Deep learning for NLP

Language modeling with RNNs

Benoit Favre

23 Feb 2017

1 Introduction

Language modeling is a core component of most natural language processing systems. It consists in, given a history, predicting the next symbol in a sequence. Formally, we try to estimate the probability of a sequence of words to come from the same distribution as a training corpus.

$$P(w_0, \dots w_n) = P(w_0)P(w_1|w_0) \dots P(w_n|w_0 \dots w_{n-1}) \tag{1}$$

Where $w_0 \dots w_n$ are words. These probabilities can be estimated with a k-gram language model with horizon-k Markovian assumption:

$$P(w_i|w_0 \dots w_{i-1}) = p(w_i|w_{i-k+1}, \dots w_{i-1}) \tag{2}$$

In which we ignore the history after $k-1$ words. This model is easy to estimate but it is lacking in term of modeling long-term dependencies.

Recurrent neural networks (RNNs) make strive to model the full history from the beginning of the sequence by estimating that probability in the following way:

$$P(w_i|w_0 \dots w_{i-1}) = p(w_i|h_i), \quad h_i = f(w_{i-1}, h_{i-1}) \tag{3}$$

2 Character language model

In this tutorial, you will build a simple language model which predicts a character given the previous character and a hidden state. The model will be trained to predict a sentence character by character, from the start.

The dataset is a list of people extracted from `http://www.nndb.com/`. Each line contains the person's name, her job, a summary of what this person is known for, her birth date, and death date (if it applies).

```
Donald Sutherland;Actor;The Dirty Dozen;17-Jul-1935;-
Amy Yasbeck;Actor;Casey Davenport on Wings;12-Sep-1963;-
George Lucas;Film Director;Creator of Star Wars;14-May-1944;-
```

The objective is to make a language model that will predict a person's description character by character.

2.1 Data layout

The RNN classifier will predict the next character given the previous character. So let's consider the 6-character sentence "Benoit". The first input is `<start>` for which we must predict the character 'B'. Then given 'B' we must predict 'e' and so on. At the end, the last character to predict is `<eos>` (for end of sentence).

	0	1	2	3	4	5	6
Input	<start>	B	e	n	o	i	t
Output	B	e	n	o	i	t	<eos>

Keras (any many deep learning toolkits) requires to unroll the network for several time steps to optimize computations and minimize CPU-GPU communication. We will use Keras' stateless recurrent units that start with a hidden state of 0 before predicting the first symbol of an unrolled sequence, and we will consider this to be the beginning of a sentence, and therefore predict the first character of the sentence given the symbol `<start>`.

For unrolling to be efficient, Keras enforces that all sequences are the same size, so we have to pad the sequences to account for the various length they can have. Since some sentences can be very long, we will decide on a maximum length, and trim longer sequences and pad shorter sequences with the `<eos>` symbol. For instance, the example padded at length 11 would look like that.

	0	1	2	3	4	5	6	7	8	9
Input	<start>	B	e	n	o	i	t	<eos>	<eos>	<eos>
Output	B	e	n	o	i	t	<eos>	<eos>	<eos>	<eos>

The model will use some capacity to model this phenomenon but RNNs are pretty good at figuring out that when a given symbol is seen they should output it for the rest of the sequence. Keras can automate this process, but instead of manipulating character strings, it needs to deal with numbers. So the first step is to load the dataset sentence by sentence, as a sequence of characters, and prefix each sequence with a `<start>` symbol. The characters should be mapped to integers with a dictionary and you should also keep track of a reversed dictionary to be able to display the character that corresponds to a given integer when generating text. The following python function loads a text file in this fashion and returns an array of integer sequences. Note that `<eos>` needs to have an id of 0.

```
def load_text(filename):
    vocab = collections.defaultdict(lambda: len(vocab))
    vocab['<eos>'] = 0
    text = []
    with open(filename) as fp:
        for line in fp:
            text.append([vocab['<start>']] + [vocab[char] for
↪   char in line.strip()])
```

```
    rev_vocab = {i: char for char, i in vocab.items()}
    return text, vocab, rev_vocab
```

You can pad and trim this array of sequences by calling the `pad_sequences` function from the module `keras.preprocessing.sequence`. Make sure to check the Keras documentation for this function to pad after the sequence. Once the sentences are padded, you can create numpy arrays in which to put the final training data. We will use embeddings for the input so the shape of the numpy array is the same as the padded sequences (num sentences, unrolled steps). For the output, we will use the sparse categorical loss which expects a 3-dimensional array of shape (num sentences, unrolled steps, 1). It is counter intuitive but that's to be consistent with losses operating on dense representations. The output array must be filled with the next character of the input at the same location, and must end in the `<eos>` token. The following function does all the work.

```
def vectorize(sequences, unroll, size):
    sequences = pad_sequences(text, unroll + 1, np.int32, 'post',
↪   'post', 0)

    X_train = np.zeros((len(sequences), unroll), dtype=np.int32)
    Y_train = np.zeros((len(sequences), unroll, 1),
↪   dtype=np.int32)

    for i in range(len(sequences)):
        for j in range(unroll):
            X_train[i,j] = sequences[i,j]
            Y_train[i,j,0] = sequences[i,j + 1]

    return X_train, Y_train
```

2.2 The keras model

The model contains an embedding layer which converts the n characters that are represented in the input to a lower dimension vector, setup to accept an unrolled input. Then, a recurrent layer is added which inputs the embedding and outputs its hidden state at each timestep (thanks to the `return_sequences` flag). GRU or LSTM recurrent layers will do a good job. On top of the recurrent unit, the model contains a dense layer with a softmax activation to produce the correct number of outputs (one per character). We have vectorized our data for the `sparse_categorical_crossentropy` loss function, so the model must be compiled with that loss. Here is a function that would create and compile such a model for a given `hidden` RNN state size, an `embed` embedding size, a `size` number of characters and `unroll` steps unrolled.

```
def make_model(hidden, embed, size, unroll):
    model = Sequential()
```

```
    model.add(Embedding(output_dim=embed, input_dim=size,
↪   input_length=unroll))
    model.add(GRU(hidden, return_sequences=True))
    model.add(TimeDistributed(Dense(size)))
    model.add(Activation('softmax'))
    model.compile(loss='sparse_categorical_crossentropy',
↪   optimizer='Nadam')
    return model
```

2.3 Training

A first way of training the model would be to call `model.fit` on the training data for a few epochs.

```
model = make_model(hidden, embed, size, unroll)
model.fit(X_train, Y_train, batch_size=batch, nb_epoch=5,
↪   shuffle=True)
```

If you load the first 5,000 lines of `people.txt` and train 5 epochs[1], you should get a loss of about 1.7. Note that this loss is made artificially low by the padding symbols (you can change the unroll length to increase/lower it).

```
Epoch 1/5
5000/5000 [==============================] - 33s - loss: 2.6760
Epoch 2/5
5000/5000 [==============================] - 32s - loss: 2.0012
Epoch 3/5
5000/5000 [==============================] - 35s - loss: 1.8512
Epoch 4/5
5000/5000 [==============================] - 28s - loss: 1.7645
Epoch 5/5
5000/5000 [==============================] - 27s - loss: 1.7091
```

3 Generation

The fun with language models is to generate novel text. To do that, we can input the model with the `<start>` symbol and a hidden state of 0, and compute the probability of the next character for all characters. Then we will select a character in this distribution and use it as input for generating the next character, and repeat the process until we reach the `<eos>` symbol. For simplicity, we will first use the highest probability character at each time step, and once that works, we will look into sampling from the full distribution.

The model with have built in Keras is not designed for generation and is optimized for training. In particular, it assumes that the input is known over

[1] Parameters : $hidden = 64$, $batch = 8$, $unroll = 64$, $embed = 16$, $size = 126$

all the unrolled time steps. One way to address that problem would be to setup another model initialized with the weights we have trained earlier. Instead, to keep things simple we will use the Keras model we have and repeatedly predict from it adding time steps each time until `<eos>` is generated. So the input will look like this:

1. The first symbol is `<start>` followed by arbitrary input:
   `<start> <eos> <eos> <eos>...`
2. then we use the first prediction `c1` as input for the second step:
   `<start> c1 <eos> <eos>...`
3. then we use the second prediction `c2` as input for the third step:
   `<start> c1 c2 <eos>...`
4. then we use the third prediction `c3` as input for the fourth step:
   `<start> c1 c2 c3...`

Note that under this model we can only generate sequences of length the number of unrolled steps. To go further, we would need stateful recurrent cells, which are available in Keras, but that's another story.

So we need to create a numpy array x of shape (1, unrolled steps) for the input. $x[0, 0]$ is set to the id of the `<start>` symbol. Then repeat until the number of unrolled steps is reached or the `<eos>` symbol is generated: use `model.predict` to get the score distribution for the next symbol[2], use `np.argmax(scores)` to get the most probable symbol, and use the `rev_vocab` dictionary to map the character index back to an actual character, and finally set the next input to the id of the predicted character.

```
def generate(model, unroll):
    x = np.zeros((1, unroll))
    selected = 1 # <start>
    x[0,0] = selected
    for i in range(unroll - 1):
        scores = model.predict(x, verbose=0)[0][i]
        selected = np.argmax(scores)
        if selected == 0:
            break
        print(rev_vocab[selected], end='')
        x[0,i + 1] = selected
    print()
```

Using the *argmax* is deterministic, so the model will always output its favorite character sequence. For instance, on the previously trained model, it generates:

[2]The model generates predictions for the whole unrolled input, so you need to pick the correct row.

```
<start>Jan Baller;Actor;Coriner of Corine Coriner of Corine...
```

To add some variability, we need to sample from the distribution instead of using the most probable character. Numpy provides the `np.random.choice` function which takes a `p` parameter for specifying a distribution and can be used to generate a random integer according to a distribution. The distribution is not very sharp so quite unlikely characters might be selected. In order to limit this phenomenon, people have applied a "temperature" factor to the probabilities by dividing them in log space by a constant between 0 and 1 and renormalizing with a softmax. Low temperature will generate conservative decisions (characters that the model really likes) and high temperatures lead to less conservative decisions.

```python
def sample(scores):
    scores = np.log(scores) / 0.7
    e = np.exp(scores)
    scores = e / np.sum(e)
    return np.random.choice(len(scores), 1, p=scores)[0]
```

With a temperature of 0.7, a sample of the generated sequences after 10 epochs might look like:

```
<start>Alan Britner;Compimat;President Ropelanist;28-Oct-1966;-
<start>Jofne A. Soma;Actor;Woph Howter;22-Oct-1943;-
<start>Roy Rind;Phlshels;President of Apc.;11-May-1945;-
<start>Lavi;Actor;Lidper on Stong on State;25-Sep-1957;-
<start>Robert M. Jame;Silger;Head of Hiller of Muthe;19-Mar-1942;-
<start>Doron Breiz;Actor;Amanan of the Stite;17-Sep-1944;-
<start>Dancelles Olen;Actor;A Tenter of Odano;15-Jan-1927;-
```

It is interesting to see that the model is good at capturing the repetition of semicolumn separated fields with correct semantics for each field. In addition, the model does a good job at generating job names such as "actor" or "president", and generating birth dates. Since most people in the database are still alive, it rarely generates death dates.

4 Extensions

The first extension is to train the model on much larger quantities of texts, such as all of Shakespeare's work[3], Wikipedia[4] or movie scripts[5]. A strong GPU and patience are recommended with larger datasets.

Another extension is to predict words instead of characters. The output layer of the model will be much larger, in general more than 100,000 words

[3] https://ocw.mit.edu/ans7870/6/6.006/s08/lecturenotes/files/t8.shakespeare.txt
[4] https://dumps.wikimedia.org/
[5] https://www.cs.cornell.edu/~cristian/Cornell_Movie-Dialogs_Corpus.html

(using the most frequent words and replacing infrequent ones with a `<unk>` token). For training to be fast enough, one needs to approximate the softmax layer, through methods such as Noice Contrastive Estimation (NCE) or sampled softmax which only compute the probability of a subset of words and update the model accordingly.