Docs » レイヤー » Coreレイヤー

Dense [source]

```
keras.layers.Dense(units, activation=None, use_bias=True, kernel_initializer='glorot_uniform
```

通常の全結合ニューラルネットワークレイヤー.

`Dense`が実行する操作：`output = activation(dot(input, kernel) + bias)`ただし，`activation`は`activation`引数として渡される要素単位の活性化関数で，`kernel`はレイヤーによって作成された重み行列であり，`bias`はレイヤーによって作成されたバイアスベクトルです.（`use_bias`が`True`の場合にのみ適用されます）.

- 注意：レイヤーへの入力のランクが2より大きい場合は，`kernel`を使用した最初のドット積の前に平坦化されます.

例

```
# as first layer in a sequential model:
model = Sequential()
model.add(Dense(32, input_shape=(16,)))
# now the model will take as input arrays of shape (*, 16)
# and output arrays of shape (*, 32)

# after the first layer, you don't need to specify
# the size of the input anymore:
model.add(Dense(32))
```

引数

- **units**：正の整数，出力空間の次元数
- **activation**：使用する活性化関数名（activationsを参照） もしあなたが何も指定しなければ，活性化は適用されない. （すなわち，"線形"活性化：`a(x) = x`）.
- **use_bias**：真理値，レイヤーがバイアスベクトルを使用するかどうか.
- **kernel_initializer**：`kernel`重み行列の初期化（initializationsを参照）
- **bias_initializer**：バイアスベクトルの初期化（initializationsを参照）
- **kernel_regularizer**：`kernel`重み行列に適用される正則化関数（regularizersを参照）
- **bias_regularizer**：バイアスベクトルに適用される正則化関数（regularizersを参照）
- **activity_regularizer**：レイヤーの出力に適用される正則化関数（"activation"）（regularizersを参照）
- **kernel_constraint**：`kernel`重み行列に適用される制約関数（constraintsを参照）
- **bias_constraint**：バイアスベクトルに適用される制約関数（constraintsを参照）

入力のshape

以下のshapeを持つn階テンソル： `(batch_size, ..., input_dim)`. 最も一般的なのは以下のshapeを持つ2階テンソル： `(batch_size, input_dim)`.

出力のshape

以下のshapeを持つn階テンソル： `(batch_size, ..., units)`. 例えば，以下のshapeを持つ2階テンソル `(batch_size, input_dim)` の入力に対して，アウトプットは以下のshapwを持つ `(batch_size, units)`.

Activation [source]

```
keras.layers.Activation(activation)
```

出力に活性化関数を適用する.

引数

- **activation**： 使用する活性化関数名 (activationsを参照)， もしくは，TheanoかTensorFlowオペレーション.

入力のshape

任意．モデルの最初のレイヤーとしてこのレイヤーを使う時，キーワード引数 `input_shape` （整数のタプルはサンプルの軸（axis）を含まない．）を使う.

出力のshape

入力と同じshape.

Dropout [source]

```
keras.layers.Dropout(rate, noise_shape=None, seed=None)
```

入力にドロップアウトを適用する.

訓練時の更新においてランダムに入力ユニットを0とする割合であり，過学習の防止に役立ちます.

引数

- **rate**： 0と1の間の浮動小数点数．入力ユニットをドロップする割合.
- **noise_shape**： 入力と乗算されるバイナリドロップアウトマスクのshapeは1階の整数テンソルで表す．例えば入力のshapeを `(batch_size, timesteps, features)` とし，ドロップアウトマスクをす

べてのタイムステップで同じにしたい場合，`noise_shape=(batch_size, 1, features)` を使うことができる．

- **seed**： random seedとして使うPythonの整数．

参考文献

- Dropout: A Simple Way to Prevent Neural Networks from Overfitting

---

Flatten

[source]

```
keras.layers.Flatten()
```

入力を平滑化する．バッチサイズに影響を与えません．

例

```
model = Sequential()
model.add(Conv2D(64, 3, 3,
                 border_mode='same',
                 input_shape=(3, 32, 32)))
# now: model.output_shape == (None, 64, 32, 32)

model.add(Flatten())
# now: model.output_shape == (None, 65536)
```

Input

[source]

```
keras.engine.topology.Input()
```

`Input()` はKerasテンソルのインスタンス化に使われます．

Kerasテンソルは下位のバックエンド（TheanoやTensorFlow，あるいはCNTK）からなるテンソルオブジェクトです． モデルの入出力がわかっていれば，Kerasのモデルを構築するためにいくつかの属性を拡張できます．

例えばa, b, cがKerasのテンソルの場合，次のようにできます：
`model = Model(input=[a, b], output=c)`

追加されたKerasの属性：

- `_keras_shape`: Keras側の推論から伝達された整数のshapeのタプル．
- `_keras_history`: テンソルに適用される最後のレイヤー．全体のレイヤーグラフはこのレイヤーから再帰的に取り出せます．

引数

- **shape**: shapeのタプル（整数）で，バッチサイズを含みません．例えば，`shape=(32,)`は期待される入力が32次元ベクトルのバッチであることを示します．
- **batch_shape**: shapeのタプル（整数）で，バッチサイズを含みます． 例えば，`batch_shape=(10, 32)`は期待される入力が10個の32次元ベクトルのバッチであることを示します． `batch_shape=(None, 32)`は任意の数の32次元ベクトルのバッチを示します．
- **name**: オプションとなるレイヤーの名前の文字列． モデルの中でユニークな値である必要があります（同じ名前は二回使えません）． 指定しなければ自動生成されます．
- **dtype**: 入力から期待されるデータの型で，文字列で指定します(`float32`, `float64`, `int32`...)．
- **sparse**: 生成されるプレースホルダをスパースにするか指定する真理値．
- **tensor**: `Input`レイヤーをラップする既存のテンソル． 設定した場合，レイヤーはプレースホルダとなるテンソルを生成しません．

戻り値

テンソル．

例

```
# this is a logistic regression in Keras
x = Input(shape=(32,))
y = Dense(16, activation='softmax')(x)
model = Model(x, y)
```

Reshape

[source]

```
keras.layers.Reshape(target_shape)
```

あるshapeに出力を変形する．

引数

- **target_shape**： ターゲットのshape．整数のタプル，サンプルの次元を含まない（バッチサイズ）．

入力のshape

入力のshapeのすべての次元は固定されなければならないが，任意． モデルの最初のレイヤーとしてこのレイヤーを使うとき，キーワード引数`input_shape`(整数のタプルはサンプルの軸を含まない．)を使う．

出力のshape

`(batch_size,) + target_shape`

例

```
# as first layer in a Sequential model
model = Sequential()
model.add(Reshape((3, 4), input_shape=(12,)))
# now: model.output_shape == (None, 3, 4)
# note: `None` is the batch dimension

# as intermediate layer in a Sequential model
model.add(Reshape((6, 2)))
# now: model.output_shape == (None, 6, 2)

# also supports shape inference using `-1` as dimension
model.add(Reshape((-1, 2, 2)))
# now: model.output_shape == (None, 3, 2, 2)
```

Permute [source]

```
keras.layers.Permute(dims)
```

与えられたパターンにより入力の次元を入れ替える．

例えば，RNNsやconvnetsの連結に対して役立ちます．

例

```
model = Sequential()
model.add(Permute((2, 1), input_shape=(10, 64)))
# now: model.output_shape == (None, 64, 10)
# note: `None` is the batch dimension
```

引数

- **dims**：整数のタプル．配列パターン，サンプルの次元を含まない．添字は1で始まる．例えば，`(2, 1)`は入力の1番目と2番目の次元を入れ替える．

入力のshape

任意. モデルの最初のレイヤーとしてこのレイヤーを使う時，キーワード引数`input_shape`（整数のタプルはサンプルの軸を含まない）を使う．

出力のshape

入力のshapeと同じだが，特定のパターンにより並べ替えられた次元を持つ．

RepeatVector [source]

```
keras.layers.RepeatVector(n)
```

n回入力を繰り返す．

例

```
model = Sequential()
model.add(Dense(32, input_dim=32))
# now: model.output_shape == (None, 32)
# note: `None` is the batch dimension

model.add(RepeatVector(3))
# now: model.output_shape == (None, 3, 32)
```

引数

- n：整数，繰り返し因数.

入力のshape

`(num_samples, features)` のshapeを持つ2階テンソル.

出力のshape

`(num_samples, n, features)` のshapeを持つ3階テンソル.

Lambda [source]

```
keras.layers.Lambda(function, output_shape=None, mask=None, arguments=None)
```

`Layer` オブジェクトのように，任意の式をラップする.

例

```
# add a x -> x^2 layer
model.add(Lambda(lambda x: x ** 2))
```

```
# add a layer that returns the concatenation
# of the positive part of the input and
# the opposite of the negative part

def antirectifier(x):
    x -= K.mean(x, axis=1, keepdims=True)
    x = K.l2_normalize(x, axis=1)
    pos = K.relu(x)
    neg = K.relu(-x)
    return K.concatenate([pos, neg], axis=1)

def antirectifier_output_shape(input_shape):
    shape = list(input_shape)
    assert len(shape) == 2  # only valid for 2D tensors
    shape[-1] *= 2
    return tuple(shape)

model.add(Lambda(antirectifier,
                 output_shape=antirectifier_output_shape))
```

引数

- **function**： 評価される関数．第1引数としてインプットテンソルを取る
- **output_shape**： 関数からの期待される出力のshape．Theanoを使用する場合のみ関連します．タプルもしくは関数． タプルなら，入力に近いほうの次元だけを指定する，データサンプルの次元は入力と同じ： `output_shape = (input_shape[0], ) + output_shape` か入力が `None` でかつサンプル次元も `None` ： `output_shape = (None, ) + output_shape` のどちらかが推測される． 関数なら，入力のshapeの関数としてshape全体を指定する： `output_shape = f(input_shape)`
- **arguments**： 関数に通されるキーワード引数の追加辞書

入力のshape

任意．モデルの最初のレイヤーとしてこのレイヤーを使う時，キーワード引数 `input_shape` （整数のタプル，サンプルの軸（axis）を含まない）を使う．

出力のshape

`output_shape` 引数によって特定される（TensorFlowを使用していると自動推論される）．

ActivityRegularization [source]

```
keras.layers.ActivityRegularization(l1=0.0, l2=0.0)
```

コスト関数に基づく入力アクティビティに更新を適用するレイヤー

引数

- **l1**： L1正則化係数（正の浮動小数点数）．
- **l2**： L2正則化係数（正の浮動小数点数）．

入力のshape

任意．モデルの最初のレイヤーとしてこのレイヤーを使う時，キーワード引数 `input_shape` （整数のタプル，サンプルの軸（axis）を含まない）を使う．

出力のshape

入力と同じshape．

Masking [source]

```
keras.layers.Masking(mask_value=0.0)
```

タイプステップをスキップするためのマスク値を用いてシーケンスをマスクします．

入力テンソル（テンソルの次元 #1）のそれぞれのタイムステップに対して，もしそのタイムステップの入力テンソルのすべての値が `mask_value` に等しいなら，そのときそのタイムステップはすべての下流レイヤー（それらがマスキングをサポートしている限り）でマスク（スキップ）されるでしょう．

下流レイヤーがマスキングをサポートしていないのにそのような入力マスクを受け取ると例外が発生します．

例

LSTMレイヤーに与えるための `(samples, timesteps, features)` のshapeを持つのNumpy配列 `x` を考えてみましょう． あなたが#3と#5のタイムステップに関してデータを欠損しているので，これらのタイムステップをマスクしたい場合，あなたは以下のようにできます：

- `x[:, 3, :] = 0.` と `x[:, 5, :] = 0.` をセットする．
- LSTMレイヤーの前に `mask_value=0.` の `Masking` レイヤーを追加する：

```
model = Sequential()
model.add(Masking(mask_value=0., input_shape=(timesteps, features)))
model.add(LSTM(32))
```