Venelin Valkov Follow
Adventures in Artificial Intelligence
May 25 · 9 min read

Making a Predictive Keyboard using Recurrent Neural Networks — TensorFlow for Hackers (Part V)

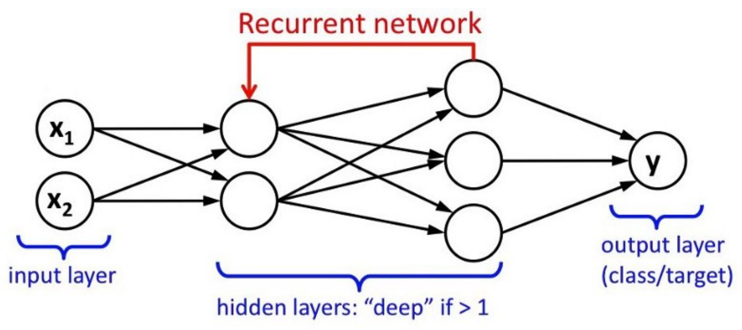
Can you predict what someone is typing?

Welcome to another part of the series. This time we will build a model that predicts the next word (a character actually) based on a few of the previous. We will extend it a bit by asking it for 5 suggestions instead of only 1. Similar models are widely used today. You might be using one without even knowing! Here's one example:

Recurrent Neural Networks

Our weapon of choice for this task will be Recurrent Neural Networks (RNNs). But why? What's wrong with the type of networks we've used so far? Nothing! Yet, they lack something that proves to be quite useful in practice — memory!

In short, `RNN` models provide a way to not only examine the current input but the one that was provided one step back, as well. If we turn that around, we can say that the decision reached at time step t-1 directly affects the future at step t.



source: Leonardo Araujo dos Santos's Artificial Intelligence

It seems like a waste to throw out the memory of what you've seen so far and start from scratch every time. That's what other types of Neural Networks do. Let's end this madness!

Definition

RNNs define a recurrence relation over time steps which is given by:

$$S_t = f(S_{t-1} * W_{rec} + X_t * W_x)$$

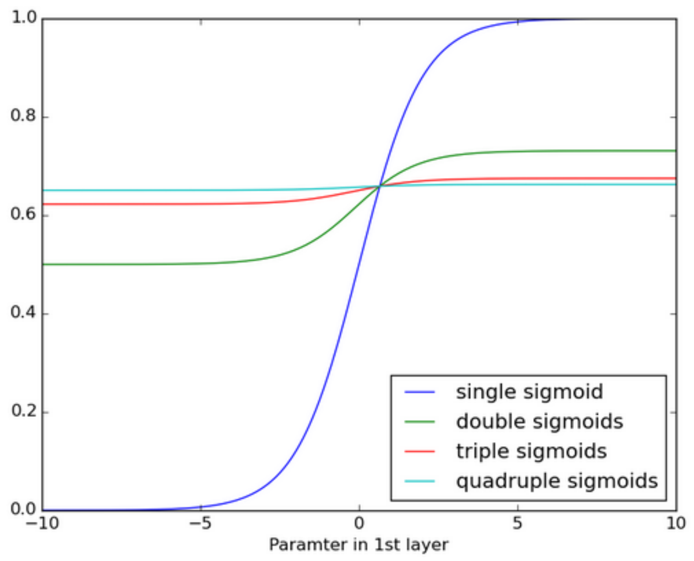
Where St is the state at time step t, Xt an exogenous input at time t, Wrec and Wx are weights parameters. The feedback loops gives memory to the model because it can remember information between time steps.

`RNNs` can compute the current state St from the current input Xt and previous state St−1 or predict the next state from St+1 from the current St and current input Xt. Concretely, we will pass a sequence of 40 characters and ask the model to predict the next one. We will append the new character and drop the first one and predict again. This will continue until we complete a whole word.

LSTMs

Two major problems torment the `RNNs` —vanishing and exploding gradients. In traditional `RNNs` the gradient signal can be multiplied a large number of times by the weight matrix. Thus, the magnitude of the weights of the transition matrix can play an important role.

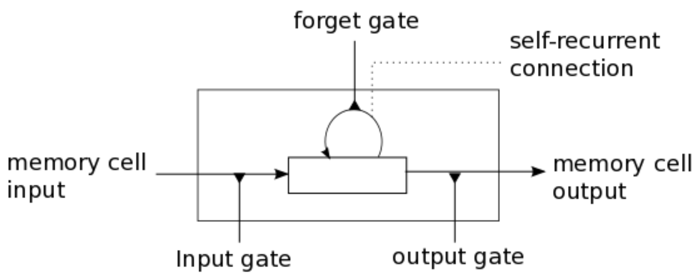
If the weights in the matrix are small, the gradient signal becomes smaller at every training step, thus making learning very slow or completely stops it. This is called vanishing gradient. Let's have a look at applying the sigmoid function multiple times, thus simulating the effect of vanishing gradient:



source: deeplearning4j.org

Conversely, the exploding gradient refers to the weights in this matrix being so large that it can cause learning to diverge.

`LSTM` model is a special kind of `RNN` that learns long-term dependencies. It introduces new structure—the memory cell that is composed of four elements: input, forget and output gates and a neuron that connects to itself:



source: deeplearning.net

`LSTMs` fight the gradient vanishing problem by preserving the error that can be backpropagated through time and layers. By maintaining a more constant error, they allow for learning long-term dependencies.

On another hand, exploding is controlled with gradient clipping, that is the gradient is not allowed to go above some predefined value.

Setup

Let's properly seed our random number generator and import all required modules:

```
import numpy as np
np.random.seed(42)
import tensorflow as tf
tf.set_random_seed(42)
from keras.models import Sequential, load_model
from keras.layers import Dense, Activation
from keras.layers import LSTM, Dropout
from keras.layers import TimeDistributed
from keras.layers.core import Dense, Activation, Dropout,
RepeatVector
from keras.optimizers import RMSprop
import matplotlib.pyplot as plt
import pickle
import sys
import heapq
import seaborn as sns
from pylab import rcParams

%matplotlib inline

sns.set(style='whitegrid', palette='muted', font_scale=1.5)

rcParams['figure.figsize'] = 12, 5
```

This code works with `TensorFlow` 1.1 and `Keras` 2.

Loading the data

We will use Friedrich Nietzsche's *Beyond Good and Evil* as a training corpus for our model. The text is not that large and our model can be trained relatively fast using a modest `GPU` . Let's use the lowercase version of it:

```
path = 'nietzsche.txt'
text = open(path).read().lower()
print('corpus length:', len(text))
```

>

```
corpus length: 600893
```

Preprocessing

Let's find all unique chars in the corpus and create char to index and index to char maps:

```
chars = sorted(list(set(text)))
char_indices = dict((c, i) for i, c in enumerate(chars))
indices_char = dict((i, c) for i, c in enumerate(chars))

print(f'unique chars: {len(chars)}')
```

>

```
unique chars: 57
```

Next, let's cut the corpus into chunks of `40` characters, spacing the sequences by `3` characters. Additionally, we will store the next character (the one we need to predict) for every sequence:

```
SEQUENCE_LENGTH = 40
step = 3
sentences = []
next_chars = []
for i in range(0, len(text) - SEQUENCE_LENGTH, step):
    sentences.append(text[i: i + SEQUENCE_LENGTH])
    next_chars.append(text[i + SEQUENCE_LENGTH])
print(f'num training examples: {len(sentences)}')
```

>

```
num training examples: 200285
```

It is time for generating our features and labels. We will use the previously generated sequences and characters that need to be predicted to create one-hot encoded vectors using the `char_indices` map:

```
X = np.zeros((len(sentences), SEQUENCE_LENGTH, len(chars)),
dtype=np.bool)
y = np.zeros((len(sentences), len(chars)), dtype=np.bool)
for i, sentence in enumerate(sentences):
    for t, char in enumerate(sentence):
        X[i, t, char_indices[char]] = 1
    y[i, char_indices[next_chars[i]]] = 1
```

Let’s have a look at a single training sequence:

```
sentences[100]
```

>

```
've been unskilled and unseemly methods f'
```

The character that needs to be predicted for it is:

```
next_chars[100]
```

>

```
'o'
```

The encoded (one-hot) data looks like this:

```
X[0][0]
```

>

```
array([False, False, False, False, False, False, False,
False, False,
       False, False, False, False, False, False, False,
False, False,
       False, False, False, False, False, False, False,
False, False,
       False, False, False, False, False, False, False,
False, False,
       False, False, False, False, False, False,  True,
False, False,
       False, False, False, False, False, False, False,
False, False,
       False, False, False], dtype=bool)
```

>

```
y[0]
```

>

```
array([False, False, False, False, False, False, False,
False, False,
       False, False, False, False, False, False, False,
False, False,
       False, False, False, False, False, False, False,
False, False,
       False, False, False, False, False, False, False,
False, False,
       False, False, False, False,  True, False, False,
False, False,
       False, False, False, False, False, False, False,
False, False,
       False, False, False], dtype=bool)
```

And for the dimensions:

```
X.shape
```

>

```
(200285, 40, 57)
```

>

```
y.shape
```

>

```
(200285, 57)
```

We have `200285` training examples, each sequence has length of `40` with `57` unique chars.

Building the model

The model we're going to train is pretty straight forward. Single `LSTM` layer with `128` neurons which accepts input of shape (`40` —the length of a sequence, `57` —the number of unique characters in our dataset). A fully connected layer (for our output) is added after that. It has `57` neurons and softmax for activation function:

```
model = Sequential()
model.add(LSTM(128, input_shape=(SEQUENCE_LENGTH,
len(chars))))
model.add(Dense(len(chars)))
model.add(Activation('softmax'))
```

Training

Our model is trained for `20` epochs using `RMSProp` optimizer and uses `5%` of the data for validation:

```
optimizer = RMSprop(lr=0.01)
model.compile(loss='categorical_crossentropy',
optimizer=optimizer, metrics=['accuracy'])


history = model.fit(X, y, validation_split=0.05,
batch_size=128, epochs=20, shuffle=True).history
```

Saving

It took a lot of time to train our model. Let's save our progress:

```
model.save('keras_model.h5')
pickle.dump(history, open("history.p", "wb"))
```

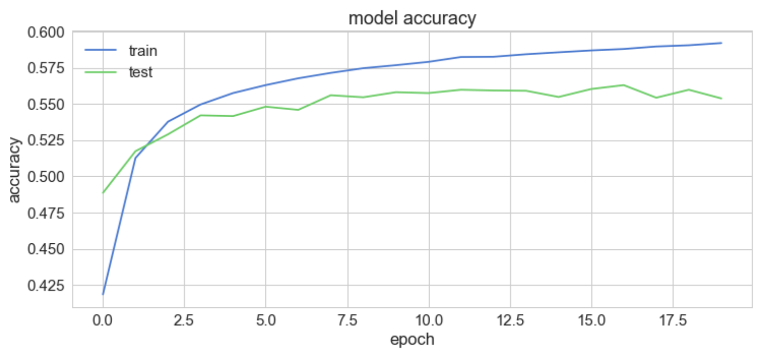
And load it back, just to make sure it works:

```
model = load_model('keras_model.h5')
history = pickle.load(open("history.p", "rb"))
```

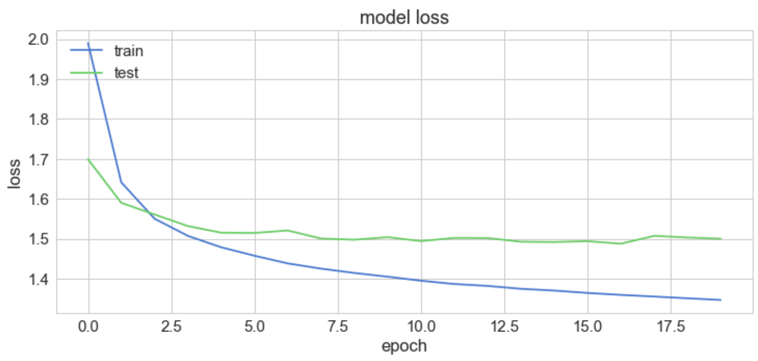
Evaluation

Let's have a look at how our accuracy and loss change over training epochs:

```
plt.plot(history['acc'])
plt.plot(history['val_acc'])
plt.title('model accuracy')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper left');
```



```
plt.plot(history['loss'])
plt.plot(history['val_loss'])
plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper left');
```



Let’s put our model to the test

Finally, it is time to predict some word completions using our model! First, we need some helper functions. Let’s start by preparing our input text:

```
def prepare_input(text):
    x = np.zeros((1, SEQUENCE_LENGTH, len(chars)))

    for t, char in enumerate(text):
        x[0, t, char_indices[char]] = 1.

    return x
```

Remember that our sequences must be `40` characters long. So we make a tensor with shape `(1, 40, 57)`, initialized with zeros. Then, a value of `1` is placed for each character in the passed text. We must not forget to use the lowercase version of the text:

```
prepare_input("This is an example of input for our
LSTM".lower())
```

>

```
array([[[ 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., ...,  0.,  0.,  0.],
        [ 0.,  0.,  0., ...,  0.,  0.,  0.]]])
```

Next up, the sample function:

```
def sample(preds, top_n=3):
    preds = np.asarray(preds).astype('float64')
    preds = np.log(preds)
    exp_preds = np.exp(preds)
    preds = exp_preds / np.sum(exp_preds)

    return heapq.nlargest(top_n, range(len(preds)),
preds.take)
```

This function allows us to ask our model what are the next `n` most probable characters. Isn't that heap just cool?

Now for the prediction functions themselves:

```
def predict_completion(text):
    original_text = text
    generated = text
    completion = ''
    while True:
        x = prepare_input(text)
        preds = model.predict(x, verbose=0)[0]
        next_index = sample(preds, top_n=1)[0]
        next_char = indices_char[next_index]


        text = text[1:] + next_char
        completion += next_char

        if len(original_text + completion) + 2 >
len(original_text) and next_char == ' ':
            return completion
```

This function predicts next character until space is predicted (you can extend that to punctuation symbols, right?). It does so by repeatedly preparing input, asking our model for predictions and sampling from them.

The final piece of the puzzle— `predict_completions` wraps everything and allow us to predict multiple completions:

```
def predict_completions(text, n=3):
    x = prepare_input(text)
    preds = model.predict(x, verbose=0)[0]
    next_indices = sample(preds, n)
    return [indices_char[idx] + predict_completion(text[1:]
+ indices_char[idx]) for idx in next_indices]
```

Let's use sequences `of` 40 characters that we will use as seed for our completions. All of these are quotes from Friedrich Nietzsche himself:

```
quotes = [
    "It is not a lack of love, but a lack of friendship that
makes unhappy marriages.",
    "That which does not kill us makes us stronger.",
    "I'm not upset that you lied to me, I'm upset that from
now on I can't believe you.",
    "And those who were seen dancing were thought to be
insane by those who could not hear the music.",
    "It is hard enough to remember my opinions, without also
remembering my reasons for them!"
]
```

>

```
for q in quotes:
    seq = q[:40].lower()
    print(seq)
    print(predict_completions(seq, 5))
    print()
```

>

```
it is not a lack of love, but a lack of
['the ', 'an ', 'such ', 'man ', 'present, ']


that which does not kill us makes us str
['ength ', 'uggle ', 'ong ', 'ange ', 'ive ']


i'm not upset that you lied to me, i'm u
['nder ', 'pon ', 'ses ', 't ', 'uder ']


and those who were seen dancing were tho
['se ', 're ', 'ugh ', ' servated ', 't ']


it is hard enough to remember my opinion
[' of ', 's ', ', ', '\nof ', 'ed ']
```

Apart from the fact that the completions look like proper words (remember, we are training our model on characters, not words), they look pretty reasonable as well! Perhaps better model and/or more training will provide even better results?

Conclusion

We've built a model using just a few lines of code in `Keras` that performs reasonably well after just 20 training epochs. Can you try it with your own text? Why not predict whole sentences? Will it work that well in other languages?

References

Recurrent Nets in TensorFlow

The Unreasonable Effectiveness of Recurrent Neural Networks

Understanding LSTM Networks

How to implement RNN in Python

LSTM Networks for Sentiment Analysis

cs231n — Recurrent Neural Networks

. . .

More from TensorFlow for Hackers series:

1. TensorFlow Basics
2. Building a Simple Neural Network
3. Building a Cat Detector using Convolutional Neural Networks
4. Creating a Neural Network from Scratch
5. **Making a Predictive Keyboard using Recurrent Neural Networks**
6. Human Activity Recognition using LSTMs on Android
7. Credit Card Fraud Detection using Autoencoders in Keras

. . .

Originally published at curiousily.com on May 25, 2017.