Model class API

In the functional API, given an input tensor and output tensor, you can instantiate a `Model` via:

```
from keras.models import Model
from keras.layers import Input, Dense

a = Input(shape=(32,))
b = Dense(32)(a)
model = Model(input=a, output=b)
```

This model will include all layers required in the computation of `b` given `a`.

In the case of multi-input or multi-output models, you can use lists as well:

```
model = Model(input=[a1, a2], output=[b1, b3, b3])
```

For a detailed introduction of what `Model` can do, read this guide to the Keras functional API.

Useful attributes of Model

- `model.layers` is a flattened list of the layers comprising the model graph.
- `model.inputs` is the list of input tensors.
- `model.outputs` is the list of output tensors.

Methods

compile

```
compile(self, optimizer, loss, metrics=[], loss_weights=None, sample_weight_mode=None)
```

Configures the model for training.

Arguments

- **optimizer**: str (name of optimizer) or optimizer object. See optimizers.
- **loss**: str (name of objective function) or objective function. See objectives. If the model has multiple outputs, you can use a different loss on each output by passing a dictionary or a list of objectives.
- **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'}`.

- **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.
- **kwargs**: when using the Theano backend, these arguments are passed into K.function. Ignored for Tensorflow backend.

---

fit

```
fit(self, x, y, batch_size=32, nb_epoch=10, verbose=1, callbacks=[], validation_split=0.0, vali
```

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

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.
- **batch_size**: integer. Number of samples per gradient update.
- **nb_epoch**: integer, the number of times to iterate over the training data arrays.
- **verbose**: 0, 1, or 2. Verbosity mode. 0 = silent, 1 = verbose, 2 = one log line per epoch.
- **callbacks**: list of callbacks to be called during training. 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.
- **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. This could be a tuple (x_val, y_val) or a tuple (val_x, val_y, val_sample_weights).
- **shuffle**: boolean, whether to shuffle the training data before each epoch.
- **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.
- **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().

Returns

A `History` instance. Its `history` attribute contains all information collected during training.

---

evaluate

```
evaluate(self, x, y, batch_size=32, verbose=1, sample_weight=None)
```

Returns the loss value and metrics values for the model in test mode. Computation is done in batches.

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.
- **batch_size**: integer. Number of samples per gradient update.

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(self, x, batch_size=32, verbose=0)
```

Generates output predictions for the input samples, processing the samples in a batched way.

Arguments

- **x**: the input data, as a Numpy array (or list of Numpy arrays if the model has multiple outputs).
- **batch_size**: integer.
- **verbose**: verbosity mode, 0 or 1.

Returns

A Numpy array of predictions.

---

train_on_batch

```
train_on_batch(self, x, y, sample_weight=None, class_weight=None)
```

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.

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(self, x, y, sample_weight=None)
```

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().

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(self, x)
```

Returns predictions for a single batch of samples.

---

fit_generator

```
fit_generator(self, generator, samples_per_epoch, nb_epoch, verbose=1, callbacks=[], validation
```

Fits the model on data generated batch-by-batch by a Python generator. 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.

Arguments

- **generator**: a generator. The output of the generator must be either
  - a tuple (inputs, targets)
  - a tuple (inputs, targets, sample_weights). All arrays should contain the same number of samples. The generator is expected to loop over its data indefinitely. An epoch finishes when `samples_per_epoch` samples have been seen by the model.
- **samples_per_epoch**: integer, number of samples to process before going to the next epoch.
- **nb_epoch**: integer, total number of iterations on the data.
- **verbose**: verbosity mode, 0, 1, or 2.
- **callbacks**: list of callbacks to be called during training.
- **validation_data**: this can be either
  - a generator for the validation data
  - a tuple (inputs, targets)
  - a tuple (inputs, targets, sample_weights).
- **nb_val_samples**: only relevant if `validation_data` is a generator. number of samples to use from validation generator at the end of every epoch.
- **class_weight**: dictionary mapping class indices to a weight for the class.
- **max_q_size**: maximum size for the generator queue
- **nb_worker**: maximum number of processes to spin up when using process based threading

- **pickle_safe**: if True, use process based threading. Note that because this implementation relies on multiprocessing, you should not pass non non picklable arguments to the generator as they can't be passed easily to children processes.

Returns

A `History` object.

Example

```
def generate_arrays_from_file(path):
    while 1:
    f = open(path)
    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})
    f.close()

model.fit_generator(generate_arrays_from_file('/my_file.txt'),
        samples_per_epoch=10000, nb_epoch=10)
```

---

evaluate_generator

```
evaluate_generator(self, generator, val_samples, max_q_size=10, nb_worker=1, pickle_safe=False)
```

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)
- **val_samples**: total number of samples to generate from `generator` before returning.
- **max_q_size**: maximum size for the generator queue
- **nb_worker**: maximum number of processes to spin up when using process based threading
- **pickle_safe**: if True, use process based threading. Note that because this implementation relies on multiprocessing, you should not pass non non picklable arguments to the generator as they can't be passed easily to children processes.

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.

get_layer

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

Returns a layer based on either its name (unique) or its index in the graph. 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.