32 , 32 , 3 ) )

output <- input %>%
  # we are using a pre-trained MobileNet model!
  layer_hub( handle =
"https://tfhub.dev/google/tf2-preview/mobilenet_v2/feature_vector/2") %>%
  layer_dense( units = 10 , activation = "softmax")

model <- keras_model( input , output )
```

While the tfhub feature columns vignette illustrates the second one:

```
spec <- dataset_train %>%
  feature_spec( AdoptionSpeed ~ . ) %>%
  step_text_embedding_column(
    Description,
    module_spec = "https://tfhub.dev/google/universal-sentence-encoder/2"
    ) %>%
  step_image_embedding_column(
    img ,
    module_spec =
"https://tfhub.dev/google/imagenet/resnet_v2_50/feature_vector/3"
  ) %>%
  step_numeric_column( Age , Fee , Quantity , normalizer_fn =
scaler_standard( ) ) %>%
  step_categorical_column_with_vocabulary_list(
    has_type ( "string") , - Description, - RescuerID, -
img_path , - PetID , - Name
  ) %>%
  step_embedding_column( Breed1 : Health , State )
```

Both usage modes illustrate the high potential of working with Hub modules. Just be cautioned that, as of today, not every model published will work with TF 2.

`tf_function`, TF autograph and the R package `tfautograph`

As explained above, the default execution mode in TF 2 is eager. For performance reasons however, in many cases it will be desirable to compile parts of your code into a graph. Calls to Keras layers, for example, are run in graph mode.

To compile a function into a graph, wrap it in a call to `tf_function`, as done e.g. in the post Modeling censored data with tfprobability:

```
run_mcmc <-       function(       kernel   )        {
  kernel    %>%       mcmc_sample_chain(
    num_results =         n_steps  ,
    num_burnin_steps =         n_burnin ,
    current_state =         tf       $       ones_like(       initial_betas)       ,
    trace_fn =         trace_fn
  )
}

# important for performance: run HMC in graph mode
run_mcmc <-       tf_function(       run_mcmc )
```

On the Python side, the `tf.autograph` module automatically translates Python control flow statements into appropriate graph operations.

Independently of `tf.autograph`, the R package tfautograph, developed by Tomasz Kalinowski, implements control flow conversion directly from R to TensorFlow. This lets you use R's `if`, `while`, `for`, `break`, and `next` when writing custom training flows. Check out the package's extensive documentation for instructive examples!

Conclusion

With that, we end our introduction of TF 2 and the new developments that surround it.

If you have been using `keras` in traditional ways, how much changes *for you* is mainly up to you: Most everything will still work, but new options exist to write more performant, more modular, more elegant code. In particular, check out `tfdatasets` pipelines for efficient data loading.

If you're an advanced user requiring non-standard setup, have a look into custom training and custom models, and consult the `tfautograph` documentation to see how the package can help.

In any case, stay tuned for upcoming posts showing some of the above-mentioned functionality in action. Thanks for reading!

Footnotes

1. Original Python Keras, and thus, R `keras`, supported additional backends: Theano and CNTK. But the default backend in R `keras` always was TensorFlow. [↩]
2. Note the terminology: in R `keras`, *implementation* referred to the Python library (Keras or TensorFlow, with its module `tf.keras`) bound to, while *backend* referred to the framework providing low-level operations, which could be one of Theano, TensorFlow and CNTK.) [↩]
3. E.g., parameters like *learning_rate* may have to be adapted. [↩]
4. See More flexible models with TensorFlow eager execution and Keras for an overview and annotated links. [↩]
5. Here the nominal input is an R vector that gets converted to a Python list by `reticulate`, and to a tensor by TensorFlow. [↩]
6. This is still a tensor though. To continue working with its values in R, we need to convert it to R using `as.numeric`, `as.matrix`, `as.array` etc. [↩]
7. For example, see Generating images with Keras and TensorFlow eager execution on GANs, Neural style transfer with eager execution and Keras on neural style transfer, or Representation learning with MMD-VAE on Variational Autoencoders. [↩]
8. `step_indicator_column` is there (twice) for technical reasons. Our post on feature columns explains. [↩]

9. As readers working in e.g. image segmentation will know, data augmentation is not as easy as just using `image_data_generator` on the input images, as analogous distortions have to be applied to the masks. [↩]

10. or block of layers [↩]

Reuse

Citation

For attribution, please cite this work as

```
Keydana (2019, Oct. 8). RStudio AI Blog: TensorFlow 2.0 is here - what changes for R users?.
Retrieved from https://blogs.rstudio.com/tensorflow/posts/2019-10-08-tf2-whatchanges/
```

BibTeX citation

```
@misc{keydana2019tf2,
  author = {Keydana, Sigrid},
  title = {RStudio AI Blog: TensorFlow 2.0 is here - what changes for R users?},
  url = {https://blogs.rstudio.com/tensorflow/posts/2019-10-08-tf2-whatchanges/},
  year = {2019}
}
```