Docs » コールバック

コールバックの使い方

コールバックは訓練中で適用される関数集合です．訓練中にモデル内部の状態と統計量を可視化する際に，コールバックを使います． `Sequential` と `Model` クラスの `.fit()` メソッドに（キーワード引数 `callbacks` として）コールバックのリストを渡すことができます．コールバックに関連するメソッドは，訓練の各段階で呼び出されます．

Callback [source]

```
keras.callbacks.Callback()
```

この抽象基底クラスは新しいコールバックを構築するために使用されます．

プロパティ

- **params**: 辞書．訓練のパラメータ（例: 冗長性，バッチサイズ，エポック数...）．
- **model**: `keras.models.Model` のインスタンス．学習されたモデルへの参照．

コールバック関数が引数としてとる辞書の `logs` は，現在のバッチ数かエポック数に関連したデータのキーを含みます．

現在， `Sequential` モデルクラスの `.fit()` メソッドは，そのコールバックに渡す `logs` に以下のデータが含まれます．

- **on_epoch_end**: ログは `acc` と `loss` を含み，オプションとして（ `fit` 内のバリデーションが有効になっている場合は） `val_loss` ，（バリデーションと精度の監視が有効になっている場合は） `val_acc` を含みます．
- **on_batch_begin**: ログは現在のバッチのサンプル数 `size` を含みます．
- **on_batch_end**: ログは `loss` と（精度の監視が有効になっている場合は）オプションとして `acc` を含みます．

BaseLogger [source]

```
keras.callbacks.BaseLogger()
```

監視されている評価値のエポック平均を蓄積するコールバックです．

このコールバックは全Kerasモデルに自動的に適用されます．

TerminateOnNaN [source]

```
keras.callbacks.TerminateOnNaN()
```

損失がNaNになった時に訓練を終了するコールバックです．

ProgbarLogger [source]

```
keras.callbacks.ProgbarLogger(count_mode='samples')
```

標準出力に評価値を出力するコールバックです．

引数

- **count_mode**: "steps"か"samples"の一方．サンプルかステップ（バッチ）のどちらをプログレスバーの集計に使うか．

Raises

- **ValueError**: `count_mode` の値が不正のとき．

History [source]

```
keras.callbacks.History()
```

`History` オブジェクトにイベントを記録するコールバックです．

このコールバックは全Kerasモデルに自動的に適用されます． `History` オブジェクトはモデルの `fit` メソッドの戻り値として取得します．

ModelCheckpoint [source]

```
keras.callbacks.ModelCheckpoint(filepath, monitor='val_loss', verbose=0, save_best_only=Fals
```

各エポック終了後にモデルを保存します．

`filepath` は，（ `on_epoch_end` で渡された） `epoch` の値と `logs` のキーで埋められた書式設定オプションを含むことができます．

例えば， `filepath` が `weights.{epoch:02d}-{val_loss:.2f}.hdf5` の場合，複数のファイルがエポック数とバリデーションロスの値を付与して保存されます．

引数

- **filepath**: 文字列，モデルファイルを保存するパス.
- **monitor**: 監視する値.
- **verbose**: 冗長モード, 0 または 1.
- **save_best_only**: `save_best_only=True` の場合，監視しているデータによって最新の最良モデルが上書きされません.
- **mode**: {auto, min, max}の内の一つが選択されます． `save_best_only=True` ならば，現在保存されているファイルを上書きするかは，監視されている値の最大化か最小化によって決定されます． `val_acc` の場合，この引数は `max` となり， `val_loss` の場合は `min` になります． `auto` モードでは，この傾向は自動的に監視されている値から推定します.
- **save_weights_only**: Trueなら，モデルの重みが保存されます (`model.save_weights(filepath)`)，そうでないなら，モデルの全体が保存されます (`model.save(filepath)`).
- **period**: チェックポイント間の間隔（エポック数）.

EarlyStopping

[source]

```
keras.callbacks.EarlyStopping(monitor='val_loss', min_delta=0, patience=0, verbose=0, mode='
```

監視する値の変化が停止した時に訓練を終了します.

引数

- **monitor**: 監視する値.
- **min_delta**: 監視する値について改善として判定される最小変化値．つまり，min_deltaよりも絶対値の変化が小さければ改善していないとみなします.
- **patience**: 訓練が停止し，値が改善しなくなってからのエポック数.
- **verbose**: 冗長モード.
- **mode**: {auto, min, max}の内，一つが選択されます． `min` モードでは，監視する値の減少が停止した際に，訓練を終了します．また， `max` モードでは，監視する値の増加が停止した際に，訓練を終了します． `auto` モードでは，この傾向は自動的に監視されている値から推定します.

RemoteMonitor

[source]

```
keras.callbacks.RemoteMonitor(root='http://localhost:9000', path='/publish/epoch/end/', fiel
```

このコールバックはサーバーにイベントをストリームするときに使用されます.

`requests` ライブラリが必要です．イベントはデフォルトで `root + '/publish/epoch/end/'` に送信されます． コールすることによって，イベントデータをJSONエンコードした辞書の `data` 引数をHTTP POSTされます.

引数

- **root**: 文字列；対象サーバのルートURL．
- **path**: 文字列；イベントを送る `root` への相対パス．
- **field**: 文字列；データを保存するJSONのフィールド．
- **headers**: 辞書; オプションでカスタムできるHTTPヘッダー．

LearningRateScheduler [source]

```
keras.callbacks.LearningRateScheduler(schedule, verbose=0)
```

学習率のスケジューラ．

引数

- **schedule**: この関数はエポックのインデックス（整数, 0から始まるインデックス）を入力とし，新しい学習率（浮動小数点数）を返します．
- **verbose**: 整数．0:：何も表示しない．1：更新メッセージを表示．

TensorBoard [source]

```
keras.callbacks.TensorBoard(log_dir='./logs', histogram_freq=0, batch_size=32, write_graph=T
```

Tensorboardによる基本的な可視化．

TensorBoardはTensorFlowによって提供されている可視化ツールです

このコールバックはTensorBoardのログを出力します．TensorBoardでは，異なる層への活性化ヒストグラムと同様に，訓練とテストの評価値を動的にグラフ化し，可視化できます．

pipからTensorFlowをインストールしているならば，コマンドラインからTensorBoardを起動できます．

```
tensorboard --logdir=/full_path_to_your_logs
```

引数

- **log_dir**: TensorfBoardによって解析されたログファイルを保存するディレクトリのパス
- **histogram_freq**: モデルの層の活性化ヒストグラムを計算する（エポック中の）頻度．この値を0に設定するとヒストグラムが計算されません．ヒストグラムの可視化にはバリデーションデータを指定しておく必要があります．
- **write_graph**: TensorBoardのグラフを可視化するか． `write_graph` がTrueの場合，ログファイルが非常に大きくなることがあります．
- **write_grads**: TensorBoardに勾配のヒストグラフを可視化するかどうか． `histogram_freq` は0より大きくしなければなりません．

- **batch_size**: ヒストグラム計算のネットワークに渡す入力のバッチサイズ.
- **write_images**: TensorfBoardで可視化するモデルの重みを画像として書き出すかどうか.
- **embeddings_freq**: 選択したembeddingsレイヤーを保存する（エポックに対する）頻度.
- **embeddings_layer_names**: 観察するレイヤー名のリスト．もしNoneか空リストなら全embeddingsレイヤーを観察します.
- **embeddings_metadata**: レイヤー名からembeddingsレイヤーに関するメタデータの保存ファイル名へマップする辞書． メタデータのファイルフォーマットの詳細． 全embeddingsレイヤーに対して同じメタデータファイルを使う場合は文字列を渡します.

ReduceLROnPlateau

[source]

```
keras.callbacks.ReduceLROnPlateau(monitor='val_loss', factor=0.1, patience=10, verbose=0, mo
```

評価値の改善が止まった時に学習率を減らします.

モデルは訓練が停滞した時に学習率を2〜10で割ることで恩恵を受けることがあります． このコールバックは評価値を監視し，'patience'で指定されたエポック数の間改善が見られなかった場合，学習率を減らします.

例

```
reduce_lr = ReduceLROnPlateau(monitor='val_loss', factor=0.2,
                              patience=5, min_lr=0.001)
model.fit(X_train, Y_train, callbacks=[reduce_lr])
```

引数

- **monitor**: 監視する値.
- **factor**: 学習率を減らす割合． new_lr = lr * factor
- **patience**: 何エポック改善が見られなかったら学習率の削減を行うか.
- **verbose**: 整数． 0: 何も表示しない． 1: 学習率削減時メッセージを表示.
- **mode**: `auto`， `min`， `max` のいずれか． `min` の場合，監視する値の減少が停止した際に，学習率を更新します． `max` の場合，監視する値の増加が停止した時に，学習率を更新します． `auto` の場合，監視する値の名前から自動で判断します.
- **epsilon**: 改善があったと判断する閾値． 有意な変化だけに注目するために用います.
- **cooldown**: 学習率を減らした後，通常の学習を再開するまで待機するエポック数.
- **min_lr**: 学習率の下限.

CSVLogger

[source]

```
keras.callbacks.CSVLogger(filename, separator=',', append=False)
```

各エポックの結果をcsvファイルに保存するコールバックです.

np.ndarrayのような1次元イテラブルを含む，文字列表現可能な値をサポートしています．

例

```
csv_logger = CSVLogger('training.log')
model.fit(X_train, Y_train, callbacks=[csv_logger])
```

引数

- **filename**: csvファイル名．例えば'run/log.csv'．
- **separator**: csvファイルで各要素を区切るために用いられる文字．
- **append**: True: ファイルが存在する場合，追記します．（訓練を続ける場合に便利です） False: 既存のファイルを上書きします．

LambdaCallback

[source]

```
keras.callbacks.LambdaCallback(on_epoch_begin=None, on_epoch_end=None, on_batch_begin=None,
```

シンプルな自作コールバックを急いで作るためのコールバックです．

このコールバックは，適切なタイミングで呼び出される無名関数で構築されます． 以下のような位置引数が必要であることに注意してください:

- `on_epoch_begin` と `on_epoch_end` は2つの位置引数が必要です: `epoch` , `logs`
- `on_batch_begin` と `on_batch_end` は2つの位置引数が必要です: `batch` , `logs`
- `on_train_begin` と `on_train_end` は1つの位置引数が必要です: `logs`

引数

- **on_epoch_begin**: すべてのエポックの開始時に呼ばれます．
- **on_epoch_end**: すべてのエポックの終了時に呼ばれます．
- **on_batch_begin**: すべてのバッチの開始時に呼ばれます．
- **on_batch_end**: すべてのバッチの終了時に呼ばれます．
- **on_train_begin**: 訓練の開始時に呼ばれます．
- **on_train_end**: 訓練の終了時に呼ばれます．

例

```
# すべてのバッチの開始時にバッチ番号を表示
batch_print_callback = LambdaCallback(
    on_batch_begin=lambda batch,logs: print(batch))

# Stream the epoch loss to a file in JSON format. The file content
# is not well-formed JSON but rather has a JSON object per line.
import json
json_log = open('loss_log.json', mode='wt', buffering=1)
json_logging_callback = LambdaCallback(
    on_epoch_end=lambda epoch, logs: json_log.write(
        json.dumps({'epoch': epoch, 'loss': logs['loss']}) + '\n'),
    on_train_end=lambda logs: json_log.close()
)

# 訓練終了時にいくつかのプロセスを終了
processes = ...
cleanup_callback = LambdaCallback(
    on_train_end=lambda logs: [
        p.terminate() for p in processes if p.is_alive()])

model.fit(...,
          callbacks=[batch_print_callback,
                     json_logging_callback,
                     cleanup_callback])
```

コールバックを作成

基底クラスの `keras.callbacks.Callback` を拡張することで，カスタムコールバックを作成できます． コールバックは， `self.model` プロパティによって，関連したモデルにアクセスできます．

訓練中の各バッチの損失のリストを保存する簡単な例は，以下のようになります．

```
class LossHistory(keras.callbacks.Callback):
    def on_train_begin(self, logs={}):
        self.losses = []

    def on_batch_end(self, batch, logs={}):
        self.losses.append(logs.get('loss'))
```

例: 損失の履歴を記録する

```
class LossHistory(keras.callbacks.Callback):
    def on_train_begin(self, logs={}):
        self.losses = []

    def on_batch_end(self, batch, logs={}):
        self.losses.append(logs.get('loss'))

model = Sequential()
model.add(Dense(10, input_dim=784, kernel_initializer='uniform'))
model.add(Activation('softmax'))
model.compile(loss='categorical_crossentropy', optimizer='rmsprop')

history = LossHistory()
model.fit(x_train, y_train, batch_size=128, epochs=20, verbose=0, callbacks=[history])

print(history.losses)
# 出力
'''
[0.66047596406559383, 0.3547245744908703, ..., 0.25953155204159617, 0.25901699725311789]
'''
```

例: モデルのチェックポイント

```
from keras.callbacks import ModelCheckpoint

model = Sequential()
model.add(Dense(10, input_dim=784, kernel_initializer='uniform'))
model.add(Activation('softmax'))
model.compile(loss='categorical_crossentropy', optimizer='rmsprop')

'''
バリデーションロスが減少した場合に，各エポック終了後，モデルの重みを保存します
'''
checkpointer = ModelCheckpoint(filepath='/tmp/weights.hdf5', verbose=1, save_best_only=True)
model.fit(x_train, y_train, batch_size=128, epochs=20, verbose=0, validation_data=(X_test, Y
```