A Deep Learning Model to Learn the Hangman Game

Yan Zeng

Version 1.0, last updated on 2023-04-12

Abstract

We explain a deep learning model to play the Hangman game.

Contents

1 Formulation of the Problem

We formulate the Hangman game as a filtering/estimation problem. At each stage of a game, we observe two states: the guessed letters and the masked word. Our goal is to estimate the probability distribution of unrevealed letters in the original word. Using categorical cross entropy as the optimization criterion, we can assume the unrevealed letters have equal probabilities and we set up a deep neural network (DNN) model to estimate the locations of the unrevealed letters in the alphabet.

An example. Suppose the original word is `cutch`. Table 1 records one scenario of how the Hangman game could be played.

Table 1: One scenario of Hangman game play

stage	word	masked word	guessed letters	unrevealed letters
0	`cutch`	`_ _ _ _ _`	{}	{c, u, t, h}
1	`cutch`	`c _ _ c _`	{c}	{u, t, h}
2	`cutch`	`c _ _ c _`	{c, f}	{u, t, h}
3	`cutch`	`c _ t c _`	{c, f, t}	{u, h}
4	`cutch`	`c _ t c _`	{c, f, t, w}	{u, h}
5	`cutch`	`c _ t c _`	{c, f, t, w, j}	{u, h}
6	`cutch`	`c _ t c _`	{c, f, t, w, j, s}	{u, h}
7	`cutch`	`c _ t c _`	{c, f, t, w, j, s, r}	{u, h}
8	`cutch`	`c _ t c _`	{c, f, t, w, j, s, r, b}	{u, h}

Problem formulation. At each stage, the estimation problem is to estimate the (unobservable) state of `unrevealed letters` according to the (observable) states of `masked word` and `guessed letters`:

$$\texttt{masked word}, \texttt{guessed letters} \longrightarrow \texttt{unrevealed letters}$$

The state machine for masked word can be represented by a vector of integers: 1-26 for a-z and 27 for _. Table 2 records the evolution of `masked word` state machine in the scenario illustrated by Table 1.

Table 2: Evolution of the state machine for `masked word`

stage	masked word
0	[27, 27, 27, 27, 27]
1	[3, 27, 27, 3, 27]
2	[3, 27, 27, 3, 27]
3	[3, 27, 20, 3, 27]
4	[3, 27, 20, 3, 27]
5	[3, 27, 20, 3, 27]
6	[3, 27, 20, 3, 27]
7	[3, 27, 20, 3, 27]
8	[3, 27, 20, 3, 27]

The state machine for guessed letters can be represented by a binary vector of length 26, where the ith element corresponds to the $(i+1)$th element in the alphabet. 1 means the letter has been

used, and 0 means the letter has not been used. Table 3 records the evolution of `guessed letters` state machine in the scenario illustrated by Table 1.

Table 3: Evolution of the state machine for `guessed letters`

stage	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	**1**	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	0	0	**1**	0	0	**1**	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3	0	0	**1**	0	0	**1**	0	0	0	0	0	0	0	0	0	0	0	0	0	**1**	0	0	0	0	0	0
4	0	0	**1**	0	0	**1**	0	0	0	0	0	0	0	0	0	0	0	0	0	**1**	0	0	**1**	0	0	0
5	0	0	**1**	0	0	**1**	0	0	0	**1**	0	0	0	0	0	0	0	0	0	**1**	0	0	**1**	0	0	0
6	0	0	**1**	0	0	**1**	0	0	0	**1**	0	0	0	0	0	0	0	0	**1**	**1**	0	0	**1**	0	0	0
7	0	0	**1**	0	0	**1**	0	0	0	**1**	0	0	0	0	0	0	0	**1**	**1**	**1**	0	0	**1**	0	0	0
8	0	**1**	**1**	0	0	**1**	0	0	0	**1**	0	0	0	0	0	0	0	**1**	**1**	**1**	0	0	**1**	0	0	0

Finally, the state machine for the unrevealed letters are represented by a vector of length 26, where the ith element corresponds to the (unobservable) probability of the $(i+1)$th element in the alphabet being an unrevealed letter in the word. Table 4 records the evolution of probability distribution of the unrevealed letters in the scenario illustrated by Table 1.

Table 4: Evolution of the state machine for probability distribution of unrevealed letters

stage	0	1	2	3-6	7	8-18	19	20	21-25
0	0.0	0.0	**0.25**	0.0	**0.25**	0.0	**0.25**	**0.25**	0.0
1	0.0	0.0	0.00	0.0	**0.33**	0.0	**0.33**	**0.33**	0.0
2	0.0	0.0	0.00	0.0	**0.33**	0.0	**0.33**	**0.33**	0.0
3	0.0	0.0	0.00	0.0	**0.50**	0.0	0.00	**0.50**	0.0
4	0.0	0.0	0.00	0.0	**0.50**	0.0	0.00	**0.50**	0.0
5	0.0	0.0	0.00	0.0	**0.50**	0.0	0.00	**0.50**	0.0
6	0.0	0.0	0.00	0.0	**0.50**	0.0	0.00	**0.50**	0.0
7	0.0	0.0	0.00	0.0	**0.50**	0.0	0.00	**0.50**	0.0
8	0.0	0.0	0.00	0.0	**0.50**	0.0	0.00	**0.50**	0.0

Choice of the solution. The above problem formulation indicates a deep neural network (DNN) model could be a suitable solution. In particular,

• The data are path dependent and are inherently high dimensional.

• There are inherent patterns within a given word (even though some words in the training dictionary are just gibberish).

• Letter orders in a word may impact the best guess (`in_` may suggest `g` whereas `ni_` may suggest `n`).

2 Heuristics of the Architecture

In view of the problem characteristics analyzed above, we choose the following building blocks for our DNN model:

• Multilayer Perceptrons (MLPs). These are basic building blocks of a deep neural network. We'll use the `keras.layers.Dense` class for fully connected neurons.

• Convolutional layers and pooling layers. The within-word patterns hint at a usage of Convolutional Neural Networks (CNNs). We'll therefore use the `keras.layers.Conv1D` class and the `keras.layers.MaxPooling1D` class.

• Word embedding. This is a standard technique for natural language processing (NLP). We'll use the `keras.layers.Embedding` class.

• Long Short-Term Memory (LSTM). As letter orders could be useful to the best guess, we want our DNN model to be able to retain memories. We'll therefore use the `keras.layer.LSTM` class.

Architecture for masked word. The architecture for masked word state machine is shown in Figure 1. We have chosen `kernel_size` = 3 to capture patterns like `th`, `tch`, `ion`, etc.

Figure 1: Architecture for masked word

```
# CNN+LSTM state machine for masked words as 1x(max_word_size) arrays filled by letter codes:
# 1-26 for a-z, 27 for _, 0 for padding, a total of 28 dimensions. E.g. 'x_z' -> [0, ..., 0,
# 24, 27, 26]
masked_st = keras.models.Sequential(layers=[
    keras.layers.Input(shape=(self.max_word_size,)),
    keras.layers.Embedding(input_dim=28, output_dim=32,
                           mask_zero=True, input_length=self.max_word_size),
    keras.layers.Conv1D(filters=32, kernel_size=3, padding='same', activation='relu'),
    keras.layers.MaxPooling1D(pool_size=2),
    keras.layers.LSTM(units=100, dropout=0.2, recurrent_dropout=0.2),
    keras.layers.Dense(units=60, activation='relu'),
], name='MaskedWordModel')
```

Architecture for guessed letters. We use regular MLPs for guessed letters state machine (Figure 2). This means we are modeling guessed letters as a set, not as a sequence.

Figure 2: Architecture for guessed letters

```
# MLP state machine for guessed letters as a 1x26 array filled by 0's and 1's.
# E.g. ['a', 'c'] -> [1, 0, 1, 0, ..., 0]
guessed_st = keras.models.Sequential(layers=[
    keras.layers.Input(shape=(26,)),
    keras.layers.Dense(units=40, activation='relu'),
    keras.layers.Dense(units=40, activation='relu'),
], name='GuessedLettersModel')
```

Architecture of the final model. The final model is just stacking the concatenation of guessed letters model and masked word model on top of two layers of fully connected neurons (Figure 3). Since this is a multiclass classification problem, the out layer has 26 neurons with `softmax` as the activation function.

To compile the model, we choose `categorical_crossentropy` as the loss function and the Adam algorithm as the stochastic gradient descent method.

Figure 3: Architecture for the final model

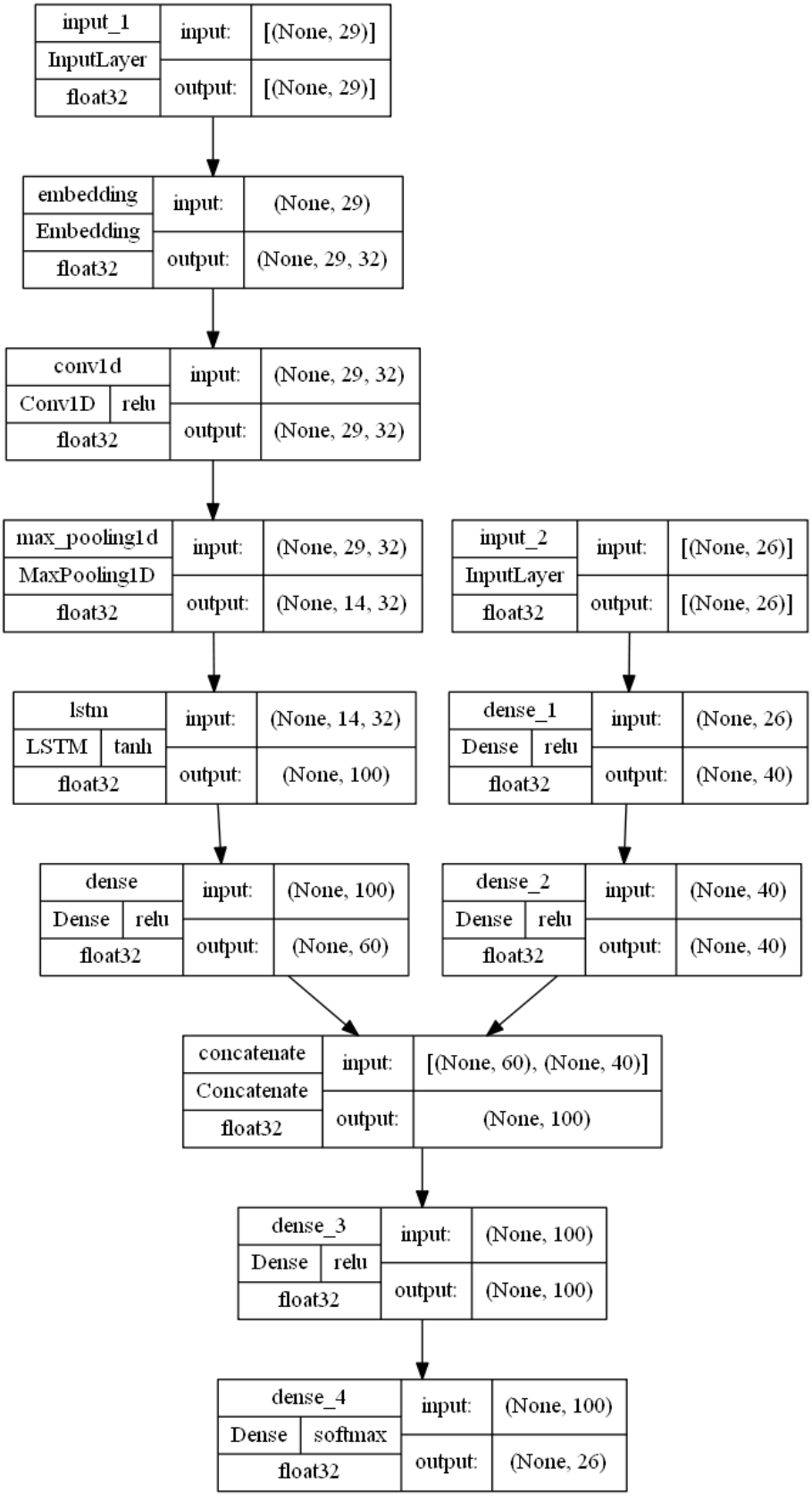
```
# final DNN Model
x = keras.layers.Concatenate()([masked_st.output, guessed_st.output])
x = keras.layers.Dense(units=100, activation='relu')(x)
x = keras.layers.Dense(units=26, activation='softmax')(x)
self.model = keras.models.Model(inputs=[masked_st.input, guessed_st.input],
                                 outputs=x, name='DNNModel')
```

The model is summarized in Figure 4, with 78,706 parameters. The full architecture is captured in Figure 5.

Figure 4: Architecture for final model

```
Model: "DNNModel"
__________________________________________________________________________________________________
 Layer (type)                   Output Shape         Param #     Connected to
==================================================================================================
 input_1 (InputLayer)           [(None, 29)]         0           []

 embedding (Embedding)          (None, 29, 32)       896         ['input_1[0][0]']

 conv1d (Conv1D)                (None, 29, 32)       3104        ['embedding[0][0]']

 max_pooling1d (MaxPooling1D)   (None, 14, 32)       0           ['conv1d[0][0]']

 input_2 (InputLayer)           [(None, 26)]         0           []

 lstm (LSTM)                    (None, 100)          53200       ['max_pooling1d[0][0]']

 dense_1 (Dense)                (None, 40)           1080        ['input_2[0][0]']

 dense (Dense)                  (None, 60)           6060        ['lstm[0][0]']

 dense_2 (Dense)                (None, 40)           1640        ['dense_1[0][0]']

 concatenate (Concatenate)      (None, 100)          0           ['dense[0][0]',
                                                                  'dense_2[0][0]']

 dense_3 (Dense)                (None, 100)          10100       ['concatenate[0][0]']

 dense_4 (Dense)                (None, 26)           2626        ['dense_3[0][0]']

==================================================================================================
Total params: 78,706
Trainable params: 78,706
Non-trainable params: 0
__________________________________________________________________________________________________
```

Figure 5: Architecture of the Deep Neural Networks model



3 Discussion for Performance Improvement

Based on 1,000 tests, the in-sample accuracy is 34% with a standard deviation of 1.5%.

The following could be done to improve model performance.

Try bidirectional LSTM network. If letter orders in masked words give us additional information, it's common to use bidirectional LSTM (for recurrent neural networks) instead of plain vanilla LSTM. We did not use bidirectional layers simply because it significantly increased the number of parameters and the training became very slow.

Increase the kernel size in CNN. We set `kernel_size` = 3 to capture patterns like `th`, `tch`, and `ion`. We could try bigger sizes to capture patterns like `tion`, `ality`, etc..

Increase the number of neurons. For `Dense` layers, it's common to start with 100 neurons. We have chosen smaller numbers (40, 60) to speed up the training. Due to the high dimensional nature of the problem, more neurons will obviously be helpful.

To conclude, there is always this debate of "wide vs. deep" in neural networks design and there are many interesting building blocks other than what we have used so far to express our intuitions about the Hangman game. Much to be done with more computational resources.