Introduction to Keras

Sources: keras.io, blog.keras.io, github.com/fchollet/keras

Branch: master ▾ keras / examples / mnist_mlp.py

Find file | Copy path

kemaswill Remove unused imports and unused variables (#4930) c0d95fd 25 days ago

3 contributors

61 lines (48 sloc) | 1.69 KB

Raw | Blame | History

```
'''Trains a simple deep NN on the MNIST dataset.

Gets to 98.40% test accuracy after 20 epochs
(there is *a lot* of margin for parameter tuning).
2 seconds per epoch on a K520 GPU.
'''

from __future__ import print_function
import numpy as np
np.random.seed(1337)  # for reproducibility

from keras.datasets import mnist
from keras.models import Sequential
from keras.layers.core import Dense, Dropout, Activation
from keras.optimizers import RMSprop
from keras.utils import np_utils


batch_size = 128
nb_classes = 10
nb_epoch = 20

```

```
8   from __future__ import print_function
9   import numpy as np
10  np.random.seed(1337)  # for reproducibility
11
12  from keras.datasets import mnist
13  from keras.models import Sequential
14  from keras.layers.core import Dense, Dropout, Activation
15  from keras.optimizers import RMSprop
16  from keras.utils import np_utils
```

```
19  batch_size = 128
20  nb_classes = 10
21  nb_epoch = 20
22
23  # the data, shuffled and split between train and test sets
24  (X_train, y_train), (X_test, y_test) = mnist.load_data()
25
26  X_train = X_train.reshape(60000, 784)
27  X_test = X_test.reshape(10000, 784)
28  X_train = X_train.astype('float32')
29  X_test = X_test.astype('float32')
30  X_train /= 255
31  X_test /= 255
32  print(X_train.shape[0], 'train samples')
33  print(X_test.shape[0], 'test samples')
```

```
35  # convert class vectors to binary class matrices
36  Y_train = np_utils.to_categorical(y_train, nb_classes)
37  Y_test = np_utils.to_categorical(y_test, nb_classes)
```

```
39    model = Sequential()
40    model.add(Dense(512, input_shape=(784,)))
41    model.add(Activation('relu'))
42    model.add(Dropout(0.2))
43    model.add(Dense(512))
44    model.add(Activation('relu'))
45    model.add(Dropout(0.2))
46    model.add(Dense(10))
47    model.add(Activation('softmax'))
```

```
49    model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_1 (Dense)              (None, 512)               401920
_________________________________________________________________
activation_1 (Activation)    (None, 512)               0
_________________________________________________________________
dropout_1 (Dropout)          (None, 512)               0
_________________________________________________________________
dense_2 (Dense)              (None, 512)               262656
_________________________________________________________________
activation_2 (Activation)    (None, 512)               0
_________________________________________________________________
dropout_2 (Dropout)          (None, 512)               0
_________________________________________________________________
dense_3 (Dense)              (None, 10)                5130
_________________________________________________________________
activation_3 (Activation)    (None, 10)                0
=================================================================
Total params: 669,706
Trainable params: 669,706
Non-trainable params: 0
```

```
51  model.compile(loss='categorical_crossentropy',
52                optimizer=RMSprop(),
53                metrics=['accuracy'])
54
55  history = model.fit(X_train, Y_train,
56                      batch_size=batch_size, nb_epoch=nb_epoch,
57                      verbose=1, validation_data=(X_test, Y_test))
58  score = model.evaluate(X_test, Y_test, verbose=0)
59  print('Test score:', score[0])
60  print('Test accuracy:', score[1])
```

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_dim=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'))
```

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

Layers

Branch: **master** ▾ **keras** / keras / **layers** /

the-moliver committed with **fchollet** change abs to K.abs (#5200) ...

..

__init__.py	Fix review
advanced_activations.py	change abs to K.abs (#5200)
convolutional.py	Further style fixes in layers.
convolutional_recurrent.py	Further style fixes in layers.
core.py	Fix custom_objects for regularizers and other issues (#5012)
embeddings.py	Further style fixes.
local.py	Further style fixes in layers.
noise.py	Further docstring fixes.
normalization.py	Further style fixes.
pooling.py	Further style fixes in layers.
recurrent.py	Further style fixes in layers.
wrappers.py	TimeDistributedDense deprecated, doc change (#5097)

keras / keras / layers / core.py

```
model.add(Flatten())
model.add(Dense(512))
model.add(Activation('relu'))
model.add(Dropout(0.5))
```

keras / keras / activations.py

- linear
- sigmoid
- tanh
- relu

keras / keras / layers / advanced_activations.py

- PReLU
- Leaky ReLU
- SReLU

keras / keras / regularizers.py

- L1 weight penalty
- L2 weight penalty

```
class L1L2Regularizer(Regularizer):
    """Regularizer for L1 and L2 regularization.

    # Arguments
        l1: Float; L1 regularization factor.
        l2: Float; L2 regularization factor.
    """

    def __init__(self, l1=0., l2=0.):
```

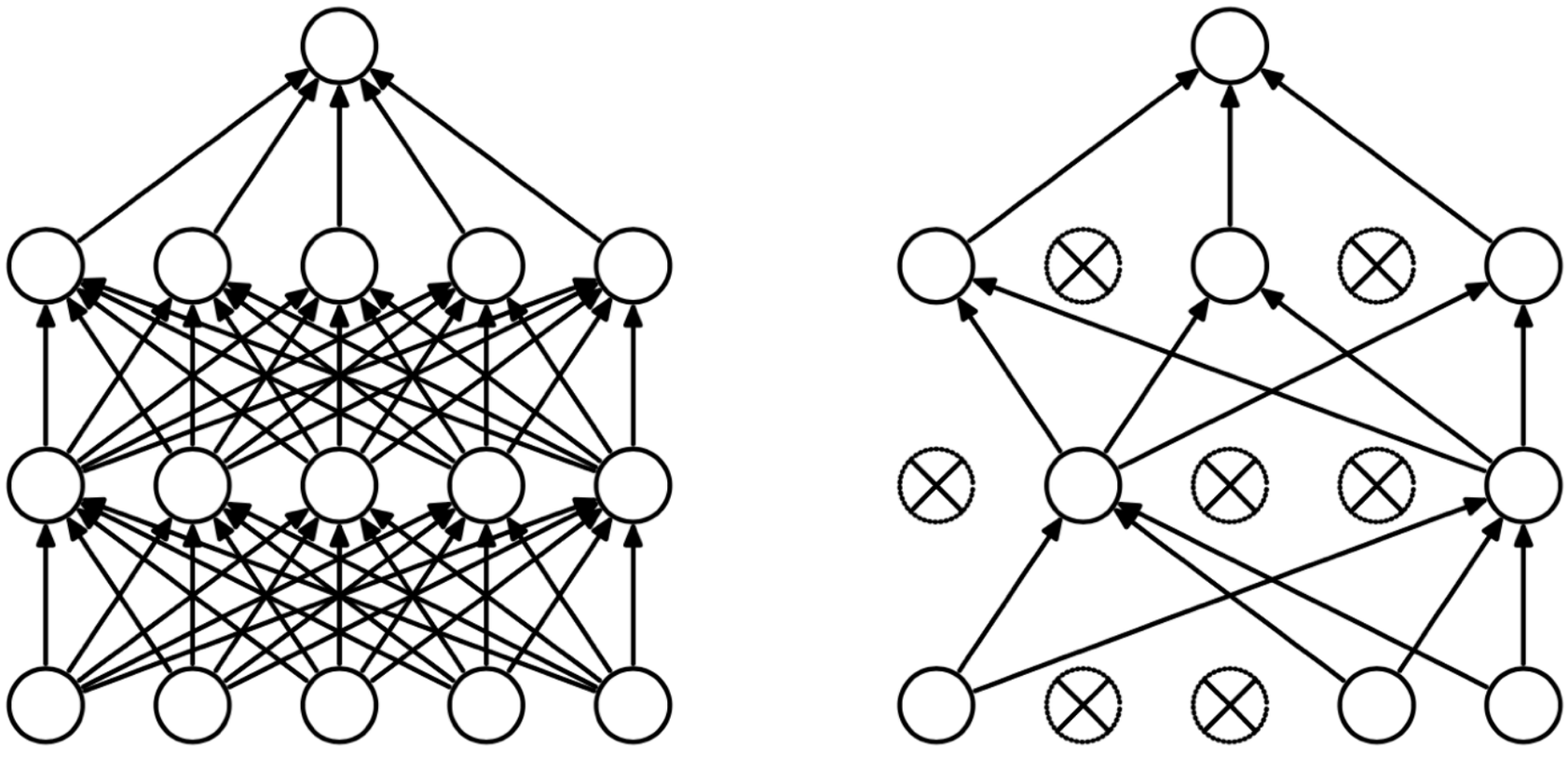
keras / keras / layers / noise.py

```
class GaussianNoise(Layer):
    """Apply additive zero-centered Gaussian noise.

    This is useful to mitigate overfitting
    (you could see it as a form of random data augmentation).
    Gaussian Noise (GS) is a natural choice as corruption process
    for real valued inputs.

    As it is a regularization layer, it is only active at training time.

    # Arguments
        sigma: float, standard deviation of the noise distribution.

    # Input shape
        Arbitrary. Use the keyword argument `input_shape`
        (tuple of integers, does not include the samples axis)
        when using this layer as the first layer in a model.

    # Output shape
        Same shape as input.
    """

    def __init__(self, sigma, **kwargs):
        self.supports_masking = True
        self.sigma = sigma
        self.uses_learning_phase = True
        super(GaussianNoise, self).__init__(**kwargs)
```

Journal of Machine Learning Research 15 (2014) 1929-1958 Submitted 11/13; Published 6/14

Dropout: A Simple Way to Prevent Neural Networks from Overfitting

Nitish Srivastava NITISH@CS.TORONTO.EDU
Geoffrey Hinton HINTON@CS.TORONTO.EDU
Alex Krizhevsky KRIZ@CS.TORONTO.EDU
Ilya Sutskever ILYA@CS.TORONTO.EDU
Ruslan Salakhutdinov RSALAKHU@CS.TORONTO.EDU
Department of Computer Science
University of Toronto
10 Kings College Road, Rm 3302
Toronto, Ontario, M5S 3G4, Canada.



(a) Standard Neural Net (b) After applying dropout.

Figure 1: Dropout Neural Net Model. **Left**: A standard neural net with 2 hidden layers. **Right**: An example of a thinned net produced by applying dropout to the network on the left. Crossed units have been dropped.

```
model.add(Dropout(0.25))
```

keras / keras / layers / core.py

```
class Dropout(Layer):
    """Applies Dropout to the input.

    Dropout consists in randomly setting
    a fraction `p` of input units to 0 at each update during training time,
    which helps prevent overfitting.

    # Arguments
        p: float between 0 and 1. Fraction of the input units to drop.
        noise_shape: 1D integer tensor representing the shape of the
            binary dropout mask that will be multiplied with the input.
            For instance, if your inputs ahve shape
            `(batch_size, timesteps, features)` and
            you want the dropout mask to be the same for all timesteps,
            you can use `noise_shape=(batch_size, 1, features)`.
        seed: A Python integer to use as random seed.
```

```
class SpatialDropout2D(Dropout):
    """Spatial 2D version of Dropout.

    This version performs the same function as Dropout, however it drops
    entire 2D feature maps instead of individual elements. If adjacent pixels
    within feature maps are strongly correlated (as is normally the case in
    early convolution layers) then regular dropout will not regularize the
    activations and will otherwise just result in an effective learning rate
    decrease. In this case, SpatialDropout2D will help promote independence
    between feature maps and should be used instead.

    # Arguments
        p: float between 0 and 1. Fraction of the input units to drop.
        dim_ordering: 'th' or 'tf'. In 'th' mode, the channels dimension
            (the depth) is at index 1, in 'tf' mode is it at index 3.
            It defaults to the `image_dim_ordering` value found in your
            Keras config file at `~/.keras/keras.json`.
            If you never set it, then it will be "tf".

    # Input shape
        4D tensor with shape:
        `(samples, channels, rows, cols)` if dim_ordering='th'
        or 4D tensor with shape:
        `(samples, rows, cols, channels)` if dim_ordering='tf'.

    # Output shape
        Same as input
```

Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift

Sergey Ioffe
Google Inc., *sioffe@google.com*

Christian Szegedy
Google Inc., *szegedy@google.com*

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β
Output: $\{y_i = \text{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \text{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Algorithm 1: Batch Normalizing Transform, applied to activation x over a mini-batch.

Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift

Sergey Ioffe
Google Inc., *sioffe@google.com*

Christian Szegedy
Google Inc., *szegedy@google.com*

4.2.1 Accelerating BN Networks

Simply adding Batch Normalization to a network does not take full advantage of our method. To do so, we further changed the network and its training parameters, as follows:

Increase learning rate. In a batch-normalized model, we have been able to achieve a training speedup from higher learning rates, with no ill side effects (Sec. 3.3).

Remove Dropout. As described in Sec. 3.4, Batch Normalization fulfills some of the same goals as Dropout. Removing Dropout from Modified BN-Inception speeds up training, without increasing overfitting.

Reduce the L_2 weight regularization. While in Inception an L_2 loss on the model parameters controls overfitting, in Modified BN-Inception the weight of this loss is reduced by a factor of 5. We find that this improves the accuracy on the held-out validation data.

Accelerate the learning rate decay. In training Inception, learning rate was decayed exponentially. Because our network trains faster than Inception, we lower the learning rate 6 times faster.

Reduce the photometric distortions. Because batch-normalized networks train faster and observe each training example fewer times, we let the trainer focus on more "real" images by distorting them less.

keras / keras / layers / normalization.py

```
class BatchNormalization(Layer):
    """Batch normalization layer (Ioffe and Szegedy, 2014).

    Normalize the activations of the previous layer at each batch,
    i.e. applies a transformation that maintains the mean activation
    close to 0 and the activation standard deviation close to 1.

    # Arguments
        epsilon: small float > 0. Fuzz parameter.
            Theano expects epsilon >= 1e-5.
        mode: integer, 0, 1 or 2.
            - 0: feature-wise normalization.
                Each feature map in the input will
                be normalized separately. The axis on which
                to normalize is specified by the `axis` argument.
                Note that if the input is a 4D image tensor
                using Theano conventions (samples, channels, rows, cols)
                then you should set `axis` to `1` to normalize along
                the channels axis.
                During training we use per-batch statistics to normalize
                the data, and during testing we use running averages
                computed during the training phase.
            - 1: sample-wise normalization. This mode assumes a 2D input.
            - 2: feature-wise normalization, like mode 0, but
                using per-batch statistics to normalize the data during both
                testing and training.
        axis: integer, axis along which to normalize in mode 0. For instance,
            if your input tensor has shape (samples, channels, rows, cols),
            set axis to 1 to normalize per feature map (channels axis).
```

keras / keras / layers / convolutional.py

```
model.add(Convolution2D(32, 3, 3, border_mode='same',
                        input_shape=X_train.shape[1:]))
model.add(Activation('relu'))
model.add(Convolution2D(32, 3, 3))
```

keras / keras / layers / convolutional.py

```
class Convolution2D(Layer):
    """Convolution operator for filtering windows of two-dimensional inputs.


    # Arguments
        nb_filter: Number of convolution filters to use.
        nb_row: Number of rows in the convolution kernel.
        nb_col: Number of columns in the convolution kernel.
        init: name of initialization function for the weights of the layer
            (see [initializations](../initializations.md)), or alternatively,
            Theano function to use for weights initialization.
            This parameter is only relevant if you don't pass
            a `weights` argument.
        activation: name of activation function to use
            (see [activations](../activations.md)),
            or alternatively, elementwise Theano function.
            If you don't specify anything, no activation is applied
            (ie. "linear" activation: a(x) = x).
        weights: list of numpy arrays to set as initial weights.
        border_mode: 'valid', 'same' or 'full'
            ('full' requires the Theano backend).
```

keras / keras / layers / convolutional.py

```
class Convolution2D(Layer):
    """Convolution operator for filtering windows of two-dimensional inputs.


        subsample: tuple of length 2. Factor by which to subsample output.
            Also called strides elsewhere.
        W_regularizer: instance of [WeightRegularizer](../regularizers.md)
            (eg. L1 or L2 regularization), applied to the main weights matrix.
        b_regularizer: instance of [WeightRegularizer](../regularizers.md),
            applied to the bias.
        activity_regularizer: instance of [ActivityRegularizer](../regularizers.md),
            applied to the network output.
        W_constraint: instance of the [constraints](../constraints.md) module
            (eg. maxnorm, nonneg), applied to the main weights matrix.
        b_constraint: instance of the [constraints](../constraints.md) module,
            applied to the bias.
        dim_ordering: 'th' or 'tf'. In 'th' mode, the channels dimension
            (the depth) is at index 1, in 'tf' mode is it at index 3.
            It defaults to the `image_dim_ordering` value found in your
            Keras config file at `~/.keras/keras.json`.
            If you never set it, then it will be "tf".
        bias: whether to include a bias
```

keras / keras / layers / convolutional.py

```
class Convolution2D(Layer):
    """Convolution operator for filtering windows of two-dimensional inputs.
```

```
 def __init__(self, nb_filter, nb_row, nb_col,
              init='glorot_uniform', activation=None, weights=None,
              border_mode='valid', subsample=(1, 1), dim_ordering='default',
              W_regularizer=None, b_regularizer=None,
              activity_regularizer=None,
              W_constraint=None, b_constraint=None,
              bias=True, **kwargs):
```

A guide to convolution arithmetic for deep learning

Vincent Dumoulin[1]★ and Francesco Visin[2]★†

★MILA, Université de Montréal
†AIRLab, Politecnico di Milano

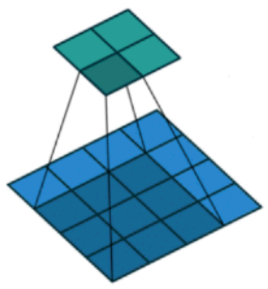
March 24, 2016

https://github.com/vdumoulin/conv_arithmetic

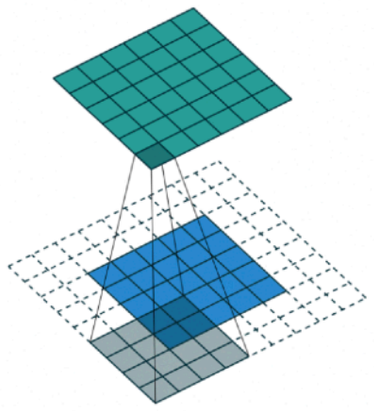
Convolution arithmetic

A technical report on convolution arithmetic in the context of deep learning.

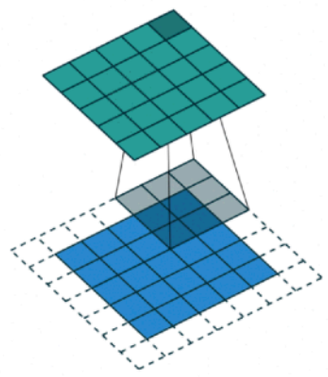
Convolution animations



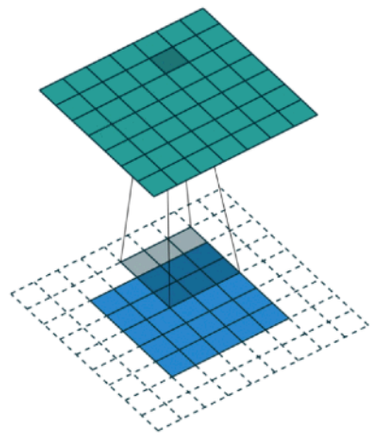
No padding, no strides



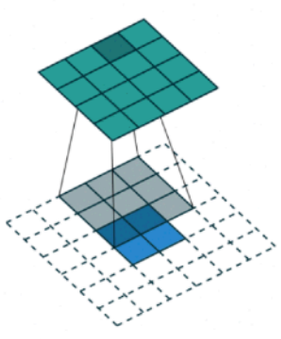
Arbitrary padding, no strides



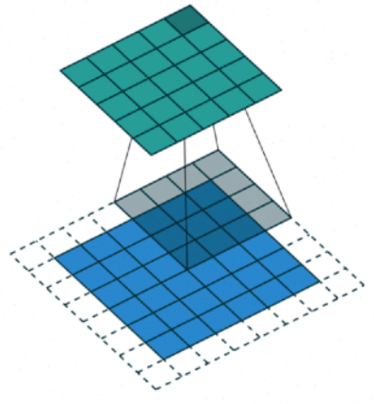
Half padding, no strides



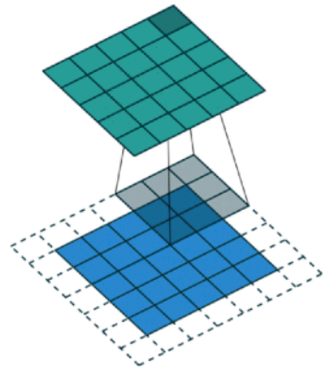
Full padding, no strides



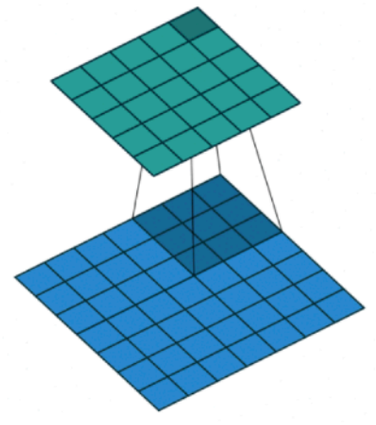
No padding, no strides, transposed



Arbitrary padding, no strides, transposed

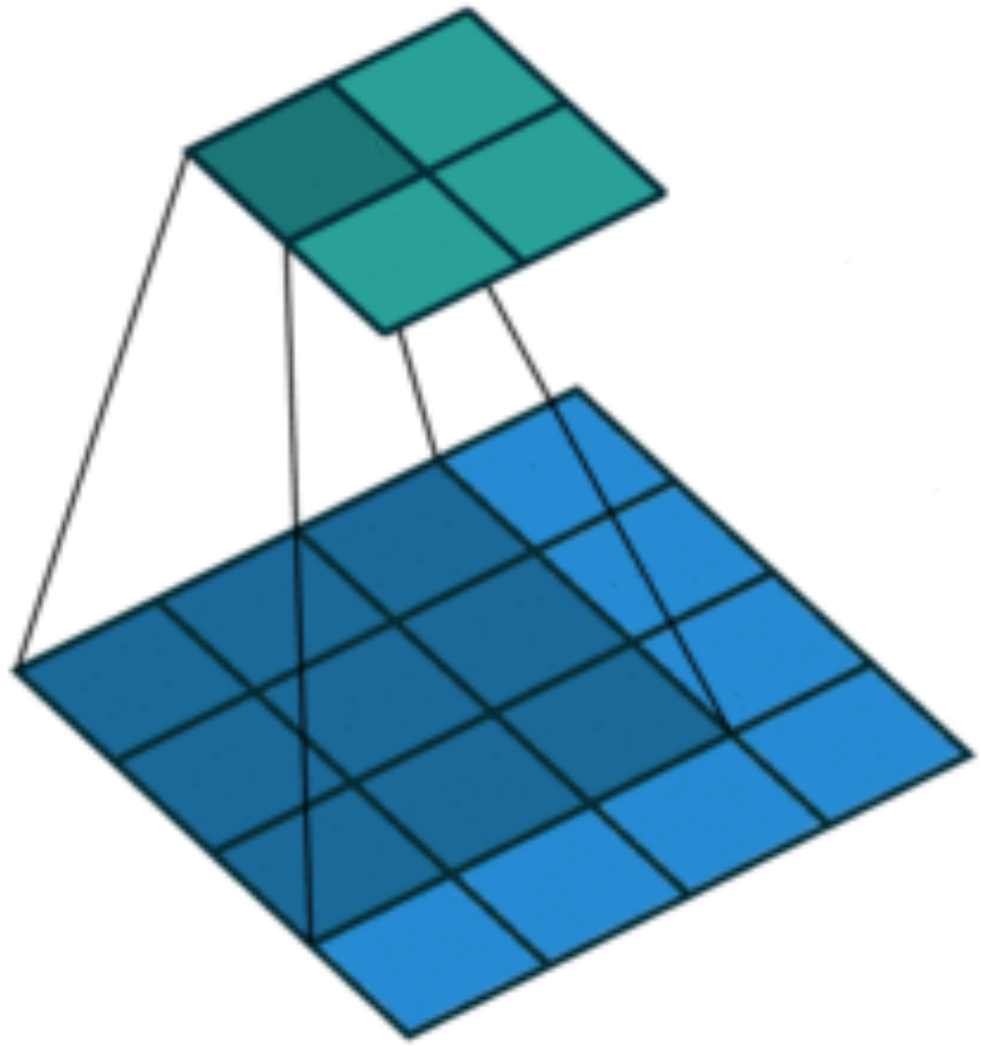


Half padding, no strides, transposed



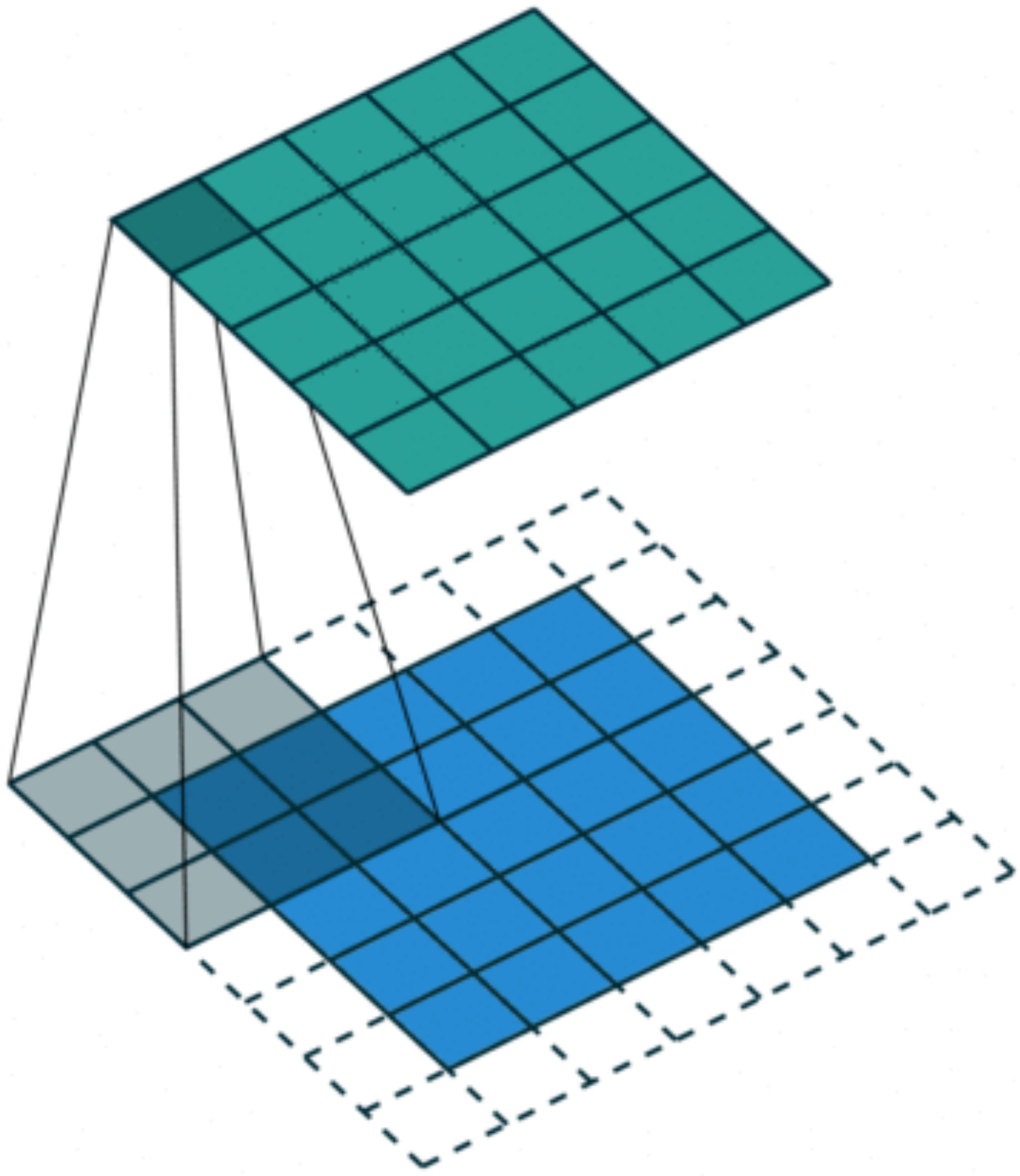
Full padding, no strides, transposed

https://github.com/vdumoulin/conv_arithmetic



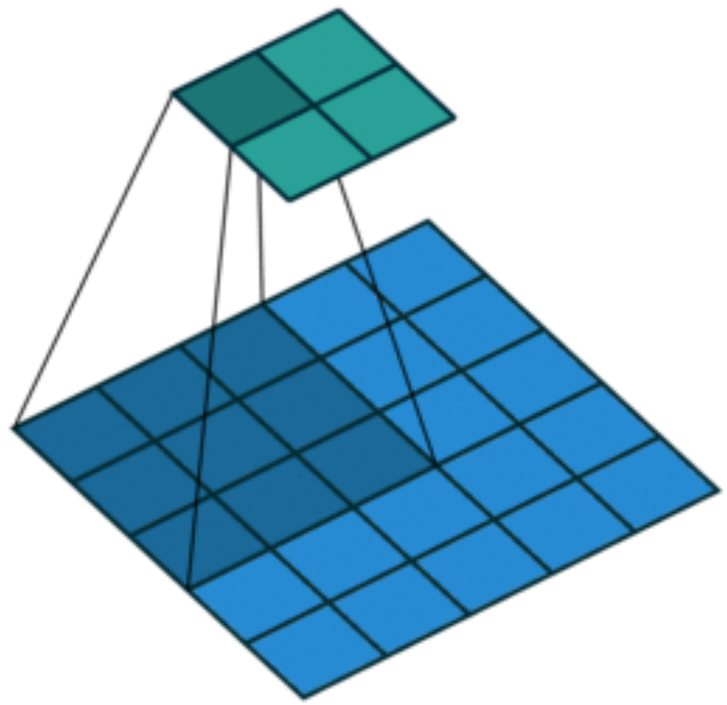
border_mode = ‘valid’
no strides: subsample= (1,1)

https://github.com/vdumoulin/conv_arithmetic



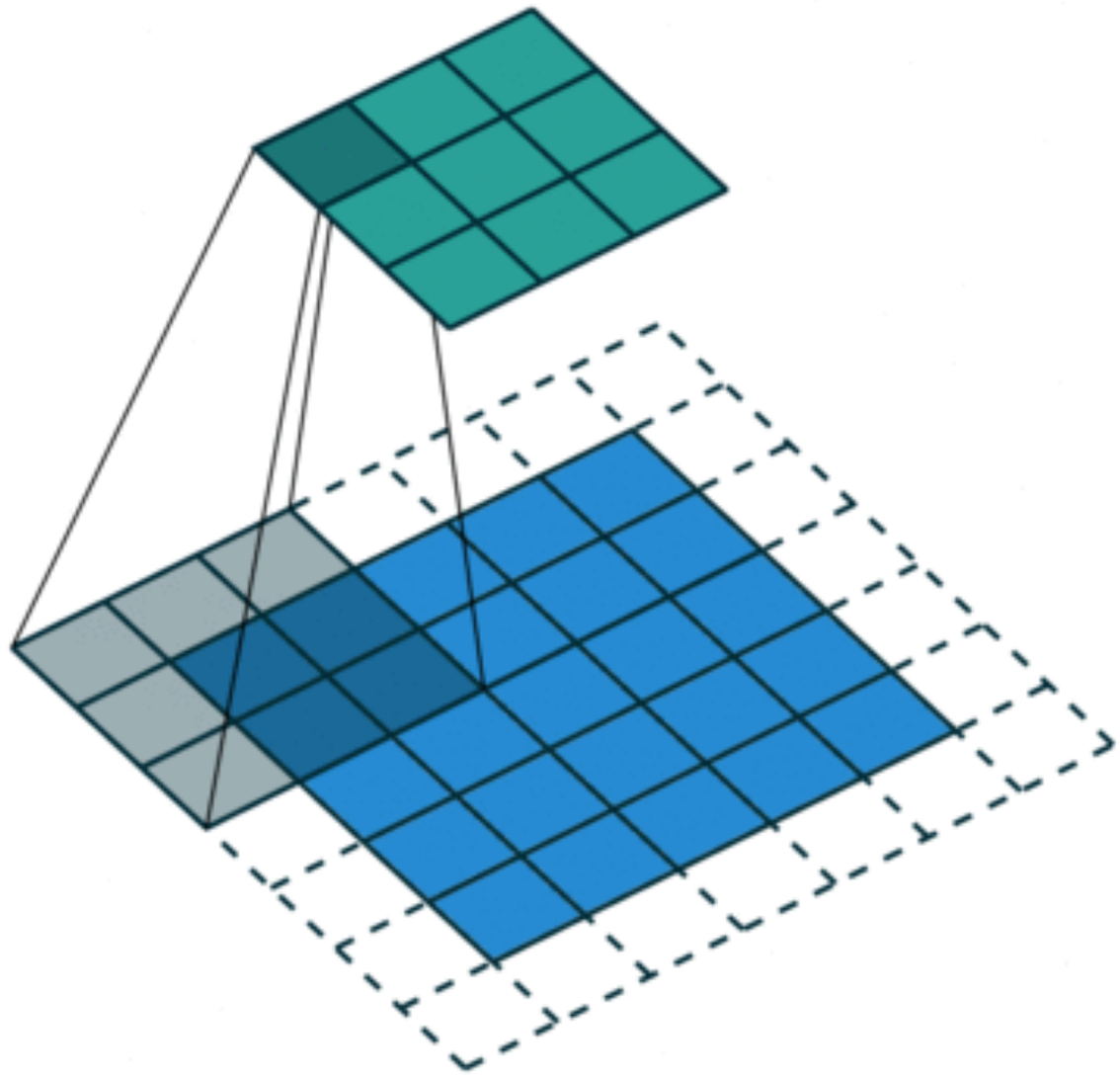
border_mode = ‘same’
no strides: subsample= (1,1)

https://github.com/vdumoulin/conv_arithmetic



border_mode = ‘valid’
2x2 strides: subsample= (2,2)

https://github.com/vdumoulin/conv_arithmetic



border_mode = ‘same’
2x2 strides: subsample= (2,2)

keras / keras / layers / pooling.py

```
class MaxPooling2D(_Pooling2D):
    """Max pooling operation for spatial data.

    # Arguments
        pool_size: tuple of 2 integers,
            factors by which to downscale (vertical, horizontal).
            (2, 2) will halve the image in each dimension.
        strides: tuple of 2 integers, or None. Strides values.
            If None, it will default to `pool_size`.
        border_mode: 'valid' or 'same'.
        dim_ordering: 'th' or 'tf'. In 'th' mode, the channels dimension
            (the depth) is at index 1, in 'tf' mode is it at index 3.
            It defaults to the `image_dim_ordering` value found in your
            Keras config file at `~/.keras/keras.json`.
            If you never set it, then it will be "tf".

    # Input shape
        4D tensor with shape:
        `(samples, channels, rows, cols)` if dim_ordering='th'
        or 4D tensor with shape:
        `(samples, rows, cols, channels)` if dim_ordering='tf'.

    # Output shape
        4D tensor with shape:
        `(nb_samples, channels, pooled_rows, pooled_cols)` if dim_ordering='th'
        or 4D tensor with shape:
        `(samples, pooled_rows, pooled_cols, channels)` if dim_ordering='tf'.
    """

    def __init__(self, pool_size=(2, 2), strides=None, border_mode='valid',
                 dim_ordering='default', **kwargs):
        super(MaxPooling2D, self).__init__(pool_size, strides, border_mode,
                                           dim_ordering, **kwargs)
```

Initializations

Usage of initializations

Initializations define the way to set the initial random weights of Keras layers.

The keyword arguments used for passing initializations to layers will depend on the layer. Usually it is simply `init`:

```
model.add(Dense(64, init='uniform'))
```

Available initializations

- **uniform**
- **lecun_uniform**: Uniform initialization scaled by the square root of the number of inputs (LeCun 98).
- **normal**
- **identity**: Use with square 2D layers (`shape[0] == shape[1]`).
- **orthogonal**: Use with square 2D layers (`shape[0] == shape[1]`).
- **zero**
- **one**
- **glorot_normal**: Gaussian initialization scaled by fan_in + fan_out (Glorot 2010)
- **glorot_uniform**
- **he_normal**: Gaussian initialization scaled by fan_in (He et al., 2014)
- **he_uniform**

For a discussion of weight initializations:
http://cs231n.github.io/neural-networks-2/#init

keras / keras / objectives.py

```
def mean_squared_error(y_true, y_pred):
    return K.mean(K.square(y_pred - y_true), axis=-1)


def mean_absolute_error(y_true, y_pred):
    return K.mean(K.abs(y_pred - y_true), axis=-1)


def categorical_crossentropy(y_true, y_pred):
    return K.categorical_crossentropy(y_pred, y_true)


def binary_crossentropy(y_true, y_pred):
    return K.mean(K.binary_crossentropy(y_pred, y_true), axis=-1)
```

keras / keras / optimizers.py

SGD

RMSProp

Adam

Tune the learning rate!

keras / keras / callbacks.py

History [source]

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

Callback that records events into a `History` object.

This callback is automatically applied to every Keras model. The `History` object gets returned by the `fit` method of models.

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

- Use *metrics* to specify what you want in history
- Up to you to save it!

keras / keras / callbacks.py

LearningRateScheduler [source]

```
keras.callbacks.LearningRateScheduler(schedule)
```

Learning rate scheduler.

Arguments

- **schedule**: a function that takes an epoch index as input (integer, indexed from 0) and returns a new learning rate as output (float).

---

keras / keras / callbacks.py

ReduceLROnPlateau [source]

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

Reduce learning rate when a metric has stopped improving.

Models often benefit from reducing the learning rate by a factor of 2-10 once learning stagnates. This callback monitors a quantity and if no improvement is seen for a 'patience' number of epochs, the learning rate is reduced.

Example

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

Arguments

- **monitor**: quantity to be monitored.
- **factor**: factor by which the learning rate will be reduced. new_lr = lr * factor
- **patience**: number of epochs with no improvement after which learning rate will be reduced.
- **verbose**: int. 0: quiet, 1: update messages.
- **mode**: one of {auto, min, max}. In `min` mode, lr will be reduced when the quantity monitored has stopped decreasing; in `max` mode it will be reduced when the quantity monitored has stopped increasing; in `auto` mode, the direction is automatically inferred from the name of the monitored quantity.
- **epsilon**: threshold for measuring the new optimum, to only focus on significant changes.
- **cooldown**: number of epochs to wait before resuming normal operation after lr has been reduced.
- **min_lr**: lower bound on the learning rate.

keras / keras / callbacks.py

```
class ModelCheckpoint(Callback):
    """Save the model after every epoch.

    `filepath` can contain named formatting options,
    which will be filled the value of `epoch` and
    keys in `logs` (passed in `on_epoch_end`).

    For example: if `filepath` is `weights.{epoch:02d}-{val_loss:.2f}.hdf5`,
    then the model checkpoints will be saved with the epoch number and
    the validation loss in the filename.

    # Arguments
        filepath: string, path to save the model file.
        monitor: quantity to monitor.
        verbose: verbosity mode, 0 or 1.
        save_best_only: if `save_best_only=True`,
            the latest best model according to
            the quantity monitored will not be overwritten.
        mode: one of {auto, min, max}.
            If `save_best_only=True`, the decision
            to overwrite the current save file is made
            based on either the maximization or the
            minimization of the monitored quantity. For `val_acc`,
            this should be `max`, for `val_loss` this should
            be `min`, etc. In `auto` mode, the direction is
            automatically inferred from the name of the monitored quantity.
        save_weights_only: if True, then only the model's weights will be
            saved (`model.save_weights(filepath)`), else the full model
            is saved (`model.save(filepath)`).
        period: Interval (number of epochs) between checkpoints.
    """

    def __init__(self, filepath, monitor='val_loss', verbose=0,
                 save_best_only=False, save_weights_only=False,
                 mode='auto', period=1):
```

keras / keras / callbacks.py

```
from keras.callbacks import ModelCheckpoint

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

'''
saves the model weights after each epoch if the validation loss decreased
'''
checkpointer = ModelCheckpoint(filepath="/tmp/weights.hdf5", verbose=1, save_best_only=True)
model.fit(X_train, Y_train, batch_size=128, nb_epoch=20, verbose=0,
    validation_data=(X_test, Y_test), callbacks=[checkpointer])
```

Saving and loading weights

- `model.save_weights(filepath)` : saves the weights of the model as a HDF5 file.
- `model.load_weights(filepath, by_name=False)` : loads the weights of the model from a HDF5 file (created by `save_weights`). By default, the architecture is expected to be unchanged. To load weights into a different architecture (with some layers in common), use `by_name=True` to load only those layers with the same name.

Saving and loading a model

- `model.summary()` : prints a summary representation of your model.
- `model.get_config()` : returns a dictionary containing the configuration of the model. The model can be reinstantiated from its config via:

```
config = model.get_config()
model = Model.from_config(config)
# or, for Sequential:
model = Sequential.from_config(config)
```

- `model.get_weights()` : returns a list of all weight tensors in the model, as Numpy arrays.
- `model.set_weights(weights)` : sets the values of the weights of the model, from a list of Numpy arrays. The arrays in the list should have the same shape as those returned by `get_weights()` .
- `model.to_json()` : returns a representation of the model as a JSON string. Note that the representation does not include the weights, only the architecture. You can reinstantiate the same model (with reinitialized weights) from the JSON string via:

```
from models import model_from_json

json_string = model.to_json()
model = model_from_json(json_string)
```

Building powerful image classification models using very little data

In this tutorial, we will present a few simple yet effective methods that you can use to build a powerful image classifier, using only very few training examples --just a few hundred or thousand pictures from each class you want to be able to recognize.

Sun 05 June 2016
By Francois Chollet
In Tutorials.

We will go over the following options:

- training a small network from scratch (as a baseline)
- using the bottleneck features of a pre-trained network
- fine-tuning the top layers of a pre-trained network

This will lead us to cover the following Keras features:

- `fit_generator` for training Keras a model using Python data generators
- `ImageDataGenerator` for real-time data augmentation
- layer freezing and model fine-tuning
- ...and more.

Loading a retrained model: first approach

fchollet / **classifier_from_little_data_script_2.py**

```
# build the VGG16 network
model = Sequential()
model.add(ZeroPadding2D((1, 1), input_shape=(3, img_width, img_height)))

model.add(Convolution2D(64, 3, 3, activation='relu', name='conv1_1'))
model.add(ZeroPadding2D((1, 1)))
model.add(Convolution2D(64, 3, 3, activation='relu', name='conv1_2'))
model.add(MaxPooling2D((2, 2), strides=(2, 2)))

model.add(ZeroPadding2D((1, 1)))
model.add(Convolution2D(128, 3, 3, activation='relu', name='conv2_1'))
model.add(ZeroPadding2D((1, 1)))
model.add(Convolution2D(128, 3, 3, activation='relu', name='conv2_2'))
model.add(MaxPooling2D((2, 2), strides=(2, 2)))

model.add(ZeroPadding2D((1, 1)))
model.add(Convolution2D(256, 3, 3, activation='relu', name='conv3_1'))
model.add(ZeroPadding2D((1, 1)))
model.add(Convolution2D(256, 3, 3, activation='relu', name='conv3_2'))
model.add(ZeroPadding2D((1, 1)))
model.add(Convolution2D(256, 3, 3, activation='relu', name='conv3_3'))
model.add(MaxPooling2D((2, 2), strides=(2, 2)))

model.add(ZeroPadding2D((1, 1)))
model.add(Convolution2D(512, 3, 3, activation='relu', name='conv4_1'))
model.add(ZeroPadding2D((1, 1)))
model.add(Convolution2D(512, 3, 3, activation='relu', name='conv4_2'))
model.add(ZeroPadding2D((1, 1)))
model.add(Convolution2D(512, 3, 3, activation='relu', name='conv4_3'))
model.add(MaxPooling2D((2, 2), strides=(2, 2)))

model.add(ZeroPadding2D((1, 1)))
model.add(Convolution2D(512, 3, 3, activation='relu', name='conv5_1'))
model.add(ZeroPadding2D((1, 1)))
model.add(Convolution2D(512, 3, 3, activation='relu', name='conv5_2'))
model.add(ZeroPadding2D((1, 1)))
model.add(Convolution2D(512, 3, 3, activation='relu', name='conv5_3'))
model.add(MaxPooling2D((2, 2), strides=(2, 2)))
```

Loading a retrained model: first approach

fchollet / **classifier_from_little_data_script_2.py**

```
# load the weights of the VGG16 networks
# (trained on ImageNet, won the ILSVRC competition in 2014)
# note: when there is a complete match between your model definition
# and your weight savefile, you can simply call model.load_weights(filename)
assert os.path.exists(weights_path), 'Model weights not found (see "weights_path" variable in script).'
f = h5py.File(weights_path)
for k in range(f.attrs['nb_layers']):
    if k >= len(model.layers):
        # we don't look at the last (fully-connected) layers in the savefile
        break
    g = f['layer_{}'.format(k)]
    weights = [g['param_{}'.format(p)] for p in range(g.attrs['nb_params'])]
    model.layers[k].set_weights(weights)
f.close()
print('Model loaded.')
```

Loading a retrained model: second approach

keras / examples / **deep_dream.py**

```
from keras.applications import vgg16

# build the VGG16 network with our placeholder
# the model will be loaded with pre-trained ImageNet weights
model = vgg16.VGG16(input_tensor=dream,
                    weights='imagenet', include_top=False)
print('Model loaded.')
```

Loading a retrained model: second approach

keras / keras / applications / **vgg16.py**

```
def VGG16(include_top=True, weights='imagenet',
          input_tensor=None, input_shape=None,
          classes=1000):
    """Instantiate the VGG16 architecture,
    optionally loading weights pre-trained
    on ImageNet. Note that when using TensorFlow,
    for best performance you should set
    `image_dim_ordering="tf"` in your Keras config
    at ~/.keras/keras.json.

    The model and the weights are compatible with both
    TensorFlow and Theano. The dimension ordering
    convention used by the model is the one
    specified in your Keras config file.
```

Loading a retrained model: second approach

keras / keras / applications / **vgg16.py**

```
# Arguments
    include_top: whether to include the 3 fully-connected
        layers at the top of the network.
    weights: one of `None` (random initialization)
        or "imagenet" (pre-training on ImageNet).
    input_tensor: optional Keras tensor (i.e. output of `layers.Input()`)
        to use as image input for the model.
    input_shape: optional shape tuple, only to be specified
        if `include_top` is False (otherwise the input shape
        has to be `(224, 224, 3)` (with `tf` dim ordering)
        or `(3, 224, 244)` (with `th` dim ordering).
        It should have exactly 3 inputs channels,
        and width and height should be no smaller than 48.
        E.g. `(200, 200, 3)` would be one valid value.
    classes: optional number of classes to classify images
        into, only to be specified if `include_top` is True, and
        if no `weights` argument is specified.
```

Functional API

First example: fully connected network

The `Sequential` model is probably a better choice to implement such a network, but it helps to start with something really simple.

- A layer instance is callable (on a tensor), and it returns a tensor
- Input tensor(s) and output tensor(s) can then be used to define a `Model`
- Such a model can be trained just like Keras `Sequential` models.

Functional API

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

# this returns a tensor
inputs = Input(shape=(784,))

# a layer instance is callable on a tensor, and returns a tensor
x = Dense(64, activation='relu')(inputs)
x = Dense(64, activation='relu')(x)
predictions = Dense(10, activation='softmax')(x)

# this creates a model that includes
# the Input layer and three Dense layers
model = Model(input=inputs, output=predictions)
model.compile(optimizer='rmsprop',
              loss='categorical_crossentropy',
              metrics=['accuracy'])
model.fit(data, labels)  # starts training
```

Functional API

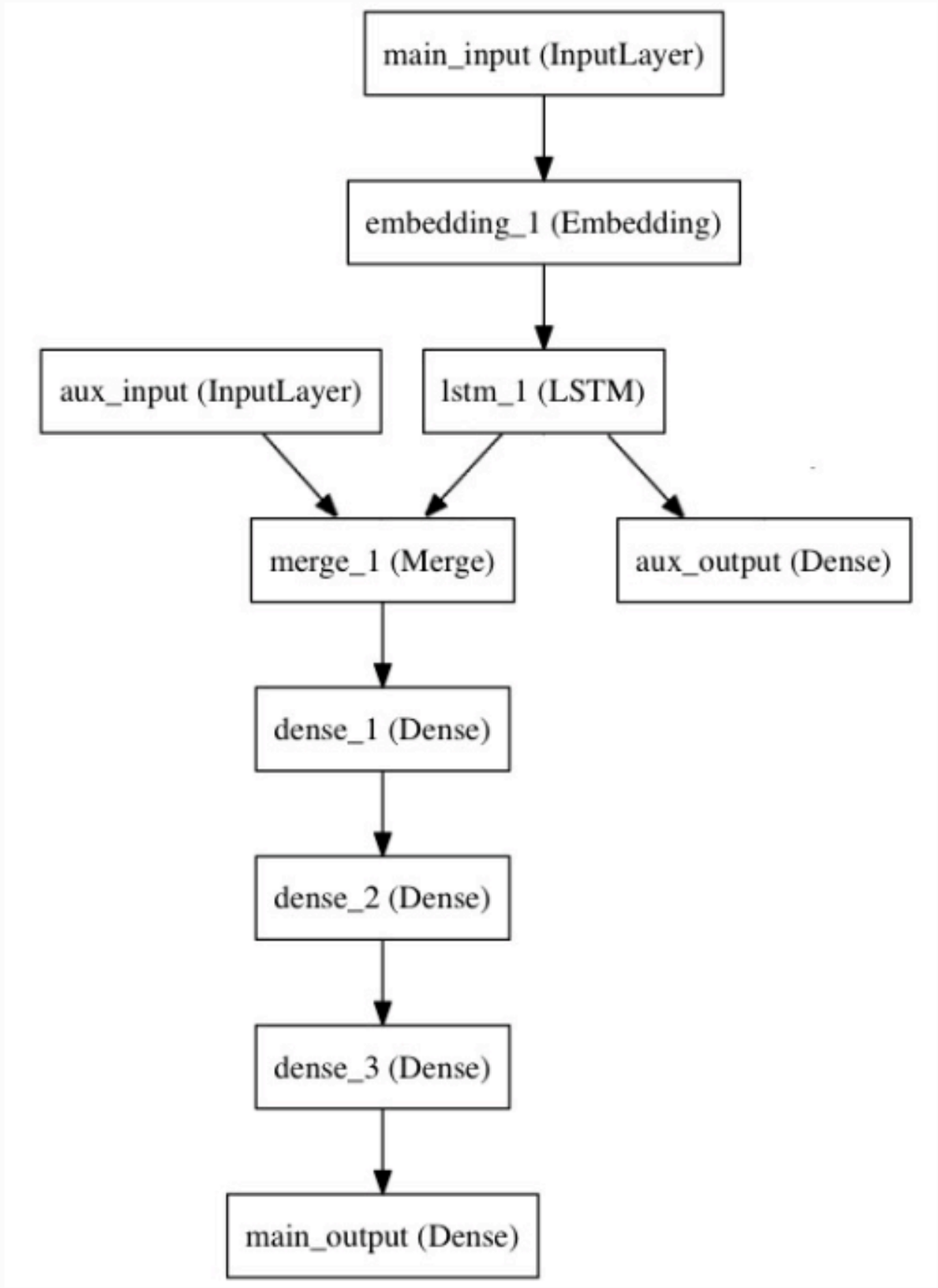
Multi-input and multi-output models

Here's a good use case for the functional API: models with multiple inputs and outputs. The functional API makes it easy to manipulate a large number of intertwined datastreams.

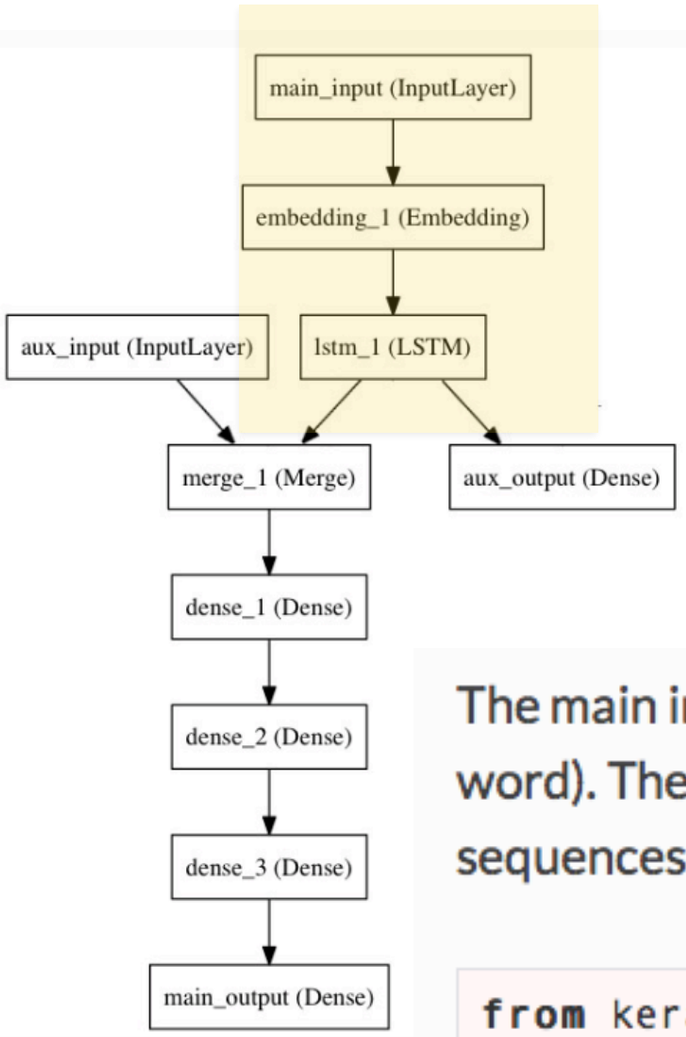
Let's consider the following model. We seek to predict how many retweets and likes a news headline will receive on Twitter. The main input to the model will be the headline itself, as a sequence of words, but to spice things up, our model will also have an auxiliary input, receiving extra data such as the time of day when the headline was posted, etc. The model will also be supervised via two loss functions. Using the main loss function earlier in a model is a good regularization mechanism for deep models.

Functional API

Here's what our model looks like:



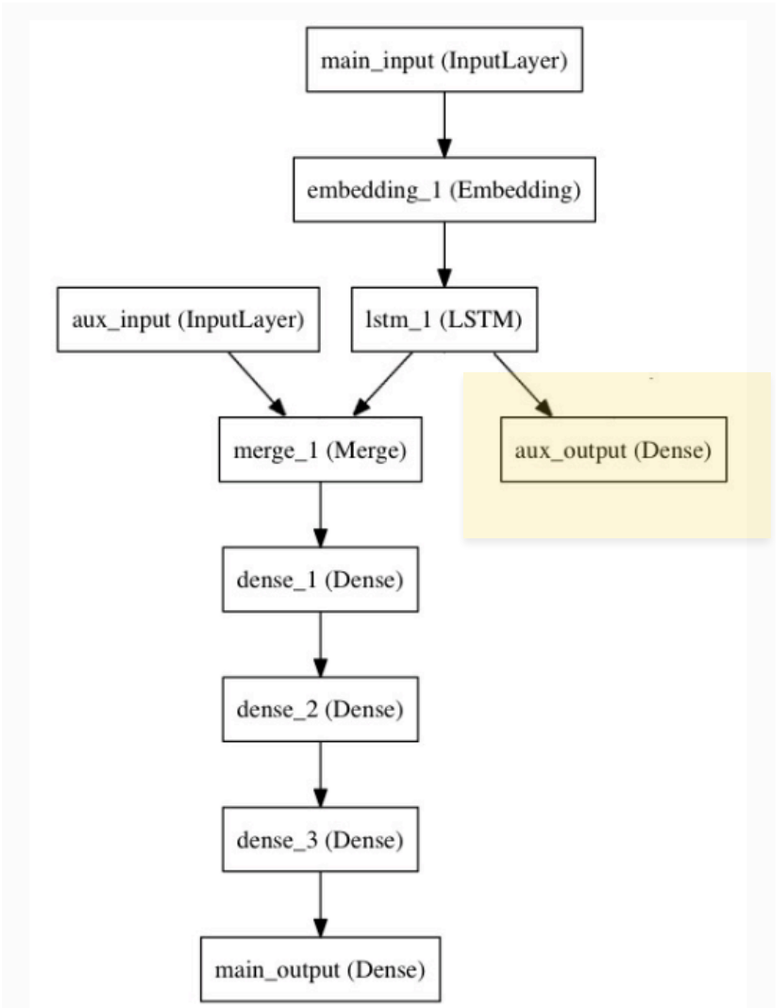
Functional API



The main input will receive the headline, as a sequence of integers (each integer encodes a word). The integers will be between 1 and 10,000 (a vocabulary of 10,000 words) and the sequences will be 100 words long.

```
from keras.layers import Input, Embedding, LSTM, Dense, merge
from keras.models import Model

# headline input: meant to receive sequences of 100 integers, between 1 and 10000.
# note that we can name any layer by passing it a "name" argument.
main_input = Input(shape=(100,), dtype='int32', name='main_input')

# this embedding layer will encode the input sequence
# into a sequence of dense 512-dimensional vectors.
x = Embedding(output_dim=512, input_dim=10000, input_length=100)(main_input)

# a LSTM will transform the vector sequence into a single vector,
# containing information about the entire sequence
lstm_out = LSTM(32)(x)
```

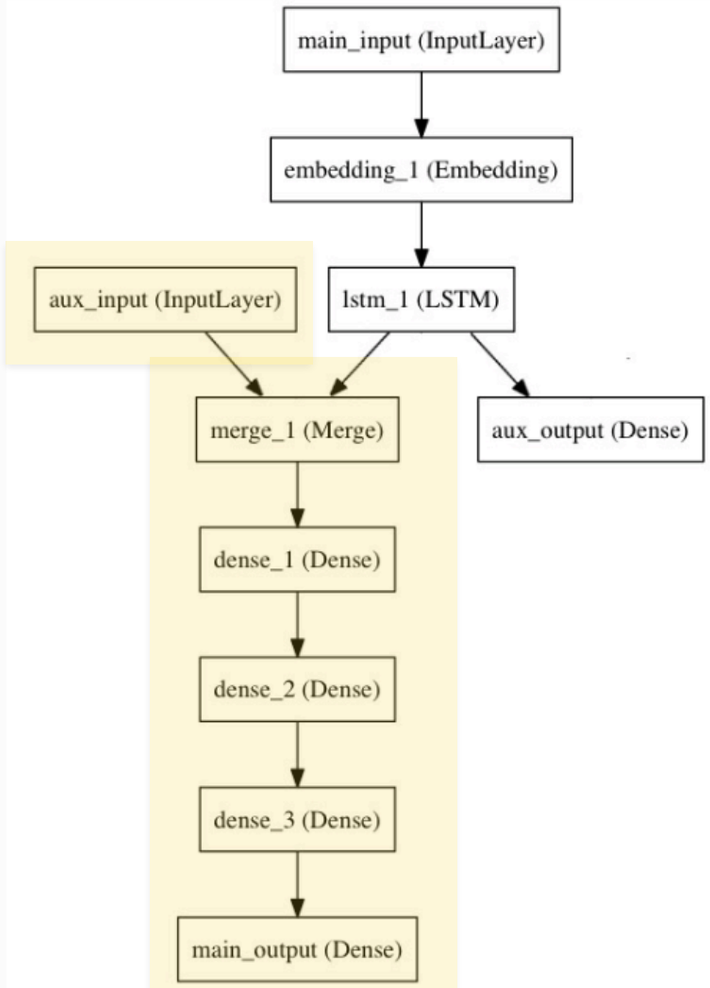
Functional API



Here we insert the auxiliary loss, allowing the LSTM and Embedding layer to be trained smoothly even though the main loss will be much higher in the model.

```
auxiliary_output = Dense(1, activation='sigmoid', name='aux_output')(lstm_out)
```

Functional API



At this point, we feed into the model our auxiliary input data by concatenating it with the LSTM output:

```
auxiliary_input = Input(shape=(5,), name='aux_input')
x = merge([lstm_out, auxiliary_input], mode='concat')

# we stack a deep fully-connected network on top
x = Dense(64, activation='relu')(x)
x = Dense(64, activation='relu')(x)
x = Dense(64, activation='relu')(x)

# and finally we add the main logistic regression layer
main_output = Dense(1, activation='sigmoid', name='main_output')(x)
```

Functional API

This defines a model with two inputs and two outputs:

```
model = Model(input=[main_input, auxiliary_input], output=[main_output, auxiliary_output])
```

We compile the model and assign a weight of 0.2 to the auxiliary loss. To specify different `loss_weights` or `loss` for each different output, you can use a list or a dictionary. Here we pass a single loss as the `loss` argument, so the same loss will be used on all outputs.

```
model.compile(optimizer='rmsprop', loss='binary_crossentropy',
              loss_weights=[1., 0.2])
```

We can train the model by passing it lists of input arrays and target arrays:

```
model.fit([headline_data, additional_data], [labels, labels],
          nb_epoch=50, batch_size=32)
```

Functional API

Since our inputs and outputs are named (we passed them a "name" argument), We could also have compiled the model via:

```
model.compile(optimizer='rmsprop',
              loss={'main_output': 'binary_crossentropy', 'aux_output': 'binary_crossentropy'},
              loss_weights={'main_output': 1., 'aux_output': 0.2})

# and trained it via:
model.fit({'main_input': headline_data, 'aux_input': additional_data},
          {'main_output': labels, 'aux_output': labels},
          nb_epoch=50, batch_size=32)
```