Sequential models, attention and ensemble with meta-embedding for news classification

Bjenk Ellefsen, Ph.D.

The strategy employed in the following methods is to identify as many articles that have been selected over two years to brief the Minsiters. Therefore, we seek to select as much ones as possible. We are also seeking a balance with good predictions on zeros (not part of the briefing packages) as to reduce penalties on an overall quality f1 and roc-auc score.

Most of the decisions on methods and strategies in this notebook were derived from a previous baseline contained in a separate notebook where metrics such as precision, recall, roc-auc and log loss were monitored over many models.

Note on reproducibility of this notebook: the training was not run deterministically. While we could have factored for all randomness, we simply ran the training multiple times, averaging to monitor for progress and keeping the best metrics as the best models. In running this code, there will be some randomness. The reasons for this are many. CuDNN has two initializers that can be seeded, Keras fit shuffles as a default, regularization such as dropout can take a seed value, and so on. GPU libraries also carry some randomness that may be difficult to account for. It should also be said that in Pytorch, it is possible to completely set the process to be deterministic, and in the future we will move our work to the Pytorch framework. e.g:

In [1]:
```
# This is meant as an example of Pytorch's deterministic methods.

# SEED = 1337
# np.random.seed(SEED)
# torch.manual_seed(SEED)
# torch.cuda.manual_seed(SEED)
# torch.backends.cudnn.deterministic = True
```

In [2]:
```
%load_ext autoreload
%autoreload 2
%matplotlib inline
```

In [3]:
```
# Check for GPU
# import tensorflow as tf
# from tensorflow.python.client import device_lib
# print(device_lib.list_local_devices())
```

In [4]:
```
import math
import numpy as np
import pandas as pd
import random
import os
import re
import matplotlib.pyplot as plt

# scikit-sklearn
import sklearn.metrics as sklm
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, roc_auc_score
from sklearn.metrics import precision_score, recall_score, confusion_mat
rix, f1_score, classification_report

# Keras
from keras.preprocessing.text import Tokenizer
from keras.preprocessing import text, sequence
from keras.optimizers import Adam
from keras.callbacks import EarlyStopping, ModelCheckpoint, LearningRate
Scheduler, EarlyStopping,ModelCheckpoint, Callback
from keras.preprocessing.sequence import pad_sequences
from keras.layers import Dense, Flatten, LSTM, GRU, Conv1D, Conv2D, MaxP
ooling1D, Dropout, Concatenate, concatenate, Input, Embedding, MaxPool2D
from keras.layers import Reshape, SpatialDropout1D, Activation, Bidirect
ional, GlobalAveragePooling1D, GlobalMaxPooling1D, BatchNormalization
from keras.layers import CuDNNLSTM, CuDNNGRU, Lambda
from keras import optimizers
from keras.models import Model, load_model, Sequential
from keras import initializers, regularizers, constraints, layers, callb
acks, Input, optimizers
from keras.engine import InputSpec, Layer
from keras import backend as K
from keras.engine.topology import Layer
```

```
Using TensorFlow backend.
```

In [8]:
```
 # Get the filenames in each directory (train and test)
data_dir = 'data/no_dup_en'
filenames = os.listdir(data_dir)
filenames = [os.path.join(data_dir, f) for f in filenames if f.endswith(
'.json')]
```

In [5]:
```
programs = pd.read_csv("data/program_descriptions/all_programs.csv", enc
oding="utf8").dropna()
names = pd.read_csv("data/Important_Names.csv" ,encoding="latin-1")
#program_names = pd.read_csv("data/program_names.csv", encoding="latin-
1")

names.columns = ["important","other"]

important_names = names.important.dropna().map(lambda x: x.strip()).str.
lower().values
other_names = names.other.dropna().map(lambda x: x.strip()).str.lower().
values
#program_names = program_names.names.dropna().map(lambda x: x.strip()).s
tr.lower().values
```

In [10]:
```
len(filenames)
```

Out[10]:
```
681
```

In [11]:
```
# Split the images in 'train_signs' into 80% train and 20% dev
# Make sure to always shuffle with a fixed seed so that the split is rep
roducible
# This does not shuffle the order. To do so, random.shuffle would need t
o be used.

random.seed(230)
filenames.sort()
#random.shuffle(filenames)
split = int(0.96 * len(filenames))
train_filenames = filenames[:split]
test_filenames = filenames[split:]
filenames[0:5]
```

Out[11]:
```
['data/no_dup_en/2015-12-28.json',
 'data/no_dup_en/2015-12-29.json',
 'data/no_dup_en/2015-12-30.json',
 'data/no_dup_en/2015-12-31.json',
 'data/no_dup_en/2016-01-01.json']
```

In [12]:
```
print(len(train_filenames), len(test_filenames))
train_filenames[0:5]
```

```
653 28
```

Out[12]:
```
['data/no_dup_en/2015-12-28.json',
 'data/no_dup_en/2015-12-29.json',
 'data/no_dup_en/2015-12-30.json',
 'data/no_dup_en/2015-12-31.json',
 'data/no_dup_en/2016-01-01.json']
```

In [13]:
```
# Read files after split and import news articles into dataframes
train_df = pd.concat([pd.read_json(file, encoding='UTF-8') for file in t
rain_filenames], ignore_index = True)
test_df = pd.concat([pd.read_json(file, encoding='UTF-8') for file in te
st_filenames], ignore_index = True)
```

In [24]:
```
```

Out[24]:
```
array([0, 3, 5, 4, 2, 1])
```

In [18]:
```
# Near duplicates were identified with cosine similarity and removed her
e. This script is not contained in this notebook. W
# We found that removing near duplicates with cosine similarity do not s
ignificantly impact the results.
train_df = train_df[train_df.duplicate != 1]
test_df = test_df[test_df.duplicate != 1]
```

In [19]:
```
len(train_df), len(test_df)
```

Out[19]:
```
(590391, 138143)
```

The objective is to train a classifier for the 1 to 5 categories which translates to "in package or pdf". To do so, we subset for the news articles that are filtered by NewsDesk based on keywords detemrined by ESDC. Therefore, the subset removes the news articles that do not contain a tag identifying it as an ESDC article. Future work will explore obtaining a learned subset from the main articles pool.

In [20]:
```
train_df = train_df[~(train_df['categories'].astype(str) == '[]')].reset
_index(drop=True)
test_df = test_df[~(test_df['categories'].astype(str) == '[]')]
```

In [21]:
```
print(len(train_df), len(test_df))
```

```
541011 22986
```

Classes count

In [12]:
```
print(train_df.in_pdf.value_counts()), print(""), print(test_df.in_pdf.v
alue_counts())
```

```
0    516362
1     24649
Name: in_pdf, dtype: int64

0    21773
1     1213
Name: in_pdf, dtype: int64
```

Out[12]:
```
(None, None, None)
```

Classes imbalance: undersampling the majority class

As it can be seen above, the data has a significant imbalance between the classes 0 and 1. An imbalance can lead models to be biased as it is seeing many more examples of one over the other. In such cases, oversampling the minority class or undersampling the majority are valid approaches. This is done only on the train set, never on the test set. Note that accuracy is useless in the case of imbalances and we relied on f1 score, roc-auc and log loss for error analysis. To undersample the majority, we used a resample with replacement. This will shuffle the news articles so sort_values on the variable "creationDate" must be used after the join. The reason for this step is that we made the assumption that we should preserve the timeline in the news article.

In [13]:
```
from sklearn.utils import resample
# class counts

class_0, class_1 = train_df.in_pdf.value_counts()

# Split majority and minority

train_df_maj = train_df[train_df.in_pdf == 0]
train_df_min = train_df[train_df.in_pdf == 1]

# Resample majority to under sample it
train_df_maj_under = resample(train_df_maj, replace = False, n_samples = 
class_1, random_state = 230) # random_state is like seed, it allows for
 reproducibility

# Join the classes back together
train_df_undersampled = pd.concat([train_df_maj_under, train_df_min]).so
rt_values('creationDate')

# Check the balance between majority and minority
train_df_undersampled.in_pdf.value_counts()
```

Out[13]:
```
1    24649
0    24649
Name: in_pdf, dtype: int64
```

Cleaning and tagging

The strategy for cleaning is to remove occurences of html tags, punctuations while leaving hyphenated words. We are laso tagging names with an "anchor" which provides a consistent marker for when the name of an important individual from ESDC or stakeholder relevant to ESDC are mentioned.

In [14]:
```
#Tagging is taking a lot of time with programname. Using compile().sub()
appears to improve time.
def clean_text(text):

    text = text.lower()

    for name in important_names:
        text = re.compile(r"\b{}\b".format(name)).sub("importantname", t
ext)

    for name in other_names:
        text = re.compile(r"\b{}\b".format(name)).sub("othername", text)

#     for name in program_names:
#         text = re.compile(r"\b{}\b".format(name)).sub("programname", t
ext)

    text  = re.compile('<[^<]+?>').sub('', text)
    text = re.compile('[^a-zA-Z0-9\s\-]').sub('', text)
    # text = re.compile("[^a-zA-Z0-9\s]+").sub('', text)
    # text = re.compile('[0-9]+').sub('#', text)

    return text
```

In [15]:
```
%%time
train_df_undersampled['body_clean'] = train_df_undersampled['body'].appl
y(lambda x: clean_text(str(x)))
test_df['body_clean'] = test_df['body'].apply(lambda x: clean_text(str(x
)))
```

```
CPU times: user 11min 27s, sys: 54.9 ms, total: 11min 27s
Wall time: 11min 27s
```

In [16]:
```
#train_df['body'].iloc[12]
#train_df_undersampled['body_clean'].iloc[87]
```

Tokenizing and creating the word index

We use teh Keras tokenizer class to vectorize the news articles into tokens; sentences split into words with a defined length and total vocabulary size. We use a word index to create a dictionary of words and assign a unique integer for each. Tokenizer removes all punctuations and

The method fit_on_texts creates the vocabulary index based on word frequency so every word gets a unique integer value. A lower integer means more frequent words. The, the texts_to_sequences transforms text into a sequence of integers. So it basically takes each word in the text and replaces it with its corresponding integer value from the word_index dictionary.

In other words, Tokenizer transforms sentences into a word representation of numbers with the most common word being represented as 1, the second most common with 2 and so on. By setting a number to the vocabulary, we keep only the N most common words found in the news articles. We have used a rather large N here. Tokenizer in Keras is a two steps process which begins with computing the word frequencies where the most common ones are assigned a low integer. Then, the text is transformed into numerical tokens. The word frequencies computation corresponds to fit_on_texts, as in "fitting" the tokeniser and the transformation to numerical tokens as texts_to_sequences. Keras refers to numerical representation as "sequences".

In [17]:
```
%%time
embed_size = 300
vocabulary_size = 200000
max_len = 900

X_train = train_df_undersampled['body_clean'].fillna("_##_").values
X_test = test_df['body_clean'].fillna("_##_").values

tokenizer = Tokenizer(num_words= vocabulary_size)
tokenizer.fit_on_texts(list(X_train))

train_sequences = tokenizer.texts_to_sequences(X_train)
test_sequences = tokenizer.texts_to_sequences(X_test)

x_train = pad_sequences(train_sequences, maxlen=max_len)
x_test = pad_sequences(test_sequences, maxlen=max_len)
word_index = tokenizer.word_index

y_train = train_df_undersampled['in_pdf'].values
y_test = test_df['in_pdf'].values
```

```
CPU times: user 47.8 s, sys: 88.5 ms, total: 47.9 s
Wall time: 47.9 s
```

In [18]:
```
# Save the tokenizer model
import pickle
# saving
with open('tokenizer.pickle', 'wb') as handle:
    pickle.dump(tokenizer, handle, protocol=pickle.HIGHEST_PROTOCOL)
```

In [19]:
```
print(len(x_train), len(x_test))
```

```
49298 22986
```

In [20]:
```
print(len(x_train), len(y_train))
```

```
49298 49298
```

Validation split

In [21]:
```
# Split again for validation set
X_tra, X_val, y_tra, y_val = train_test_split(x_train, y_train, test_siz
e=0.17, shuffle = False, random_state=233)
```

X_val.shape, y_val.shape

Models

The use of sequence models proved to be a fundamental part of the solution for the NewsDesk dataset. First, sequence means such data as dna, audio signals, heart rate, and text. Sentences are sequences of words and so are a natural fit for models that are designed to operate on sequences. As such, recurrent neural networks have proven to be powerful for language models.

The problems that arise with sequence data is that sequences can have different lenghts, sentences do vary considerably. Also, a naive network does not share features learned across different positions of texts. So, we want a similar effect for sequences as the CNN has on images, that is to say, one thing learned from part of the image generalized to the other parts.

Recurrent neural networks do not have either of these disadvantages. A bidirectional RNN will use information from earlier in the sequence but also from the future in the sequence. We are using a combination of bidirectional Long short term memory (Hochreiter, Schmidhuber, 1997 (https://www.mitpressjournals.org/doi/10.1162/neco.1997.9.8.1735)) and gated recurrent unit (Cho et. al., 2014 (https://arxiv.org/abs/1406.1078)) networks. Both types are very similar and are good at long range connections in sequences such as sentences. LSTM is usually more powerful than GRU and more general as well. The point is that they are both very robist for long range sequences.

LSTM and GRU learn representations from previous time steps but to capture context, it capures information ahead in the sequence through the bidirectional feature. As such, they will have forward and backward connections. This is not like backpropagation, it is still forward propagation but it has part of the computation going backward and forward. We combine these with a deeper structure by using them together as layers. Usually, two or three layers is considered deep enough for RNNs as the dimensions can become quite large.

Ensembling

A word on ensembling and stacking: they are methods used to increase the predictive performance of models. The general intuition is to combine the predictions of multiple models and in combining information we can generate a more powerful model that outpermforms the individual ones.

An ensemble is, usually, outperforming single models because of its smoothing nature: it captures the best performances of the single models, and discredits the poorest.

An ensemble is most effective when the single models are significantly different. This is important because we are using the predictions of the individual models as features for the ensemble. So, with an ensemble, we can find where the single models do well and where they don't and with different models, we can minimize the correlations between the errors of the single models (Hinton et al. 2015).

Below, we define two models making use of both LSTM and GRU but also CNN as a layer in the first model and as a model on its own with 2 dimensions. The objectives of these three models are to provide significantly different learning approaches for the ensemble. The three models were chosen in a baseline previous work where up to ten models were tested. IN the training phase of these test models, we monitored precision, recall, f1 score, roc-auc and log-loss. These metrics allowed us to decide which models to use for the ensembling test phase.

- Geoffrey Hinton, Oriol Vinyals, Jeff Dean. 2015. Distilling the Knowledge in a Neural Network.

Attention mechanism

The attention mechanism layer is a keras custom layer adaptation of Raffel et.al, 2016 (https://arxiv.org/abs/1512.08756). This will be used in one of the models below.

github keras code (https://github.com/craffel/ff-attention/blob/master/layers.py).

In [23]:
```
class Attention(Layer):
    def __init__(self, step_dim,
                 W_regularizer=None, b_regularizer=None,
                 W_constraint=None, b_constraint=None,
                 bias=True, **kwargs):
        self.supports_masking = True
        self.init = initializers.get('glorot_uniform')

        self.W_regularizer = regularizers.get(W_regularizer)
        self.b_regularizer = regularizers.get(b_regularizer)

        self.W_constraint = constraints.get(W_constraint)
        self.b_constraint = constraints.get(b_constraint)

        self.bias = bias
        self.step_dim = step_dim
        self.features_dim = 0
        super(Attention, self).__init__(**kwargs)

    def build(self, input_shape):
        assert len(input_shape) == 3

        self.W = self.add_weight((input_shape[-1],),
                                 initializer=self.init,
                                 name='{}_W'.format(self.name),
                                 regularizer=self.W_regularizer,
                                 constraint=self.W_constraint)
        self.features_dim = input_shape[-1]

        if self.bias:
            self.b = self.add_weight((input_shape[1],),
                                     initializer='zero',
                                     name='{}_b'.format(self.name),
                                     regularizer=self.b_regularizer,
                                     constraint=self.b_constraint)
        else:
            self.b = None

        self.built = True

    def compute_mask(self, input, input_mask=None):
        return None

    def call(self, x, mask=None):
        features_dim = self.features_dim
        step_dim = self.step_dim

        eij = K.reshape(K.dot(K.reshape(x, (-1, features_dim)),
                        K.reshape(self.W, (features_dim, 1))), (-1, step
_dim))

        if self.bias:
            eij += self.b

        eij = K.tanh(eij)
```

```
        a = K.exp(eij)

        if mask is not None:
            a *= K.cast(mask, K.floatx())

        a /= K.cast(K.sum(a, axis=1, keepdims=True) + K.epsilon(), K.flo
atx())

        a = K.expand_dims(a)
        weighted_input = x * a
        return K.sum(weighted_input, axis=1)

    def compute_output_shape(self, input_shape):
        return input_shape[0],  self.features_dim

    def get_config(self):
        config = {
            'step_dim': self.step_dim

        }
        base_config = super(Attention, self).get_config()
        return dict(list(base_config.items()) + list(config.items()))
```

LSTM, GRU, CNN

In [24]:
```
def model_lstm_gru_cnn(embedding_matrix):
    inp = Input(shape=(max_len,))
    x = Embedding(vocabulary_size, embed_size, weights=[embedding_matrix
], trainable=False)(inp)
    x = SpatialDropout1D(0.50)(x)
    x = Bidirectional(CuDNNLSTM(100, return_sequences=True))(x)
    x = Bidirectional(CuDNNGRU(100, return_sequences=True))(x)
    x = Conv1D(64, kernel_size = 3, padding = 'valid', kernel_initialize
r = 'glorot_uniform')(x)
    avg_pool = GlobalAveragePooling1D()(x)
    max_pool = GlobalMaxPooling1D()(x)
    x = concatenate([avg_pool, max_pool])
    out = Dense(1, activation = 'sigmoid')(x)

    model = Model(inp, out)
    model.compile(loss='binary_crossentropy', optimizer = 'adam', metric
s = ['accuracy'])
    return model
```

LSTM GRU 2 att

In [26]:
```
def model_lstm_gru_2att(embedding_matrix):
    inp = Input(shape=(max_len,))
    x = Embedding(vocabulary_size, embed_size, weights=[embedding_matrix
], trainable=False)(inp)
    x = SpatialDropout1D(0.1)(x)
    x = Bidirectional(CuDNNLSTM(100, return_sequences=True))(x)
    y = Bidirectional(CuDNNGRU(100, return_sequences=True))(x)

    atten_1 = Attention(max_len)(x)
    atten_2 = Attention(max_len)(y)
    avg_pool = GlobalAveragePooling1D()(y)
    max_pool = GlobalMaxPooling1D()(y)

    conc = concatenate([atten_1, atten_2, avg_pool, max_pool])
    conc = Dense(16, activation="relu")(conc)
    conc = Dropout(0.1)(conc)
    outp = Dense(1, activation="sigmoid")(conc)

    model = Model(inputs=inp, outputs=outp)
    model.compile(loss='binary_crossentropy', optimizer='adam', metrics=
['accuracy'])

    return model
```

CNN2D

In [27]:
```
def model_cnn(embedding_matrix):
    filter_sizes = [1,2,3,5]
    num_filters = 36

    inp = Input(shape=(max_len,))
    x = Lambda(lambda x: K.reverse(x,axes=-1))(inp)
    x = Embedding(vocabulary_size, embed_size, weights=[embedding_matrix
])(inp)
    x = Reshape((max_len, embed_size, 1))(x)

    maxpool_pool = []
    for i in range(len(filter_sizes)):
        conv = Conv2D(num_filters, kernel_size=(filter_sizes[i], embed_s
ize),
                                     kernel_initializer='he_normal', act
ivation='elu')(x)
        maxpool_pool.append(MaxPool2D(pool_size=(max_len - filter_sizes[
i] + 1, 1))(conv))

    z = Concatenate(axis=1)(maxpool_pool)
    z = Flatten()(z)
    z = Dropout(0.1)(z)

    outp = Dense(1, activation="sigmoid")(z)

    model = Model(inputs=inp, outputs=outp)
    model.compile(loss='binary_crossentropy', optimizer='adam', metrics=
['accuracy'])

    return model
```

Train and predict models functions

In [28]:
```
def train_pred(model, model_name, epochs = 1):
    filepath= "models_ens/{}.h5".format(model_name)
    for e in range(epochs):
        checkpoint = ModelCheckpoint(filepath, monitor='loss', verbose=0
, save_weights_only=False, save_best_only=True, mode='auto', period=1)
        model.fit(X_tra, y_tra, batch_size = 512, epochs = 18, validatio
n_data=(X_val, y_val), callbacks=[checkpoint])
        y_pred_val = model.predict([X_val], batch_size = 1024, verbose=0
)

        best_thresh = 0.5
        best_score = 0.0
        for thresh in np.arange(0.1, 0.501, 0.01):
            thresh = np.round(thresh, 2)
            score = f1_score(y_val, (y_pred_val > thresh).astype(int))
            if score > best_score:
                best_thresh = thresh
                best_score = score

        print("Val F1 Score: {:.4f}".format(best_score))

    y_pred_test = model.predict([x_test], batch_size=1024, verbose=0)
    return y_pred_val, y_pred_test, best_score


def model_pred(model):
    y_pred_val = model.predict([X_val], batch_size = 1024, verbose=0)

    best_thresh = 0.5
    best_score = 0.0
    for thresh in np.arange(0.1, 0.501, 0.01):
        thresh = np.round(thresh, 2)
        score = f1_score(y_val, (y_pred_val > thresh).astype(int))
        if score > best_score:
            best_thresh = thresh
            best_score = score

    print("Val F1 Score: {:.4f}".format(best_score))

    y_pred_test = model.predict([x_test], batch_size=1024, verbose=0)
    return y_pred_val, y_pred_test, best_score
```

Meta-embedding

It is a pretty standard practice tehse days to use word embeddings in NLP given the results they achieved with language models. However, there are well documented differences in performance depending on which sets are used (Yin and Schutz, 2016; Bakarov, 2017). The reality i sthat performance is difficult to predict depending on which task and which embedding sets are used as there can be a poor correlation between the tasks and word-level benchmarks (Kiela et al. 2018).

A way around such issues is to combine the strengths of different embeddings. In using them together, we can leverage an observed complementarity (Bollega et al. 2017; Coates and Bollegala, 2018). This approach has two main advantages: 1) it enhances the representations: they perform better than single sets; 2) they solve the problem of coverage with out of vocabulary words as they cover more words together (Yin and Schutze, 2016).

In order to do that, embeddings can be concatenated but this lead to a very large set with high dimensionlaity which can lead to severely reduced efficiency in modeling. Recent work showed that embeddings can be averaged without loss of performance and with the added benefit of dimensionality reduction (Coates and Bollega, 2018).

Therefore, we are using a meta-embedding the average between multiple sets: Fasttext with subword, GloVe, and paragram embeddings. This last set is based on teh Paraphrase Database (PPDB), a considerable semantic resource from sentence pairs with heuristic confidence estimates (Wieting, et al. 2015). As an example, PPDB contains millions of sentence pairs with very little lexical overlap but with a common semantic such as: {we must do our utmost, we must make every effort}. The embeddings set was built using learned paragram vectors from the paraphrase pairs.

Sources:

- kyela et. al. 2018. Dynamic meta-embedding for improved sentence representations.
- Coates, Joshua N, Bollegala, Danushla. 2018. Frustratingly easy meta-embedding - Computing meta-embeddings by averaging source word embeddings
- complete the sources

Load embedding functions

In [25]:
```
def glove(word_index):
    EMBEDDING_FILE = '../embeddings/glove.840B.300d.txt'
    def get_coefs(word,*arr): return word, np.asarray(arr, dtype='float3
2')
    embeddings_index = dict(get_coefs(*o.split(" ")) for o in open(EMBED
DING_FILE, encoding='utf-8'))

    all_embs = np.stack(embeddings_index.values())
    emb_mean,emb_std = all_embs.mean(), all_embs.std()
    embed_size = all_embs.shape[1]

    # word_index = tokenizer.word_index
    nb_words = min(vocabulary_size, len(word_index))
    embedding_matrix = np.random.normal(emb_mean, emb_std, (nb_words, em
bed_size))
    for word, i in word_index.items():
        if i >= vocabulary_size: continue
        embedding_vector = embeddings_index.get(word)
        if embedding_vector is not None: embedding_matrix[i] = embedding
_vector

    return embedding_matrix

def fasttext(word_index):
    EMBEDDING_FILE = '../embeddings/fasttext/wiki-news-300d-1M-subword.v
ec'
    def get_coefs(word,*arr): return word, np.asarray(arr, dtype='float3
2')
    embeddings_index = dict(get_coefs(*o.split(" ")) for o in open(EMBED
DING_FILE, encoding='utf-8') if len(o)>100)

    all_embs = np.stack(embeddings_index.values())
    emb_mean,emb_std = all_embs.mean(), all_embs.std()
    embed_size = all_embs.shape[1]

    # word_index = tokenizer.word_index
    nb_words = min(vocabulary_size, len(word_index))
    embedding_matrix = np.random.normal(emb_mean, emb_std, (nb_words, em
bed_size))
    for word, i in word_index.items():
        if i >= vocabulary_size: continue
        embedding_vector = embeddings_index.get(word)
        if embedding_vector is not None: embedding_matrix[i] = embedding
_vector

    return embedding_matrix

def para(word_index):
    EMBEDDING_FILE = '/data_disk/embeddings/paragram_300_sl999.txt'
    def get_coefs(word,*arr): return word, np.asarray(arr, dtype='float3
2')
    embeddings_index = dict(get_coefs(*o.split(" ")) for o in open(EMBED
DING_FILE, encoding="utf8", errors='ignore') if len(o)>100)

    all_embs = np.stack(embeddings_index.values())
    emb_mean,emb_std = all_embs.mean(), all_embs.std()
```

```
    embed_size = all_embs.shape[1]

    # word_index = tokenizer.word_index
    nb_words = min(vocabulary_size, len(word_index))
    embedding_matrix = np.random.normal(emb_mean, emb_std, (nb_words, em
bed_size))
    for word, i in word_index.items():
        if i >= vocabulary_size: continue
        embedding_vector = embeddings_index.get(word)
        if embedding_vector is not None: embedding_matrix[i] = embedding
_vector

    return embedding_matrix
```

In [26]:
```
# Load embeddings.
embedding_1 = glove(word_index)
embedding_2 = fasttext(word_index)
embedding_4 = para(word_index)
```

In [27]:
```
embedding_matrix = np.mean([embedding_1, embedding_2, embedding_4], axis
= 0)
np.shape(embedding_matrix)
```

Out[27]:
```
(200000, 300)
```

Models training

We run the functions defined above to train the models and predict outputs on both validation and test data. We collect the outputs in a list for the ensemble later.

In [67]:
```
outputs = []
y_pred_val, y_pred_test, best_score = train_pred(model_lstm_gru_cnn(embe
dding_matrix), epochs = 1, model_name = "lstm_gru_cnn")
outputs.append([y_pred_val, y_pred_test, best_score, 'lstm-gru-cnn'])
```

```
Train on 41337 samples, validate on 8467 samples
Epoch 1/18
41337/41337 [==============================] - 38s 909us/step - loss:
0.5565 - acc: 0.7122 - val_loss: 0.5071 - val_acc: 0.7585
Epoch 2/18
41337/41337 [==============================] - 36s 860us/step - loss:
0.4850 - acc: 0.7741 - val_loss: 0.4782 - val_acc: 0.7744
Epoch 3/18
41337/41337 [==============================] - 36s 865us/step - loss:
0.4614 - acc: 0.7854 - val_loss: 0.4588 - val_acc: 0.7846
Epoch 4/18
41337/41337 [==============================] - 36s 864us/step - loss:
0.4402 - acc: 0.7985 - val_loss: 0.4438 - val_acc: 0.7939
Epoch 5/18
41337/41337 [==============================] - 36s 865us/step - loss:
0.4226 - acc: 0.8085 - val_loss: 0.4189 - val_acc: 0.8109
Epoch 6/18
41337/41337 [==============================] - 36s 864us/step - loss:
0.4124 - acc: 0.8164 - val_loss: 0.4335 - val_acc: 0.8107
Epoch 7/18
41337/41337 [==============================] - 36s 863us/step - loss:
0.3943 - acc: 0.8252 - val_loss: 0.4142 - val_acc: 0.8163
Epoch 8/18
41337/41337 [==============================] - 36s 864us/step - loss:
0.3880 - acc: 0.8297 - val_loss: 0.4114 - val_acc: 0.8192
Epoch 9/18
41337/41337 [==============================] - 36s 862us/step - loss:
0.3768 - acc: 0.8365 - val_loss: 0.4198 - val_acc: 0.8193
Epoch 10/18
41337/41337 [==============================] - 35s 859us/step - loss:
0.3728 - acc: 0.8384 - val_loss: 0.4086 - val_acc: 0.8189
Epoch 11/18
41337/41337 [==============================] - 35s 858us/step - loss:
0.3577 - acc: 0.8454 - val_loss: 0.4188 - val_acc: 0.8192
Epoch 12/18
41337/41337 [==============================] - 36s 859us/step - loss:
0.3540 - acc: 0.8459 - val_loss: 0.4166 - val_acc: 0.8199
Epoch 13/18
41337/41337 [==============================] - 36s 860us/step - loss:
0.3433 - acc: 0.8529 - val_loss: 0.4061 - val_acc: 0.8207
Epoch 14/18
41337/41337 [==============================] - 36s 859us/step - loss:
0.3356 - acc: 0.8563 - val_loss: 0.4088 - val_acc: 0.8230
Epoch 15/18
41337/41337 [==============================] - 35s 857us/step - loss:
0.3330 - acc: 0.8589 - val_loss: 0.4095 - val_acc: 0.8200
Epoch 16/18
41337/41337 [==============================] - 35s 857us/step - loss:
0.3198 - acc: 0.8653 - val_loss: 0.4275 - val_acc: 0.8198
Epoch 17/18
41337/41337 [==============================] - 36s 860us/step - loss:
0.3144 - acc: 0.8673 - val_loss: 0.4266 - val_acc: 0.8200
Epoch 18/18
41337/41337 [==============================] - 35s 857us/step - loss:
0.3048 - acc: 0.8709 - val_loss: 0.4220 - val_acc: 0.8116
Val F1 Score: 0.8253
```

In [68]:
```
y_pred_val, y_pred_test, best_score = train_pred(model_lstm_gru_2att(emb
edding_matrix), epochs = 1, model_name = "lstm_gru_2att")
outputs.append([y_pred_val, y_pred_test, best_score, 'lstm gru 2attentio
n'])
```

```
Train on 41337 samples, validate on 8467 samples
Epoch 1/18
41337/41337 [==============================] - 39s 953us/step - loss:
0.5488 - acc: 0.7206 - val_loss: 0.4901 - val_acc: 0.7716
Epoch 2/18
41337/41337 [==============================] - 37s 890us/step - loss:
0.4616 - acc: 0.7895 - val_loss: 0.4606 - val_acc: 0.7813
Epoch 3/18
41337/41337 [==============================] - 37s 889us/step - loss:
0.4348 - acc: 0.8030 - val_loss: 0.4537 - val_acc: 0.7911
Epoch 4/18
41337/41337 [==============================] - 37s 890us/step - loss:
0.4076 - acc: 0.8182 - val_loss: 0.4389 - val_acc: 0.7943
Epoch 5/18
41337/41337 [==============================] - 37s 890us/step - loss:
0.3936 - acc: 0.8276 - val_loss: 0.4052 - val_acc: 0.8213
Epoch 6/18
41337/41337 [==============================] - 37s 889us/step - loss:
0.3727 - acc: 0.8403 - val_loss: 0.4000 - val_acc: 0.8211
Epoch 7/18
41337/41337 [==============================] - 37s 888us/step - loss:
0.3606 - acc: 0.8468 - val_loss: 0.4083 - val_acc: 0.8161
Epoch 8/18
41337/41337 [==============================] - 37s 889us/step - loss:
0.3394 - acc: 0.8579 - val_loss: 0.4022 - val_acc: 0.8240
Epoch 9/18
41337/41337 [==============================] - 37s 889us/step - loss:
0.3249 - acc: 0.8661 - val_loss: 0.3998 - val_acc: 0.8298
Epoch 10/18
41337/41337 [==============================] - 37s 889us/step - loss:
0.3043 - acc: 0.8776 - val_loss: 0.4156 - val_acc: 0.8237
Epoch 11/18
41337/41337 [==============================] - 37s 888us/step - loss:
0.2858 - acc: 0.8879 - val_loss: 0.4173 - val_acc: 0.8221
Epoch 12/18
41337/41337 [==============================] - 37s 890us/step - loss:
0.2648 - acc: 0.8965 - val_loss: 0.4318 - val_acc: 0.8228
Epoch 13/18
41337/41337 [==============================] - 37s 892us/step - loss:
0.2479 - acc: 0.9051 - val_loss: 0.4420 - val_acc: 0.8167
Epoch 14/18
41337/41337 [==============================] - 37s 894us/step - loss:
0.2267 - acc: 0.9153 - val_loss: 0.4644 - val_acc: 0.8194
Epoch 15/18
41337/41337 [==============================] - 37s 893us/step - loss:
0.2064 - acc: 0.9233 - val_loss: 0.4781 - val_acc: 0.8152
Epoch 16/18
41337/41337 [==============================] - 37s 886us/step - loss:
0.1916 - acc: 0.9289 - val_loss: 0.5020 - val_acc: 0.8084
Epoch 17/18
41337/41337 [==============================] - 37s 893us/step - loss:
0.1787 - acc: 0.9339 - val_loss: 0.5114 - val_acc: 0.8189
Epoch 18/18
41337/41337 [==============================] - 37s 893us/step - loss:
0.1517 - acc: 0.9459 - val_loss: 0.5466 - val_acc: 0.8119
Val F1 Score: 0.8194
```

In [70]:
```
y_pred_val, y_pred_test, best_score = train_pred(model_cnn(embedding_mat
rix), epochs = 1, model_name = "cnn2d")
outputs.append([y_pred_val, y_pred_test, best_score, 'cnn2d'])
```

```
Train on 41337 samples, validate on 8467 samples
Epoch 1/18
41337/41337 [==============================] - 15s 370us/step - loss:
0.4999 - acc: 0.7539 - val_loss: 0.4438 - val_acc: 0.7946
Epoch 2/18
41337/41337 [==============================] - 13s 304us/step - loss:
0.3569 - acc: 0.8478 - val_loss: 0.4100 - val_acc: 0.8171
Epoch 3/18
41337/41337 [==============================] - 12s 301us/step - loss:
0.2743 - acc: 0.8963 - val_loss: 0.4036 - val_acc: 0.8210
Epoch 4/18
41337/41337 [==============================] - 12s 302us/step - loss:
0.2004 - acc: 0.9348 - val_loss: 0.3985 - val_acc: 0.8246
Epoch 5/18
41337/41337 [==============================] - 13s 303us/step - loss:
0.1387 - acc: 0.9611 - val_loss: 0.4158 - val_acc: 0.8175
Epoch 6/18
41337/41337 [==============================] - 12s 301us/step - loss:
0.0944 - acc: 0.9767 - val_loss: 0.4339 - val_acc: 0.8133
Epoch 7/18
41337/41337 [==============================] - 13s 303us/step - loss:
0.0650 - acc: 0.9845 - val_loss: 0.4502 - val_acc: 0.8129
Epoch 8/18
41337/41337 [==============================] - 13s 304us/step - loss:
0.0485 - acc: 0.9875 - val_loss: 0.4647 - val_acc: 0.8107
Epoch 9/18
41337/41337 [==============================] - 12s 302us/step - loss:
0.0394 - acc: 0.9895 - val_loss: 0.4745 - val_acc: 0.8148
Epoch 10/18
41337/41337 [==============================] - 13s 305us/step - loss:
0.0316 - acc: 0.9913 - val_loss: 0.5172 - val_acc: 0.8029
Epoch 11/18
41337/41337 [==============================] - 13s 303us/step - loss:
0.0279 - acc: 0.9921 - val_loss: 0.5017 - val_acc: 0.8143
Epoch 12/18
41337/41337 [==============================] - 12s 301us/step - loss:
0.0262 - acc: 0.9919 - val_loss: 0.5339 - val_acc: 0.8063
Epoch 13/18
41337/41337 [==============================] - 12s 300us/step - loss:
0.0239 - acc: 0.9928 - val_loss: 0.5273 - val_acc: 0.8148
Epoch 14/18
41337/41337 [==============================] - 12s 302us/step - loss:
0.0220 - acc: 0.9932 - val_loss: 0.5542 - val_acc: 0.8081
Epoch 15/18
41337/41337 [==============================] - 12s 301us/step - loss:
0.0221 - acc: 0.9934 - val_loss: 0.5474 - val_acc: 0.8086
Epoch 16/18
41337/41337 [==============================] - 12s 300us/step - loss:
0.0200 - acc: 0.9936 - val_loss: 0.5683 - val_acc: 0.8069
Epoch 17/18
41337/41337 [==============================] - 12s 301us/step - loss:
0.0195 - acc: 0.9941 - val_loss: 0.5721 - val_acc: 0.8038
Epoch 18/18
41337/41337 [==============================] - 12s 302us/step - loss:
0.0182 - acc: 0.9945 - val_loss: 0.6152 - val_acc: 0.7940
Val F1 Score: 0.8169
```

Ensemble model

We trained over many iterations until we got an acceptable output in terms of how many articles it would predict correctly "in package" with as little error as possible on the "out of package". Basically, we wanted the best outcome for the false negatives, of course the true positives, and the true negatives. It is not an issue if there are over-predictions for the false positives. As we stated above, our objectives are to capture as many relevant articles while minimizing the penalty on positive predictions of non-relevant articles.

In [43]:
```
outputs = []

model_lstm_gru_cnn = load_model('models_ens/lstm_gru_cnn_best.h5')
y_pred_val, y_pred_test, best_score = model_pred(model_lstm_gru_cnn)
outputs.append([y_pred_val, y_pred_test, best_score, 'lstm-gru-cnn'])
```

```
Val F1 Score: 0.8253
```

In [44]:
```
model_lstm_gru_2att = load_model('models_ens/lstm_gru_2att_best.h5', cus
tom_objects={'Attention': Attention})
y_pred_val, y_pred_test, best_score = model_pred(model_lstm_gru_2att)
outputs.append([y_pred_val, y_pred_test, best_score, 'lstm gru 2attentio
n'])
```

```
Val F1 Score: 0.8194
```

In [31]:
```
model_cnn = load_model('models_ens/cnn2d_best.h5')
y_pred_val, y_pred_test, best_score = model_pred(model_cnn)
outputs.append([y_pred_val, y_pred_test, best_score, 'cnn2d'])
```

```
Val F1 Score: 0.8169
```

In [32]:
```
outputs.sort(key=lambda x: x[2]) # Sort the output by val f1 score
print(len(outputs))
for output in outputs:
    print(output[2], output[3])
```

```
3
0.8168918151890112 cnn2d
0.8194263363754888 lstm gru 2attention
0.825293056807935 lstm-gru-cnn
```

Linear regression to select the ensemble coefficients

In [33]:
```
from sklearn.linear_model import LinearRegression
X = np.asarray([outputs[i][0] for i in range(len(outputs))])
X = X[...,0]
reg_pred = LinearRegression().fit(X.T, y_val)
print(reg_pred.score(X.T, y_val),reg_pred.coef_)
```

```
0.511826161216943 [0.27373523 0.23183241 0.43300602]
```

In [35]:
```
reg_pred.coef_
```

Out[35]:
```
array([0.27373523, 0.23183241, 0.43300602], dtype=float32)
```

In [34]:
```
y_pred_val = np.sum([outputs[i][0] * reg_pred.coef_[i] for i in range(le
n(outputs))], axis = 0)
# y_pred_val= np.mean([outputs[i][0] for i in range(len(outputs))], axis
= 0) # to avoid overfitting, just take average

thresholds = []
for thresh in np.arange(0.1, 0.501, 0.01):
    thresh = np.round(thresh, 2)
    res = f1_score(y_val, (y_pred_val > thresh).astype(int))
    thresholds.append([thresh, res])
    #print("F1 score at threshold {0} is {1}".format(thresh, res))

thresholds.sort(key=lambda x: x[1], reverse=True)
opt_thresh = thresholds[0][0]
print("Optimal threshold: ", opt_thresh)
```

```
Optimal threshold:  0.3
```

In [35]:
```
y_pred_test = np.sum([outputs[i][1] * reg_pred.coef_[i] for i in range(l
en(outputs))], axis = 0)
#y_pred_test = np.mean([outputs[i][1] for i in range(len(outputs))], axi
s = 0)
y_pred_test = (y_pred_test > opt_thresh).astype(int)
```

In [47]:
```
len(y_pred_test)
```

Out[47]:
```
21470
```

Confusion matrix

In [41]:
```
print(classification_report(y_test, y_pred_test))
print(confusion_matrix(y_test, y_pred_test))
```

```
             precision    recall  f1-score   support

          0       1.00      0.80      0.89     20510
          1       0.18      0.94      0.30       960

avg / total       0.96      0.80      0.86     21470

[[16372  4138]
 [   59   901]]
```

In [31]:
```
import gc
gc.collect()
```

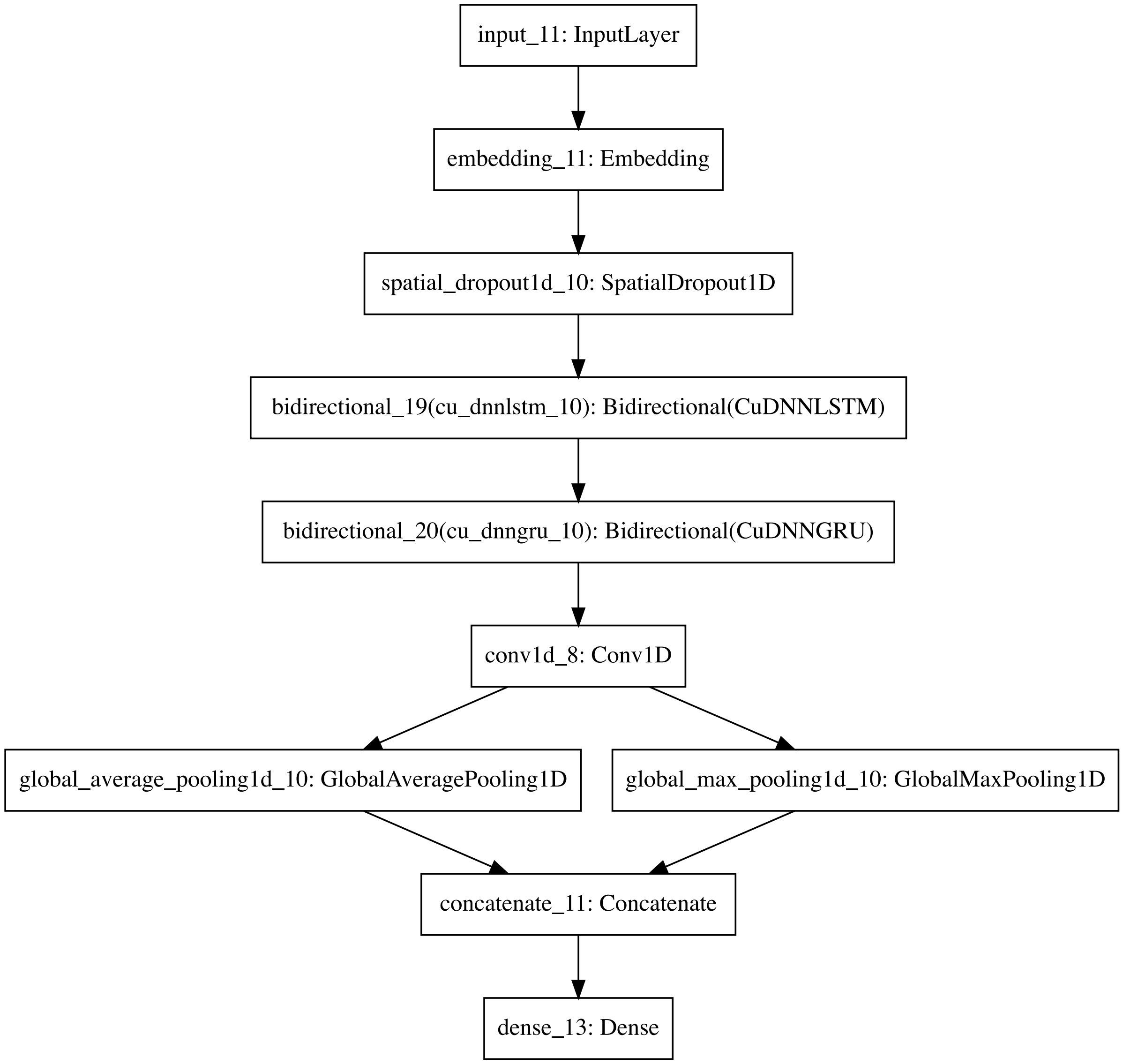
Out[31]:
```
290
```

Model graphs

In [28]:
```
from IPython.display import SVG
from keras.utils.vis_utils import model_to_dot

SVG(model_to_dot(model_lstm_gru_cnn).create(prog='dot', format='svg'))
```

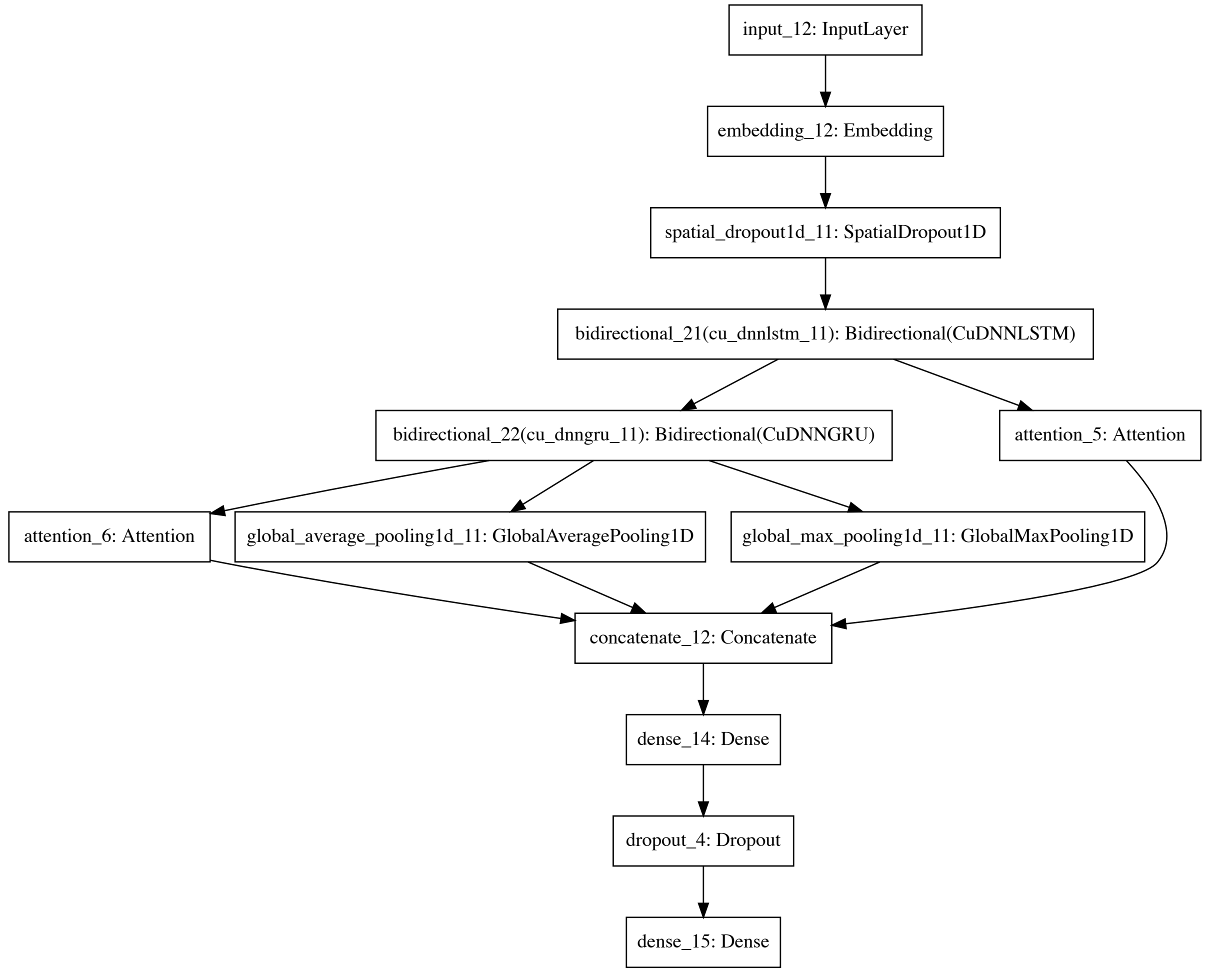
Out[28]:



LSTM, GRU, 2 attention

In [29]:
```
SVG(model_to_dot(model_lstm_gru_2att).create(prog='dot', format='svg'))
```

Out[29]:



CNN flip 2 dimensions

In [30]:
```
SVG(model_to_dot(model_cnn).create(prog='dot', format='svg'))
```

Out[30]: