hw02

March 13, 2020

HW2 Convolutional neural networks for text classification

Due: March 2020 13th, 23:59

In this homework you will learn how to build a simple convolutional neural networks (1 convolution layer with max pooling + 1 activation layer) from scratch, and use the model to solve text classification problem. As optional, you also have a chance to build real life CNN models using Keras + Tensorflow and use it to challenge the model you build from scratch.

1. Math preliminaries

Please answer all these questions:

1. What is the form of sigmoid function $\sigma(z)$? Show that $\sigma\ (z) = \sigma(z)[1-\sigma(z)]$.

Answer: $\sigma(z) = \frac{1}{1+e^{-z}}$

$\sigma'(z) = (1+e^{-z})' * (-\frac{1}{(1+e^{-z})}^2)$

$= \frac{e^{-z}}{(1+e^{-z})^2}$

$= \frac{1}{1+e^{-z}} * \frac{e^{-z}}{(1+e^{-z})}$

$= \sigma(z)[1-\sigma(z)]$

2. Another popular activation function is $tanh(z) = \frac{e^z-e^{-z}}{e^z+e^{-z}}$, show that $tanh\ (z) = 1-tanh(z)^2$.

Answer: $tanh(z) = \frac{e^z-e^{-z}}{e^z+e^{-z}}$

denote $a(z) = e^z - e^{-z}$, $b(z) = e^z + e^{-z}$, easy to see that $a'(z) = b$ and $b'(z) = a$

$tanh(z) = \frac{a}{b}$

$tanh'(z) = \frac{a'*b-b'*a}{b^2} = \frac{b^2-a^2}{b^2} = 1-(a/b)^2 = 1-tanh(z)^2$

3. For a single variable single layer perceptron with sigmoid activation function (equivalent to LR) and loss function defined as:
   $\hat{y}_i = \sigma(w_1 x_i + w_0)$
   $L(w_0, w_1) = \sum_i y_i lg(\hat{y}_i) + (1-y_i) lg(1-\hat{y}_i)$
   Show that:
   $\frac{\partial L}{\partial w_1} = \sum_i (y_i-\hat{y}_i) x_i$
   $\frac{\partial L}{\partial w_0} = \sum_i (y-\hat{y}_i)$

Answer: $\frac{\partial L}{\partial w_1} = \sum_i \frac{\partial L}{\partial \hat{y}_i} \frac{\partial \hat{y}_i}{\partial w_1}$

$= \sum_i (\frac{y_i}{\hat{y}_i} - \frac{1-y_i}{1-\hat{y}_i}) x_i \sigma'(w_1 x_i + w_0)$

$= \sum_i (\frac{y_i}{\hat{y}_i} - \frac{1-y_i}{1-\hat{y}_i}) x_i \hat{y}_i (1-\hat{y}_i)$

$= \sum_i (y_i(1-\hat{y}_i) - (1-y_i)\hat{y}_i) x_i$

$= \sum_i (y_i - \hat{y}_i) x_i$

$\frac{\partial L}{\partial w_0} = \sum_i \frac{\partial L}{\partial \hat{y}_i} \frac{\partial \hat{y}_i}{\partial w_0}$

$= \sum_i (\frac{y_i}{\hat{y}_i} - \frac{1-y_i}{1-\hat{y}_i}) \sigma'(w_1 x_i + w_0)$

$= \sum_i (\frac{y_i}{\hat{y}_i} - \frac{1-y_i}{1-\hat{y}_i}) \hat{y}_i (1-\hat{y}_i)$

$= \sum_i (y_i(1-\hat{y}_i) - (1-y_i)\hat{y}_i)$

$= \sum_i (y_i - \hat{y}_i)$

4. For column vectors $\vec{x}$ and $\vec{w}$, and a symmetric matrix $\overleftrightarrow{M}$, define the gradient operator:
   $\nabla_x = (\frac{\partial}{\partial x_0}, \frac{\partial}{\partial x_1}, ..., \frac{\partial}{\partial x_n})^T$
   show that:
   $\nabla_x(\vec{w}^T\vec{x}) = \vec{w}$
   $\nabla_x(\vec{x}^T\vec{w}) = \vec{w}$
   $\nabla_x(\vec{w}^T\overleftrightarrow{M}\vec{x}) = \overleftrightarrow{M}\vec{w}$

Answer: $\vec{w}^T\vec{x} = \sum_i w_i x_i$

$\frac{\partial \vec{w}^T\vec{x}}{\partial x_i} = w_i$

$\nabla_x(\vec{w}^T\vec{x}) = (w_0, w_1, ..., w_n)^T = \vec{w}$

$\vec{x}^T\vec{w} = \sum_i x_i w_i$

$\frac{\partial \vec{x}^T\vec{w}}{\partial x_i} = w_i$

$\nabla_x(\vec{x}^T\vec{w}) = (w_0, w_1, ..., w_n)^T = \vec{w}$

$\nabla_x(\vec{w}^T\overleftrightarrow{M}\vec{x}) = \nabla_x(\sum_i \sum_j w_i M_{ij} x_j)$

$= (\sum_j w_0 M_{0j}, \sum_j w_1 M_{1j}, ..., \sum_j w_n M_{nj})$

and $\overleftrightarrow{M}\vec{w} = (\sum_j M_{0j} w_0, \sum_j M_{1j} w_1, ..., \sum_j M_{nj} w_n)$

so $\nabla_x(\vec{w}^T\overleftrightarrow{M}\vec{x}) = \overleftrightarrow{M}\vec{w}$

5. Let's expand Q3 to a more general case. Suppose there is a single layer perceptron with multiple variables:

   $\hat{y}_i = \sigma(\vec{w}^T\vec{x_i})$

   $L(\vec{w}) = \sum_i y_i lg(\hat{y}_i) + (1-y_i) lg(1-\hat{y}_i)$

   show that:

   $\nabla_{\vec{w}} L(\vec{w}) = \sum_i (y_i - \hat{y}_i)\vec{x_i}$

   (hint: use the notation defined in Q4)

Answer: $\nabla_{\vec{w}} L(\vec{w}) = \sum_i \frac{\partial L}{\partial \hat{y}_i} \nabla_{\vec{w}} \hat{y}_i$

$= \sum_i (\frac{y_i}{\hat{y}_i} - \frac{1-y_i}{1-\hat{y}_i})[\hat{y}_i(1-\hat{y}_i)\nabla_{\vec{w}}(\vec{w}^T \vec{x_i})]$

$= \sum_i (y_i - \hat{y}_i)\vec{x_i}$

6. In a CNN illustrated as Fig 1, suppose the loss function is:

   $L(\overleftrightarrow{U}, \vec{w}) = \sum_i y_i lg(\hat{y}_i) + (1-y_i) lg(1-\hat{y}_i)$

   From the conclusion in Q5, we can get that:

   $\nabla_w L(\overleftrightarrow{U}, \vec{w}) = \sum_i (y_i - \hat{y}_i)\vec{h}^{(i)}$

   Can you calculate $\nabla_{u_i} L(U, w)$ using similar techniques?

Answer: $\hat{y}_i = \sigma(\vec{w}^T \vec{h_i}) = \sigma(\vec{w}^T tanh(\overleftrightarrow{U_i} \vec{x_i}))$

$\nabla_{\overleftrightarrow{u_i}} L(\overleftrightarrow{U}, \vec{w}) = \nabla_{\overleftrightarrow{u_i}} [y_i lg(\hat{y}_i) + (1-y_i) lg(1-\hat{y}_i)]$

$= (y_i - \hat{y}_i)\nabla_{\overleftrightarrow{u_i}} (\vec{w_i}^T \vec{h_i})$

$= (y_i - \hat{y}_i)\nabla_{\overleftrightarrow{u_i}} [\vec{w_i}^T tanh(\overleftrightarrow{U_i} \vec{x_i})]$

$= (y_i - \hat{y}_i)\nabla_{\overleftrightarrow{u_i}} (\sum_j \sum_k w_{ij} tanh(U_{ijk} x_{ik}))$

denote the answer $= \overleftrightarrow{A}$

$\overleftrightarrow{A}_{jk} = (y_i - \hat{y}_i) w_{ij} (1 - h_{ij}^2) x_{ik}$

2. Coding exercise

Follow the instruction in the notebook, and implement the missing code to build the CNN classifier from scratch. Note that the training might be very slow. Consider reducing the training data size and vocabulary size for testing your code. Ask questions in Piazza if you get blocked.

Hint: In this CNN, words should be one-hot encoded, but we actually numerically encoded it in the code. This is a sparse trick we did to boost the efficiency, try to understand how it works.

Some of the key details you will have a chance to implement: - Forward propagation of a CNN network - Backward propagation of a CNN network - Numerical gradient checking - Use Keras and TensorFlow to implement more complex CNN networks

You are given the following files: - `hw02.ipynb`: Notebook file with starter code - `train.txt`: Training set to train your model - `test.txt`: Test set to report your model's performance - `sample_prediction.csv`: Sample file your prediction result should look like - `utils/`: folder containing all utility code for the series of homeworks

3. Deliverables (zip them all)

- pdf version of your final notebook.
- Use the best model you trained, generate the prediction for test.txt, name the output file prediction.csv (Be careful: the best model in your training set might not be the best model for the test set).
- After you finished the run, does the model perform better than the bag of words model you built last week? What do you think that contributes to the difference in performance?

- HW1_writeup.pdf: summarize the method you used and report their performance. If you worked on the optional task, add the discussion. Add a short essay discussing the biggest challenges you encounter during this assignment and what you have learnt.

(**You are encouraged to add the writeup doc into your notebook using markdown/html langauge, just like how this notes is prepared**)

HW2 write up

Generally, it is hard to reach to higher score than the naive bag-of-word model.

Manual CNN

We have use a lot of mathamatically knowledge including linear algebra, matrix operations, calculas. The main challenage here is to understand the numerically encoding trick, and to derive the matrix form gradient descent formula correctly. The sparse trick is to only use/update those matrix elements in U, which has a numerically coding in the dataset sliding window.

I think the best way for me, to derive the vector/matrix gradient is just to write it out explicitly.

I also find that if the window size is 2, then we only reach ~70% of accuracy, but if the window size is 3, then we can reach to 74%.

Keras CNN

Without further tuning. The dev accuracy on the same dataset performs worse than the Logistic regression bag of words method. Although the CNN utilizes the position information between the words, if the window_size is small like 2, then it would similar as a 2-gram bag of words problem. We performed a maxpooling on the sentences to extract the window that contains the most important information, the bag of words use all the words in the corpus, so the maxpooling might lose information potentially. Also, the CNN model has a lot of parameters, so it would easily overfit without proper regulazations. We can see that the training accuracy is very hight like 90% and keep incrasing while the dev accuracy just keep on the 70% level, that is a sign of overfitting.

Keras Own model

Observing that the present Keras CNN model needs to add regulazations and decrease learning rate. I did not change the structure of the network, and add l2 regulazation with $\lambda = 0.01$, and decreased learning rate. It is obvious that if we increase the truncate size from 100 to 300, that we can also increase our accuracy by acquiring more information in the text. Of course I want to learn more lessons on how to tune hyperparameters to make the model improve more.

1 ================ Coding Starts Here ====================

```
In [ ]: %load_ext autoreload
        %autoreload 2
        %matplotlib inline
        import os
        import sys
        import pandas as pd
        import numpy as np
        from nltk.stem import WordNetLemmatizer
        from nltk import word_tokenize
```

```
# add utils folder to path
p = os.path.dirname(os.getcwd())
if p not in sys.path:
    sys.path = [p] + sys.path

from utils.hw2 import load_data, save_prediction, read_vocab
from utils.general import sigmoid, tanh, show_keras_model
```

2 CNN model

Complete the code block in the cells in this section.

- step1: Implement the pipeline method to process the raw input
- step2: Implement the forward method
- step3: Implement the backward method
- step4: Run the cell below to train your model

In [46]:
```
"""
This cell shows you how the model will be used, you have to finish the cell below before
you
can run this cell.

Once the implementation is done, you should hype tune the parameters to find the best
config

Note I only selected 2000 data points to speed up debugging, you should use all the data
to train the
final model
"""
from sklearn.model_selection import train_test_split
data = load_data("train.txt")
vocab = read_vocab("vocab.txt")
X, y = data.text, data.target
X_train, X_dev, y_train, y_dev = train_test_split(X, y, test_size=0.3)
cls = CNNTextClassificationModel(vocab,window_size=3, F=100, alpha=0.02)
cls.train(X_train, y_train, X_dev, y_dev, nEpoch=20)
```

```
Epoch: 0        Train accuracy: 0.558   Dev accuracy: 0.549
Epoch: 1        Train accuracy: 0.608   Dev accuracy: 0.593
Epoch: 2        Train accuracy: 0.676   Dev accuracy: 0.625
Epoch: 3        Train accuracy: 0.772   Dev accuracy: 0.662
Epoch: 4        Train accuracy: 0.868   Dev accuracy: 0.687
Epoch: 5        Train accuracy: 0.947   Dev accuracy: 0.711
Epoch: 6        Train accuracy: 0.985   Dev accuracy: 0.724
Epoch: 7        Train accuracy: 0.995   Dev accuracy: 0.731
Epoch: 8        Train accuracy: 0.998   Dev accuracy: 0.732
Epoch: 9        Train accuracy: 0.999   Dev accuracy: 0.735
Epoch: 10       Train accuracy: 0.999   Dev accuracy: 0.736
Epoch: 11       Train accuracy: 0.999   Dev accuracy: 0.740
Epoch: 12       Train accuracy: 0.998   Dev accuracy: 0.741
Epoch: 13       Train accuracy: 0.998   Dev accuracy: 0.742
Epoch: 14       Train accuracy: 0.998   Dev accuracy: 0.742
Epoch: 15       Train accuracy: 0.998   Dev accuracy: 0.743
Epoch: 16       Train accuracy: 0.998   Dev accuracy: 0.743
Epoch: 17       Train accuracy: 0.998   Dev accuracy: 0.741
Epoch: 18       Train accuracy: 0.998   Dev accuracy: 0.740
Epoch: 19       Train accuracy: 0.998   Dev accuracy: 0.740
```

In [39]:
```
class CNNTextClassificationModel:
    def __init__(self, vocab, window_size=2, F=100, alpha=0.1):
        """
        F: number of filters
        alpha: back propagatoin learning rate
        """
        self.vocab = vocab
        self.window_size = window_size
        self.F = F
        self.alpha = alpha

        # U and w are the weights of the hidden layer, see Fig 1 in the pdf file
        # U is the 1D convolutional layer with shape: voc_size * num_filter *
window_size
        self.U = np.random.normal(loc=0, scale=0.01, size=(len(vocab), F, window_size))
        # w is the weights of the activation layer (after max pooling)
        self.w = np.random.normal(loc=0, scale=0.01, size=(F + 1))

    def pipeline(self, X):
        """
        Data processing pipeline to:
        1. Tokenize, Normalize the raw input
        2. Translate raw data input into numerical encoded vectors

        :param X: raw data input
        :return: list of lists

        For example:
        X = [["Apples orange banana"]
         ["orange apple bananas"]]
        returns:
        [[0, 1, 2],
         [1, 0, 2]]
        """

        X2 = []
        for row in X:
            words = word_tokenize(row.lower())
            if len(words) < self.window_size:
                words.extend((self.window_size-len(words))*'__unknown__')
            row2 = []
            for word in words:
                row2.append(self.vocab.get(WordNetLemmatizer().lemmatize(word),
len(self.vocab)-1))
            X2.append(row2)
        return X2

    @staticmethod
    def accuracy(probs, labels):
        assert len(probs) == len(labels), "Wrong input!!"
        a = np.array(probs)
        b = np.array(labels)
        return 1.0 * (a==b).sum() / len(b)

    def train(self, X_train, y_train, X_dev, y_dev, nEpoch=50):
        """
        Function to fit the model
        :param X_train, X_dev: raw data input
        :param y_train, y_dev: label
        :nEpoch: number of training epoches
        """
        X_train = self.pipeline(X_train)
        X_dev = self.pipeline(X_dev)
        for epoch in range(nEpoch):
            self.fit(X_train, y_train)

            accuracy_train = self.accuracy(self.predict(X_train), y_train)
            accuracy_dev = self.accuracy(self.predict(X_dev), y_dev)
```

```python
            print("Epoch: {}\tTrain accuracy: {:.3f}\tDev accuracy: {:.3f}"
                  .format(epoch, accuracy_train, accuracy_dev))

    def fit(self, X, y):
        """
        :param X: numerical encoded input
        """
        for (data, label) in zip(X, y):
            self.backward(data, label)

        return self

    def predict(self, X):
        """
        :param X: numerical encoded input
        """
        result = []
        for data in X:
            if self.forward(data)["prob"] > 0.5:
                result.append(1)
            else:
                result.append(0)

        return result

    def forward(self, word_indices):
        """
        :param word_indices: a list of numerically ecoded words
        :return: a result dictionary containing 3 items -
        result['prob']: \hat y in Fig 1.
        result['h']: the hidden layer output after max pooling, h = [h1, ..., hf]
        result['hid']: argmax of F filters, e.g. j of x_j
        e.g. for the ith filter u_i, tanh(word[hid[j], hid[j] + width]*u_i) = h_i
        """
        assert len(word_indices) >= self.window_size, "Input length cannot be shorter
than the window size"

        h = np.zeros(self.F + 1, dtype=float)
        hid = np.zeros(self.F, dtype=int)
        prob = 0.0

        # layer 1. compute h and hid
        # loop through the input data of word indices and
        # keep track of the max filtered value h_i and its position index x_j
        # h_i = max(tanh(weighted sum of all words in a given window)) over all windows
for u_i

        # for each training data
        # size: U: vocab * F * window_size
        # size: X: (vocab * window_size) * token_num
        # for each filter: u^T * x = vocab * vocab (make sure windows has interacts?)
        # max-pooling: size h -> F + h0 -> 1
        # weight: F + 1

        for f in range(self.F):
            #prodlist = []
            for index in range(len(word_indices)-self.window_size+1): #[2,3,0...]
                #x = word_indices[index:index+self.window_size]
#[[2,3],[3,0],...,],#[[0,0,1,0,0,0,0,1],[0,0,0,1,1,0,0,0]]
                # now x is numerical coding, x = (x_0,x_1,...,x_j,...x_ws)
                # x_j = i means \vec{x}_j = [0,0,...,1,...0]^T, \matrix{x}_ij = 1
                prod = 0
                # (u^T * x)_ij = \sum_k u_ki*x_kj (if \num{x}_j = l) = u_li
                for j in range(self.window_size):
                    prod += self.U[word_indices[index+j],f,j]
                #prodlist.append(prod)
            #h[f] = np.max(prodlist)
```

```
            #hid[f] = np.argmax(prodlist)
                if index == 0:
                    h[f] = prod
                if prod > h[f]:
                    h[f] = prod
                    hid[f] = index

            h[f] = tanh(h[f])
        h[self.F] = 1e-4
        # layer 2. compute probability
        # once h and hid are computed, compute the probabiliy by sigmoid(h^TV)
        #print(self.w[self.F],self.w.dot(h))
        prob = sigmoid(self.w.dot(h))
        #print('hid,w,h,prob',hid,self.w,h,prob)

        # return result
        return {"prob": prob, "h": h, "hid": hid}

    def backward(self, word_indices, label):
        """
        Update the U, w using backward propagation

        :param word_indices: a list of numerically ecoded words
        :param label: int 0 or 1
        :return: None

        update weight matrix/vector U and V based on the loss function
        """

        pred = self.forward(word_indices)
        prob = pred["prob"]
        h = pred["h"]
        hid = pred["hid"]
        #print('prob,y','y-prob',prob,label,(label - pred['prob']))
        # update U and w here
        # to update V: w_new = w_current + d(loss_function)/d(w)*alpha
        # to update U: U_new = U_current + d(loss_function)/d(U)*alpha
        # Hint: use Q6 in the first part of your homework
        #print(label,pred['prob'])
        #print('grad,w',self.calc_gradients_w(pred, label),self.w)
        self.w = self.w + self.calc_gradients_w(pred, label) * self.alpha
        dU = (label - pred['prob']) * self.w * (1 - h**2)
        for f in range(self.F):
            for j in range(self.window_size):
                index = word_indices[hid[f]+j]
                self.U[index,f,j] = self.U[index,f,j] +  dU[f] * self.alpha

    def calc_gradients_w(self, pred, y):
        #print('w grad',(y - pred['prob']) * pred['h'])
        return (y - pred['prob']) * pred['h']
```

3 Optional: Build your model using Keras + Tensorflow

So far we have always forced you to implement things from scratch. You may feel it's overwhelming, but fortunately, it is not how the real world works. In the real world, there are existing tools you can leverage, so you can focus on the most innovative part of your work. We asked you to do all the previous execises for learning purpose, and since you have already reached so far, it's time to unleash yourself and allow you the access to the real world toolings.

3.1 Sample model

```
In [48]: # First let's see how you can build a similar CNN model you just had using Keras
         from sklearn.pipeline import Pipeline
         from sklearn.preprocessing import OneHotEncoder

         MAX_LENGTH = 100

In [49]: # Yes! it is a good practice to do data processing outside the ML model
         wnet = WordNetLemmatizer()
         # Numerical encode all the words
         unknown = vocab['__unknown__']
         X_train2 = [[vocab.get(wnet.lemmatize(w), unknown) for w in word_tokenize(sent)] for
         sent in X_train]
         X_dev2 = [[vocab.get(wnet.lemmatize(w), unknown)for w in word_tokenize(sent)] for sent
         in X_dev]

         # Tensorflow does not handle variable length input well, let's unify all input to the
         same length
         def trim_X(X, max_length=100, default=vocab['.']):
             for i in range(len(X)):
                 if len(X[i]) > max_length:
                     X[i] = X[i][:max_length]
                 elif len(X[i]) < max_length:
                     X[i] = X[i] + [default] * (max_length - len(X[i]))

             return np.array(X)

         X_train2 = trim_X(X_train2, MAX_LENGTH)
         X_dev2 = trim_X(X_dev2, MAX_LENGTH)


         # Now we have all the input data nicely encoded with numerical label, and each of the
         input data are trimmed
         # to have the same length. We would have needed to further apply one-hot encode for each
         word. However, this
         # would be very expensive, since each word will be expanded into a len(vocab) (~10000)
         length vector. Keras does
         # not support sparse matrix input at this moment. But don't worry, we will use an
         advanced technique called embedding
         # layer. This concept will be introduced in the next lesson. At this moment, you don't
         have to understand why.

In [50]: from keras.models import Sequential
         from keras.layers import Embedding, Conv1D, MaxPooling1D, Dense, GlobalMaxPooling1D

         model = Sequential()
         model.add(Embedding(input_dim=len(vocab), input_length=MAX_LENGTH, output_dim=1024,
         name="Embedding-1"))
         model.add(Conv1D(filters=100, kernel_size=2, activation="tanh", name="Conv1D-1"))
         model.add(GlobalMaxPooling1D(name="MaxPooling1D-1"))
         model.add(Dense(1, activation="sigmoid", name="Dense-1"))
         print(model.summary())

         show_keras_model(model)
```

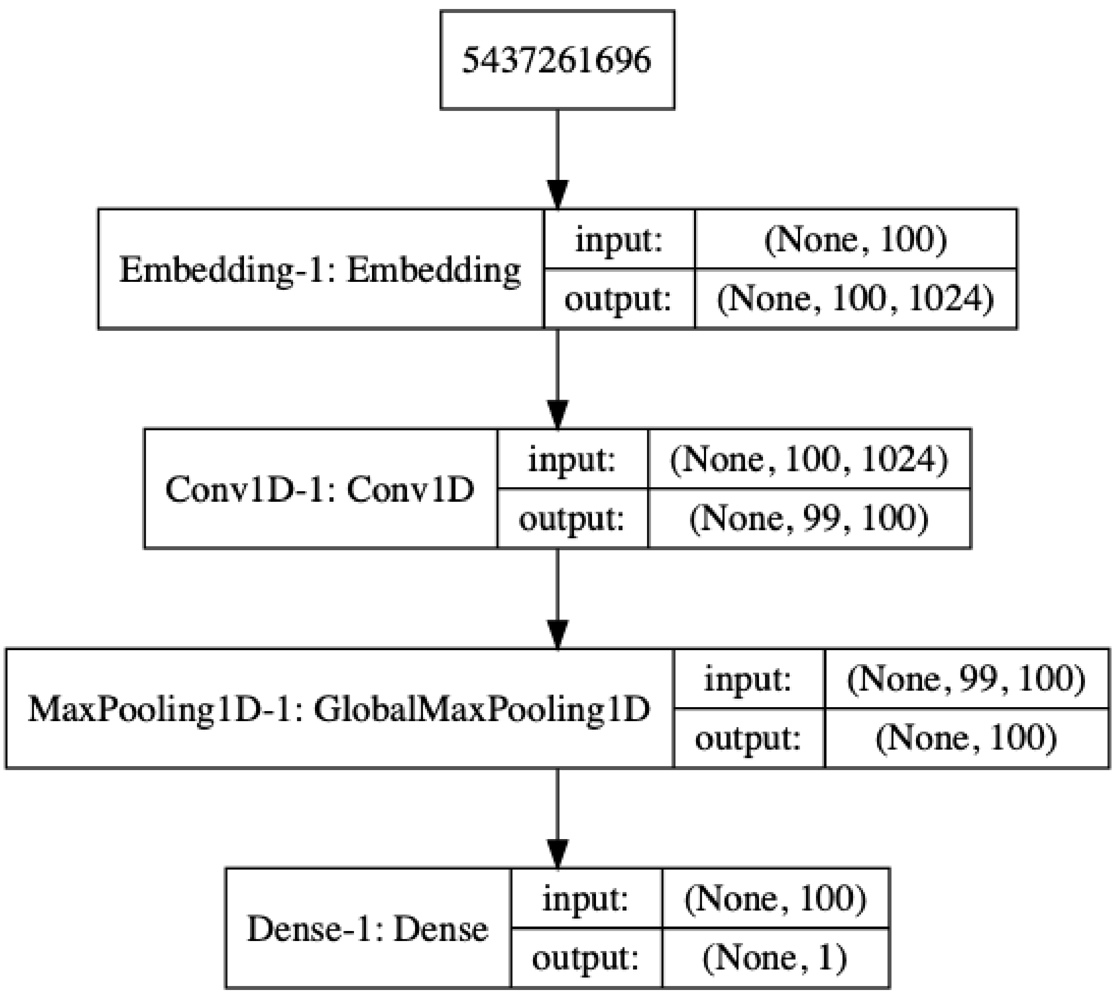
```
WARNING: Logging before flag parsing goes to stderr.
W0313 17:48:33.565071 4701199808 deprecation_wrapper.py:119] From
/usr/local/lib/python3.6/site-packages/keras/backend/tensorflow_backend.py:74: The
name tf.get_default_graph is deprecated. Please use tf.compat.v1.get_default_graph
instead.

W0313 17:48:33.620762 4701199808 deprecation_wrapper.py:119] From
/usr/local/lib/python3.6/site-packages/keras/backend/tensorflow_backend.py:517: The
name tf.placeholder is deprecated. Please use tf.compat.v1.placeholder instead.
```

```
W0313 17:48:33.636304 4701199808 deprecation_wrapper.py:119] From
/usr/local/lib/python3.6/site-packages/keras/backend/tensorflow_backend.py:4138: The
name tf.random_uniform is deprecated. Please use tf.random.uniform instead.
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
Embedding-1 (Embedding)      (None, 100, 1024)         10241024
_________________________________________________________________
Conv1D-1 (Conv1D)            (None, 99, 100)           204900
_________________________________________________________________
MaxPooling1D-1 (GlobalMaxPoo (None, 100)               0
_________________________________________________________________
Dense-1 (Dense)              (None, 1)                 101
=================================================================
Total params: 10,446,025
Trainable params: 10,446,025
Non-trainable params: 0
_________________________________________________________________
None
```

[50]:



```
In [51]: # Train the model
         model.compile(loss="binary_crossentropy", optimizer='adam', metrics=['accuracy'])
         model.fit(X_train2, y_train, epochs=10, validation_data=[X_dev2, y_dev])
```

```
W0313 17:48:33.940645 4701199808 deprecation_wrapper.py:119] From
/usr/local/lib/python3.6/site-packages/keras/optimizers.py:790: The name
tf.train.Optimizer is deprecated. Please use tf.compat.v1.train.Optimizer instead.

W0313 17:48:33.970345 4701199808 deprecation_wrapper.py:119] From
/usr/local/lib/python3.6/site-packages/keras/backend/tensorflow_backend.py:3376: The
name tf.log is deprecated. Please use tf.math.log instead.

W0313 17:48:33.975730 4701199808 deprecation.py:323] From
/usr/local/lib/python3.6/site-packages/tensorflow/python/ops/nn_impl.py:180:
add_dispatch_support.<locals>.wrapper (from tensorflow.python.ops.array_ops) is
deprecated and will be removed in a future version.
Instructions for updating:
Use tf.where in 2.0, which has the same broadcast rule as np.where
W0313 17:48:34.191435 4701199808 deprecation_wrapper.py:119] From
/usr/local/lib/python3.6/site-packages/keras/backend/tensorflow_backend.py:986: The
name tf.assign_add is deprecated. Please use tf.compat.v1.assign_add instead.
```

```
Train on 7000 samples, validate on 3000 samples
Epoch 1/10
7000/7000 [==============================] - 34s 5ms/step - loss: 0.6198 - acc: 0.6487
- val_loss: 0.5654 - val_acc: 0.6970
Epoch 2/10
7000/7000 [==============================] - 37s 5ms/step - loss: 0.3878 - acc: 0.8349
- val_loss: 0.5320 - val_acc: 0.7410
Epoch 3/10
7000/7000 [==============================] - 40s 6ms/step - loss: 0.1807 - acc: 0.9477
- val_loss: 0.5990 - val_acc: 0.7333
Epoch 4/10
7000/7000 [==============================] - 43s 6ms/step - loss: 0.0665 - acc: 0.9877
- val_loss: 0.7008 - val_acc: 0.7270
Epoch 5/10
7000/7000 [==============================] - 37s 5ms/step - loss: 0.0208 - acc: 0.9974
- val_loss: 0.8237 - val_acc: 0.7210
Epoch 6/10
7000/7000 [==============================] - 37s 5ms/step - loss: 0.0077 - acc: 0.9996
- val_loss: 0.9294 - val_acc: 0.7247
Epoch 7/10
7000/7000 [==============================] - 38s 5ms/step - loss: 0.0040 - acc: 0.9997
- val_loss: 1.0003 - val_acc: 0.7233
Epoch 8/10
7000/7000 [==============================] - 36s 5ms/step - loss: 0.0027 - acc: 0.9994
- val_loss: 1.0617 - val_acc: 0.7223
Epoch 9/10
7000/7000 [==============================] - 44s 6ms/step - loss: 0.0021 - acc: 0.9997
- val_loss: 1.1127 - val_acc: 0.7173
Epoch 10/10
7000/7000 [==============================] - 45s 6ms/step - loss: 0.0016 - acc: 0.9999
- val_loss: 1.1623 - val_acc: 0.7133
```

```
[51]: <keras.callbacks.History at 0x1442f3860>
```

3.2 Play with your own model

We have shown you have to use an industry level tool to build a CNN model. Hopefully you think it is simpler than the version we built from scratch. Not really? Read Keras Documentation and learn more: https://keras.io/

```
In [52]: # # Now it's your turn to build some more complicated CNN models
         from keras import regularizers
         from keras import optimizers
         from keras.layers import GlobalAveragePooling1D

         X_train2 = [[vocab.get(wnet.lemmatize(w), unknown) for w in word_tokenize(sent)] for
         sent in X_train]
         X_dev2 = [[vocab.get(wnet.lemmatize(w), unknown)for w in word_tokenize(sent)] for sent
         in X_dev]
         X_train3 = trim_X(X_train2, 300)
         X_dev3 = trim_X(X_dev2, 300)
```

```
In [60]: model = Sequential()
         model.add(Embedding(input_dim=len(vocab), input_length=300, output_dim=512,
         name="Embedding-1"))
         model.add(Conv1D(filters=100, kernel_size=4,
         activation="tanh",kernel_regularizer=regularizers.l2(0.01), name="Conv1D-1"))
         model.add(GlobalAveragePooling1D(name="AveragePooling1D-1"))
         model.add(Dense(1, activation="sigmoid",name="Dense-1"))
         print(model.summary())
         adam = optimizers.Adam(lr=0.01, beta_1=0.9, beta_2=0.999, amsgrad=False)
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
Embedding-1 (Embedding)      (None, 300, 512)          5120512
_________________________________________________________________
Conv1D-1 (Conv1D)            (None, 297, 100)          204900
_________________________________________________________________
AveragePooling1D-1 (GlobalAv (None, 100)               0
_________________________________________________________________
Dense-1 (Dense)              (None, 1)                 101
=================================================================
Total params: 5,325,513
Trainable params: 5,325,513
Non-trainable params: 0
_________________________________________________________________
None
```

```
In [61]: model.compile(loss="binary_crossentropy", optimizer='adam', metrics=['accuracy'])
         model.fit(X_train3, y_train, epochs=20, validation_data=[X_dev3, y_dev])
```

```
Train on 7000 samples, validate on 3000 samples
Epoch 1/20
7000/7000 [==============================] - 37s 5ms/step - loss: 0.8270 - acc: 0.4977
- val_loss: 0.6930 - val_acc: 0.5043
Epoch 2/20
7000/7000 [==============================] - 37s 5ms/step - loss: 0.6879 - acc: 0.5554
- val_loss: 0.6753 - val_acc: 0.6607
Epoch 3/20
7000/7000 [==============================] - 37s 5ms/step - loss: 0.6264 - acc: 0.6753
```

```
- val_loss: 0.6301 - val_acc: 0.6150
Epoch 4/20
7000/7000 [==============================] - 37s 5ms/step - loss: 0.5314 - acc: 0.7743
- val_loss: 0.5892 - val_acc: 0.7517
Epoch 5/20
7000/7000 [==============================] - 37s 5ms/step - loss: 0.4630 - acc: 0.8181
- val_loss: 0.5723 - val_acc: 0.7473
Epoch 6/20
7000/7000 [==============================] - 40s 6ms/step - loss: 0.4088 - acc: 0.8520
- val_loss: 0.5865 - val_acc: 0.7360
Epoch 7/20
7000/7000 [==============================] - 37s 5ms/step - loss: 0.3605 - acc: 0.8757
- val_loss: 0.6022 - val_acc: 0.7357
Epoch 8/20
7000/7000 [==============================] - 37s 5ms/step - loss: 0.3272 - acc: 0.8946
- val_loss: 0.6255 - val_acc: 0.7663
Epoch 9/20
7000/7000 [==============================] - 37s 5ms/step - loss: 0.3001 - acc: 0.9067
- val_loss: 0.6678 - val_acc: 0.7510
Epoch 10/20
7000/7000 [==============================] - 37s 5ms/step - loss: 0.2741 - acc: 0.9154
- val_loss: 0.6420 - val_acc: 0.7667
Epoch 11/20
7000/7000 [==============================] - 37s 5ms/step - loss: 0.2505 - acc: 0.9271
- val_loss: 0.6735 - val_acc: 0.7477
Epoch 12/20
7000/7000 [==============================] - 37s 5ms/step - loss: 0.2319 - acc: 0.9364
- val_loss: 0.7006 - val_acc: 0.7600
Epoch 13/20
7000/7000 [==============================] - 37s 5ms/step - loss: 0.2168 - acc: 0.9429
- val_loss: 0.7634 - val_acc: 0.7507
Epoch 14/20
7000/7000 [==============================] - 39s 6ms/step - loss: 0.2033 - acc: 0.9437
- val_loss: 0.7449 - val_acc: 0.7597
Epoch 15/20
7000/7000 [==============================] - 38s 5ms/step - loss: 0.1855 - acc: 0.9543
- val_loss: 0.8727 - val_acc: 0.7370
Epoch 16/20
7000/7000 [==============================] - 37s 5ms/step - loss: 0.1784 - acc: 0.9569
- val_loss: 0.8091 - val_acc: 0.7550
Epoch 17/20
7000/7000 [==============================] - 38s 5ms/step - loss: 0.1604 - acc: 0.9656
- val_loss: 0.8831 - val_acc: 0.7560
Epoch 18/20
7000/7000 [==============================] - 37s 5ms/step - loss: 0.1571 - acc: 0.9630
- val_loss: 0.8416 - val_acc: 0.7550
Epoch 19/20
7000/7000 [==============================] - 37s 5ms/step - loss: 0.1462 - acc: 0.9664
- val_loss: 0.9316 - val_acc: 0.7527
Epoch 20/20
7000/7000 [==============================] - 38s 5ms/step - loss: 0.1363 - acc: 0.9710
- val_loss: 0.9500 - val_acc: 0.7403
```

```
[61]: <keras.callbacks.History at 0x11af80710>
```

```
In [64]: X_test = load_data("test.txt").text
         X_test = [[vocab.get(wnet.lemmatize(w), unknown)for w in word_tokenize(sent)] for sent
         in X_test]
```

```
X_test = trim_X(X_test, 300)
y_test = model.predict(X_test)
y_test = [y > 0.5 for y in y_test]
save_prediction(y_test)
```

In [ ]: