DNN Speech Recognizer

REVIEW

CODE REVIEW 6

HISTORY

▼ sample_models.py 6

```
from keras import backend as K
from keras.models import Model
from keras.layers import (BatchNormalization, Conv1D, Dense, Input,
    TimeDistributed, Activation, Bidirectional, SimpleRNN, GRU, LSTM, MaxPool1D, Dropout)

def simple_rnn_model(input_dim, output_dim=29):
    """ Build a recurrent network for speech
    """
    # Main acoustic input
    input_data = Input(name='the_input', shape=(None, input_dim))
```

```
    # Add recurrent layer
    simp_rnn = GRU(output_dim, return_sequences=True,
                 implementation=2, name='rnn')(input_data)
    # Add softmax activation layer
    y_pred = Activation('softmax', name='softmax')(simp_rnn)
    # Specify the model
    model = Model(inputs=input_data, outputs=y_pred)
    model.output_length = lambda x: x
    print(model.summary())
    return model

def rnn_model(input_dim, units, activation, output_dim=29):
    """ Build a recurrent network for speech
    """
    # Main acoustic input
    input_data = Input(name='the_input', shape=(None, input_dim))
    # Add recurrent layer
    simp_rnn = GRU(units, activation=activation,
        return_sequences=True, implementation=2, name='rnn')(input_data)
    # TODO: Add batch normalization
    bn_rnn = BatchNormalization(name='bn_rnn')(simp_rnn)
```

SUGGESTION

Best use of BatchNorm can be done just after input layer

```
    # TODO: Add a TimeDistributed(Dense(output_dim)) layer
    time_dense = TimeDistributed(Dense(output_dim))(bn_rnn)
    # Add softmax activation layer
    y_pred = Activation('softmax', name='softmax')(time_dense)
    # Specify the model
    model = Model(inputs=input_data, outputs=y_pred)
    model.output_length = lambda x: x
    print(model.summary())
    return model


def cnn_rnn_model(input_dim, filters, kernel_size, conv_stride,
    conv_border_mode, units, output_dim=29):
    """ Build a recurrent + convolutional network for speech
    """
    # Main acoustic input
    input_data = Input(name='the_input', shape=(None, input_dim))
    # Add convolutional layer
```

```
    conv_1d = Conv1D(filters, kernel_size,
                     strides=conv_stride,
                     padding=conv_border_mode,
                     activation='relu',
                     name='conv1d')(input_data)
    # Add batch normalization
    bn_cnn = BatchNormalization(name='bn_conv_1d')(conv_1d)
    # Add a recurrent layer
    simp_rnn = SimpleRNN(units, activation='relu',
```

SUGGESTION

Try using a better RNN network like GRU here

```
        return_sequences=True, implementation=2, name='rnn')(bn_cnn)
    # TODO: Add batch normalization
    bn_rnn = BatchNormalization(name='bn_simp_rnn')(simp_rnn)
    # TODO: Add a TimeDistributed(Dense(output_dim)) layer
    time_dense = TimeDistributed(Dense(output_dim))(bn_rnn)
    # Add softmax activation layer
    y_pred = Activation('softmax', name='softmax')(time_dense)
    # Specify the model
    model = Model(inputs=input_data, outputs=y_pred)
    model.output_length = lambda x: cnn_output_length(
        x, kernel_size, conv_border_mode, conv_stride)
    print(model.summary())
    return model

def cnn_output_length(input_length, filter_size, border_mode, stride,
                      dilation=1):
    """ Compute the length of the output sequence after 1D convolution along
        time. Note that this function is in line with the function used in
        Convolution1D class from Keras.
    Params:
        input_length (int): Length of the input sequence.
        filter_size (int): Width of the convolution kernel.
        border_mode (str): Only support `same` or `valid`.
        stride (int): Stride size used in 1D convolution.
        dilation (int)
    """
    if input_length is None:
        return None
    assert border_mode in {'same', 'valid'}
    dilated_filter_size = filter_size + (filter_size - 1) * (dilation - 1)
```

```
    if border_mode == 'same':
        output_length = input_length
    elif border_mode == 'valid':
        output_length = input_length - dilated_filter_size + 1
    return (output_length + stride - 1) // stride

def deep_rnn_model(input_dim, units, recur_layers, output_dim=29):
    """ Build a deep recurrent network for speech
    """
    # Main acoustic input
    input_data = Input(name='the_input', shape=(None, input_dim))
    # TODO: Add recurrent layers, each with batch normalization
    for i in range(recur_layers):
```

AWESOME

Well done on robust implementation using `recur_layers`

```
        if i==0:
            simp_rnn = GRU(units, return_sequences=True)(input_data)
        else:
            simp_rnn = GRU(units, return_sequences=True)(bn_rnn)
        bn_rnn = BatchNormalization()(simp_rnn)
    # TODO: Add a TimeDistributed(Dense(output_dim)) layer
    time_dense = TimeDistributed(Dense(output_dim))(bn_rnn)
    # Add softmax activation layer
    y_pred = Activation('softmax', name='softmax')(time_dense)
    # Specify the model
    model = Model(inputs=input_data, outputs=y_pred)
    model.output_length = lambda x: x
    print(model.summary())
    return model


def bidirectional_rnn_model(input_dim, units, output_dim=29):
    """ Build a bidirectional recurrent network for speech
    """
    # Main acoustic input
    input_data = Input(name='the_input', shape=(None, input_dim))
    # TODO: Add bidirectional recurrent layer
    bidir_rnn = Bidirectional(GRU(units, return_sequences=True, name = 'bidir_rnn'), merge_mode='concat')(input_data)
```

AWESOME

Correct Implementation of Bidirectional layer

```
    # TODO: Add a TimeDistributed(Dense(output_dim)) layer
    time_dense = TimeDistributed(Dense(output_dim))(bidir_rnn)
    # Add softmax activation layer
    y_pred = Activation('softmax', name='softmax')(time_dense)
    # Specify the model
    model = Model(inputs=input_data, outputs=y_pred)
    model.output_length = lambda x: x
    print(model.summary())
    return model


# def final_model(input_dim, filters, kernel_size, conv_stride,
#     conv_border_mode, units, output_dim=29):
#     """ Build a deep network for speech
#     """
#     # Main acoustic input
#     input_data = Input(name='the_input', shape=(None, input_dim))
#     # TODO: Specify the layers in your network

#     # Add convolutional layer
#     conv_1d = Conv1D(filters, kernel_size,
#                      strides=conv_stride,
#                      padding=conv_border_mode,
#                      activation='relu',
#                      name='conv1d')(input_data)
#     # dropout layer
#     drop = Dropout(0.2)(conv_1d)
#     # Add batch normalization
#     bn_cnn = BatchNormalization(name='bn_conv_1d')(drop)

    # Add bidirectional recurrent layer
#     bidir_rnn = Bidirectional(GRU(units, return_sequences=True, name = 'bidir_rnn'), merge_mode='concat')(bn_cnn)
#     bn_bi_rnn = BatchNormalization(name='bn__bi_rnn')(bidir_rnn)

#     bidir_rnn = Bidirectional(GRU(units, activation='relu',
#         return_sequences=True, implementation=2, name='bid_rnn0', dropout=0.2))(bn_cnn)
#     bn_bi_rnn = BatchNormalization(name='bn_rnn1')(bidir_rnn)
#     bidir_rnn = Bidirectional(GRU(units, activation='relu',
#         return_sequences=True, implementation=2, name='bid_rnn1', dropout=0.2))(bn_bi_rnn)
#     bn_bi_rnn = BatchNormalization(name='bn_rnn2')(bidir_rnn)
```

```
#     # Add two RNN layers
#     simp_rnn1 = GRU(units, return_sequences=True, dropout_W=0.0, dropout_U=0.1)(bn_cnn)
#     bn_rnn1 = BatchNormalization(name='bn_conv_1d1')(simp_rnn1)
#     simp_rnn2 = GRU(units, return_sequences=True, dropout_W=0.5, dropout_U=0.1)(bn_rnn1)
#     bn_rnn2 = BatchNormalization(name='bn_conv_1d2')(simp_rnn2)

#     # Add three bidirectional recurrent layers
#     bidir_rnn = Bidirectional(GRU(units, return_sequences=True, implementation=2, name='bidirnn0', dropout=0.2, recurren
#     bidir_rnn = Bidirectional(GRU(units, return_sequences=True, implementation=2, name='bidirnn1', dropout=0.2, recurren
#     bidir_rnn = Bidirectional(GRU(units, return_sequences=True, implementation=2, name='bidirnn2', dropout=0.2, recurren


    # Add a TimeDistributed(Dense(output_dim)) layer
#     time_dense = TimeDistributed(Dense(output_dim))(bn_rnn2)
#     # TODO: Add softmax activation layer
#     y_pred = Activation('softmax', name='softmax')(time_dense)

#     # Specify the model
#     model = Model(inputs=input_data, outputs=y_pred)

#     # Specify model.output_length
#     model.output_length = lambda x: cnn_output_length(
#         x, kernel_size, conv_border_mode, conv_stride)

#     print(model.summary())
#     return model

def final_model(input_dim, filters, kernel_size, conv_stride,
    conv_border_mode, units, output_dim=29):
    """ Build a deep network for speech
        cnn + 2 bidirectional recurrent network for speech
    """
    # Main acoustic input
    input_data = Input(name='the_input', shape=(None, input_dim))
    # TODO: Specify the layers in your network
    # Add convolutional layer
    conv_1d = Conv1D(filters, kernel_size,
                     strides=conv_stride,
                     padding=conv_border_mode,
                     activation='relu',
                     name='conv1d')(input_data)

#     # dropout layer
#     conv_1d = Dropout(0.2)(conv_1d)
    # Add batch normalization
```

```
    bn_cnn = BatchNormalization(name='bn_conv_1d')(conv_1d)

    # Add two RNN layers
    simp_rnn1 = GRU(units, return_sequences=True, dropout_W=0.0, dropout_U=0.1)(bn_cnn)
```

SUGGESTION

Try using the concept of `recur_layers` here

You may remove `dropout_W` if you are setting it to 0

```
    bn_rnn1 = BatchNormalization(name='bn_conv_1d1')(simp_rnn1)
    simp_rnn2 = GRU(units, return_sequences=True, dropout_W=0.5, dropout_U=0.1)(bn_rnn1)
```

AWESOME

Well done on using dropout here.

You may also add dropout layer

```
    bn_rnn2 = BatchNormalization(name='bn_conv_1d2')(simp_rnn2)

    # Add a TimeDistributed(Dense(output_dim)) layer
    time_dense = TimeDistributed(Dense(output_dim))(bn_rnn2)

    # TODO: Add softmax activation layer
    y_pred = Activation('softmax', name='softmax')(time_dense)
    # Specify the model
    model = Model(inputs=input_data, outputs=y_pred)
    # TODO: Specify model.output_length
    model.output_length = lambda x: cnn_output_length(
        x, kernel_size, conv_border_mode, conv_stride)
    print(model.summary())
    return model
```

- workspace-utils.py
- utils.py
- train_utils.py
- data_generator.py
- char_map.py

RETURN TO PATH