Keras FAQ: Frequently Asked Keras Questions

How should I cite Keras?

Please cite Keras in your publications if it helps your research. Here is an example BibTeX entry:

```
@misc{chollet2015keras,
  title={Keras},
  author={Chollet, Fran\c{c}ois},
  year={2015},
  publisher={GitHub},
  howpublished={\url{https://github.com/fchollet/keras}},
}
```

How can I run Keras on GPU?

If you are running on the TensorFlow backend, your code will automatically run on GPU if any available GPU is detected. If you are running on the Theano backend, you can use one of the following methods:

Method 1: use Theano flags.

```
THEANO_FLAGS=device=gpu,floatX=float32 python my_keras_script.py
```

The name 'gpu' might have to be changed depending on your device's identifier (e.g. `gpu0`, `gpu1`, etc).

Method 2: set up your `.theanorc`: Instructions

Method 3: manually set `theano.config.device`, `theano.config.floatX` at the beginning of your code:

```
import theano
theano.config.device = 'gpu'
theano.config.floatX = 'float32'
```

How can I save a Keras model?

It is not recommended to use pickle or cPickle to save a Keras model.

You can use `model.save(filepath)` to save a Keras model into a single HDF5 file which will contain:

- the architecture of the model, allowing to re-create the model
- the weights of the model
- the training configuration (loss, optimizer)
- the state of the optimizer, allowing to resume training exactly where you left off.

You can then use `keras.models.load_model(filepath)` to reinstantiate your model. `load_model` will also take care of compiling the model using the saved training configuration (unless the model was never compiled in the first place).

Example:

```
from keras.models import load_model

model.save('my_model.h5')  # creates a HDF5 file 'my_model.h5'
del model  # deletes the existing model

# returns a compiled model
# identical to the previous one
model = load_model('my_model.h5')
```

If you only need to save the **architecture of a model**, and not its weights or its training configuration, you can do:

```
# save as JSON
json_string = model.to_json()

# save as YAML
yaml_string = model.to_yaml()
```

The generated JSON / YAML files are human-readable and can be manually edited if needed.

You can then build a fresh model from this data:

```
# model reconstruction from JSON:
from keras.models import model_from_json
model = model_from_json(json_string)

# model reconstruction from YAML
model = model_from_yaml(yaml_string)
```

If you need to save the **weights of a model**, you can do so in HDF5 with the code below.

Note that you will first need to install HDF5 and the Python library h5py, which do not come bundled with Keras.

```
model.save_weights('my_model_weights.h5')
```

Assuming you have code for instantiating your model, you can then load the weights you saved into a model with the same architecture:

```
model.load_weights('my_model_weights.h5')
```

Why is the training loss much higher than the testing loss?

A Keras model has two modes: training and testing. Regularization mechanisms, such as Dropout and L1/L2 weight regularization, are turned off at testing time.

Besides, the training loss is the average of the losses over each batch of training data. Because your model is changing over time, the loss over the first batches of an epoch is generally higher than over the last batches. On the other hand, the testing loss for an epoch is computed using the model as it is at the end of the epoch, resulting in a lower loss.

How can I visualize the output of an intermediate layer?

You can build a Keras function that will return the output of a certain layer given a certain input, for example:

```
from keras import backend as K

# with a Sequential model
get_3rd_layer_output = K.function([model.layers[0].input],
                                  [model.layers[3].output])
layer_output = get_3rd_layer_output([X])[0]
```

Similarly, you could build a Theano and TensorFlow function directly.

Note that if your model has a different behavior in training and testing phase (e.g. if it uses `Dropout`, `BatchNormalization`, etc.), you will need to pass the learning phase flag to your function:

```
get_3rd_layer_output = K.function([model.layers[0].input, K.learning_phase()],
                                  [model.layers[3].output])

# output in test mode = 0
layer_output = get_3rd_layer_output([X, 0])[0]

# output in train mode = 1
layer_output = get_3rd_layer_output([X, 1])[0]
```

Another more flexible way of getting output from intermediate layers is to use the functional API. For example, if you have created an autoencoder for MNIST:

```
inputs = Input(shape=(784,))
encoded = Dense(32, activation='relu')(inputs)
decoded = Dense(784)(encoded)
model = Model(input=inputs, output=decoded)
```

After compiling and training the model, you can get the output of the data from the encoder like this:

```
encoder = Model(input=inputs, output=encoded)
X_encoded = encoder.predict(X)
```

How can I use Keras with datasets that don't fit in memory?

You can do batch training using `model.train_on_batch(X, y)` and `model.test_on_batch(X, y)`. See the models documentation.

Alternatively, you can write a generator that yields batches of training data and use the method `model.fit_generator(data_generator, samples_per_epoch, nb_epoch)`.

You can see batch training in action in our CIFAR10 example.

How can I interrupt training when the validation loss isn't decreasing anymore?

You can use an `EarlyStopping` callback:

```
from keras.callbacks import EarlyStopping
early_stopping = EarlyStopping(monitor='val_loss', patience=2)
model.fit(X, y, validation_split=0.2, callbacks=[early_stopping])
```

Find out more in the callbacks documentation.

How is the validation split computed?

If you set the `validation_split` argument in `model.fit` to e.g. 0.1, then the validation data used will be the *last 10%* of the data. If you set it to 0.25, it will be the last 25% of the data, etc.

Is the data shuffled during training?

Yes, if the `shuffle` argument in `model.fit` is set to `True` (which is the default), the training data will be randomly shuffled at each epoch.

Validation data is never shuffled.

How can I record the training / validation loss / accuracy at each epoch?

The `model.fit` method returns an `History` callback, which has a `history` attribute containing the lists of successive losses and other metrics.

```
hist = model.fit(X, y, validation_split=0.2)
print(hist.history)
```

How can I "freeze" Keras layers?

To "freeze" a layer means to exclude it from training, i.e. its weights will never be updated. This is useful in the context of fine-tuning a model, or using fixed embeddings for a text input.

You can pass a `trainable` argument (boolean) to a layer constructor to set a layer to be non-trainable:

```
frozen_layer = Dense(32, trainable=False)
```

Additionally, you can set the `trainable` property of a layer to `True` or `False` after instantiation. For this to take effect, you will need to call `compile()` on your model after modifying the `trainable` property. Here's an example:

```
x = Input(shape=(32,))
layer = Dense(32)
layer.trainable = False
y = layer(x)

frozen_model = Model(x, y)
# in the model below, the weights of `layer` will not be updated during training
frozen_model.compile(optimizer='rmsprop', loss='mse')

layer.trainable = True
trainable_model = Model(x, y)
# with this model the weights of the layer will be updated during training
# (which will also affect the above model since it uses the same layer instance)
trainable_model.compile(optimizer='rmsprop', loss='mse')

frozen_model.fit(data, labels)  # this does NOT update the weights of `layer`
trainable_model.fit(data, labels)  # this updates the weights of `layer`
```

How can I use stateful RNNs?

Making a RNN stateful means that the states for the samples of each batch will be reused as initial states for the samples in the next batch.

When using stateful RNNs, it is therefore assumed that:

- all batches have the same number of samples
- If `X1` and `X2` are successive batches of samples, then `X2[i]` is the follow-up sequence to `X1[i]`, for every `i`.

To use statefulness in RNNs, you need to:

- explicitly specify the batch size you are using, by passing a `batch_input_shape` argument to the first layer in your model. It should be a tuple of integers, e.g. `(32, 10, 16)` for a 32-samples batch of sequences of 10 timesteps with 16 features per timestep.
- set `stateful=True` in your RNN layer(s).

To reset the states accumulated:

- use `model.reset_states()` to reset the states of all layers in the model
- use `layer.reset_states()` to reset the states of a specific stateful RNN layer

Example:

```
X  # this is our input data, of shape (32, 21, 16)
# we will feed it to our model in sequences of length 10

model = Sequential()
model.add(LSTM(32, batch_input_shape=(32, 10, 16), stateful=True))
model.add(Dense(16, activation='softmax'))

model.compile(optimizer='rmsprop', loss='categorical_crossentropy')

# we train the network to predict the 11th timestep given the first 10:
model.train_on_batch(X[:, :10, :], np.reshape(X[:, 10, :], (32, 16)))

# the state of the network has changed. We can feed the follow-up sequences:
model.train_on_batch(X[:, 10:20, :], np.reshape(X[:, 20, :], (32, 16)))

# let's reset the states of the LSTM layer:
model.reset_states()

# another way to do it in this case:
model.layers[0].reset_states()
```

Notes that the methods `predict`, `fit`, `train_on_batch`, `predict_classes`, etc. will *all* update the states of the stateful layers in a model. This allows you to do not only stateful training, but also stateful prediction.

How can I remove a layer from a Sequential model?

You can remove the last added layer in a Sequential model by calling `.pop()`:

```
model = Sequential()
model.add(Dense(32, activation='relu', input_dim=784))
model.add(Dense(32, activation='relu'))

print(len(model.layers))  # "2"

model.pop()
print(len(model.layers))  # "1"
```

How can I use pre-trained models in Keras?

Code and pre-trained weights are available for the following image classification models:

- VGG-16
- VGG-19
- AlexNet

For an example of how to use such a pre-trained model for feature extraction or for fine-tuning, see this blog post.

The VGG-16 model is also the basis for several Keras example scripts:

- Style transfer
- Feature visualization
- Deep dream