1 config.py
2 models.py
3 time_frequency.py

----

1 config.py

```
import pandas as pd

train_dir = '../input/audio_train/'

submit = pd.read_csv('../input/sample_submission.csv')
i2label = label_columns = submit.columns[1:].tolist()
label2i = {label:i for i,label in enumerate(i2label)}

n_classes = 80

assert len(label2i) == n_classes




class Config(object):
    def __init__(self,
        batch_size=32,
        n_folds=5,
        lr=0.0005,
        duration = 5,
        name = 'v1',
        milestones = (14,21,28),
        rnn_unit = 128,
        lm = 0.0,
        momentum = 0.85,
        mixup_prob = -1,
        folds=None,
        pool_mode = ('max','avemax1'),
        pretrained = None,
        gamma = 0.5,
        x1_rate = 0.7,
        w_ratio = 1,
        get_backbone = None
    ):
```

```
        self.maxlen = int((duration*44100))
        self.bs = batch_size
        self.n_folds = n_folds
        self.name = name
        self.lr = lr
        self.milestones = milestones
        self.rnn_unit = rnn_unit
        self.lm = lm
        self.momentum = momentum
        self.mixup_prob = mixup_prob
        self.folds = list(range(n_folds)) if folds is
None else folds
        self.pool_mode = pool_mode
        self.pretrained = pretrained
        self.gamma = gamma
        self.x1_rate = x1_rate
        self.w_ratio = w_ratio
        self.get_backbone = get_backbone

    def __str__(self):
        return ',\t'.join(['%s:%s' % item for item in
self.__dict__.items()])





----------
2 models.py

from keras.layers import *
from time_frequency import Melspectrogram, AdditiveNoise
from keras.optimizers import Nadam,SGD
from keras.constraints import *
from keras.initializers import *
from keras.models import Model
from config import *

EPS = 1e-8

def squeeze_excitation_layer(x, out_dim, ratio = 4):
    '''
    SE module performs inter-channel weighting.
    '''
    squeeze = GlobalAveragePooling2D()(x)
    excitation = Dense(units=out_dim // ratio)(squeeze)
```

```
    excitation = Activation('relu')(excitation)
    excitation = Dense(units=out_dim)(excitation)
    excitation = Activation('sigmoid')(excitation)
    excitation = Reshape((1, 1, out_dim))(excitation)

    scale = multiply([x, excitation])
    return scale

def
conv_se_block(x,filters,pool_stride,pool_size,pool_mode,cfg,
ratio = 4):

    x = Conv2D(filters=filters, kernel_size=3,
strides=1, padding='same')(x)
    x = BatchNormalization(momentum=cfg.momentum)(x)
    x = Activation('relu')(x)
    x = squeeze_excitation_layer(x,
out_dim=filters,ratio=ratio)
    x = pooling_block(x, pool_size[0], pool_stride[0],
pool_mode[0], cfg)

    x = Conv2D(filters=filters, kernel_size=3,
strides=1, padding='same')(x)
    x = BatchNormalization(momentum=cfg.momentum)(x)
    x = Activation('relu')(x)
    x = squeeze_excitation_layer(x,
out_dim=filters,ratio=ratio)
    x = pooling_block(x, pool_size[1], pool_stride[1],
pool_mode[1], cfg)

    return x

def AveMaxPool(x, pool_size,stride, ave_axis):
    if isinstance(pool_size,int):
        pool_size1,pool_size2 = pool_size, pool_size
    else:
        pool_size1,pool_size2 = pool_size
    if ave_axis == 2:
        x = AveragePooling2D(pool_size=(1,pool_size1),
padding='same', strides=(1,stride))(x)
        x = MaxPool2D(pool_size=(pool_size2,1),
padding='same', strides=(stride,1))(x)
    elif ave_axis == 1:
        x = AveragePooling2D(pool_size=(pool_size1,1),
padding='same', strides=(stride,1))(x)
        x = MaxPool2D(pool_size=(1,pool_size2),
```

```
padding='same', strides=(1,stride))(x)
    elif ave_axis == 3:
        x = MaxPool2D(pool_size=(1,pool_size1),
padding='same', strides=(1,stride))(x)
        x = AveragePooling2D(pool_size=(pool_size2, 1),
padding='same', strides=(stride, 1))(x)
    elif ave_axis == 4:
        x = MaxPool2D(pool_size=(pool_size1, 1),
padding='same', strides=(stride, 1))(x)
        x = AveragePooling2D(pool_size=(1, pool_size2),
padding='same', strides=(1, stride))(x)
    else:
        raise RuntimeError("axis error")
    return x

def pooling_block(x,pool_size,stride,pool_mode, cfg):
    if pool_mode == 'max':
        x = MaxPool2D(pool_size=pool_size,
padding='same', strides=stride)(x)
    elif pool_mode == 'ave':
        x = AveragePooling2D(pool_size=pool_size,
padding='same', strides=stride)(x)
    elif pool_mode == 'avemax1':
        x = AveMaxPool(x, pool_size=pool_size,
stride=stride, ave_axis=1)
    elif pool_mode == 'avemax2':
        x = AveMaxPool(x, pool_size=pool_size,
stride=stride, ave_axis=2)
    elif pool_mode == 'avemax3':
        x = AveMaxPool(x, pool_size=pool_size,
stride=stride, ave_axis=3)
    elif pool_mode == 'avemax4':
        x = AveMaxPool(x, pool_size=pool_size,
stride=stride, ave_axis=4)
    elif pool_mode == 'conv':
        x = Lambda(lambda
x:K.expand_dims(K.permute_dimensions(x,
(0,3,1,2)),axis=-1))(x)
        x = TimeDistributed(Conv2D(filters=1,
kernel_size=pool_size, strides=stride, padding='same',
use_bias=False))(x)
        x = Lambda(lambda
x:K.permute_dimensions(K.squeeze(x,axis=-1),(0,2,3,1)))
(x)
    elif pool_mode is None:
        x = x
```

```
    else:
        raise RuntimeError('pool mode error')
    return x

def
conv_block(x,filters,pool_stride,pool_size,pool_mode,cfg):


    x = Conv2D(filters=filters, kernel_size=3,
strides=1, padding='same')(x)
    x = BatchNormalization(momentum=cfg.momentum)(x)
    x = Activation('relu')(x)
    x = pooling_block(x, pool_size[0], pool_stride[0],
pool_mode[0], cfg)

    x = Conv2D(filters=filters, kernel_size=3,
strides=1, padding='same')(x)
    x = BatchNormalization(momentum=cfg.momentum)(x)
    x = Activation('relu')(x)
    x = pooling_block(x, pool_size[1], pool_stride[1],
pool_mode[1], cfg)
    return x


def conv_cat_block(x, filters, pool_stride, pool_size,
pool_mode, cfg):
    x = Conv2D(filters=filters, kernel_size=3,
strides=1, padding='same')(x)
    x = BatchNormalization(momentum=cfg.momentum)(x)
    x = Activation('relu')(x)
    x = pooling_block(x, pool_size[0], pool_stride[0],
pool_mode[0], cfg)

    x1 = x
    x = Conv2D(filters=filters, kernel_size=3,
strides=1, padding='same')(x)
    x = BatchNormalization(momentum=cfg.momentum)(x)
    x = Activation('relu')(x)

    ## concat
    x = concatenate([x1, x])
    x = Conv2D(filters=filters, kernel_size=1,
strides=1, padding='same')(x)
    x = pooling_block(x, pool_size[1], pool_stride[1],
pool_mode[1], cfg)
```

```
    return x


def conv_se_cat_block(x, filters, pool_stride,
pool_size, pool_mode, cfg):
    x = Conv2D(filters=filters, kernel_size=3,
strides=1, padding='same')(x)
    x = BatchNormalization(momentum=cfg.momentum)(x)
    x = Activation('relu')(x)
    x = squeeze_excitation_layer(x, out_dim=filters,
ratio=4)
    x = pooling_block(x, pool_size[0], pool_stride[0],
pool_mode[0], cfg)
    x1 = x
    x = Conv2D(filters=filters, kernel_size=3,
strides=1, padding='same')(x)
    x = BatchNormalization(momentum=cfg.momentum)(x)
    x = Activation('relu')(x)
    x = squeeze_excitation_layer(x, out_dim=filters,
ratio=4)
    ## concat
    x = concatenate([x1, x])
    x = Conv2D(filters=filters, kernel_size=1,
strides=1, padding='same')(x)
    x = pooling_block(x, pool_size[1], pool_stride[1],
pool_mode[1], cfg)

    return x



def pixelShuffle(x):
    _,h,w,c = K.int_shape(x)
    bs = K.shape(x)[0]
    assert w%2==0
    x = K.reshape(x,(bs,h,w//2,c*2))

    # assert h % 2 == 0
    # x = K.permute_dimensions(x,(0,2,1,3))
    # x = K.reshape(x,(bs,w//2,h//2,c*4))
    # x = K.permute_dimensions(x,(0,2,1,3))
    return x

def get_se_backbone(x, cfg):

    x = Conv2D(64, kernel_size=3, padding='same',
```

```
use_bias=False)(x)
    x = BatchNormalization(momentum=cfg.momentum)(x)
    x = Activation('relu')(x)
    x = squeeze_excitation_layer(x, out_dim=64, ratio=4)
    # backbone
    x = conv_se_block(x, 96, (1, 2), (3, 2),
cfg.pool_mode, cfg)
    x = conv_se_block(x, 128, (1, 2), (3, 2),
cfg.pool_mode, cfg)
    x = conv_se_block(x, 256, (1, 2), (3, 3),
cfg.pool_mode, cfg)
    x = conv_se_block(x, 512, (1, 2), (3, 2), (None,
None), cfg)  ## [bs,  54, 8, 512]

    # global pooling
    x = Lambda(pixelShuffle)(x)  ## [bs,  54, 4, 1024]
    x = Lambda(lambda x: K.max(x, axis=1))(x)
    x = Lambda(lambda x: K.mean(x, axis=1))(x)

    return x

def get_conv_backbone(x, cfg):

    # input stem
    x = Conv2D(64, kernel_size=3, padding='same',
use_bias=False)(x)
    x = BatchNormalization(momentum=cfg.momentum)(x)
    x = Activation('relu')(x)

    # backbone
    x = conv_block(x, 96, (1, 2), (3, 2), cfg.pool_mode,
cfg)
    x = conv_block(x, 128, (1, 2), (3, 2),
cfg.pool_mode, cfg)
    x = conv_block(x, 256, (1, 2), (3, 3),
cfg.pool_mode, cfg)
    x = conv_block(x, 512, (1, 2), (3, 2), (None, None),
cfg)  ## [bs,  54, 8, 512]

    # global pooling
    x = Lambda(pixelShuffle)(x)  ## [bs,  54, 4, 1024]
    x = Lambda(lambda x: K.max(x, axis=1))(x)
    x = Lambda(lambda x: K.mean(x, axis=1))(x)

    return x
```

```
def get_se_cat_backbone(x,cfg):


    x = Conv2D(64, kernel_size=3,
padding='same',use_bias=False)(x)
    x = BatchNormalization(momentum=cfg.momentum)(x)
    x = Activation('relu')(x)
    x = squeeze_excitation_layer(x, out_dim=64,ratio=4)
    # backbone
    x = conv_se_cat_block(x, 96, (1,2), (3,2),
cfg.pool_mode, cfg)
    x = conv_se_cat_block(x, 128, (1,2), (3,2),
cfg.pool_mode, cfg)
    x = conv_se_cat_block(x, 256, (1,2), (3,3),
cfg.pool_mode, cfg)
    x = conv_se_cat_block(x, 512, (1,2), (3,2),
(None,None), cfg) ## [bs,  54, 8, 512]

    # global pooling
    x = Lambda(pixelShuffle)(x)  ## [bs,  54, 4, 1024]
    x = Lambda(lambda x: K.max(x, axis=1))(x)
    x = Lambda(lambda x: K.mean(x, axis=1))(x)
    return x

def get_concat_backbone(x, cfg):

    # input stem
    x = Conv2D(64, kernel_size=3, padding='same',
use_bias=False)(x)
    x = BatchNormalization(momentum=cfg.momentum)(x)
    x = Activation('relu')(x)

    # backbone
    x = conv_cat_block(x, 96, (1, 2), (3, 2),
cfg.pool_mode, cfg)
    x = conv_cat_block(x, 128, (1, 2), (3, 2),
cfg.pool_mode, cfg)
    x = conv_cat_block(x, 256, (1, 2), (3, 3),
cfg.pool_mode, cfg)
    x = conv_cat_block(x, 512, (1, 2), (3, 2), (None,
None), cfg)  ## [bs,  54, 8, 512]

    # global pooling
    x = Lambda(pixelShuffle)(x)  ## [bs,  54, 4, 1024]
    x = Lambda(lambda x: K.max(x, axis=1))(x)
    x = Lambda(lambda x: K.mean(x, axis=1))(x)
```

```
    return x

def model_se_MSC(x, cfg):
    ratio = 4
    # input stem
    x_3 = Conv2D(32, kernel_size=3, padding='same',
use_bias=False)(x)
    x_5 = Conv2D(32, kernel_size=5, padding='same',
use_bias=False)(x)
    x_7 = Conv2D(32, kernel_size=7, padding='same',
use_bias=False)(x)

    x = concatenate([x_3, x_5, x_7])
    x = BatchNormalization(momentum=cfg.momentum)(x)
    x = Activation('relu')(x)
    x = squeeze_excitation_layer(x, out_dim=96,
ratio=ratio)

    w_ratio = cfg.w_ratio
    # backbone
    x = conv_se_block(x, int(96 * w_ratio), (1, 2), (3,
2), cfg.pool_mode, cfg, ratio=ratio)
    x = conv_se_block(x, int(128 * w_ratio), (1, 2), (3,
2), cfg.pool_mode, cfg, ratio=ratio)
    x = conv_se_block(x, int(256 * w_ratio), (1, 2), (3,
3), cfg.pool_mode, cfg, ratio=ratio)
    x = conv_se_block(x, int(512 * w_ratio), (1, 2), (3,
2), (None, None), cfg, ratio=ratio)

    # global pooling
    x = Lambda(pixelShuffle)(x)
    x = Lambda(lambda x: K.max(x, axis=1))(x)
    x = Lambda(lambda x: K.mean(x, axis=1))(x)
    return x

def cnn_model(cfg):


    x_in = Input((cfg.maxlen,), name='audio')
    feat_in = Input((1,), name='other')
    feat = feat_in

    gfeat_in = Input((128, 12), name='global_feat')
    gfeat = BatchNormalization()(gfeat_in)
    gfeat = Bidirectional(CuDNNGRU(cfg.rnn_unit,
```

```
return_sequences=True), merge_mode='sum')(gfeat)
    gfeat = Bidirectional(CuDNNGRU(cfg.rnn_unit,
return_sequences=True), merge_mode='sum')(gfeat)
    gfeat = GlobalMaxPooling1D()(gfeat)

    x = Lambda(lambda t: K.expand_dims(t, axis=1))(x_in)
    x_mel = Melspectrogram(n_dft=1024, n_hop=512,
input_shape=(1, K.int_shape(x_in)[1]),
                           # n_hop -> stride   n_dft
kernel_size
                           padding='same', sr=44100,
n_mels=64,
                           power_melgram=2,
return_decibel_melgram=True,
                           trainable_fb=False,
trainable_kernel=False,

image_data_format='channels_last', trainable=False)(x)

    x_mel = Lambda(lambda x: K.permute_dimensions(x,
pattern=(0, 2, 1, 3)))(x_mel)
    x = cfg.get_backbone(x_mel, cfg)
    x = concatenate([x, gfeat, feat])
    output = Dense(units=n_classes, activation='sigmoid')
(x)

    y_in = Input((n_classes,), name='y')
    y = y_in

    def get_loss(x):
        y_true, y_pred = x
        loss1 = K.mean(K.binary_crossentropy(y_true,
y_pred))
        return loss1

    loss = Lambda(get_loss)([y, output])
    model = Model(inputs=[x_in, feat_in, gfeat_in,
y_in], outputs=[output])

    if cfg.pretrained is not None:
        model.load_weights("../model/
{}.h5".format(cfg.pretrained))
        print('load_pretrained_success...')

    model.add_loss(loss)
    model.compile(
```

```
        # loss=get_loss,
        optimizer=Nadam(lr=cfg.lr),
    )
    return model


class normNorm(Constraint):
    def __init__(self, axis=0):
        self.axis = axis

    def __call__(self, w):
        # w = K.relu(w)
        # w = K.clip(w,-0.5,1)
        w /= (K.sum(w**2, axis=self.axis,
keepdims=True)**0.5)
        return w

    def get_config(self):
        return {'axis': self.axis}

def stacker(cfg,n):
    def kinit(shape, name=None):
        value = np.zeros(shape)
        value[:, -1] = 1
        return K.variable(value, name=name)


    x_in = Input((80,n))
    x = x_in
    # x = Lambda(lambda x: 1.5*x)(x)
    x =
LocallyConnected1D(1,1,kernel_initializer=kinit,kernel_constraint=n
(x)
    x = Flatten()(x)
    x = Dense(80, use_bias=False,
kernel_initializer=Identity(1))(x)
    x = Lambda(lambda x: (x - 1.6))(x)
    x = Activation('tanh')(x)
    x = Lambda(lambda x:(x+1)*0.5)(x)

    model = Model(inputs=x_in, outputs=x)
    model.compile(
        loss='binary_crossentropy',
        optimizer=Nadam(lr=cfg.lr),
    )
    return model
```

```
if __name__ == '__main__':
    cfg = Config()
    model = cnn_model(cfg)
    print(model.summary())
```

----
3 time_frequency.py

```
# -*- coding: utf-8 -*-
from __future__ import absolute_import
import numpy as np
import keras
from keras import backend as K
from keras.engine import Layer
from keras.utils.conv_utils import conv_output_length
import librosa


def mel(sr, n_dft, n_mels=128, fmin=0.0, fmax=None,
htk=False, norm=1):
    """[np] create a filterbank matrix to combine stft
bins into mel-frequency bins
    use Slaney (said Librosa)

    n_mels: numbre of mel bands
    fmin : lowest frequency [Hz]
    fmax : highest frequency [Hz]
        If `None`, use `sr / 2.0`
    """
    return librosa.filters.mel(sr=sr, n_fft=n_dft,
n_mels=n_mels,
                               fmin=fmin, fmax=fmax,
                               htk=htk,
norm=norm).astype(K.floatx())

def amplitude_to_decibel(x, amin=1e-10,
dynamic_range=80.0):
    """[K] Convert (linear) amplitude to decibel
(log10(x)).
```

```
    x: Keras *batch* tensor or variable. It has to be
batch because of sample-wise `K.max()`.
    amin: minimum amplitude. amplitude smaller than
`amin` is set to this.
    dynamic_range: dynamic_range in decibel
    """
    log_spec = 10 * K.log(K.maximum(x, amin)) /
np.log(10).astype(K.floatx())
    if K.ndim(x) > 1:
        axis = tuple(range(K.ndim(x))[1:])
    else:
        axis = None

    log_spec = log_spec - K.max(log_spec, axis=axis,
keepdims=True)  # [-?, 0]
    log_spec = K.maximum(log_spec, -1 * dynamic_range)
# [-80, 0]
    return log_spec

def get_stft_kernels(n_dft):
    """[np] Return dft kernels for real/imagnary parts
assuming
        the input . is real.
    An asymmetric hann window is used
(scipy.signal.hann).

    Parameters
    ----------
    n_dft : int > 0 and power of 2 [scalar]
        Number of dft components.

    Returns
    -------
        |  dft_real_kernels : np.ndarray
[shape=(nb_filter, 1, 1, n_win)]
        |  dft_imag_kernels : np.ndarray
[shape=(nb_filter, 1, 1, n_win)]

    * nb_filter = n_dft/2 + 1
    * n_win = n_dft

    """
    assert n_dft > 1 and ((n_dft & (n_dft - 1)) == 0), \
        ('n_dft should be > 1 and power of 2, but n_dft
== %d' % n_dft)
```

```
    nb_filter = int(n_dft // 2 + 1)

    # prepare DFT filters
    timesteps = np.array(range(n_dft))
    w_ks = np.arange(nb_filter) * 2 * np.pi /
float(n_dft)
    dft_real_kernels = np.cos(w_ks.reshape(-1, 1) *
timesteps.reshape(1, -1))
    dft_imag_kernels = -np.sin(w_ks.reshape(-1, 1) *
timesteps.reshape(1, -1))

    # windowing DFT filters
    dft_window = librosa.filters.get_window('hann',
n_dft, fftbins=True)  # _hann(n_dft, sym=False)
    dft_window = dft_window.astype(K.floatx())
    dft_window = dft_window.reshape((1, -1))
    dft_real_kernels = np.multiply(dft_real_kernels,
dft_window)
    dft_imag_kernels = np.multiply(dft_imag_kernels,
dft_window)

    dft_real_kernels = dft_real_kernels.transpose()
    dft_imag_kernels = dft_imag_kernels.transpose()
    dft_real_kernels = dft_real_kernels[:, np.newaxis,
np.newaxis, :]
    dft_imag_kernels = dft_imag_kernels[:, np.newaxis,
np.newaxis, :]

    return dft_real_kernels.astype(K.floatx()),
dft_imag_kernels.astype(K.floatx())

class Spectrogram(Layer):
    """
    ### `Spectrogram`

    ```python
    kapre.time_frequency.Spectrogram(n_dft=512,
n_hop=None, padding='same',

power_spectrogram=2.0, return_decibel_spectrogram=False,

trainable_kernel=False, image_data_format='default',
                                     **kwargs)
    ```
    Spectrogram layer that outputs spectrogram(s) in 2D
image format.
```

```
    #### Parameters
     * n_dft: int > 0 [scalar]
       - The number of DFT points, presumably power of 2.
       - Default: ``512``

     * n_hop: int > 0 [scalar]
       - Hop length between frames in sample,  probably
<= ``n_dft``.
       - Default: ``None`` (``n_dft / 2`` is used)

     * padding: str, ``'same'`` or ``'valid'``.
       - Padding strategies at the ends of signal.
       - Default: ``'same'``

     * power_spectrogram: float [scalar],
       -  ``2.0`` to get power-spectrogram, ``1.0`` to
get amplitude-spectrogram.
       -  Usually ``1.0`` or ``2.0``.
       -  Default: ``2.0``

     * return_decibel_spectrogram: bool,
       -  Whether to return in decibel or not, i.e.
returns log10(amplitude spectrogram) if ``True``.
       -  Recommended to use ``True``, although it's not
by default.
       -  Default: ``False``

     * trainable_kernel: bool
       -  Whether the kernels are trainable or not.
       -  If ``True``, Kernels are initialised with DFT
kernels and then trained.
       -  Default: ``False``

     * image_data_format: string, ``'channels_first'``
or ``'channels_last'``.
       -  The returned spectrogram follows this
image_data_format strategy.
       -  If ``'default'``, it follows the current Keras
session's setting.
       -  Setting is in ``./keras/keras.json``.
       -  Default: ``'default'``

    #### Notes
     * The input should be a 2D array, ``(audio_channel,
audio_length)``.
```

```
     * E.g., ``(1, 44100)`` for mono signal, ``(2,
44100)`` for stereo signal.
     * It supports multichannel signal input, so
``audio_channel`` can be any positive integer.
     * The input shape is not related to keras
`image_data_format()` config.

    #### Returns

    A Keras layer

     * abs(Spectrogram) in a shape of 2D data, i.e.,
     * `(None, n_channel, n_freq, n_time)` if
``'channels_first'``,
     * `(None, n_freq, n_time, n_channel)` if
``'channels_last'``,


    """

    def __init__(self, n_dft=512, n_hop=None,
padding='same',
                 power_spectrogram=2.0,
return_decibel_spectrogram=False,
                 trainable_kernel=False,
image_data_format='default', **kwargs):
        assert n_dft > 1 and ((n_dft & (n_dft - 1)) ==
0), \
            ('n_dft should be > 1 and power of 2, but
n_dft == %d' % n_dft)
        assert isinstance(trainable_kernel, bool)
        assert isinstance(return_decibel_spectrogram,
bool)
        # assert padding in ('same', 'valid')
        if n_hop is None:
            n_hop = n_dft // 2

        assert image_data_format in ('default',
'channels_first', 'channels_last')
        if image_data_format == 'default':
            self.image_data_format =
K.image_data_format()
        else:
            self.image_data_format = image_data_format

        self.n_dft = n_dft
```

```
        assert n_dft % 2 == 0
        self.n_filter = n_dft // 2 + 1
        self.trainable_kernel = trainable_kernel
        self.n_hop = n_hop
        self.padding = padding
        self.power_spectrogram = float(power_spectrogram)
        self.return_decibel_spectrogram =
return_decibel_spectrogram
        super(Spectrogram, self).__init__(**kwargs)

    def build(self, input_shape):
        self.n_ch = input_shape[1]
        self.len_src = input_shape[2]
        self.is_mono = (self.n_ch == 1)
        if self.image_data_format == 'channels_first':
            self.ch_axis_idx = 1
        else:
            self.ch_axis_idx = 3
        if self.len_src is not None:
            assert self.len_src >= self.n_dft, 'Hey! The
input is too short!'

        self.n_frame = conv_output_length(self.len_src,
                                          self.n_dft,
                                          self.padding,
                                          self.n_hop)

        dft_real_kernels, dft_imag_kernels =
get_stft_kernels(self.n_dft)
        self.dft_real_kernels =
K.variable(dft_real_kernels, dtype=K.floatx(),
name="real_kernels")
        self.dft_imag_kernels =
K.variable(dft_imag_kernels, dtype=K.floatx(),
name="imag_kernels")
        # kernels shapes: (filter_length, 1, input_dim,
nb_filter)?
        if self.trainable_kernel:

self.trainable_weights.append(self.dft_real_kernels)

self.trainable_weights.append(self.dft_imag_kernels)
        else:

self.non_trainable_weights.append(self.dft_real_kernels)
```

```
self.non_trainable_weights.append(self.dft_imag_kernels)

        super(Spectrogram, self).build(input_shape)
        # self.built = True

    def compute_output_shape(self, input_shape):
        if self.image_data_format == 'channels_first':
            return input_shape[0], self.n_ch,
self.n_filter, self.n_frame
        else:
            return input_shape[0], self.n_filter,
self.n_frame, self.n_ch

    def call(self, x):
        output = self._spectrogram_mono(x[:, 0:1, :])
        if self.is_mono is False:
            for ch_idx in range(1, self.n_ch):
                output = K.concatenate((output,

self._spectrogram_mono(x[:, ch_idx:ch_idx + 1, :])),

axis=self.ch_axis_idx)
        if self.power_spectrogram != 2.0:
            output = K.pow(K.sqrt(output),
self.power_spectrogram)
        if self.return_decibel_spectrogram:
            output = amplitude_to_decibel(output)
        return output

    def get_config(self):
        config = {'n_dft': self.n_dft,
                  'n_hop': self.n_hop,
                  'padding': self.padding,
                  'power_spectrogram':
self.power_spectrogram,
                  'return_decibel_spectrogram':
self.return_decibel_spectrogram,
                  'trainable_kernel':
self.trainable_kernel,
                  'image_data_format':
self.image_data_format}
        base_config = super(Spectrogram,
self).get_config()
        return dict(list(base_config.items()) +
list(config.items()))
```

```
    def _spectrogram_mono(self, x):
        '''x.shape : (None, 1, len_src),
        returns 2D batch of a mono power-spectrogram'''
        x = K.permute_dimensions(x, [0, 2, 1])
        x = K.expand_dims(x, 3)  # add a dummy dimension
(channel axis)
        subsample = (self.n_hop, 1)
        output_real = K.conv2d(x, self.dft_real_kernels,
                               strides=subsample,
                               padding=self.padding,

data_format='channels_last')
        output_imag = K.conv2d(x, self.dft_imag_kernels,
                               strides=subsample,
                               padding=self.padding,

data_format='channels_last')
        output = output_real ** 2 + output_imag ** 2
        # now shape is (batch_sample, n_frame, 1, freq)
        if self.image_data_format == 'channels_last':
            output = K.permute_dimensions(output, [0, 3,
1, 2])
        else:
            output = K.permute_dimensions(output, [0, 2,
3, 1])
        return output


class Melspectrogram(Spectrogram):
    '''
    ### `Melspectrogram`
    ```python
    kapre.time_frequency.Melspectrogram(sr=22050,
n_mels=128, fmin=0.0, fmax=None,

power_melgram=1.0, return_decibel_melgram=False,

trainable_fb=False, **kwargs)
    ```
d
    Mel-spectrogram layer that outputs mel-
spectrogram(s) in 2D image format.

    Its base class is ``Spectrogram``.

    Mel-spectrogram is an efficient representation using
```

```
the property of human
    auditory system -- by compressing frequency axis
into mel-scale axis.

    #### Parameters
     * sr: integer > 0 [scalar]
       - sampling rate of the input audio signal.
       - Default: ``22050``

     * n_mels: int > 0 [scalar]
       - The number of mel bands.
       - Default: ``128``

     * fmin: float > 0 [scalar]
       - Minimum frequency to include in Mel-spectrogram.
       - Default: ``0.0``

     * fmax: float > ``fmin`` [scalar]
       - Maximum frequency to include in Mel-spectrogram.
       - If `None`, it is inferred as ``sr / 2``.
       - Default: `None`

     * power_melgram: float [scalar]
       - Power of ``2.0`` if power-spectrogram,
       - ``1.0`` if amplitude spectrogram.
       - Default: ``1.0``

     * return_decibel_melgram: bool
       - Whether to return in decibel or not, i.e.
returns log10(amplitude spectrogram) if ``True``.
       - Recommended to use ``True``, although it's not
by default.
       - Default: ``False``

     * trainable_fb: bool
       - Whether the spectrogram -> mel-spectrogram
filterbanks are trainable.
       - If ``True``, the frequency-to-mel matrix is
initialised with mel frequencies but trainable.
       - If ``False``, it is initialised and then frozen.
       - Default: `False`

     * htk: bool
       - Check out Librosa's `mel-spectrogram` or `mel`
option.
```

```
     * norm: float [scalar]
       - Check out Librosa's `mel-spectrogram` or `mel`
option.

     * **kwargs:
       - The keyword arguments of ``Spectrogram`` such
as ``n_dft``, ``n_hop``,
       - ``padding``, ``trainable_kernel``,
``image_data_format``.

    #### Notes
     * The input should be a 2D array, ``(audio_channel,
audio_length)``.
    E.g., ``(1, 44100)`` for mono signal, ``(2, 44100)``
for stereo signal.
     * It supports multichannel signal input, so
``audio_channel`` can be any positive integer.
     * The input shape is not related to keras
`image_data_format()` config.

    #### Returns

    A Keras layer
     * abs(mel-spectrogram) in a shape of 2D data, i.e.,
     * `(None, n_channel, n_mels, n_time)` if
``'channels_first'``,
     * `(None, n_mels, n_time, n_channel)` if
``'channels_last'``,

    '''

    def __init__(self,
                 sr=22050, n_mels=128, fmin=0.0,
fmax=None,
                 power_melgram=1.0,
return_decibel_melgram=False,
                 trainable_fb=False, htk=False, norm=1,
**kwargs):

        super(Melspectrogram, self).__init__(**kwargs)
        assert sr > 0
        assert fmin >= 0.0
        if fmax is None:
            fmax = float(sr) / 2
        assert fmax > fmin
        assert isinstance(return_decibel_melgram, bool)
```

```
        if 'power_spectrogram' in kwargs:
            assert kwargs['power_spectrogram'] == 2.0, \
                'In Melspectrogram, power_spectrogram
should be set as 2.0.'

        self.sr = int(sr)
        self.n_mels = n_mels
        self.fmin = fmin
        self.fmax = fmax
        self.return_decibel_melgram =
return_decibel_melgram
        self.trainable_fb = trainable_fb
        self.power_melgram = power_melgram
        self.htk = htk
        self.norm = norm

    def build(self, input_shape):
        super(Melspectrogram, self).build(input_shape)
        self.built = False
        # compute freq2mel matrix -->
        mel_basis = mel(self.sr, self.n_dft,
self.n_mels, self.fmin, self.fmax,
                                self.htk, self.norm)  #
(128, 1025) (mel_bin, n_freq)
        mel_basis = np.transpose(mel_basis)

        self.freq2mel = K.variable(mel_basis,
dtype=K.floatx())
        if self.trainable_fb:
            self.trainable_weights.append(self.freq2mel)
        else:

self.non_trainable_weights.append(self.freq2mel)
        self.built = True

    def compute_output_shape(self, input_shape):
        if self.image_data_format == 'channels_first':
            return input_shape[0], self.n_ch,
self.n_mels, self.n_frame
        else:
            return input_shape[0], self.n_mels,
self.n_frame, self.n_ch

    def call(self, x):
        power_spectrogram = super(Melspectrogram,
self).call(x)
```

```
        # now,  channels_first: (batch_sample, n_ch,
n_freq, n_time)
        #       channels_last: (batch_sample, n_freq,
n_time, n_ch)
        if self.image_data_format == 'channels_first':
            power_spectrogram =
K.permute_dimensions(power_spectrogram, [0, 1, 3, 2])
        else:
            power_spectrogram =
K.permute_dimensions(power_spectrogram, [0, 3, 2, 1])
        # now, whatever image_data_format,
(batch_sample, n_ch, n_time, n_freq)
        output = K.dot(power_spectrogram, self.freq2mel)
        if self.image_data_format == 'channels_first':
            output = K.permute_dimensions(output, [0, 1,
3, 2])
        else:
            output = K.permute_dimensions(output, [0, 3,
2, 1])
        if self.power_melgram != 2.0:
            output = K.pow(K.sqrt(output),
self.power_melgram)
        if self.return_decibel_melgram:
            output = amplitude_to_decibel(output)
        return output

    def get_config(self):
        config = {'sr': self.sr,
                  'n_mels': self.n_mels,
                  'fmin': self.fmin,
                  'fmax': self.fmax,
                  'trainable_fb': self.trainable_fb,
                  'power_melgram': self.power_melgram,
                  'return_decibel_melgram':
self.return_decibel_melgram,
                  'htk': self.htk,
                  'norm': self.norm}
        base_config = super(Melspectrogram,
self).get_config()
        return dict(list(base_config.items()) +
list(config.items()))



class AdditiveNoise(Layer):
```

```
    def __init__(self, power=0.1, random_gain=False,
noise_type='white', **kwargs):
        assert noise_type in ['white']
        self.supports_masking = True
        self.power = power
        self.random_gain = random_gain
        self.noise_type = noise_type
        self.uses_learning_phase = True
        super(AdditiveNoise, self).__init__(**kwargs)

    def call(self, x):
        if self.random_gain:
            noise_x = x +
K.random_normal(shape=K.shape(x),
                                          mean=0.,

stddev=np.random.uniform(0.0, self.power))
        else:
            noise_x = x +
K.random_normal(shape=K.shape(x),
                                          mean=0.,

stddev=self.power)

        return K.in_train_phase(noise_x, x)

    def get_config(self):
        config = {'power': self.power,
                  'random_gain': self.random_gain,
                  'noise_type': self.noise_type}
        base_config = super(AdditiveNoise,
self).get_config()
        return dict(list(base_config.items()) +
list(config.items()))

----------
```