Serialization and saving

Authors: Kathy Wu, Francois Chollet
Date created: 2020/04/28
Last modified: 2020/04/28
Description: Complete guide to saving & serializing models.

View in Colab • GitHub source

Introduction

A Keras model consists of multiple components:

- The architecture, or configuration, which specifies what layers the model contain, and how they're connected.
- A set of weights values (the "state of the model").
- An optimizer (defined by compiling the model).
- A set of losses and metrics (defined by compiling the model or calling `add_loss()` or `add_metric()`).

The Keras API makes it possible to save all of these pieces to disk at once, or to only selectively save some of them:

- Saving everything into a single archive in the TensorFlow SavedModel format (or in the older Keras H5 format). This is the standard practice.
- Saving the architecture / configuration only, typically as a JSON file.
- Saving the weights values only. This is generally used when training the model.

Let's take a look at each of these options. When would you use one or the other, and how do they work?

How to save and load a model

If you only have 10 seconds to read this guide, here's what you need to know.

Saving a Keras model:

```
model = ...  # Get model (Sequential, Functional Model, or Model subclass)
model.save('path/to/location')
```

Loading the model back:

```
from tensorflow import keras
model = keras.models.load_model('path/to/location')
```

Now, let's look at the details.

Setup

```
import numpy as np
import tensorflow as tf
from tensorflow import keras
```

Whole-model saving & loading

You can save an entire model to a single artifact. It will include:

- The model's architecture/config
- The model's weight values (which were learned during training)
- The model's compilation information (if `compile()` was called)

- The optimizer and its state, if any (this enables you to restart training where you left)

APIs

- `model.save()` or `tf.keras.models.save_model()`
- `tf.keras.models.load_model()`

There are two formats you can use to save an entire model to disk: **the TensorFlow SavedModel format**, and the older Keras **H5 format**. The recommended format is SavedModel. It is the default when you use `model.save()`.

You can switch to the H5 format by:

- Passing `save_format='h5'` to `save()`.
- Passing a filename that ends in `.h5` or `.keras` to `save()`.

SavedModel format

SavedModel is the more comprehensive save format that saves the model architecture, weights, and the traced Tensorflow subgraphs of the call functions. This enables Keras to restore both built-in layers as well as custom objects.

Example:

```
def get_model():
    # Create a simple model.
    inputs = keras.Input(shape=(32,))
    outputs = keras.layers.Dense(1)(inputs)
    model = keras.Model(inputs, outputs)
    model.compile(optimizer="adam", loss="mean_squared_error")
    return model


model = get_model()

# Train the model.
test_input = np.random.random((128, 32))
test_target = np.random.random((128, 1))
model.fit(test_input, test_target)

# Calling `save('my_model')` creates a SavedModel folder `my_model`.
model.save("my_model")

# It can be used to reconstruct the model identically.
reconstructed_model = keras.models.load_model("my_model")

# Let's check:
np.testing.assert_allclose(
    model.predict(test_input), reconstructed_model.predict(test_input)
)

# The reconstructed model is already compiled and has retained the optimizer
# state, so training can resume:
reconstructed_model.fit(test_input, test_target)
```

```
4/4 [==============================] - 1s 7ms/step - loss: 2.9296
4/4 [==============================] - 0s 3ms/step - loss: 2.7120

<keras.callbacks.History at 0x7ba64817d390>
```

What the SavedModel contains

Calling `model.save('my_model')` creates a folder named `my_model`, containing the following:

```
!ls my_model
```

```
assets  keras_metadata.pb  saved_model.pb  variables
```

The model architecture, and training configuration (including the optimizer, losses, and metrics) are stored in `saved_model.pb`. The weights are saved in the `variables/` directory.

For detailed information on the SavedModel format, see the SavedModel guide (*The SavedModel format on disk*).

How SavedModel handles custom objects

When saving the model and its layers, the SavedModel format stores the class name, **call function**, losses, and weights (and the config, if implemented). The call function defines the computation graph of the model/layer.

In the absence of the model/layer config, the call function is used to create a model that exists like the original model which can be trained, evaluated, and used for inference.

Nevertheless, it is always a good practice to define the `get_config` and `from_config` methods when writing a custom model or layer class. This allows you to easily update the computation later if needed. See the section about Custom objects for more information.

Example:

```
class CustomModel(keras.Model):
    def __init__(self, hidden_units):
        super(CustomModel, self).__init__()
        self.hidden_units = hidden_units
        self.dense_layers = [keras.layers.Dense(u) for u in hidden_units]

    def call(self, inputs):
        x = inputs
        for layer in self.dense_layers:
            x = layer(x)
        return x

    def get_config(self):
        return {"hidden_units": self.hidden_units}

    @classmethod
    def from_config(cls, config):
        return cls(**config)


model = CustomModel([16, 16, 10])
# Build the model by calling it
input_arr = tf.random.uniform((1, 5))
outputs = model(input_arr)
model.save("my_model")

# Option 1: Load with the custom_object argument.
loaded_1 = keras.models.load_model(
    "my_model", custom_objects={"CustomModel": CustomModel}
)

# Option 2: Load without the CustomModel class.

# Delete the custom-defined model class to ensure that the loader does not have
# access to it.
del CustomModel

loaded_2 = keras.models.load_model("my_model")
np.testing.assert_allclose(loaded_1(input_arr), outputs)
np.testing.assert_allclose(loaded_2(input_arr), outputs)

print("Original model:", model)
print("Model Loaded with custom objects:", loaded_1)
print("Model loaded without the custom object class:", loaded_2)
```

```
Original model: <__main__.CustomModel object at 0x7ba6480b96a0>
Model Loaded with custom objects: <__main__.CustomModel object at 0x7ba648064518>
Model loaded without the custom object class: <keras.saving.saved_model.load.CustomModel object
at 0x7ba648159940>
```

The first loaded model is loaded using the config and `CustomModel` class. The second model is loaded by dynamically creating the model class that acts like the original model.

Configuring the SavedModel

New in TensorFlow 2.4 The argument `save_traces` has been added to `model.save`, which allows you to toggle SavedModel function tracing. Functions are saved to allow the Keras to re-load custom objects without the original class definitions, so when `save_traces=False`, all custom objects must have defined `get_config`/`from_config` methods. When loading, the custom objects must be passed to the `custom_objects` argument. `save_traces=False` reduces the disk space used by the SavedModel and saving time.

Keras H5 format

Keras also supports saving a single HDF5 file containing the model's architecture, weights values, and `compile()` information. It is a light-weight alternative to SavedModel.

Example:

```
model = get_model()

# Train the model.
test_input = np.random.random((128, 32))
test_target = np.random.random((128, 1))
model.fit(test_input, test_target)

# Calling `save('my_model.h5')` creates a h5 file `my_model.h5`.
model.save("my_h5_model.h5")

# It can be used to reconstruct the model identically.
reconstructed_model = keras.models.load_model("my_h5_model.h5")

# Let's check:
np.testing.assert_allclose(
    model.predict(test_input), reconstructed_model.predict(test_input)
)

# The reconstructed model is already compiled and has retained the optimizer
# state, so training can resume:
reconstructed_model.fit(test_input, test_target)
```

```
4/4 [==============================] - 0s 4ms/step - loss: 0.2223
4/4 [==============================] - 0s 3ms/step - loss: 0.2035

<keras.callbacks.History at 0x7ba6404ca588>
```

Format Limitations

Keras SavedModel format limitations:

The tracing done by SavedModel to produce the graphs of the layer call functions allows SavedModel be more portable than H5, but it comes with drawbacks.

- Can be slower and bulkier than H5.
- Cannot serialize the ops generated from the mask argument (i.e. if a layer is called with `layer(..., mask=mask_value)`, the mask argument is not saved to SavedModel).
- Does not save the overridden `train_step()` in subclassed models.

Custom objects that use masks or have a custom training loop can still be saved and loaded from SavedModel, except they must override `get_config()`/`from_config()`, and the classes must be passed to the `custom_objects` argument when loading.

H5 limitations:

- External losses & metrics added via `model.add_loss()` & `model.add_metric()` are not saved (unlike SavedModel). If you have such losses & metrics on your model and you want to resume training, you need to add these losses back yourself after loading the model. Note that this does not apply to losses/metrics created *inside* layers via `self.add_loss()` & `self.add_metric()`. As long as the layer gets loaded, these losses & metrics are kept, since they are part of the `call` method of the layer.
- The *computation graph of custom objects* such as custom layers is not included in the saved file. At loading time, Keras will need access to the Python classes/functions of these objects in order to reconstruct the model. See Custom objects.
- Does not support preprocessing layers.

Saving the architecture

The model's configuration (or architecture) specifies what layers the model contains, and how these layers are connected*. If you have the configuration of a model, then the model can be created with a freshly initialized state for the weights and no compilation information.

*Note this only applies to models defined using the functional or Sequential apis not subclassed models.

Configuration of a Sequential model or Functional API model

These types of models are explicit graphs of layers: their configuration is always available in a structured form.

APIs

- `get_config()` and `from_config()`
- `tf.keras.models.model_to_json()` and `tf.keras.models.model_from_json()`

`get_config()` and `from_config()`

Calling `config = model.get_config()` will return a Python dict containing the configuration of the model. The same model can then be reconstructed via `Sequential.from_config(config)` (for a `Sequential` model) or `Model.from_config(config)` (for a Functional API model).

The same workflow also works for any serializable layer.

Layer example:

```
layer = keras.layers.Dense(3, activation="relu")
layer_config = layer.get_config()
new_layer = keras.layers.Dense.from_config(layer_config)
```

Sequential model example:

```
model = keras.Sequential([keras.Input((32,)), keras.layers.Dense(1)])
config = model.get_config()
new_model = keras.Sequential.from_config(config)
```

Functional model example:

```
inputs = keras.Input((32,))
outputs = keras.layers.Dense(1)(inputs)
model = keras.Model(inputs, outputs)
config = model.get_config()
new_model = keras.Model.from_config(config)
```

`to_json()` and `tf.keras.models.model_from_json()`

This is similar to `get_config` / `from_config`, except it turns the model into a JSON string, which can then be loaded without the original model class. It is also specific to models, it isn't meant for layers.

Example:

```
model = keras.Sequential([keras.Input((32,)), keras.layers.Dense(1)])
json_config = model.to_json()
new_model = keras.models.model_from_json(json_config)
```

Custom objects

Models and layers

The architecture of subclassed models and layers are defined in the methods `__init__` and `call`. They are considered Python bytecode, which cannot be serialized into a JSON-compatible config -- you could try serializing the bytecode (e.g. via `pickle`), but it's completely unsafe and means your model cannot be loaded on a different system.

In order to save/load a model with custom-defined layers, or a subclassed model, you should overwrite the `get_config` and optionally `from_config` methods. Additionally, you should register the custom object so that Keras is aware of it.

Custom functions

Custom-defined functions (e.g. activation loss or initialization) do not need a `get_config` method. The function name is sufficient for loading as long as it is registered as a custom object.

Loading the TensorFlow graph only

It's possible to load the TensorFlow graph generated by the Keras. If you do so, you won't need to provide any `custom_objects`. You can do so like this:

```
model.save("my_model")
tensorflow_graph = tf.saved_model.load("my_model")
x = np.random.uniform(size=(4, 32)).astype(np.float32)
predicted = tensorflow_graph(x).numpy()
```

Note that this method has several drawbacks: * For traceability reasons, you should always have access to the custom objects that were used. You wouldn't want to put in production a model that you cannot re-create. * The object returned by `tf.saved_model.load` isn't a Keras model. So it's not as easy to use. For example, you won't have access to `.predict()` or `.fit()`

Even if its use is discouraged, it can help you if you're in a tight spot, for example, if you lost the code of your custom objects or have issues loading the model with `tf.keras.models.load_model()`.

You can find out more in the page about [`tf.saved_model.load`](https://www.tensorflow.org/api_docs/python/tf/saved_model/load)

Defining the config methods

Specifications:

- `get_config` should return a JSON-serializable dictionary in order to be compatible with the Keras architecture- and model-saving APIs.
- `from_config(config)` (`classmethod`) should return a new layer or model object that is created from the config. The default implementation returns `cls(**config)`.

Example:

```
class CustomLayer(keras.layers.Layer):
    def __init__(self, a):
        self.var = tf.Variable(a, name="var_a")

    def call(self, inputs, training=False):
        if training:
            return inputs * self.var
        else:
            return inputs

    def get_config(self):
        return {"a": self.var.numpy()}

    # There's actually no need to define `from_config` here, since returning
    # `cls(**config)` is the default behavior.
    @classmethod
    def from_config(cls, config):
        return cls(**config)


layer = CustomLayer(5)
layer.var.assign(2)

serialized_layer = keras.layers.serialize(layer)
new_layer = keras.layers.deserialize(
    serialized_layer, custom_objects={"CustomLayer": CustomLayer}
)
```

Registering the custom object

Keras keeps a note of which class generated the config. From the example above, `tf.keras.layers.serialize` generates a serialized form of the custom layer:

```
{'class_name': 'CustomLayer', 'config': {'a': 2}}
```

Keras keeps a master list of all built-in layer, model, optimizer, and metric classes, which is used to find the correct class to call `from_config`. If the class can't be found, then an error is raised (`Value Error: Unknown layer`). There are a few ways to register custom classes to this list:

1. Setting `custom_objects` argument in the loading function. (see the example in section above "Defining the config methods")
2. `tf.keras.utils.custom_object_scope` or `tf.keras.utils.CustomObjectScope`
3. `tf.keras.utils.register_keras_serializable`

Custom layer and function example

```
class CustomLayer(keras.layers.Layer):
    def __init__(self, units=32, **kwargs):
        super(CustomLayer, self).__init__(**kwargs)
        self.units = units

    def build(self, input_shape):
        self.w = self.add_weight(
            shape=(input_shape[-1], self.units),
            initializer="random_normal",
            trainable=True,
        )
        self.b = self.add_weight(
            shape=(self.units,), initializer="random_normal", trainable=True
        )

    def call(self, inputs):
        return tf.matmul(inputs, self.w) + self.b

    def get_config(self):
        config = super(CustomLayer, self).get_config()
        config.update({"units": self.units})
        return config


def custom_activation(x):
    return tf.nn.tanh(x) ** 2


# Make a model with the CustomLayer and custom_activation
inputs = keras.Input((32,))
x = CustomLayer(32)(inputs)
outputs = keras.layers.Activation(custom_activation)(x)
model = keras.Model(inputs, outputs)

# Retrieve the config
config = model.get_config()

# At loading time, register the custom objects with a `custom_object_scope`:
custom_objects = {"CustomLayer": CustomLayer, "custom_activation": custom_activation}
with keras.utils.custom_object_scope(custom_objects):
    new_model = keras.Model.from_config(config)
```

In-memory model cloning

You can also do in-memory cloning of a model via `tf.keras.models.clone_model()`. This is equivalent to getting the config then recreating the model from its config (so it does not preserve compilation information or layer weights values).

Example:

```
with keras.utils.custom_object_scope(custom_objects):
    new_model = keras.models.clone_model(model)
```

Saving & loading only the model's weights values

You can choose to only save & load a model's weights. This can be useful if:

- You only need the model for inference: in this case you won't need to restart training, so you don't need the compilation information or optimizer state.
- You are doing transfer learning: in this case you will be training a new model reusing the state of a prior model, so you don't need the compilation information of the prior model.

APIs for in-memory weight transfer

Weights can be copied between different objects by using `get_weights` and `set_weights`:

- `tf.keras.layers.Layer.get_weights()`: Returns a list of numpy arrays.
- `tf.keras.layers.Layer.set_weights()`: Sets the model weights to the values in the `weights` argument.

Examples below.

Transfering weights from one layer to another, in memory

```
def create_layer():
    layer = keras.layers.Dense(64, activation="relu", name="dense_2")
    layer.build((None, 784))
    return layer


layer_1 = create_layer()
layer_2 = create_layer()

# Copy weights from layer 1 to layer 2
layer_2.set_weights(layer_1.get_weights())
```

Transfering weights from one model to another model with a compatible architecture, in memory

```
# Create a simple functional model
inputs = keras.Input(shape=(784,), name="digits")
x = keras.layers.Dense(64, activation="relu", name="dense_1")(inputs)
x = keras.layers.Dense(64, activation="relu", name="dense_2")(x)
outputs = keras.layers.Dense(10, name="predictions")(x)
functional_model = keras.Model(inputs=inputs, outputs=outputs, name="3_layer_mlp")

# Define a subclassed model with the same architecture
class SubclassedModel(keras.Model):
    def __init__(self, output_dim, name=None):
        super(SubclassedModel, self).__init__(name=name)
        self.output_dim = output_dim
        self.dense_1 = keras.layers.Dense(64, activation="relu", name="dense_1")
        self.dense_2 = keras.layers.Dense(64, activation="relu", name="dense_2")
        self.dense_3 = keras.layers.Dense(output_dim, name="predictions")

    def call(self, inputs):
        x = self.dense_1(inputs)
        x = self.dense_2(x)
        x = self.dense_3(x)
        return x

    def get_config(self):
        return {"output_dim": self.output_dim, "name": self.name}


subclassed_model = SubclassedModel(10)
# Call the subclassed model once to create the weights.
subclassed_model(tf.ones((1, 784)))

# Copy weights from functional_model to subclassed_model.
subclassed_model.set_weights(functional_model.get_weights())

assert len(functional_model.weights) == len(subclassed_model.weights)
for a, b in zip(functional_model.weights, subclassed_model.weights):
    np.testing.assert_allclose(a.numpy(), b.numpy())
```

The case of stateless layers

Because stateless layers do not change the order or number of weights, models can have compatible architectures even if there are extra/missing stateless layers.

```
inputs = keras.Input(shape=(784,), name="digits")
x = keras.layers.Dense(64, activation="relu", name="dense_1")(inputs)
x = keras.layers.Dense(64, activation="relu", name="dense_2")(x)
outputs = keras.layers.Dense(10, name="predictions")(x)
functional_model = keras.Model(inputs=inputs, outputs=outputs, name="3_layer_mlp")

inputs = keras.Input(shape=(784,), name="digits")
x = keras.layers.Dense(64, activation="relu", name="dense_1")(inputs)
x = keras.layers.Dense(64, activation="relu", name="dense_2")(x)

# Add a dropout layer, which does not contain any weights.
x = keras.layers.Dropout(0.5)(x)
outputs = keras.layers.Dense(10, name="predictions")(x)
functional_model_with_dropout = keras.Model(
    inputs=inputs, outputs=outputs, name="3_layer_mlp"
)

functional_model_with_dropout.set_weights(functional_model.get_weights())
```

APIs for saving weights to disk & loading them back

Weights can be saved to disk by calling `model.save_weights` in the following formats:

- TensorFlow Checkpoint
- HDF5

The default format for `model.save_weights` is TensorFlow checkpoint. There are two ways to specify the save format:

1. `save_format` argument: Set the value to `save_format="tf"` or `save_format="h5"`.
2. `path` argument: If the path ends with `.h5` or `.hdf5`, then the HDF5 format is used. Other suffixes will result in a TensorFlow checkpoint unless `save_format` is set.

There is also an option of retrieving weights as in-memory numpy arrays. Each API has its pros and cons which are detailed below.

TF Checkpoint format

Example:

```
# Runnable example
sequential_model = keras.Sequential(
    [
        keras.Input(shape=(784,), name="digits"),
        keras.layers.Dense(64, activation="relu", name="dense_1"),
        keras.layers.Dense(64, activation="relu", name="dense_2"),
        keras.layers.Dense(10, name="predictions"),
    ]
)
sequential_model.save_weights("ckpt")
load_status = sequential_model.load_weights("ckpt")

# `assert_consumed` can be used as validation that all variable values have been
# restored from the checkpoint. See [`tf.train.Checkpoint.restore`](https://www.tensorflow.org
/api_docs/python/tf/train/Checkpoint#restore) for other
# methods in the Status object.
load_status.assert_consumed()
```

```
<tensorflow.python.training.tracking.util.CheckpointLoadStatus at 0x7ba6402ffbe0>
```

Format details

The TensorFlow Checkpoint format saves and restores the weights using object attribute names. For instance, consider the `tf.keras.layers.Dense` layer. The layer contains two weights: `dense.kernel` and `dense.bias`. When the layer is saved to the `tf` format, the resulting checkpoint contains the keys `"kernel"` and `"bias"` and their corresponding weight values. For more information see "Loading mechanics" in the TF Checkpoint guide.

Note that attribute/graph edge is named after **the name used in parent object, not the name of the variable**. Consider the `CustomLayer` in the example below. The variable `CustomLayer.var` is saved with `"var"` as part of key, not `"var_a"`.

```
class CustomLayer(keras.layers.Layer):
    def __init__(self, a):
        self.var = tf.Variable(a, name="var_a")


layer = CustomLayer(5)
layer_ckpt = tf.train.Checkpoint(layer=layer).save("custom_layer")

ckpt_reader = tf.train.load_checkpoint(layer_ckpt)

ckpt_reader.get_variable_to_dtype_map()
```

```
{'save_counter/.ATTRIBUTES/VARIABLE_VALUE': tf.int64,
 '_CHECKPOINTABLE_OBJECT_GRAPH': tf.string,
 'layer/var/.ATTRIBUTES/VARIABLE_VALUE': tf.int32}
```

Transfer learning example

Essentially, as long as two models have the same architecture, they are able to share the same checkpoint.

Example:

```
inputs = keras.Input(shape=(784,), name="digits")
x = keras.layers.Dense(64, activation="relu", name="dense_1")(inputs)
x = keras.layers.Dense(64, activation="relu", name="dense_2")(x)
outputs = keras.layers.Dense(10, name="predictions")(x)
functional_model = keras.Model(inputs=inputs, outputs=outputs, name="3_layer_mlp")

# Extract a portion of the functional model defined in the Setup section.
# The following lines produce a new model that excludes the final output
# layer of the functional model.
pretrained = keras.Model(
    functional_model.inputs, functional_model.layers[-1].input, name="pretrained_model"
)
# Randomly assign "trained" weights.
for w in pretrained.weights:
    w.assign(tf.random.normal(w.shape))
pretrained.save_weights("pretrained_ckpt")
pretrained.summary()

# Assume this is a separate program where only 'pretrained_ckpt' exists.
# Create a new functional model with a different output dimension.
inputs = keras.Input(shape=(784,), name="digits")
x = keras.layers.Dense(64, activation="relu", name="dense_1")(inputs)
x = keras.layers.Dense(64, activation="relu", name="dense_2")(x)
outputs = keras.layers.Dense(5, name="predictions")(x)
model = keras.Model(inputs=inputs, outputs=outputs, name="new_model")

# Load the weights from pretrained_ckpt into model.
model.load_weights("pretrained_ckpt")

# Check that all of the pretrained weights have been loaded.
for a, b in zip(pretrained.weights, model.weights):
    np.testing.assert_allclose(a.numpy(), b.numpy())

print("\n", "-" * 50)
model.summary()

# Example 2: Sequential model
# Recreate the pretrained model, and load the saved weights.
inputs = keras.Input(shape=(784,), name="digits")
x = keras.layers.Dense(64, activation="relu", name="dense_1")(inputs)
x = keras.layers.Dense(64, activation="relu", name="dense_2")(x)
pretrained_model = keras.Model(inputs=inputs, outputs=x, name="pretrained")

# Sequential example:
model = keras.Sequential([pretrained_model, keras.layers.Dense(5, name="predictions")])
model.summary()

pretrained_model.load_weights("pretrained_ckpt")

# Warning! Calling `model.load_weights('pretrained_ckpt')` won't throw an error,
# but will *not* work as expected. If you inspect the weights, you'll see that
# none of the weights will have loaded. `pretrained_model.load_weights()` is the
# correct method to call.
```

```
Model: "pretrained_model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 digits (InputLayer)         [(None, 784)]             0

 dense_1 (Dense)             (None, 64)                50240

 dense_2 (Dense)             (None, 64)                4160

=================================================================
Total params: 54,400
Trainable params: 54,400
Non-trainable params: 0
_________________________________________________________________
```

```
 --------------------------------------------------
Model: "new_model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 digits (InputLayer)         [(None, 784)]             0

 dense_1 (Dense)             (None, 64)                50240

 dense_2 (Dense)             (None, 64)                4160

 predictions (Dense)         (None, 5)                 325

=================================================================
Total params: 54,725
Trainable params: 54,725
Non-trainable params: 0
_________________________________________________________________
Model: "sequential_3"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 pretrained (Functional)     (None, 64)                54400

 predictions (Dense)         (None, 5)                 325

=================================================================
Total params: 54,725
Trainable params: 54,725
Non-trainable params: 0
_________________________________________________________________

<tensorflow.python.training.tracking.util.CheckpointLoadStatus at 0x7ba6402d4e48>
```

It is generally recommended to stick to the same API for building models. If you switch between Sequential and Functional, or Functional and subclassed, etc., then always rebuild the pre-trained model and load the pre-trained weights to that model.

The next question is, how can weights be saved and loaded to different models if the model architectures are quite different? The solution is to use `tf.train.Checkpoint` to save and restore the exact layers/variables.

Example:

```
# Create a subclassed model that essentially uses functional_model's first
# and last layers.
# First, save the weights of functional_model's first and last dense layers.
first_dense = functional_model.layers[1]
last_dense = functional_model.layers[-1]
ckpt_path = tf.train.Checkpoint(
    dense=first_dense, kernel=last_dense.kernel, bias=last_dense.bias
).save("ckpt")

# Define the subclassed model.
class ContrivedModel(keras.Model):
    def __init__(self):
        super(ContrivedModel, self).__init__()
        self.first_dense = keras.layers.Dense(64)
        self.kernel = self.add_variable("kernel", shape=(64, 10))
        self.bias = self.add_variable("bias", shape=(10,))

    def call(self, inputs):
        x = self.first_dense(inputs)
        return tf.matmul(x, self.kernel) + self.bias


model = ContrivedModel()
# Call model on inputs to create the variables of the dense layer.
_ = model(tf.ones((1, 784)))

# Create a Checkpoint with the same structure as before, and load the weights.
tf.train.Checkpoint(
    dense=model.first_dense, kernel=model.kernel, bias=model.bias
).restore(ckpt_path).assert_consumed()
```

```
<tensorflow.python.training.tracking.util.CheckpointLoadStatus at 0x7ba6402d4b00>
```

HDF5 format

The HDF5 format contains weights grouped by layer names. The weights are lists ordered by concatenating the list of trainable weights to the list of non-trainable weights (same as `layer.weights`). Thus, a model can use a hdf5 checkpoint if it has the same layers and trainable statuses as saved in the checkpoint.

Example:

```
# Runnable example
sequential_model = keras.Sequential(
    [
        keras.Input(shape=(784,), name="digits"),
        keras.layers.Dense(64, activation="relu", name="dense_1"),
        keras.layers.Dense(64, activation="relu", name="dense_2"),
        keras.layers.Dense(10, name="predictions"),
    ]
)
sequential_model.save_weights("weights.h5")
sequential_model.load_weights("weights.h5")
```

Note that changing `layer.trainable` may result in a different `layer.weights` ordering when the model contains nested layers.

```
class NestedDenseLayer(keras.layers.Layer):
    def __init__(self, units, name=None):
        super(NestedDenseLayer, self).__init__(name=name)
        self.dense_1 = keras.layers.Dense(units, name="dense_1")
        self.dense_2 = keras.layers.Dense(units, name="dense_2")

    def call(self, inputs):
        return self.dense_2(self.dense_1(inputs))


nested_model = keras.Sequential([keras.Input((784,)), NestedDenseLayer(10, "nested")])
variable_names = [v.name for v in nested_model.weights]
print("variables: {}".format(variable_names))

print("\nChanging trainable status of one of the nested layers...")
nested_model.get_layer("nested").dense_1.trainable = False

variable_names_2 = [v.name for v in nested_model.weights]
print("\nvariables: {}".format(variable_names_2))
print("variable ordering changed:", variable_names != variable_names_2)
```

```
variables: ['nested/dense_1/kernel:0', 'nested/dense_1/bias:0', 'nested/dense_2/kernel:0',
'nested/dense_2/bias:0']
```

```
Changing trainable status of one of the nested layers...
```

```
variables: ['nested/dense_2/kernel:0', 'nested/dense_2/bias:0', 'nested/dense_1/kernel:0',
'nested/dense_1/bias:0']
variable ordering changed: True
```

Transfer learning example

When loading pretrained weights from HDF5, it is recommended to load the weights into the original checkpointed model, and then extract the desired weights/layers into a new model.

Example:

```python
def create_functional_model():
    inputs = keras.Input(shape=(784,), name="digits")
    x = keras.layers.Dense(64, activation="relu", name="dense_1")(inputs)
    x = keras.layers.Dense(64, activation="relu", name="dense_2")(x)
    outputs = keras.layers.Dense(10, name="predictions")(x)
    return keras.Model(inputs=inputs, outputs=outputs, name="3_layer_mlp")


functional_model = create_functional_model()
functional_model.save_weights("pretrained_weights.h5")

# In a separate program:
pretrained_model = create_functional_model()
pretrained_model.load_weights("pretrained_weights.h5")

# Create a new model by extracting layers from the original model:
extracted_layers = pretrained_model.layers[:-1]
extracted_layers.append(keras.layers.Dense(5, name="dense_3"))
model = keras.Sequential(extracted_layers)
model.summary()
```

```
Model: "sequential_6"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_1 (Dense)             (None, 64)                50240

 dense_2 (Dense)             (None, 64)                4160

 dense_3 (Dense)             (None, 5)                 325

=================================================================
Total params: 54,725
Trainable params: 54,725
Non-trainable params: 0
_________________________________________________________________…
```