Contents

Sequential model methods 2
compile 2
fit 2
evaluate 4
predict 5
train_on_batch 6
test_on_batch 6
predict_on_batch 6
fit_generator 7
evaluate_generator 9
predict_generator 9
get_layer 10
Getting started with the Keras Sequential model 11
Specifying the input shape 11
Compilation 12
Training 12
Examples 13
Multilayer Perceptron (MLP) for multi-class softmax classification: 14
MLP for binary classification: 15
VGG-like convnet: 15
Sequence classification with LSTM: 16
Sequence classification with 1D convolutions: 17
Stacked LSTM for sequence classification 18
Same stacked LSTM model, rendered "stateful" 19
Peer-graded Assignment: Build a Regression Model in Keras 21
Peer Review Final Assignment 22
Part 1 22
Part 2 22
Part 3 22

Sequential model methods

compile

```
compile(optimizer, loss=None, metrics=None, loss_weights=None, sample_weight_mode=None, weighted_metrics=None, target_tensors=None)
```

Configures the model for training.

Arguments

- **optimizer**: String (name of optimizer) or optimizer instance. See optimizers.
- **loss**: String (name of objective function) or objective function or `Loss` instance. See losses. If the model has multiple outputs, you can use a different loss on each output by passing a dictionary or a list of losses. The loss value that will be minimized by the model will then be the sum of all individual losses.
- **metrics**: List of metrics to be evaluated by the model during training and testing. Typically you will use `metrics=['accuracy']`. To specify different metrics for different outputs of a multi-output model, you could also pass a dictionary, such as `metrics={'output_a': 'accuracy', 'output_b': ['accuracy', 'mse']}`. You can also pass a list (len = len(outputs)) of lists of metrics such as `metrics=[['accuracy'], ['accuracy', 'mse']]` or `metrics=['accuracy', ['accuracy', 'mse']]`.
- **loss_weights**: Optional list or dictionary specifying scalar coefficients (Python floats) to weight the loss contributions of different model outputs. The loss value that will be minimized by the model will then be the *weighted sum* of all individual losses, weighted by the `loss_weights` coefficients. If a list, it is expected to have a 1:1 mapping to the model's outputs. If a dict, it is expected to map output names (strings) to scalar coefficients.
- **sample_weight_mode**: If you need to do timestep-wise sample weighting (2D weights), set this to `"temporal"`. `None` defaults to sample-wise weights (1D). If the model has multiple outputs, you can use a different `sample_weight_mode` on each output by passing a dictionary or a list of modes.
- **weighted_metrics**: List of metrics to be evaluated and weighted by sample_weight or class_weight during training and testing.
- **target_tensors**: By default, Keras will create placeholders for the model's target, which will be fed with the target data during training. If instead you would like to use your own target tensors (in turn, Keras will not expect external Numpy data for these targets at training time), you can specify them via the `target_tensors` argument. It can be a single tensor (for a single-output model), a list of tensors, or a dict mapping output names to target tensors.
- ****kwargs**: When using the Theano/CNTK backends, these arguments are passed into `K.function`. When using the TensorFlow backend, these arguments are passed into `tf.Session.run`.

Raises

- **ValueError**: In case of invalid arguments for `optimizer`, `loss`, `metrics` or `sample_weight_mode`.

---

fit

```
fit(x=None, y=None, batch_size=None, epochs=1, verbose=1, callbacks=None, validation_split=0.0, validation_data=None, shuffle=True, class_weight=None, sample_weight=None, initial_epoch=0, steps_per_epoch=None, validation_steps=None, validation_freq=1, max_queue_size=10, workers=1, use_multiprocessing=False)
```

Trains the model for a fixed number of epochs (iterations on a dataset).

Arguments

- **x**: Input data. It could be:
    - A Numpy array (or array-like), or a list of arrays (in case the model has multiple inputs).
    - A dict mapping input names to the corresponding array/tensors, if the model has named inputs.
    - A generator or `keras.utils.Sequence` returning `(inputs, targets)` or `(inputs, targets, sample weights)`.

- None (default) if feeding from framework-native tensors (e.g. TensorFlow data tensors).

- **y**: Target data. Like the input data `x`, it could be either Numpy array(s), framework-native tensor(s), list of Numpy arrays (if the model has multiple outputs) or None (default) if feeding from framework-native tensors (e.g. TensorFlow data tensors). If output layers in the model are named, you can also pass a dictionary mapping output names to Numpy arrays. If `x` is a generator, or `keras.utils.Sequence` instance, `y` should not be specified (since targets will be obtained from `x`).
- **batch_size**: Integer or `None`. Number of samples per gradient update. If unspecified, `batch_size` will default to 32. Do not specify the `batch_size` if your data is in the form of symbolic tensors, generators, or `Sequence` instances (since they generate batches).
- **epochs**: Integer. Number of epochs to train the model. An epoch is an iteration over the entire `x` and `y` data provided. Note that in conjunction with `initial_epoch`, `epochs` is to be understood as "final epoch". The model is not trained for a number of iterations given by `epochs`, but merely until the epoch of index `epochs` is reached.
- **verbose**: Integer. 0, 1, or 2. Verbosity mode. 0 = silent, 1 = progress bar, 2 = one line per epoch.
- **callbacks**: List of `keras.callbacks.Callback` instances. List of callbacks to apply during training and validation (if). See callbacks.
- **validation_split**: Float between 0 and 1. Fraction of the training data to be used as validation data. The model will set apart this fraction of the training data, will not train on it, and will evaluate the loss and any model metrics on this data at the end of each epoch. The validation data is selected from the last samples in the `x` and `y` data provided, before shuffling. This argument is not supported when `x` is a generator or `Sequence` instance.
- **validation_data**: Data on which to evaluate the loss and any model metrics at the end of each epoch. The model will not be trained on this data. `validation_data` will override `validation_split`. `validation_data` could be: - tuple `(x_val, y_val)` of Numpy arrays or tensors - tuple `(x_val, y_val, val_sample_weights)` of Numpy arrays - dataset or a dataset iterator

  For the first two cases, `batch_size` must be provided. For the last case, `validation_steps` must be provided.

- **shuffle**: Boolean (whether to shuffle the training data before each epoch) or str (for 'batch'). 'batch' is a special option for dealing with the limitations of HDF5 data; it shuffles in batch-sized chunks. Has no effect when `steps_per_epoch` is not `None`.
- **class_weight**: Optional dictionary mapping class indices (integers) to a weight (float) value, used for weighting the loss function (during training only). This can be useful to tell the model to "pay more attention" to samples from an under-represented class.
- **sample_weight**: Optional Numpy array of weights for the training samples, used for weighting the loss function (during training only). You can either pass a flat (1D) Numpy array with the same length as the input samples (1:1 mapping between weights and samples), or in the case of temporal data, you can pass a 2D array with shape `(samples, sequence_length)`, to apply a different weight to every timestep of every sample. In this case you should make sure to specify `sample_weight_mode="temporal"` in `compile()`. This argument is not supported when `x` generator, or `Sequence` instance, instead provide the sample_weights as the third element of `x`.
- **initial_epoch**: Integer. Epoch at which to start training (useful for resuming a previous training run).
- **steps_per_epoch**: Integer or `None`. Total number of steps (batches of samples) before declaring one epoch finished and starting the next epoch. When training with input tensors such as TensorFlow data tensors, the default `None` is equal to the number of samples in your dataset divided by the batch size, or 1 if that cannot be determined.
- **validation_steps**: Only relevant if `steps_per_epoch` is specified. Total number of steps (batches of samples) to validate before stopping.
- **validation_steps**: Only relevant if `validation_data` is provided and is a generator. Total number of steps (batches of samples) to draw before stopping when performing validation at the end of every epoch.
- **validation_freq**: Only relevant if validation data is provided. Integer or list/tuple/set. If an integer, specifies how many training epochs to run before a new validation run is performed, e.g. `validation_freq=2` runs validation every 2 epochs. If a list, tuple, or set, specifies the epochs on which to run validation, e.g. `validation_freq=[1, 2, 10]` runs validation at the end of the 1st, 2nd, and 10th epochs.
- **max_queue_size**: Integer. Used for generator or `keras.utils.Sequence` input only. Maximum size for the generator queue. If unspecified, `max_queue_size` will default to 10.
- **workers**: Integer. Used for generator or `keras.utils.Sequence` input only. Maximum number of processes to spin up when using process-based threading. If unspecified, `workers` will default to 1. If 0, will execute the generator on the main thread.
- **use_multiprocessing**: Boolean. Used for generator or `keras.utils.Sequence` input only. If `True`, use process-based threading. If unspecified, `use_multiprocessing` will default to `False`. Note that because this implementation relies on multiprocessing, you should not pass non-picklable arguments to the generator as they can't be passed easily to children processes.

- ****kwargs**: Used for backwards compatibility.

Returns

A `History` object. Its `History.history` attribute is a record of training loss values and metrics values at successive epochs, as well as validation loss values and validation metrics values (if applicable).

Raises

- **RuntimeError**: If the model was never compiled.
- **ValueError**: In case of mismatch between the provided input data and what the model expects.

---

evaluate

```
evaluate(x=None, y=None, batch_size=None, verbose=1, sample_weight=None, steps=None, callbacks=None,
max_queue_size=10, workers=1, use_multiprocessing=False)
```

Returns the loss value & metrics values for the model in test mode.

Computation is done in batches.

Arguments

- **x**: Input data. It could be:
  - A Numpy array (or array-like), or a list of arrays (in case the model has multiple inputs).
  - A dict mapping input names to the corresponding array/tensors, if the model has named inputs.
  - A generator or `keras.utils.Sequence` returning `(inputs, targets)` or `(inputs, targets, sample weights)`.
  - None (default) if feeding from framework-native tensors (e.g. TensorFlow data tensors).
- **y**: Target data. Like the input data `x`, it could be either Numpy array(s), framework-native tensor(s), list of Numpy arrays (if the model has multiple outputs) or None (default) if feeding from framework-native tensors (e.g. TensorFlow data tensors). If output layers in the model are named, you can also pass a dictionary mapping output names to Numpy arrays. If `x` is a generator, or `keras.utils.Sequence` instance, `y` should not be specified (since targets will be obtained from `x`).
- **batch_size**: Integer or `None`. Number of samples per gradient update. If unspecified, `batch_size` will default to 32. Do not specify the `batch_size` is your data is in the form of symbolic tensors, generators, or `keras.utils.Sequence` instances (since they generate batches).
- **verbose**: 0 or 1. Verbosity mode. 0 = silent, 1 = progress bar.
- **sample_weight**: Optional Numpy array of weights for the test samples, used for weighting the loss function. You can either pass a flat (1D) Numpy array with the same length as the input samples (1:1 mapping between weights and samples), or in the case of temporal data, you can pass a 2D array with shape `(samples, sequence_length)`, to apply a different weight to every timestep of every sample. In this case you should make sure to specify `sample_weight_mode="temporal"` in `compile()`.
- **steps**: Integer or `None`. Total number of steps (batches of samples) before declaring the evaluation round finished. Ignored with the default value of `None`.
- **callbacks**: List of `keras.callbacks.Callback` instances. List of callbacks to apply during evaluation. See callbacks.
- **max_queue_size**: Integer. Used for generator or `keras.utils.Sequence` input only. Maximum size for the generator queue. If unspecified, `max_queue_size` will default to 10.
- **workers**: Integer. Used for generator or `keras.utils.Sequence` input only. Maximum number of processes to spin up when using process-based threading. If unspecified, `workers` will default to 1. If 0, will execute the generator on the main thread.
- **use_multiprocessing**: Boolean. Used for generator or `keras.utils.Sequence` input only. If `True`, use process-based threading. If unspecified, `use_multiprocessing` will default to `False`. Note that because this implementation relies on multiprocessing, you should not pass non-picklable arguments to the generator as they can't be passed easily to children processes.

Raises

- **ValueError**: in case of invalid arguments.

Returns

Scalar test loss (if the model has a single output and no metrics) or list of scalars (if the model has multiple outputs and/or metrics). The attribute `model.metrics_names` will give you the display labels for the scalar outputs.

---

predict

```
predict(x, batch_size=None, verbose=0, steps=None, callbacks=None, max_queue_size=10, workers=1,
use_multiprocessing=False)
```

Generates output predictions for the input samples.

Computation is done in batches.

Arguments

- **x**: Input data. It could be:
  - A Numpy array (or array-like), or a list of arrays (in case the model has multiple inputs).
  - A dict mapping input names to the corresponding array/tensors, if the model has named inputs.
  - A generator or `keras.utils.Sequence` returning `(inputs, targets)` or `(inputs, targets, sample weights)`.
  - None (default) if feeding from framework-native tensors (e.g. TensorFlow data tensors).
- **batch_size**: Integer or `None`. Number of samples per gradient update. If unspecified, `batch_size` will default to 32. Do not specify the `batch_size` is your data is in the form of symbolic tensors, generators, or `keras.utils.Sequence` instances (since they generate batches).
- **verbose**: Verbosity mode, 0 or 1.
- **steps**: Total number of steps (batches of samples) before declaring the prediction round finished. Ignored with the default value of `None`.
- **callbacks**: List of `keras.callbacks.Callback` instances. List of callbacks to apply during prediction. See callbacks.
- **max_queue_size**: Integer. Used for generator or `keras.utils.Sequence` input only. Maximum size for the generator queue. If unspecified, `max_queue_size` will default to 10.
- **workers**: Integer. Used for generator or `keras.utils.Sequence` input only. Maximum number of processes to spin up when using process-based threading. If unspecified, `workers` will default to 1. If 0, will execute the generator on the main thread.
- **use_multiprocessing**: Boolean. Used for generator or `keras.utils.Sequence` input only. If `True`, use process-based threading. If unspecified, `use_multiprocessing` will default to `False`. Note that because this implementation relies on multiprocessing, you should not pass non-picklable arguments to the generator as they can't be passed easily to children processes.

Returns

Numpy array(s) of predictions.

Raises

- **ValueError**: In case of mismatch between the provided input data and the model's expectations, or in case a stateful model receives a number of samples that is not a multiple of the batch size.

---

train_on_batch

```
train_on_batch(x, y, sample_weight=None, class_weight=None, reset_metrics=True)
```

Runs a single gradient update on a single batch of data.

Arguments

- **x**: Numpy array of training data, or list of Numpy arrays if the model has multiple inputs. If all inputs in the model are named, you can also pass a dictionary mapping input names to Numpy arrays.
- **y**: Numpy array of target data, or list of Numpy arrays if the model has multiple outputs. If all outputs in the model are named, you can also pass a dictionary mapping output names to Numpy arrays.
- **sample_weight**: Optional array of the same length as x, containing weights to apply to the model's loss for each sample. In the case of temporal data, you can pass a 2D array with shape (samples, sequence_length), to apply a different weight to every timestep of every sample. In this case you should make sure to specify sample_weight_mode="temporal" in compile().
- **class_weight**: Optional dictionary mapping class indices (integers) to a weight (float) to apply to the model's loss for the samples from this class during training. This can be useful to tell the model to "pay more attention" to samples from an under-represented class.
- **reset_metrics**: If `True`, the metrics returned will be only for this batch. If `False`, the metrics will be statefully accumulated across batches.

Returns

Scalar training loss (if the model has a single output and no metrics) or list of scalars (if the model has multiple outputs and/or metrics). The attribute `model.metrics_names` will give you the display labels for the scalar outputs.

---

test_on_batch

```
test_on_batch(x, y, sample_weight=None, reset_metrics=True)
```

Test the model on a single batch of samples.

Arguments

- **x**: Numpy array of test data, or list of Numpy arrays if the model has multiple inputs. If all inputs in the model are named, you can also pass a dictionary mapping input names to Numpy arrays.
- **y**: Numpy array of target data, or list of Numpy arrays if the model has multiple outputs. If all outputs in the model are named, you can also pass a dictionary mapping output names to Numpy arrays.
- **sample_weight**: Optional array of the same length as x, containing weights to apply to the model's loss for each sample. In the case of temporal data, you can pass a 2D array with shape (samples, sequence_length), to apply a different weight to every timestep of every sample. In this case you should make sure to specify sample_weight_mode="temporal" in compile().
- **reset_metrics**: If `True`, the metrics returned will be only for this batch. If `False`, the metrics will be statefully accumulated across batches.

Returns

Scalar test loss (if the model has a single output and no metrics) or list of scalars (if the model has multiple outputs and/or metrics). The attribute `model.metrics_names` will give you the display labels for the scalar outputs.

---

predict_on_batch

```
predict_on_batch(x)
```

Returns predictions for a single batch of samples.

Arguments

- **x**: Input samples, as a Numpy array.

Returns

Numpy array(s) of predictions.

---

fit_generator

```
fit_generator(generator, steps_per_epoch=None, epochs=1, verbose=1, callbacks=None, validation_data=None, validation_steps=None, validation_freq=1, class_weight=None, max_queue_size=10, workers=1, use_multiprocessing=False, shuffle=True, initial_epoch=0)
```

Trains the model on data generated batch-by-batch by a Python generator (or an instance of `Sequence`).

The generator is run in parallel to the model, for efficiency. For instance, this allows you to do real-time data augmentation on images on CPU in parallel to training your model on GPU.

The use of `keras.utils.Sequence` guarantees the ordering and guarantees the single use of every input per epoch when using `use_multiprocessing=True`.

Arguments

- **generator**: A generator or an instance of `Sequence` (`keras.utils.Sequence`) object in order to avoid duplicate data when using multiprocessing. The output of the generator must be either
  - a tuple `(inputs, targets)`
  - a tuple `(inputs, targets, sample_weights)`.

  This tuple (a single output of the generator) makes a single batch. Therefore, all arrays in this tuple must have the same length (equal to the size of this batch). Different batches may have different sizes. For example, the last batch of the epoch is commonly smaller than the others, if the size of the dataset is not divisible by the batch size. The generator is expected to loop over its data indefinitely. An epoch finishes when `steps_per_epoch` batches have been seen by the model.

- **steps_per_epoch**: Integer. Total number of steps (batches of samples) to yield from `generator` before declaring one epoch finished and starting the next epoch. It should typically be equal to `ceil(num_samples / batch_size)` Optional for `Sequence`: if unspecified, will use the `len(generator)` as a number of steps.
- **epochs**: Integer. Number of epochs to train the model. An epoch is an iteration over the entire data provided, as defined by `steps_per_epoch`. Note that in conjunction with `initial_epoch`, `epochs` is to be understood as "final epoch". The model is not trained for a number of iterations given by `epochs`, but merely until the epoch of index `epochs` is reached.
- **verbose**: Integer. 0, 1, or 2. Verbosity mode. 0 = silent, 1 = progress bar, 2 = one line per epoch.
- **callbacks**: List of `keras.callbacks.Callback` instances. List of callbacks to apply during training. See callbacks.
- **validation_data**: This can be either
  - a generator or a `Sequence` object for the validation data
  - tuple `(x_val, y_val)`
  - tuple `(x_val, y_val, val_sample_weights)`

  on which to evaluate the loss and any model metrics at the end of each epoch. The model will not be trained on this data.

- **validation_steps**: Only relevant if `validation_data` is a generator. Total number of steps (batches of samples) to yield from `validation_data` generator before stopping at the end of every epoch. It should typically be equal to the number of samples of your validation dataset divided by the batch size. Optional for `Sequence`: if unspecified, will use the `len(validation_data)` as a number of steps.
- **validation_freq**: Only relevant if validation data is provided. Integer or `collections.Container` instance (e.g. list, tuple, etc.). If an integer, specifies how many training epochs to run before a new validation run is performed, e.g. `validation_freq=2` runs validation every 2 epochs. If a Container, specifies the epochs on which to run validation, e.g. `validation_freq=[1, 2, 10]` runs validation at the end of the 1st, 2nd, and 10th epochs.
- **class_weight**: Optional dictionary mapping class indices (integers) to a weight (float) value, used for weighting the loss function (during training only). This can be useful to tell the model to "pay more attention" to samples from an under-represented class.
- **max_queue_size**: Integer. Maximum size for the generator queue. If unspecified, `max_queue_size` will default to 10.
- **workers**: Integer. Maximum number of processes to spin up when using process-based threading. If unspecified, `workers` will default to 1. If 0, will execute the generator on the main thread.
- **use_multiprocessing**: Boolean. If `True`, use process-based threading. If unspecified, `use_multiprocessing` will default to `False`. Note that because this implementation relies on multiprocessing, you should not pass non-picklable arguments to the generator as they can't be passed easily to children processes.
- **shuffle**: Boolean. Whether to shuffle the order of the batches at the beginning of each epoch. Only used with instances of `Sequence` (`keras.utils.Sequence`). Has no effect when `steps_per_epoch` is not `None`.
- **initial_epoch**: Integer. Epoch at which to start training (useful for resuming a previous training run).

Returns

A `History` object. Its `History.history` attribute is a record of training loss values and metrics values at successive epochs, as well as validation loss values and validation metrics values (if applicable).

Raises

- **ValueError**: In case the generator yields data in an invalid format.

Example

```
def generate_arrays_from_file(path):

    while True:

        with open(path) as f:

            for line in f:

                # create numpy arrays of input data

                # and labels, from each line in the file

                x1, x2, y = process_line(line)

                yield ({'input_1': x1, 'input_2': x2}, {'output': y})



model.fit_generator(generate_arrays_from_file('/my_file.txt'),

                    steps_per_epoch=10000, epochs=10)
```

evaluate_generator

```
evaluate_generator(generator, steps=None, callbacks=None, max_queue_size=10, workers=1, use_multiprocessing=False, verbose=0)
```

Evaluates the model on a data generator.

The generator should return the same kind of data as accepted by `test_on_batch`.

Arguments

- **generator**: Generator yielding tuples (inputs, targets) or (inputs, targets, sample_weights) or an instance of Sequence (keras.utils.Sequence) object in order to avoid duplicate data when using multiprocessing.
- **steps**: Total number of steps (batches of samples) to yield from `generator` before stopping. Optional for `Sequence`: if unspecified, will use the `len(generator)` as a number of steps.
- **callbacks**: List of `keras.callbacks.Callback` instances. List of callbacks to apply during training. See callbacks.
- **max_queue_size**: maximum size for the generator queue
- **workers**: Integer. Maximum number of processes to spin up when using process based threading. If unspecified, `workers` will default to 1. If 0, will execute the generator on the main thread.
- **use_multiprocessing**: if True, use process based threading. Note that because this implementation relies on multiprocessing, you should not pass non picklable arguments to the generator as they can't be passed easily to children processes.
- **verbose**: verbosity mode, 0 or 1.

Returns

Scalar test loss (if the model has a single output and no metrics) or list of scalars (if the model has multiple outputs and/or metrics). The attribute `model.metrics_names` will give you the display labels for the scalar outputs.

Raises

- **ValueError**: In case the generator yields data in an invalid format.

---

predict_generator

```
predict_generator(generator, steps=None, callbacks=None, max_queue_size=10, workers=1, use_multiprocessing=False, verbose=0)
```

Generates predictions for the input samples from a data generator.

The generator should return the same kind of data as accepted by `predict_on_batch`.

Arguments

- **generator**: Generator yielding batches of input samples or an instance of Sequence (keras.utils.Sequence) object in order to avoid duplicate data when using multiprocessing.
- **steps**: Total number of steps (batches of samples) to yield from `generator` before stopping. Optional for `Sequence`: if unspecified, will use the `len(generator)` as a number of steps.
- **callbacks**: List of `keras.callbacks.Callback` instances. List of callbacks to apply during training. See callbacks.
- **max_queue_size**: Maximum size for the generator queue.
- **workers**: Integer. Maximum number of processes to spin up when using process based threading. If unspecified, `workers` will default to 1. If 0, will execute the generator on the main thread.

- **use_multiprocessing**: If `True`, use process based threading. Note that because this implementation relies on multiprocessing, you should not pass non picklable arguments to the generator as they can't be passed easily to children processes.
- **verbose**: verbosity mode, 0 or 1.

Returns

Numpy array(s) of predictions.

Raises

- **ValueError**: In case the generator yields data in an invalid format.

---

get_layer

`get_layer(name=None, index=None)`

Retrieves a layer based on either its name (unique) or index.

If `name` and `index` are both provided, `index` will take precedence.

Indices are based on order of horizontal graph traversal (bottom-up).

Arguments

- **name**: String, name of layer.
- **index**: Integer, index of layer.

Returns

A layer instance.

Raises

- **ValueError**: In case of invalid layer name or index.

Getting started with the Keras Sequential model

The `Sequential` model is a linear stack of layers.

You can create a `Sequential` model by passing a list of layer instances to the constructor:

```
from keras.models import Sequential

from keras.layers import Dense, Activation

model = Sequential([

    Dense(32, input_shape=(784,)),

    Activation('relu'),

    Dense(10),

    Activation('softmax'),

])
```

You can also simply add layers via the `.add()` method:

```
model = Sequential()

model.add(Dense(32, input_dim=784))

model.add(Activation('relu'))
```

Specifying the input shape

The model needs to know what input shape it should expect. For this reason, the first layer in a `Sequential` model (and only the first, because following layers can do automatic shape inference) needs to receive information about its input shape. There are several possible ways to do this:

- Pass an `input_shape` argument to the first layer. This is a shape tuple (a tuple of integers or `None` entries, where `None` indicates that any positive integer may be expected). In `input_shape`, the batch dimension is not included.
- Some 2D layers, such as `Dense`, support the specification of their input shape via the argument `input_dim`, and some 3D temporal layers support the arguments `input_dim` and `input_length`.
- If you ever need to specify a fixed batch size for your inputs (this is useful for stateful recurrent networks), you can pass a `batch_size` argument to a layer. If you pass both `batch_size=32` and `input_shape=(6, 8)` to a layer, it will then expect every batch of inputs to have the batch shape `(32, 6, 8)`.

As such, the following snippets are strictly equivalent:

```
model = Sequential()

model.add(Dense(32, input_shape=(784,)))

model = Sequential()

model.add(Dense(32, input_dim=784))
```

Compilation

Before training a model, you need to configure the learning process, which is done via the `compile` method. It receives three arguments:

- An optimizer. This could be the string identifier of an existing optimizer (such as `rmsprop` or `adagrad`), or an instance of the `Optimizer` class. See: optimizers.
- A loss function. This is the objective that the model will try to minimize. It can be the string identifier of an existing loss function (such as `categorical_crossentropy` or `mse`), or it can be an objective function. See: losses.
- A list of metrics. For any classification problem you will want to set this to `metrics=['accuracy']`. A metric could be the string identifier of an existing metric or a custom metric function. See: metrics.

```
# For a multi-class classification problem

model.compile(optimizer='rmsprop',

              loss='categorical_crossentropy',

              metrics=['accuracy'])



# For a binary classification problem

model.compile(optimizer='rmsprop',

              loss='binary_crossentropy',

              metrics=['accuracy'])



# For a mean squared error regression problem

model.compile(optimizer='rmsprop',

              loss='mse')



# For custom metrics

import keras.backend as K



def mean_pred(y_true, y_pred):

    return K.mean(y_pred)



model.compile(optimizer='rmsprop',

              loss='binary_crossentropy',

              metrics=['accuracy', mean_pred])
```

Training

Keras models are trained on Numpy arrays of input data and labels. For training a model, you will typically use the `fit` function. Read its documentation here.

```
# For a single-input model with 2 classes (binary classification):

model = Sequential()

model.add(Dense(32, activation='relu', input_dim=100))

model.add(Dense(1, activation='sigmoid'))

model.compile(optimizer='rmsprop',

              loss='binary_crossentropy',

              metrics=['accuracy'])

# Generate dummy data

import numpy as np

data = np.random.random((1000, 100))

labels = np.random.randint(2, size=(1000, 1))



# Train the model, iterating on the data in batches of 32 samples

model.fit(data, labels, epochs=10, batch_size=32)

# For a single-input model with 10 classes (categorical classification):

model = Sequential()

model.add(Dense(32, activation='relu', input_dim=100))

model.add(Dense(10, activation='softmax'))

model.compile(optimizer='rmsprop',

              loss='categorical_crossentropy',

              metrics=['accuracy'])

# Generate dummy data

import numpy as np

data = np.random.random((1000, 100))

labels = np.random.randint(10, size=(1000, 1))

# Convert labels to categorical one-hot encoding

one_hot_labels = keras.utils.to_categorical(labels, num_classes=10)

# Train the model, iterating on the data in batches of 32 samples

model.fit(data, one_hot_labels, epochs=10, batch_size=32)
```

Examples

Here are a few examples to get you started!

In the examples folder, you will also find example models for real datasets:

- CIFAR10 small images classification: Convolutional Neural Network (CNN) with realtime data augmentation
- IMDB movie review sentiment classification: LSTM over sequences of words
- Reuters newswires topic classification: Multilayer Perceptron (MLP)
- MNIST handwritten digits classification: MLP & CNN
- Character-level text generation with LSTM

...and more.

Multilayer Perceptron (MLP) for multi-class softmax classification:

```
import keras

from keras.models import Sequential

from keras.layers import Dense, Dropout, Activation

from keras.optimizers import SGD



# Generate dummy data

import numpy as np

x_train = np.random.random((1000, 20))

y_train = keras.utils.to_categorical(np.random.randint(10, size=(1000, 1)), num_classes=10)

x_test = np.random.random((100, 20))

y_test = keras.utils.to_categorical(np.random.randint(10, size=(100, 1)), num_classes=10)



model = Sequential()

# Dense(64) is a fully-connected layer with 64 hidden units.

# in the first layer, you must specify the expected input data shape:

# here, 20-dimensional vectors.

model.add(Dense(64, activation='relu', input_dim=20))

model.add(Dropout(0.5))

model.add(Dense(64, activation='relu'))

model.add(Dropout(0.5))

model.add(Dense(10, activation='softmax'))



sgd = SGD(lr=0.01, decay=1e-6, momentum=0.9, nesterov=True)

model.compile(loss='categorical_crossentropy',

              optimizer=sgd,

              metrics=['accuracy'])
```

```
model.fit(x_train, y_train,

          epochs=20,

          batch_size=128)

score = model.evaluate(x_test, y_test, batch_size=128)
```

MLP for binary classification:

```
import numpy as np

from keras.models import Sequential

from keras.layers import Dense, Dropout


# Generate dummy data

x_train = np.random.random((1000, 20))

y_train = np.random.randint(2, size=(1000, 1))

x_test = np.random.random((100, 20))

y_test = np.random.randint(2, size=(100, 1))


model = Sequential()

model.add(Dense(64, input_dim=20, activation='relu'))

model.add(Dropout(0.5))

model.add(Dense(64, activation='relu'))

model.add(Dropout(0.5))

model.add(Dense(1, activation='sigmoid'))


model.compile(loss='binary_crossentropy',

              optimizer='rmsprop',

              metrics=['accuracy'])


model.fit(x_train, y_train,

          epochs=20,

          batch_size=128)

score = model.evaluate(x_test, y_test, batch_size=128)
```

VGG-like convnet:

```
import numpy as np

import keras

from keras.models import Sequential
from keras.layers import Dense, Dropout, Flatten

from keras.layers import Conv2D, MaxPooling2D

from keras.optimizers import SGD


# Generate dummy data

x_train = np.random.random((100, 100, 100, 3))

y_train = keras.utils.to_categorical(np.random.randint(10, size=(100, 1)), num_classes=10)

x_test = np.random.random((20, 100, 100, 3))

y_test = keras.utils.to_categorical(np.random.randint(10, size=(20, 1)), num_classes=10)


model = Sequential()

# input: 100x100 images with 3 channels -> (100, 100, 3) tensors.

# this applies 32 convolution filters of size 3x3 each.

model.add(Conv2D(32, (3, 3), activation='relu', input_shape=(100, 100, 3)))

model.add(Conv2D(32, (3, 3), activation='relu'))

model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Dropout(0.25))


model.add(Conv2D(64, (3, 3), activation='relu'))

model.add(Conv2D(64, (3, 3), activation='relu'))

model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Dropout(0.25))


model.add(Flatten())

model.add(Dense(256, activation='relu'))

model.add(Dropout(0.5))

model.add(Dense(10, activation='softmax'))


sgd = SGD(lr=0.01, decay=1e-6, momentum=0.9, nesterov=True)

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


model.fit(x_train, y_train, batch_size=32, epochs=10)

score = model.evaluate(x_test, y_test, batch_size=32)
```

Sequence classification with LSTM:

```
from keras.models import Sequential
from keras.layers import Dense, Dropout

from keras.layers import Embedding

from keras.layers import LSTM


max_features = 1024


model = Sequential()

model.add(Embedding(max_features, output_dim=256))

model.add(LSTM(128))

model.add(Dropout(0.5))

model.add(Dense(1, activation='sigmoid'))


model.compile(loss='binary_crossentropy',

              optimizer='rmsprop',

              metrics=['accuracy'])


model.fit(x_train, y_train, batch_size=16, epochs=10)

score = model.evaluate(x_test, y_test, batch_size=16)
```

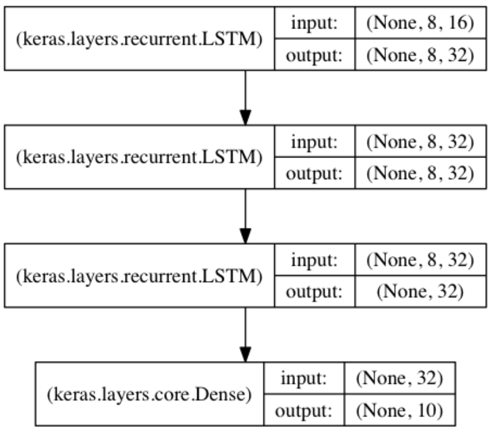
Sequence classification with 1D convolutions:

```
from keras.models import Sequential

from keras.layers import Dense, Dropout

from keras.layers import Embedding

from keras.layers import Conv1D, GlobalAveragePooling1D, MaxPooling1D


seq_length = 64


model = Sequential()

model.add(Conv1D(64, 3, activation='relu', input_shape=(seq_length, 100)))

model.add(Conv1D(64, 3, activation='relu'))

model.add(MaxPooling1D(3))

model.add(Conv1D(128, 3, activation='relu'))

model.add(Conv1D(128, 3, activation='relu'))

model.add(GlobalAveragePooling1D())

model.add(Dropout(0.5))

model.add(Dense(1, activation='sigmoid'))
model.compile(loss='binary_crossentropy',

              optimizer='rmsprop',

              metrics=['accuracy'])


model.fit(x_train, y_train, batch_size=16, epochs=10)

score = model.evaluate(x_test, y_test, batch_size=16)
```

Stacked LSTM for sequence classification

In this model, we stack 3 LSTM layers on top of each other, making the model capable of learning higher-level temporal representations.

The first two LSTMs return their full output sequences, but the last one only returns the last step in its output sequence, thus dropping the temporal dimension (i.e. converting the input sequence into a single vector).



```
from keras.models import Sequential

from keras.layers import LSTM, Dense

import numpy as np


data_dim = 16

timesteps = 8

num_classes = 10


# expected input data shape: (batch_size, timesteps, data_dim)

model = Sequential()

model.add(LSTM(32, return_sequences=True,

               input_shape=(timesteps, data_dim)))  # returns a sequence of vectors of dimension 32

model.add(LSTM(32, return_sequences=True))  # returns a sequence of vectors of dimension 32

model.add(LSTM(32))  # return a single vector of dimension 32
model.add(Dense(10, activation='softmax'))


model.compile(loss='categorical_crossentropy',
              optimizer='rmsprop',
              metrics=['accuracy'])


# Generate dummy training data
x_train = np.random.random((1000, timesteps, data_dim))
y_train = np.random.random((1000, num_classes))


# Generate dummy validation data
x_val = np.random.random((100, timesteps, data_dim))
y_val = np.random.random((100, num_classes))


model.fit(x_train, y_train,
          batch_size=64, epochs=5,
          validation_data=(x_val, y_val))
```

Same stacked LSTM model, rendered "stateful"

A stateful recurrent model is one for which the internal states (memories) obtained after processing a batch of samples are reused as initial states for the samples of the next batch. This allows to process longer sequences while keeping computational complexity manageable.

You can read more about stateful RNNs in the FAQ.

```
from keras.models import Sequential
from keras.layers import LSTM, Dense
import numpy as np


data_dim = 16
timesteps = 8
num_classes = 10
batch_size = 32


# Expected input batch shape: (batch_size, timesteps, data_dim)
# Note that we have to provide the full batch_input_shape since the network is stateful.
# the sample of index i in batch k is the follow-up for the sample i in batch k-1.
model = Sequential()
```

```
model.add(LSTM(32, return_sequences=True, stateful=True,

               batch_input_shape=(batch_size, timesteps, data_dim)))

model.add(LSTM(32, return_sequences=True, stateful=True))

model.add(LSTM(32, stateful=True))

model.add(Dense(10, activation='softmax'))


model.compile(loss='categorical_crossentropy',

              optimizer='rmsprop',

              metrics=['accuracy'])


# Generate dummy training data

x_train = np.random.random((batch_size * 10, timesteps, data_dim))

y_train = np.random.random((batch_size * 10, num_classes))


# Generate dummy validation data

x_val = np.random.random((batch_size * 3, timesteps, data_dim))

y_val = np.random.random((batch_size * 3, num_classes))


model.fit(x_train, y_train,

          batch_size=batch_size, epochs=5, shuffle=False,

          validation_data=(x_val, y_val))
```

Peer-graded Assignment: Build a Regression Model in Keras

A. Build a baseline model **(5 marks)**

Use the Keras library to build a neural network with the following:

- One hidden layer of 10 nodes, and a ReLU activation function

- Use the **adam** optimizer and the **mean squared error** as the loss function.

1. Randomly split the data into a training and test sets by holding 30% of the data for testing. You can use the **train_test_split** helper function from Scikit-learn.

2. Train the model on the training data using **50 epochs**.

3. Evaluate the model on the test data and compute the mean squared error between the predicted concrete strength and the actual concrete strength. You can use the **mean_squared_error** function from Scikit-learn.

4. Repeat steps 1 - 3, **50 times**, i.e., create a list of **50** mean squared errors.

5. Report the **mean and the standard deviation of the mean squared errors.**

Submit your Jupyter Notebook with your code and comments.

B. Normalize the data **(5 marks)**

Repeat Part A **but use a normalized version of the data**. Recall that one way to normalize the data is by subtracting the mean from the individual predictors and dividing by the standard deviation.

How does the mean of the mean squared errors compare to that from Step A?

Upload File
C. Increate the number of epochs **(5 marks)**

Repeat Part B **but use 100 epochs this time for training**.

How does the mean of the mean squared errors compare to that from Step B?

Upload File
D. Increase the number of hidden layers **(5 marks)**

Repeat part B but use a neural network with the following instead:

- Three hidden layers, each of 10 nodes and ReLU activation function.

How does the mean of the mean squared errors compare to that from Step B?

Peer Review Final Assignment

Part 1

In this part, you will design a classifier using the VGG16 pre-trained model. Just like the ResNet50 model, you can import the model `VGG16` from `keras.applications`.
You will essentially build your classifier as follows:

1. Import libraries, modules, and packages you will need. Make sure to import the *preprocess_input* function from `keras.applications.vgg16`.
2. Use a batch size of 100 images for both training and validation.
3. Construct an ImageDataGenerator for the training set and another one for the validation set. VGG16 was originally trained on 224 × 224 images, so make sure to address that when defining the ImageDataGenerator instances.
4. Create a sequential model using Keras. Add VGG16 model to it and dense layer.
5. Compile the mode using the adam optimizer and the categorical_crossentropy loss function.
6. Fit the model on the augmented data using the ImageDataGenerators.

Use the following cells to create your classifier.

Part 2

In this part, you will evaluate your deep learning models on a test data. For this part, you will need to do the following:

1. Load your saved model that was built using the ResNet50 model.
2. Construct an ImageDataGenerator for the test set. For this ImageDataGenerator instance, you only need to pass the directory of the test images, target size, and the **shuffle** parameter and set it to False.
3. Use the **evaluate_generator** method to evaluate your models on the test data, by passing the above ImageDataGenerator as an argument. You can learn more about **evaluate_generator** here.
4. Print the performance of the classifier using the VGG16 pre-trained model.
5. Print the performance of the classifier using the ResNet pre-trained model.

Use the following cells to evaluate your models.

Part 3

In this model, you will predict whether the images in the test data are images of cracked concrete or not. You will do the following:

1. Use the **predict_generator** method to predict the class of the images in the test data, by passing the test data ImageDataGenerator instance defined in the previous part as an argument. You can learn more about the **predict_generator** method here.
2. Report the class predictions of the first five images in the test set. You should print something list this:

- Positive
- Negative
- Positive
- Positive
- Negative