Save and load models

Model progress can be saved during and after training. This means a model can resume where it left off and avoid long training times. Saving also means you can share your model and others can recreate your work. When publishing research models and techniques, most machine learning practitioners share:

- code to create the model, and
- the trained weights, or parameters, for the model

Sharing this data helps others understand how the model works and try it themselves with new data.

Caution: TensorFlow models are code and it is important to be careful with untrusted code. See Using TensorFlow Securely (https://github.com/tensorflow/tensorflow/blob/master/SECURITY.md) for details.

Options

There are different ways to save TensorFlow models depending on the API you're using. This guide uses tf.keras (https://www.tensorflow.org/guide/keras)—a high-level API to build and train models in TensorFlow. The new, high-level `.keras` format used in this tutorial is recommended for saving Keras objects, as it provides robust, efficient name-based saving that is often easier to debug than low-level or legacy formats. For more advanced saving or serialization workflows, especially those involving custom objects, please refer to the Save and load Keras models guide (https://www.tensorflow.org/guide/keras/save_and_serialize). For other approaches, refer to the Using the SavedModel format guide (https://www.tensorflow.org/guide/saved_model).

Setup

Installs and imports

Install and import TensorFlow and dependencies:

```
$ pip install pyyaml h5py  # Required to save models in HDF5 format
```

```
import os

import tensorflow as tf
from tensorflow import keras

print(tf.version.VERSION)
```

Get an example dataset

To demonstrate how to save and load weights, you'll use the MNIST dataset (http://yann.lecun.com/exdb/mnist/). To speed up these runs, use the first 1000 examples:

```
(train_images, train_labels), (test_images, test_labels) = tf.keras.datasets.m

train_labels = train_labels[:1000]
test_labels = test_labels[:1000]

train_images = train_images[:1000].reshape(-1, 28 * 28) / 255.0
test_images = test_images[:1000].reshape(-1, 28 * 28) / 255.0
```

Define a model

Start by building a simple sequential model:

```
# Define a simple sequential model
def create_model():
  model = tf.keras.Sequential([
    keras.layers.Dense(512, activation='relu', input_shape=(784,)),
    keras.layers.Dropout(0.2),
    keras.layers.Dense(10)
```

```
  ])

  model.compile(optimizer='adam',
                loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits
                metrics=[tf.keras.metrics.SparseCategoricalAccuracy()])

  return model

# Create a basic model instance
model = create_model()

# Display the model's architecture
model.summary()
```

Save checkpoints during training

You can use a trained model without having to retrain it, or pick-up training where you left off in case the training process was interrupted. The **`tf.keras.callbacks.ModelCheckpoint`** (https://www.tensorflow.org/api_docs/python/tf/keras/callbacks/ModelCheckpoint) callback allows you to continually save the model both *during* and at *the end* of training.

Checkpoint callback usage

Create a **`tf.keras.callbacks.ModelCheckpoint`** (https://www.tensorflow.org/api_docs/python/tf/keras/callbacks/ModelCheckpoint) callback that saves weights only during training:

```
checkpoint_path = "training_1/cp.ckpt"
checkpoint_dir = os.path.dirname(checkpoint_path)

# Create a callback that saves the model's weights
cp_callback = tf.keras.callbacks.ModelCheckpoint(filepath=checkpoint_path,
                                                 save_weights_only=True,
                                                 verbose=1)

# Train the model with the new callback
model.fit(train_images,
          train_labels,
          epochs=10,
          validation_data=(test_images, test_labels),
```

```
          callbacks=[cp_callback])  # Pass callback to training

# This may generate warnings related to saving the state of the optimizer.
# These warnings (and similar warnings throughout this notebook)
# are in place to discourage outdated usage, and can be ignored.
```

This creates a single collection of TensorFlow checkpoint files that are updated at the end of each epoch:

```
os.listdir(checkpoint_dir)
```

As long as two models share the same architecture you can share weights between them. So, when restoring a model from weights-only, create a model with the same architecture as the original model and then set its weights.

Now rebuild a fresh, untrained model and evaluate it on the test set. An untrained model will perform at chance levels (~10% accuracy):

```
# Create a basic model instance
model = create_model()

# Evaluate the model
loss, acc = model.evaluate(test_images, test_labels, verbose=2)
print("Untrained model, accuracy: {:5.2f}%".format(100 * acc))
```

Then load the weights from the checkpoint and re-evaluate:

```
# Loads the weights
model.load_weights(checkpoint_path)

# Re-evaluate the model
loss, acc = model.evaluate(test_images, test_labels, verbose=2)
print("Restored model, accuracy: {:5.2f}%".format(100 * acc))
```

Checkpoint callback options

The callback provides several options to provide unique names for checkpoints and adjust the checkpointing frequency.

Train a new model, and save uniquely named checkpoints once every five epochs:

```
# Include the epoch in the file name (uses `str.format`)
checkpoint_path = "training_2/cp-{epoch:04d}.ckpt"
checkpoint_dir = os.path.dirname(checkpoint_path)

batch_size = 32

# Calculate the number of batches per epoch
import math
n_batches = len(train_images) / batch_size
n_batches = math.ceil(n_batches)    # round up the number of batches to the ne

# Create a callback that saves the model's weights every 5 epochs
cp_callback = tf.keras.callbacks.ModelCheckpoint(
    filepath=checkpoint_path,
    verbose=1,
    save_weights_only=True,
    save_freq=5*n_batches)

# Create a new model instance
model = create_model()

# Save the weights using the `checkpoint_path` format
model.save_weights(checkpoint_path.format(epoch=0))

# Train the model with the new callback
model.fit(train_images,
          train_labels,
          epochs=50,
          batch_size=batch_size,
          callbacks=[cp_callback],
          validation_data=(test_images, test_labels),
          verbose=0)
```

Now, review the resulting checkpoints and choose the latest one:

```
os.listdir(checkpoint_dir)
```

```
latest = tf.train.latest_checkpoint(checkpoint_dir)
latest
```

Note: The default TensorFlow format only saves the 5 most recent checkpoints.

To test, reset the model, and load the latest checkpoint:

```
# Create a new model instance
model = create_model()

# Load the previously saved weights
model.load_weights(latest)

# Re-evaluate the model
loss, acc = model.evaluate(test_images, test_labels, verbose=2)
print("Restored model, accuracy: {:5.2f}%".format(100 * acc))
```

What are these files?

The above code stores the weights to a collection of checkpoint (https://www.tensorflow.org/guide/checkpoint)-formatted files that contain only the trained weights in a binary format. Checkpoints contain:

- One or more shards that contain your model's weights.
- An index file that indicates which weights are stored in which shard.

If you are training a model on a single machine, you'll have one shard with the suffix: `.data-00000-of-00001`

Manually save weights

To save weights manually, use `tf.keras.Model.save_weights` (https://www.tensorflow.org/api_docs/python/tf/keras/Model#save_weights). By default, `tf.keras` (https://www.tensorflow.org/api_docs/python/tf/keras)—and the `Model.save_weights` (https://www.tensorflow.org/api_docs/python/tf/keras/Model#save_weights) method in particular—

uses the TensorFlow Checkpoint (https://www.tensorflow.org/guide/checkpoint) format with a `.ckpt` extension. To save in the HDF5 format with a `.h5` extension, refer to the Save and load models (https://www.tensorflow.org/guide/keras/save_and_serialize) guide.

```
# Save the weights
model.save_weights('./checkpoints/my_checkpoint')

# Create a new model instance
model = create_model()

# Restore the weights
model.load_weights('./checkpoints/my_checkpoint')

# Evaluate the model
loss, acc = model.evaluate(test_images, test_labels, verbose=2)
print("Restored model, accuracy: {:5.2f}%".format(100 * acc))
```

Save the entire model

Call `tf.keras.Model.save` (https://www.tensorflow.org/api_docs/python/tf/keras/Model#save) to save a model's architecture, weights, and training configuration in a single `model.keras` zip archive.

An entire model can be saved in three different file formats (the new `.keras` format and two legacy formats: `SavedModel`, and `HDF5`). Saving a model as `path/to/model.keras` automatically saves in the latest format.

Note: For Keras objects it's recommended to use the new high-level `.keras` format for richer, name-based saving and reloading, which is easier to debug. The low-level SavedModel format and legacy H5 format continue to be supported for existing code.

You can switch to the SavedModel format by:

- Passing `save_format='tf'` to `save()`
- Passing a filename without an extension

You can switch to the H5 format by:

- Passing `save_format='h5'` to `save()`

- Passing a filename that ends in `.h5`

Saving a fully-functional model is very useful—you can load them in TensorFlow.js (Saved Model (https://www.tensorflow.org/js/tutorials/conversion/import_saved_model), HDF5 (https://www.tensorflow.org/js/tutorials/conversion/import_keras)) and then train and run them in web browsers, or convert them to run on mobile devices using TensorFlow Lite (Saved Model (https://www.tensorflow.org/lite/models/convert/#convert_a_savedmodel_recommended_), HDF5 (https://www.tensorflow.org/lite/models/convert/#convert_a_keras_model_))

*Custom objects (for example, subclassed models or layers) require special attention when saving and loading. Refer to the **Saving custom objects** section below.

New high-level `.keras` format

The new Keras v3 saving format, marked by the `.keras` extension, is a more simple, efficient format that implements name-based saving, ensuring what you load is exactly what you saved, from Python's perspective. This makes debugging much easier, and it is the recommended format for Keras.

The section below illustrates how to save and restore the model in the `.keras` format.

```
# Create and train a new model instance.
model = create_model()
model.fit(train_images, train_labels, epochs=5)

# Save the entire model as a `.keras` zip archive.
model.save('my_model.keras')
```

Reload a fresh Keras model from the `.keras` zip archive:

```
new_model = tf.keras.models.load_model('my_model.keras')

# Show the model architecture
new_model.summary()
```

Try running evaluate and predict with the loaded model:

```
# Evaluate the restored model
loss, acc = new_model.evaluate(test_images, test_labels, verbose=2)
```

```
print('Restored model, accuracy: {:5.2f}%'.format(100 * acc))

print(new_model.predict(test_images).shape)
```

SavedModel format

The SavedModel format is another way to serialize models. Models saved in this format can be restored using **`tf.keras.models.load_model`** (https://www.tensorflow.org/api_docs/python/tf/keras/models/load_model) and are compatible with TensorFlow Serving. The SavedModel guide (https://www.tensorflow.org/guide/saved_model) goes into detail about how to **`serve/inspect`** the SavedModel. The section below illustrates the steps to save and restore the model.

```
# Create and train a new model instance.
model = create_model()
model.fit(train_images, train_labels, epochs=5)

# Save the entire model as a SavedModel.
!mkdir -p saved_model
model.save('saved_model/my_model')
```

The SavedModel format is a directory containing a protobuf binary and a TensorFlow checkpoint. Inspect the saved model directory:

```
$ # my_model directory
$ ls saved_model
$
$ # Contains an assets folder, saved_model.pb, and variables folder.
$ ls saved_model/my_model
```

Reload a fresh Keras model from the saved model:

```
new_model = tf.keras.models.load_model('saved_model/my_model')

# Check its architecture
new_model.summary()
```

The restored model is compiled with the same arguments as the original model. Try running evaluate and predict with the loaded model:

```
# Evaluate the restored model
loss, acc = new_model.evaluate(test_images, test_labels, verbose=2)
print('Restored model, accuracy: {:5.2f}%'.format(100 * acc))

print(new_model.predict(test_images).shape)
```

HDF5 format

Keras provides a basic legacy high-level save format using the HDF5 (https://en.wikipedia.org/wiki/Hierarchical_Data_Format) standard.

```
# Create and train a new model instance.
model = create_model()
model.fit(train_images, train_labels, epochs=5)

# Save the entire model to a HDF5 file.
# The '.h5' extension indicates that the model should be saved to HDF5.
model.save('my_model.h5')
```

Now, recreate the model from that file:

```
# Recreate the exact same model, including its weights and the optimizer
new_model = tf.keras.models.load_model('my_model.h5')

# Show the model architecture
new_model.summary()
```

Check its accuracy:

```
loss, acc = new_model.evaluate(test_images, test_labels, verbose=2)
print('Restored model, accuracy: {:5.2f}%'.format(100 * acc))
```

Keras saves models by inspecting their architectures. This technique saves everything:

- The weight values
- The model's architecture
- The model's training configuration (what you pass to the `.compile()` method)
- The optimizer and its state, if any (this enables you to restart training where you left off)

Keras is not able to save the `v1.x` optimizers (from `tf.compat.v1.train` (https://www.tensorflow.org/api_docs/python/tf/compat/v1/train)) since they aren't compatible with checkpoints. For v1.x optimizers, you need to re-compile the model after loading—losing the state of the optimizer.

Saving custom objects

If you are using the SavedModel format, you can skip this section. The key difference between high-level `.keras`/HDF5 formats and the low-level SavedModel format is that the `.keras`/HDF5 formats uses object configs to save the model architecture, while SavedModel saves the execution graph. Thus, SavedModels are able to save custom objects like subclassed models and custom layers without requiring the original code. However, debugging low-level SavedModels can be more difficult as a result, and we recommend using the high-level `.keras` format instead due to its name-based, Keras-native nature.

To save custom objects to `.keras` and HDF5, you must do the following:

1. Define a `get_config` method in your object, and optionally a `from_config` classmethod.
    - `get_config(self)` returns a JSON-serializable dictionary of parameters needed to recreate the object.
    - `from_config(cls, config)` uses the returned config from `get_config` to create a new object. By default, this function will use the config as initialization kwargs (`return cls(**config)`).
2. Pass the custom objects to the model in one of three ways:
    - Register the custom object with the `@tf.keras.utils.register_keras_serializable` (https://www.tensorflow.org/api_docs/python/tf/keras/utils/register_keras_serializable) decorator. **(recommended)**

- Directly pass the object to the `custom_objects` argument when loading the model. The argument must be a dictionary mapping the string class name to the Python class. E.g., `tf.keras.models.load_model(path, custom_objects={'CustomLayer': CustomLayer})`
- Use a `tf.keras.utils.custom_object_scope` (https://www.tensorflow.org/api_docs/python/tf/keras/utils/CustomObjectScope) with the object included in the `custom_objects` dictionary argument, and place a `tf.keras.models.load_model(path)` (https://www.tensorflow.org/api_docs/python/tf/keras/models/load_model) call within the scope.

Refer to the Writing layers and models from scratch (https://www.tensorflow.org/guide/keras/custom_layers_and_models) tutorial for examples of custom objects and `get_config`.

```
# MIT License
#
# Copyright (c) 2017 François Chollet
#
# Permission is hereby granted, free of charge, to any person obtaining a
# copy of this software and associated documentation files (the "Software"),
# to deal in the Software without restriction, including without limitation
# the rights to use, copy, modify, merge, publish, distribute, sublicense,
# and/or sell copies of the Software, and to permit persons to whom the
# Software is furnished to do so, subject to the following conditions:
#
# The above copyright notice and this permission notice shall be included in
# all copies or substantial portions of the Software.
#
# THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
# IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
# FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL
# THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
# LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
# FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER
# DEALINGS IN THE SOFTWARE.
```

Except as otherwise noted, the content of this page is licensed under the Creative Commons Attribution 4.0 License (https://creativecommons.org/licenses/by/4.0/), and code samples are licensed under the Apache 2.0 License (https://www.apache.org/licenses/LICENSE-2.0). For details, see the Google Developers Site Policies (https://developers.google.com/site-policies). Java is a registered trademark of Oracle and/or its affiliates.

Last updated 2024-04-03 UTC.