Week 09: Word Sense Disambiguation

This week, we introduced a hot topic in Natural Language Proccessing: *Word Sense Disambiguation (WSD)* .
Many words in natural languages have ambiguous meanings. For example, the word *party* can refer to 1) a social gathering (派對), 2) a political organization (政黨), or 3) an entity in law (當事人；……方).
As a human, we can distinguish different meanings easily, but can a machine do the same? This is what WSD aims for.

Introduction

tl; dr

You have to

1. preprocess the data
2. (stage 1) generate a small training dataset from the given collocation seed,
3. (stage 1) train a weak model on that small dataset,
4. (stage 2) use the weak model to generate more labeled data, and
5. (stage 2) train your final model
6. Evaluate your model on testing data (requirement: accuracy > 0.7)

Concept

In Lesk's assumption, each word has only one sense when it appears in the same collocation. For example, if *party* shows up with the word *court* (法庭), most likely the sense of this *party* is the 3rd one: an entity in law (當事人；……方).
However, we are not implementing Lesk's algorithm this week. Instead, we will combine his assumption with Yarowsky's *bootstrap technique* .

You are given some pre-defined collocations, or called *seeds*, of the word *party*, along with which sense each collocation belongs to.
With the given seeds, you can generate a small set of labeled data by rule. Then with this small set, we can train a small model with limited accuracy.
The current classifier might not perform well on the whole dataset, sure, but it's already enough to generate more reliable labeled data. With the newly labeled training data, we can now train another sense-classification model with more robustness, which aims for the real WSD task.
This process, about training on smaller dataset, generating more data, and then improving the model itself, is called *bootstrapping* .

I. Data preparation

First thing first. To make natural language understandable for machines, we have to transform sentences into embeddings.
So here are four things to do:

1. load data
2. preprocess the sentences
3. transform sentences into embeddings
4. pad the sentences to the same length

To make the task simple and easy to understand, we will only work on a single word *party* . Three senses of *party* is defined as below with their corresponding `sense id` s.

In [1]:
```
SENSE = {
    1: 'a social event at which a group of people meet to talk, eat, drink, d
    2: 'an organization of people with particular political beliefs', # 政黨
    3: 'a single entity which can be identified as one for the purposes of th
}
```

1. Load data

The data is a set of sentences containing the word *party*, all extracted from wikipedia. The uniqueness of each sentence is guaranteed.

In [2]:
```
import os
```

In [3]:
```
with open(os.path.join('data', 'party.train.txt'), 'r') as f:
    data = f.read().strip().split('\n')

# this dict maps sentence_id to the sentence itself
pure_data = { sent_id: text for sent_id, text in [line.split('\t', 1)
                                                  for line in data] }
```

Let's see what the data looks like.

In [4]:
```
for sent_id, sentence in pure_data.items():
    if int(sent_id) > 1003: break

    print(f'{sent_id}: {sentence}')
```

```
1001: A naked party, also known as nude party, is a party where the participa
nts are required to be nude.
1002: The town center bears the hallmarks of a typical migration-accepting Tu
rkish rural town, with traditional structures coexisting with a collection of
concrete apartment blocks providing public housing, as well as amenities such
as basic shopping and fast-food restaurants, and essential infrastructure but
little in the way of culture except for cinemas and large rooms hired out for
wedding parties.
1003: Elections Alberta oversees the creation of political parties and riding
associations, compiles election statistics on ridings, and collects financial
statements from party candidates and riding associations.
```

In [5]:
```
# a look up table from sentence to id
id_mapper = {v: k for k, v in pure_data.items()}
# a table for id to embedding; we will deal with this later
processed_data = {}
```

We define 2 samples here to validate the preprocess during our coding.

In [6]:
```
samples = [
    'Adnan Al-Hakim (died May 26, 1990) was the leader of the Najjadeh Party,
```

```
    'A block party or street party is a party in which many members of a sing
]
```

2. Preprocess the sentences

[TODO] Define your preprocessing function to transform a sentence into tokens here.

-

*hint: If you can't get a high accuracy in the final result, you may want to come back and modify your preprocessing here.
*hint: Think about what words are useful and what are useless when distinguishing a sense.

In [7]:
```
def preprocess(text):
    # [ TODO ]
    ...
    return [text]
```

In [8]:
```
sent_tokens = [preprocess(sent) for sent in samples]
sent_tokens[0][:5]
```

Out[8]:
```
['Adnan', 'Al', 'Hakim', 'died', 'May']
```

3. Transform sentences into embeddings

For the simplicity, we are still using word2vec here, so you can copy-paste your code from previous week.
This is not required; you don't have to use word2vec if you want to train a embedding model along with the classifier.

*Download w2v: [Google Code Archive]
(https://code.google.com/archive/p/word2vec/#Pretrained-word-and-phrase-vectors)

In [9]:
```
import numpy as np
from gensim.models import KeyedVectors
```

In [10]:
```
w2v = KeyedVectors.load_word2vec_format(
        os.path.join('data', 'GoogleNews-vectors-negative300.bin'),
        binary = True
      )
```

In [11]:
```
def to_embedding(tokens):
    # [ TODO ]
    ...
    return tokens
```

In [12]:
```
embeddings = [to_embedding(tokens) for tokens in sent_tokens]
embeddings[0]
```

Out[12]:
```
array([[-0.140625  ,  0.20703125, -0.12988281, ...,  0.03076172,
         0.07080078,  0.484375  ],
       [-0.0390625 ,  0.24804688,  0.00540161, ...,  0.2265625 ,
```

```
         0.02404785, -0.01477051],
       [ 0.05541992,  0.31640625,  0.27929688, ..., -0.06689453,
         0.34179688,  0.27929688],
       ...,
       [-0.09130859, -0.08691406, -0.01208496, ...,  0.02832031,
        -0.23242188,  0.11962891],
       [ 0.13183594,  0.00268555, -0.12792969, ..., -0.01263428,
        -0.07080078,  0.09863281],
       [-0.12695312,  0.20898438, -0.10644531, ...,  0.13476562,
         0.01879883, -0.1484375 ]], dtype=float32)
```

4. Pad the sentences to the same length

The input size of model is fixed. However, the sentence lengths are various.
An intuitive solution is to stuff some dummy values into arrays util they share the same size, and this is called *padding*.

*tf.keras.preprocessing.sequence.pad_sequences

In [13]:
```
# if you prefer numpy
import numpy as np
# or if you prefer tensorflow
from tensorflow.keras.preprocessing.sequence import pad_sequences
```

In [14]:
```
def add_padding(embeddings, padding_width = None):
    # [ TODO ]
    # Pad all embeddings to padding_width, or detect it automatically when it
    # ps. tensorflow's `pad_sequences` can detect that for you
    ...
    return embeddings
```

In [15]:
```
emb_padded = add_padding(embeddings)
emb_padded[0]
```

Out[15]:
```
array([[ 0.        ,  0.        ,  0.        , ...,  0.        ,
         0.        ,  0.        ],
       [ 0.        ,  0.        ,  0.        , ...,  0.        ,
         0.        ,  0.        ],
       [ 0.        ,  0.        ,  0.        , ...,  0.        ,
         0.        ,  0.        ],
       ...,
       [-0.09130859, -0.08691406, -0.01208496, ...,  0.02832031,
        -0.23242188,  0.11962891],
       [ 0.13183594,  0.00268555, -0.12792969, ..., -0.01263428,
        -0.07080078,  0.09863281],
       [-0.12695312,  0.20898438, -0.10644531, ...,  0.13476562,
         0.01879883, -0.1484375 ]])
```

In [16]:
```
print(len(embeddings[0]), len(embeddings[1]))
print(emb_padded[0].shape, emb_padded[1].shape)
```

```
12 17
(17, 300) (17, 300)
```

You should see the embedding of shorter sentence is padded by empty arrays, and they are at the same length now.

In [17]:
```
# record the width for the future use.
PADDING_WIDTH = emb_padded[0].shape[0]
```

```
print(PADDING_WIDTH)
```

```
17
```

5. all-in-one

Define a function to setup the pipeline, and transform all sentences into embeddings!

*Your embedding shape might not be the same with ours due to our different preprocessing procedure.

In [18]:
```
def process_text(sentences, padding = None):
    result = [ preprocess(sentence) for sentence in sentences ]
    result = [ to_embedding(sentence) for sentence in result ]
    result = add_padding(result, padding)
    return result
```

In [19]:
```
X = process_text(pure_data.values())
```

In [20]:
```
X[0] # should be an embedding with padding
```

Out[20]:
```
array([[ 0.        ,  0.        ,  0.        , ...,  0.        ,
         0.        ,  0.        ],
       [ 0.        ,  0.        ,  0.        , ...,  0.        ,
         0.        ,  0.        ],
       [ 0.        ,  0.        ,  0.        , ...,  0.        ,
         0.        ,  0.        ],
       ...,
       [-0.06298828, -0.01531982, -0.01708984, ..., -0.02392578,
         0.01843262, -0.06542969],
       [-0.12890625, -0.18261719,  0.10351562, ..., -0.07714844,
        -0.11572266, -0.02832031],
       [ 0.21582031, -0.12207031,  0.09765625, ..., -0.06201172,
        -0.17089844,  0.02563477]])
```

In [21]:
```
X.shape # should be (637, *, 300), * depends on your preprocessing
```

Out[21]:
```
(637, 59, 300)
```

Let's use a dictionary to store all embeddings with their sentence_id.

In [22]:
```
processed_data = {
    sent_id: embedding for sent_id, embedding in zip(pure_data, X)
}
```

In [23]:
```
print(pure_data['1001'])
processed_data['1001']
```

```
A naked party, also known as nude party, is a party where the participants ar
e required to be nude.
```

Out[23]:
```
array([[ 0.        ,  0.        ,  0.        , ...,  0.        ,
         0.        ,  0.        ],
       [ 0.        ,  0.        ,  0.        , ...,  0.        ,
         0.        ,  0.        ],
       [ 0.        ,  0.        ,  0.        , ...,  0.        ,
         0.        ,  0.        ],
       ...,
```

```
[-0.06298828, -0.01531982, -0.01708984, ..., -0.02392578,
  0.01843262, -0.06542969],
[-0.12890625, -0.18261719,  0.10351562, ..., -0.07714844,
 -0.11572266, -0.02832031],
[ 0.21582031, -0.12207031,  0.09765625, ..., -0.06201172,
 -0.17089844,  0.02563477]])
```

II. First stage

After preprocessing the training data, now we are going to train our first-stage model!

According to the method described at the beginning, we can train a simple model on a smaller dataset, and this dataset can be generated by rule from seeds.

Steps

1. Prepare the training data
2. Encode labels
3. Split training and testing dataset
4. Build classifier
5. Train

1. Prepare the training data

Given the seed collocationss, you can add a sentence into the training data with label if that sentence contains that collocation.
For example, we can say *"A party is a **social** gathering."* should be the first sense, because it contains the keyword *social*. Hence, your training data will have this sentence with its label `1`.

Don't worry about the false-positive cases for now.
If the seed is generally good enough, the model will learn to ignore those wrong data by itself.
(though yeah, you can get better results if you deal with it beforehand)

In [26]:
```
SEEDS = {
    1: ['social', 'events'],
    2: ['system', 'coalition'],
    3: ['court', 'law']
}
```

[TODO] Get the initial training data from the given seeds.

In [27]:
```
# [TODO]
indice, first_X, first_Y = [], [], [] # sentence id of selected samples, sele
for sent_id, sentence in pure_data.items():
    ...
```

Examine training data.
The labels might not be 100% correct, but it should look reasonable.

In [28]:
```
for i in range(5):
    print(pure_data[indice[i]])
    print(f' -> {first_Y[i]}: {SENSE[first_Y[i]]}')
    print()
```

```
From these social conventions derive in turn also the variants worn on relate
d occasions of varying solemnity, such as formal political, diplomatic, and a
cademic events, in addition to certain parties including award ceremonies, ba
lls, fraternal orders, high school proms, etc.
 -> 1: a social event at which a group of people meet to talk, eat, drink, da
nce, etc.

The Free-minded People's Party () or Radical People's Party was a social libe
ral party in the German Empire, founded as a result of the split of the Germa
n Free-minded Party in 1893.
 -> 1: a social event at which a group of people meet to talk, eat, drink, da
nce, etc.

Typically, a party has the right to object in court to a line of questioning
or at the introduction of a particular piece of evidence.
 -> 3: a single entity which can be identified as one for the purposes of the
law

Dizzy bat is commonly played at parties, colleges and universities, bars, and
other drinking festivities such as a tailgate party at sporting events and co
ncerts.
 -> 1: a social event at which a group of people meet to talk, eat, drink, da
nce, etc.

Party of Peace and Unity (Партия Мира и Единства, "Partiya Mira i Yedinstva")
was a socialist party in Russia.
 -> 1: a social event at which a group of people meet to talk, eat, drink, da
nce, etc.
```

Transform X and Y into numpy array for future use.

In [30]:
```
first_X = np.array(first_X)
first_Y = np.array(first_Y)
first_X.shape
```

Out[30]:
```
(201, 59, 300)
```

2. Encode labels

The labels now are all categorical, which are `1`, `2`, and `3`. However, it's hard to teach a machine this kind of answers.
Most of the time, machine learning generates a *numeric probability*, like `0.329`, rather than a categorical result.
That's why we want to encode the label into a floating point between 0 ~ 1, so that the machine can generate the probability of each answer.

Here we suggest you use the one-hot encoding, which is suitable for categorical classification.
So the label `2` will look like

```
 Sense 1, Sense 2, Sense 3
[      0,       1,       0]
```

*Why One-Hot Encode Data in Machine Learning?

In [31]:
```
# if you prefer tensorflow
from tensorflow import one_hot
# or if you don't like tensorflow
from sklearn.preprocessing import OneHotEncoder
```

[TODO] one-hot encode `first_Y`

*tf.one_hot
*sklearn.preprocessing.OneHotEncoder

In [32]:
```
# [TODO]
...
```

In [34]:
```
first_Y[:5]
```

Out[34]:
```
array([[1., 0., 0.],
       [1., 0., 0.],
       [0., 0., 1.],
       [1., 0., 0.],
       [1., 0., 0.]])
```

3. Prepare training and validation set

Split the dataset into training set and validation set.
The reason for splitting is because, you may not want the model to see what you'll use to test it when it is still learning.

Machine is very smart; sometimes it just *memorizes* the answers, rather than *learns* them. Even that the model has yielded a perfect accuracy in the test, it still might fail miserably when facing the cruel, real world. *(heh)*
That's why we need a validation set. We reserve a partition of data that will never be learnt by the model, and use it to validate whether the model really learns someting.

*Train, Validation and Test Sets

In [35]:
```
# if you prefer sklearn
from sklearn.model_selection import train_test_split
# or if you don't like sklearn. **Remember to shuffle your data before splitt
import numpy as np
```

In [36]:
```
X_train, X_val, Y_train, Y_val = train_test_split(
    first_X, first_Y,
    test_size = ...,   # [TODO] How much data you want to used as validation
    shuffle = True
)
```

In [37]:
```
print(X_train.shape, X_val.shape, Y_train.shape, Y_val.shape)
```

```
(160, 59, 300) (41, 59, 300) (160, 3) (41, 3)
```

4. Build your multi-labeling classifier

Now the data is all prepared.
Let's build a model to learn from it!

Note that, different from last week, your output dimension should be the size of all categories, rather than `2` .

-

*Although tensorflow is used below, you can always change it to any other framework you are familiar with.

*tf.keras.layers

In [38]:
```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Dense#, and all the other layers yo
```

In [39]:
```
_, PADDING_WIDTH, EMBEDDING_DIM = X_train.shape
OUTPUT_CATEGORY = len(SENSE)

print(PADDING_WIDTH, EMBEDDING_DIM, OUTPUT_CATEGORY)
```

```
59 300 3
```

[TODO] Build a classifier

In [40]:
```
model_1 = Sequential()

# [TODO]
...

print(model_1.summary())
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
lstm (LSTM)                  (None, 32)                42624
_________________________________________________________________
dense (Dense)                (None, 3)                 99
=================================================================
Total params: 42,723
Trainable params: 42,723
Non-trainable params: 0
_________________________________________________________________
None
2021-11-10 00:36:12.063567: I tensorflow/core/platform/cpu_feature_guard.cc:142] This TensorFlow binary is optimized with oneAPI Deep Neural Network Library (oneDNN) to use the following CPU instructions in performance-critical operations:  SSE4.1 SSE4.2 AVX AVX2 AVX512F FMA
To enable them in other operations, rebuild TensorFlow with the appropriate compiler flags.
```

Time to choose the optimizer and loss function.

Loss function is an equation evaluating how wrong your model has answered (the lower the better), while optimizer tells the model how to improve itself.
But seriously, we are not asking you to fine-tune these parameters. That is for Machine Learning class, not for NLP class, so if you are not able to pass the baseline, go check your processing procedure first. Something might go wrong there.

-

*tf.keras.model#compile
*tf.keras.optimizers
*tf.keras.losses

[TODO] Compile your model

In [41]:
```
# [TODO]
model_1.compile(...)
```

5. Train

Time to train your model!

You should always prevent the model from overfitting, so take validation accuracy into consideration and choose your epoch number wisely.

*What is Overfitting?

[TODO] Train and tune your model

In [42]:
```
history = model_1.fit(
    X_train, Y_train,
    validation_data=(X_val, Y_val),
    epochs = ...,          # [TODO] how many iterations you want to run
    # initial_epoch = ?    # set this if you're continuing previous training
)
```

```
Epoch 1/7
5/5 [==============================] - 2s 119ms/step - loss: 1.0196 - accurac
y: 0.5813 - val_loss: 1.0157 - val_accuracy: 0.5122
Epoch 2/7
5/5 [==============================] - 0s 37ms/step - loss: 0.9182 - accurac
y: 0.5250 - val_loss: 0.9659 - val_accuracy: 0.4634
Epoch 3/7
5/5 [==============================] - 0s 34ms/step - loss: 0.8392 - accurac
y: 0.5250 - val_loss: 0.9217 - val_accuracy: 0.4634
Epoch 4/7
5/5 [==============================] - 0s 37ms/step - loss: 0.7653 - accurac
y: 0.5562 - val_loss: 0.8495 - val_accuracy: 0.5610
Epoch 5/7
5/5 [==============================] - 0s 29ms/step - loss: 0.6787 - accurac
y: 0.7250 - val_loss: 0.7483 - val_accuracy: 0.5854
Epoch 6/7
5/5 [==============================] - 0s 35ms/step - loss: 0.5739 - accurac
y: 0.8562 - val_loss: 0.6339 - val_accuracy: 0.8293
Epoch 7/7
5/5 [==============================] - 0s 37ms/step - loss: 0.4666 - accurac
y: 0.9187 - val_loss: 0.5100 - val_accuracy: 0.8780
```

In [43]:
```
# example of continued training

# history = model_1.fit(
#     X_train, Y_train,
#     validation_data=(X_val, Y_val),
#     epochs = 10,          # how many iterations you want to run
#     initial_epoch = 7     # set this if you're continuing previous training
# )
```

```
Epoch 8/10
5/5 [==============================] - 0s 46ms/step - loss: 0.3565 - accurac
y: 0.9250 - val_loss: 0.4169 - val_accuracy: 0.8780
Epoch 9/10
5/5 [==============================] - 0s 37ms/step - loss: 0.2713 - accurac
y: 0.9312 - val_loss: 0.3433 - val_accuracy: 0.9024
Epoch 10/10
```

```
5/5 [==============================] - 0s 37ms/step - loss: 0.2119 - accurac
y: 0.9563 - val_loss: 0.2912 - val_accuracy: 0.9024
```

6. Examine your model

Let's see how good your model does.

In [44]:
```
testcases = [
    # 1
    'A block party or street party is a party in which many members of a sing
    'A party is a social gathering.',
    # 2
    'Ukraine has a multi-party system, with numerous parties in which often r
    'Serbia has a multi-party system, with numerous parties in which no one p
    # 3
    'In a civil lawsuit, a nominal party is one named as a party on the recor
]
```

In [45]:
```
# you must specify the padding width here, since the input size of model shou
test_X = process_text(testcases, padding = PADDING_WIDTH)
```

In [46]:
```
predictions = model_1.predict(test_X)
```

In [47]:
```
predictions[0]
```

Out[47]:
```
array([0.7178152 , 0.65034765, 0.18324545], dtype=float32)
```

What does the result mean?

As you can see, a list of floats are generated, and since we used one-hot encoding when preparing the training data, each number presents the result of corresponding categories.

```
 Sense 1, Sense 2, Sense 3
[   0.89,    0.12,    0.21]
```

You can consider these values as the probability of each column, or said category. Hence, the true predicted label should be the one with the highest probability, which is Sense 1 for this sample.

Now let's get all the predicted labels from these probabilities.

In [48]:
```
for idx, result in enumerate(predictions):
    predict_id = result.argmax() # select the index of the maximum value
    sense_id = predict_id + 1    # sense_id starts from 1
    print(testcases[idx])
    print(f'-> Sense {sense_id} (prob={result[predict_id]:.2f}): {SENSE[sense
    print()
```

```
A block party or street party is a party in which many members of a single co
mmunity congregate, either to observe an event of some importance or simply f
or mutual enjoyment.
-> Sense 1 (prob=0.72): a social event at which a group of people meet to tal
k, eat, drink, dance, etc.

A party is a social gathering.
```

```
-> Sense 1 (prob=0.75): a social event at which a group of people meet to tal
k, eat, drink, dance, etc.

Ukraine has a multi-party system, with numerous parties in which often not a
single party has a chance of gaining power alone, and parties must work with
each other to form coalition governments.
-> Sense 2 (prob=0.97): an organization of people with particular political b
eliefs

Serbia has a multi-party system, with numerous parties in which no one party
often has a chance of gaining power alone, and parties must work with each ot
her to form coalition governments.
-> Sense 2 (prob=0.97): an organization of people with particular political b
eliefs

In a civil lawsuit, a nominal party is one named as a party on the record of
an action, but having no interest in the action.
-> Sense 3 (prob=0.80): a single entity which can be identified as one for th
e purposes of the law
```

Again, the label might not be 100% correct, but it should look reasonable somehow.

III. Second stage

The previous model might not be enough for real-world use; another model with better ability is needed.

*Most contents of this section are the same as previous one, so you can make use of your code above.

1. Prepare the training data

The model from the previous section is weak, yet it still has learned some valuable knowledge. Let's ask that model to label more training data for us!

In [49]:
```
# Get the probability on the whold dataset
predictions = model_1.predict(np.array(list(processed_data.values())))
```

[TODO] Get the labels of all data, and reserve only those labels with high probabilities.

In [52]:
```
THRESHOLD = 0.  # you may want to change this :)
indice, second_X, second_Y = [], [], [] # sentence id of selected samples, se

for sent_id, result in zip(processed_data, predictions):
    # [TODO]
    ...
```

Observe the selected data size and the quality of labels.
You might want to go back and modify your preprocessing, first model, or the threshold until you get a better training data.

In [ ]:
```
for i in range(5):
    print(pure_data[indice[i]])
    print(f' -> {second_Y[i]}: {SENSE[second_Y[i]]}')
    print()
```

In [53]:
```
second_X = np.array(second_X)
```

```
second_Y = np.array(second_Y)
second_X.shape
```

Out[53]:
```
(220, 59, 300)
```

2. Encode labels

[TODO] one-hot encode secone_Y

In [54]:
```
# [TODO]
```

In [55]:
```
second_Y[:3]
```

Out[55]:
```
array([[1., 0., 0.],
       [0., 1., 0.],
       [1., 0., 0.]])
```

3. Prepare training and validating dataset

In [56]:
```
X_train, X_val, Y_train, Y_val = train_test_split(
    second_X, second_Y,
    test_size = ...,    # [TODO] How much data you want to used as validation
    shuffle = True
)
```

In [77]:
```
X_train.shape
```

Out[77]:
```
(176, 59, 300)
```

4. Build model

In [57]:
```
# the number comes from previous setting
print(PADDING_WIDTH, EMBEDDING_DIM, OUTPUT_CATEGORY)
```

```
59 300 3
```

[TODO] Build your second model

*This model can be different from the previous one.

In [58]:
```
model_2 = Sequential()

# [TODO]
...

print(model_2.summary())
```

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
lstm_1 (LSTM)                (None, 32)                42624
_________________________________________________________________
dense_1 (Dense)              (None, 3)                 99
=================================================================
Total params: 42,723
Trainable params: 42,723
```

```
Non-trainable params: 0
_________________________________________________________________
None
```

[TODO] Compile your model

In [59]:
```
# [TODO]
model_2.compile(...)
```

5. Train model

[TODO] Train it!

In [60]:
```
history = model_2.fit(
    X_train, Y_train,
    validation_data=(X_val, Y_val),
    epochs = ...,        # [TODO] how many iterations you want to run
    # initial_epoch = ?  # set this if you're continuing previous training
)
```

```
Epoch 1/5
6/6 [==============================] - 2s 113ms/step - loss: 1.0455 - accurac
y: 0.5568 - val_loss: 0.9039 - val_accuracy: 0.6591
Epoch 2/5
6/6 [==============================] - 0s 42ms/step - loss: 0.8431 - accurac
y: 0.6420 - val_loss: 0.7327 - val_accuracy: 0.6591
Epoch 3/5
6/6 [==============================] - 0s 36ms/step - loss: 0.7112 - accurac
y: 0.6193 - val_loss: 0.6196 - val_accuracy: 0.6591
Epoch 4/5
6/6 [==============================] - 0s 37ms/step - loss: 0.5894 - accurac
y: 0.7045 - val_loss: 0.5077 - val_accuracy: 0.8182
Epoch 5/5
6/6 [==============================] - 0s 34ms/step - loss: 0.4455 - accurac
y: 0.8523 - val_loss: 0.3919 - val_accuracy: 0.8864
```

6. Examine the result

In [62]:
```
testcases = [
    # 1
    'Green Beer Day (GBD) is a day-long party, where celebrants drink beer dy
    'When the siblings grew up, they held parties and introduced the traditic
    # 2
    'Politicians from the two main parties tend to win elections when not cor
    'After the general election on 22 March 1992, five parties (Rassadorn, Ju
    # 3
    'Typically, a party has the right to object in court to a line of questic
    'In the practice of law, judicial estoppel (also known as estoppel by inc
]
```

In [63]:
```
# you must specify the padding width!
test_X = process_text(testcases, padding = PADDING_WIDTH)
```

In [65]:
```
predictions = model_2.predict(test_X)
```

In [66]:
```
for idx, result in enumerate(predictions):
    predict_id = result.argmax()
    sense_id = predict_id + 1    # sense_id starts from 1
```

```
    print(testcases[idx])
    print(f'-> Sense {sense_id} (prob={result[predict_id]:.2f}): {SENSE[sense
    print()
```

```
Green Beer Day (GBD) is a day-long party, where celebrants drink beer dyed gr
een with artificial coloring or natural processes.
-> Sense 1 (prob=0.93): a social event at which a group of people meet to tal
k, eat, drink, dance, etc.

When the siblings grew up, they held parties and introduced the tradition to
friends while in college, and the tradition began to spread.
-> Sense 1 (prob=0.90): a social event at which a group of people meet to tal
k, eat, drink, dance, etc.

Politicians from the two main parties tend to win elections when not confront
ed by strong challengers from their own party (in which cases their tradition
al opponents tend to win).
-> Sense 2 (prob=0.99): an organization of people with particular political b
eliefs

After the general election on 22 March 1992, five parties (Rassadorn, Justice
Unity, Social Action, Thai Citizen, Chart Thai) designated Suchinda as the pr
ime minister.
-> Sense 2 (prob=0.98): an organization of people with particular political b
eliefs

Typically, a party has the right to object in court to a line of questioning
or at the introduction of a particular piece of evidence.
-> Sense 3 (prob=0.73): a single entity which can be identified as one for th
e purposes of the law

In the practice of law, judicial estoppel (also known as estoppel by inconsis
tent positions) is an estoppel that precludes a party from taking a position
in a case that is contrary to a position it has taken in earlier legal procee
dings.
-> Sense 3 (prob=0.92): a single entity which can be identified as one for th
e purposes of the law
```

Yet again, the label might not be 100% correct, but it still should look reasonable.

IV. Evaluation

We have our model built! It's time to see how good it is on the testing dataset.

Get the predictions from the final model and examine the results.

In [67]:

```
with open(os.path.join('data', 'party.test.txt'), 'r') as f:
    data = f.read().strip().split('\n')

# this dict maps sentence_id to the sentence itself
test_data = { sent_id: text for sent_id, text in [line.split('\t', 1)
                                                  for line in data] }
```

In [78]:

```
for idx, (sent_id, sentence) in enumerate(test_data.items()):
    if idx > 3: break

    print(f'{sent_id}: {sentence}')
```

```
1638: Patent ambiguity is that ambiguity which is apparent on the face of an
instrument to any one perusing it, even if unacquainted with the circumstance
s of the parties.
1639: Smith played at parties, juke joints, and fish fries.
1640: Turkey has a multi-party system, with two or three strong parties and o
ften a fourth party that is electorally successful.
```

```
1641: The Christian Liberation Movement ( or simply MCL) is a Cuban dissident
party advocating political change in Cuba.
```

[TODO] Get the labels of testing data.

Try to reserve the sentence id, because you will need it while requesting your accuracy.

Recommended format of `final_predictions` :

```
{ sent_id: sense_id }
```

In [ ]:

```
final_predictions = {}

# [TODO]
...
```

In [72]:

```
for idx, (sent_id, pred) in enumerate(final_predictions.items()):
    if idx > 5: break

    print(test_data[sent_id])
    print(f'-> Sense {pred}: {SENSE[pred]}')
    print()
```

```
Patent ambiguity is that ambiguity which is apparent on the face of an instru
ment to any one perusing it, even if unacquainted with the circumstances of t
he parties.
-> Sense 3: a single entity which can be identified as one for the purposes o
f the law

Smith played at parties, juke joints, and fish fries.
-> Sense 1: a social event at which a group of people meet to talk, eat, drin
k, dance, etc.

Turkey has a multi-party system, with two or three strong parties and often a
fourth party that is electorally successful.
-> Sense 2: an organization of people with particular political beliefs

The Christian Liberation Movement ( or simply MCL) is a Cuban dissident party
advocating political change in Cuba.
-> Sense 1: a social event at which a group of people meet to talk, eat, drin
k, dance, etc.

Greens Party () was a green liberal party in Turkey.
-> Sense 2: an organization of people with particular political beliefs

Under the Constitution of North Korea, all citizens 17 and older, regardless
of party affiliation, political views, or religion, are eligible to be electe
d to the legislature and vote in elections.
-> Sense 2: an organization of people with particular political beliefs
```

Get your accuracy

Send your predictions in json format to our server, and we will calculate the accuracy for you.

The format should be

```
{ sentence_id: sense_id }
```

Example,

```
{
    1001: 1,
```

```
    1002: 1,
    ...
}
```

In [73]:
```
import json
import requests
```

In [74]:
```
data = json.dumps(final_predictions)
ret = requests.post('http://jedi.nlplab.cc:4500/check',
                    json = { 'data': data }
                   )
```

In [79]:
```
if not ret.ok:
    print('Something wrong :o')
print(ret.json())
```

```
{'accuracy': 0.8142857142857143}
```

REQUIREMENT

Your accuracy should be higher than 0.70 to get the full points.

But do note that your assignment is mostly scored on your implementation, not just on the accuracy.
So even if you brute-forcely attack our server and get 100% accuracy, you still can't get your points if your code doesn't make sense to TA.

TA's note

Congratuation! You've finished the assignment this week.
Don't forget to **[make an appoiment with TA](https://docs.google.com/spreadsheets/d/1QGeYl5dsD9sFO9SYg4DlKk-xr-yGjRDOOLKZqCLDv2E/edit#gid=1902646609) to demo/explain your implementation before 11/18 15:30** .
Also make sure you submit your {student_id}.ipynb to eeclass.

Please note that we will announce our final project on 11/18. Again, **we strongly suggest you join and listen** .
We will have 2 Ph.D. students introduce the selected topics in class and give you some guidelines about how to approach your project.
Also, we will have a team-matching session at the end of the class, in which you may want to participate to find teammates.

In [ ]:
```
```