The Sequential model API

To get started, read this guide to the Keras Sequential model.

Useful attributes of Model

- `model.layers` is a list of the layers added to the model.

Sequential model methods

compile

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

Configures the learning process.

Arguments

- **optimizer**: str (name of optimizer) or optimizer object. See optimizers.
- **loss**: str (name of objective function) or objective function. See objectives.
- **metrics**: list of metrics to be evaluated by the model during training and testing. Typically you will use `metrics=['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).
- **kwargs**: for Theano backend, these are passed into K.function. Ignored for Tensorflow backend.

Example

```
    model = Sequential()
    model.add(Dense(32, input_shape=(500,)))
    model.add(Dense(10, activation='softmax'))
    model.compile(optimizer='rmsprop',
          loss='categorical_crossentropy',
          metrics=['accuracy'])
```

fit

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

Trains the model for a fixed number of epochs.

Arguments

- **x**: input data, as a Numpy array or list of Numpy arrays (if the model has multiple inputs).
- **y**: labels, as a Numpy array.
- **batch_size**: integer. Number of samples per gradient update.
- **nb_epoch**: integer, the number of epochs to train the model.
- **verbose**: 0 for no logging to stdout, 1 for progress bar logging, 2 for one log line per epoch.
- **callbacks**: list of `keras.callbacks.Callback` instances. List of callbacks to apply during training. See callbacks.
- **validation_split**: float (0. < x < 1). Fraction of the data to use as held-out validation data.
- **validation_data**: tuple (X, y) to be used as held-out validation data. Will override validation_split.
- **shuffle**: boolean or str (for 'batch'). Whether to shuffle the samples at each epoch. 'batch' is a special option for dealing with the limitations of HDF5 data; it shuffles in batch-sized chunks.
- **class_weight**: dictionary mapping classes to a weight value, used for scaling the loss function (during training only).
- **sample_weight**: Numpy array of weights for the training samples, used for scaling 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().

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).

---

evaluate

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

Computes the loss on some input data, batch by batch.

Arguments

- **x**: input data, as a Numpy array or list of Numpy arrays (if the model has multiple inputs).
- **y**: labels, as a Numpy array.
- **batch_size**: integer. Number of samples per gradient update.
- **verbose**: verbosity mode, 0 or 1.
- **sample_weight**: sample weights, as a Numpy array.

Returns

Scalar test loss (if the model has no metrics) or list of scalars (if the model computes other 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.
- **batch_size**: integer.
- **verbose**: verbosity mode, 0 or 1.

Returns

A Numpy array of predictions.

---

predict_classes

```
predict_classes(self, x, batch_size=32, verbose=1)
```

Generate class predictions for the input samples batch by batch.

Arguments

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

Returns

A numpy array of class predictions.

---

predict_proba

```
predict_proba(self, x, batch_size=32, verbose=1)
```

Generates class probability predictions for the input samples batch by batch.

Arguments

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

Returns

A Numpy array of probability predictions.

---

train_on_batch

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

Single gradient update over one batch of samples.

Arguments

- **x**: input data, as a Numpy array or list of Numpy arrays (if the model has multiple inputs).
- **y**: labels, as a Numpy array.
- **class_weight**: dictionary mapping classes to a weight value, used for scaling the loss function (during training only).
- **sample_weight**: sample weights, as a Numpy array.

Returns

Scalar training loss (if the model has no metrics) or list of scalars (if the model computes other 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)
```

Evaluates the model over a single batch of samples.

Arguments

- **x**: input data, as a Numpy array or list of Numpy arrays (if the model has multiple inputs).
- **y**: labels, as a Numpy array.
- **sample_weight**: sample weights, as a Numpy array.

Returns

Scalar test loss (if the model has no metrics) or list of scalars (if the model computes other 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_data=
```

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
- **pickle_safe**: 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.

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
        x, y = process_line(line)
        yield (x, 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
- **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.

---

predict_generator

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

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.
- **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
- **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 Numpy array of predictions.