Understanding what RNN learns: Part 2

Vishnu Sharma
Jun 20, 2018 · 9 min read

This post is next part of blog post at Medium (Part 1) and notebook at Part 1 notebook. Please make sure that you have gone through them as I will be using some concept from there.

In this part, I will focus more on embeddings as the concept seems obscure to beginners.

In last part, we have seen that we can use embeddings to learn representation for input. This concept will get clearer in this part.

Setup:
In this part, our aim is to pedict whether a given sequence of numbers contains ‘4’ or not. This is a classification problem. You can draw parallels with a problem where your aim is to find whether the sentence contains the word ‘not’ in it(a very basic form of sentiment analysis).

Let’s get started

In:

```
# Import modules
import numpy as np
from keras.models import Model
from keras.layers import Input, SimpleRNN, Activation
from keras.layers import Embedding
from keras.optimizers import Adam

Using TensorFlow backend.
```

In :

```
# Data and model parameters
seq_len = 3   #Length of each sequence
rnn_size = 1  #Output shape of RNN
input_size = 10000 #Numbers of instances

all_feat = np.random.randint(low=0, high=10, size=(input_size,3))
all_label = np.apply_along_axis(func1d=lambda x: int(np.any(x==4)),
axis=1, arr=all_feat)

print('\nInput Sample:\n', all_feat[:5])
print('\nOutput Sample:\n',all_label[:5])
```

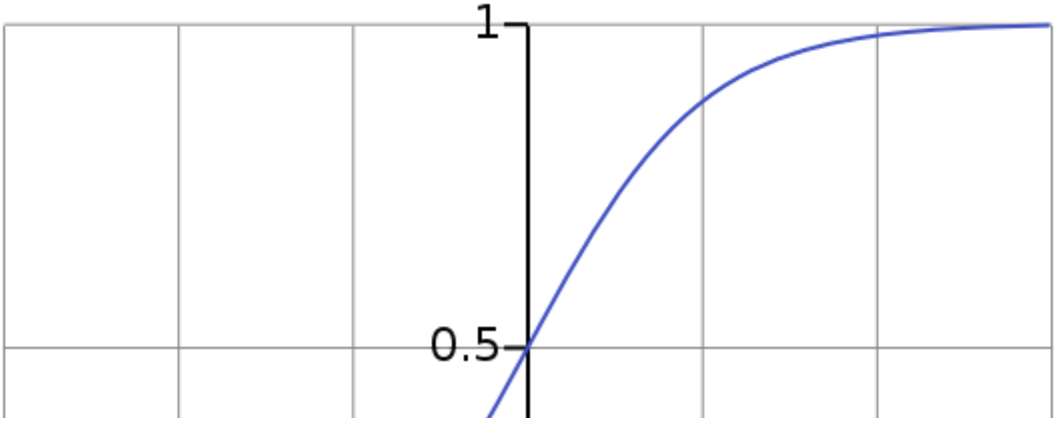
Out:

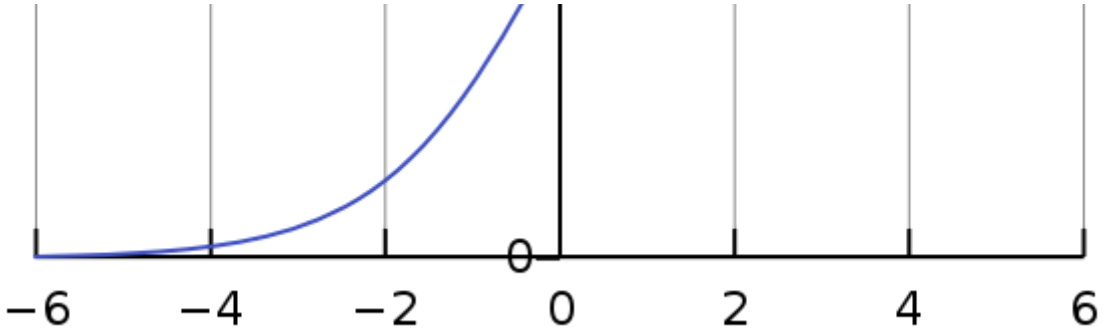
```
Input Sample:
 [[8 8 6]
 [8 5 5]
 [7 1 4]
 [7 1 0]
 [7 8 5]]

Output Sample:
 [0 0 1 0 0]
```

In this model, we are again going to choose embedding size=10, as we wish to use embeddings as a replacement to one-hot encoding.

But we can't directly use previous model(which solves a regression problem). This time we want to predict probability. So after our RNN, we'll use a sigmoid activation.

$$f(x) = \frac{1}{1 + e^{-x}}$$





Sigmoid Function

As you can see, sigmoid squashes output to a number between 0–1, such that higher the number, closer it is to 1. Similary, lower the number,closer it is to 0.

We will use binary crossentropy as the loss, which is generally used for classification problems.

. . .

Classification

In:

```
input_1 = Input(shape=(3,), name='Input_Layer')
x = Embedding(input_dim=10, output_dim=10, name='Embedding_Layer')
(input_1)
x = SimpleRNN(rnn_size, activation='linear', name='RNN_Layer')(x)
y = Activation('sigmoid', name='Activation_Layer')(x)

model = Model(inputs=input_1, outputs=y)

print(model.summary())
```

Out:

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
Input_Layer (InputLayer)     (None, 3)                 0
_________________________________________________________________
Embedding_Layer (Embedding)  (None, 3, 10)             100
_________________________________________________________________
RNN_Layer (SimpleRNN)        (None, 1)                 12
_________________________________________________________________
```

```
Activation_Layer (Activation (None, 1)                 0
=================================================================
Total params: 112.0
Trainable params: 112.0
Non-trainable params: 0.0
_________________________________________________________________
```

In :

```
model.compile(optimizer=Adam(0.02), loss='binary_crossentropy',
metrics=['acc'])
history = model.fit(x=all_feat, y=all_label, batch_size=8, epochs=4,
validation_split=0.2, verbose=1)
```

Out:

```
Train on 8000 samples, validate on 2000 samples
Epoch 1/4
8000/8000 [==============================] - 4s - loss: 0.0106 -
acc: 0.9956 - val_loss: 6.5674e-05 - val_acc: 1.0000
Epoch 2/4
8000/8000 [==============================] - 4s - loss: 3.2502e-05 -
acc: 1.0000 - val_loss: 1.6959e-05 - val_acc: 1.0000
Epoch 3/4
8000/8000 [==============================] - 4s - loss: 1.0502e-05 -
acc: 1.0000 - val_loss: 6.8796e-06 - val_acc: 1.0000
Epoch 4/4
8000/8000 [==============================] - 4s - loss: 4.6197e-06 -
acc: 1.0000 - val_loss: 3.3005e-06 - val_acc: 1.0000
```

In :

```
print('\nInput features: \n', all_feat[-10:,:])
print('\nLabels: \n', all_label[-10:])
print('\nPredictions: \n', model.predict(all_feat[-10:,:]))
```

Out:

```
Input features:
 [[5 6 9]
 [9 9 6]
 [5 8 4]
```

```
 [5 8 1]
 [6 8 1]
 [9 2 7]
 [6 0 0]
 [9 7 8]
 [8 7 9]
 [0 2 7]]

Labels:
 [0 0 1 0 0 0 0 0 0 0]

Predictions:
 [[  2.12621512e-06]
 [  2.11419456e-06]
 [  9.99977827e-01]
 [  1.95994267e-06]
 [  2.24871815e-06]
 [  2.05333026e-06]
 [  2.08695269e-06]
 [  1.90066737e-06]
 [  1.92356879e-06]
 [  1.98162775e-06]]
```

In :

```
embd_layer = model.get_layer('Embedding_Layer')
embd_mats = embd_layer.get_weights()

wgt_layer = model.get_layer('RNN_Layer')
wgts_mats = wgt_layer.get_weights()
```

Verify if we got expected shapes:

In :

```
print('Embedding W shape: ', embd_mats[0].shape)
print('W shape: ', wgts_mats[0].shape)
print('U shape: ', wgts_mats[1].shape)
print('b shape: ', wgts_mats[2].shape)
```

Out:

```
Embedding W shape:  (10, 10)
W shape:  (10, 1)
```

```
U shape:  (1, 1)
b shape:  (1,)
```

Before looking at the weights, let's see what are our expectations from them:

Reminder: **f** is linear.

We would expect that combination of embedding matrix and **W** has a high value for position corresponding to input=4. **U** should simply forward the output to next cell.

This way, if model sees 4 anywhere, output will become high. **U** will help carry this score to next cell.

Embedding weights matrix:

In:

```
embd_mats
```

Out:

```
[array([[-0.14044982,  0.21658441, -0.22791751,  0.21502978,
-0.19579233,
         0.18623219,  0.2239477 , -0.14737135,  0.1834836 ,
0.17848013],
        [-0.16119297,  0.15270022, -0.22112088,  0.1870423 ,
-0.21034855,
         0.22094215,  0.22617932, -0.19690502,  0.16201828,
0.16872635],
        [-0.20279713,  0.17129959, -0.18022029,  0.17888239,
-0.19878508,
         0.18757217,  0.18948197, -0.15959248,  0.21192279,
0.1793922 ],
        [-0.16744758,  0.18689834, -0.20116815,  0.16811925,
-0.21271777,
         0.16176091,  0.23074993, -0.22508025,  0.17736089,
0.19697021],
        [ 0.87358165, -0.92919093,  1.27148986, -0.97707129,
1.06918406,
        -1.25646746, -1.07512939,  1.01008677, -1.22653592,
```

```
-1.07279408],
        [-0.20252103,  0.13714807, -0.17961763,  0.20893431,
-0.21239041,
          0.19352351,  0.20141117, -0.19705266,  0.16560002,
0.20322703],
        [-0.20628117,  0.14427678, -0.21260698,  0.15537232,
-0.14697219,
          0.18460469,  0.15442781, -0.18720369,  0.2582643 ,
0.21449965],
        [-0.15330791,  0.21459399, -0.22178707,  0.13402544,
-0.13827942,
          0.2318393 ,  0.21597138, -0.20058507,  0.23698436,
0.19769941],
        [-0.15754659,  0.1512261 , -0.20155624,  0.22907215,
-0.17636189,
          0.18587922,  0.17176367, -0.19894339,  0.22978529,
0.18345623],
        [-0.13863607,  0.22530088, -0.15114433,  0.19402944,
-0.21190034,
          0.1904562 ,  0.22017194, -0.18479121,  0.2103454 ,
0.18715787]], dtype=float32)]
```

Doesn't give a very clear picture, right.
Let's check RNN weights

In :

```
wgts_mats
```

Out:

```
[array([[ 2.06557703],
        [-1.83462012],
        [ 1.95859563],
        [-2.21554518],
        [ 1.99453723],
        [-2.04525447],
        [-1.56783688],
        [ 1.71975875],
        [-1.72644722],
        [-1.72969031]], dtype=float32),
 array([[ 1.21534526]], dtype=float32),
 array([ 0.02591785], dtype=float32)]
```

U seems fine. **W** looks somewhat regular. Let's check our regular construct:

In :

```
print('\n W_embd * W + b: \n', np.matmul(embd_mats[0], wgts_mats[0])
+ wgts_mats[2])
print('\nU: \n', wgts_mats[1])
```

Out:

```
W_embd * W + b:
 [[ -3.58580279]
 [ -3.57090545]
 [ -3.48436666]
 [ -3.58020806]
 [ 20.28836632]
 [ -3.57022905]
 [ -3.47717571]
 [ -3.60041976]
 [ -3.53663325]
 [ -3.56173849]]

U:
 [[ 1.21534526]]
```

At index 4, we get a high positive value, while others are negative. This way if 4 is encontered in sequence, model creates high positive output, otherwise a negative value.

U is slightly higher than 1, thus the previous score with get amplified (i.e. positive input will give higher positive value, negative input will give more negative value)

Makes sense, right?

. . .

Lower embedding size

Embeddings are supposed to learn representations, right? Let's try lower embedding.

In:

```
del model

input_1 = Input(shape=(3,), name='Input_Layer')
x = Embedding(input_dim=10, output_dim=4, name='Embedding_Layer')
(input_1)
x = SimpleRNN(rnn_size, activation='linear', name='RNN_Layer')(x)
y = Activation('sigmoid', name='Activation_Layer')(x)

model = Model(inputs=input_1, outputs=y)

model.compile(optimizer=Adam(0.02), loss='binary_crossentropy',
metrics=['acc'])
history = model.fit(x=all_feat, y=all_label, batch_size=8, epochs=4,
validation_split=0.2, verbose=1)
```

Out:

```
Train on 8000 samples, validate on 2000 samples
Epoch 1/4
8000/8000 [==============================] - 4s - loss: 0.0180 -
acc: 0.9916 - val_loss: 1.4316e-04 - val_acc: 1.0000
Epoch 2/4
8000/8000 [==============================] - 4s - loss: 6.9616e-05 -
acc: 1.0000 - val_loss: 3.6003e-05 - val_acc: 1.0000
Epoch 3/4
8000/8000 [==============================] - 4s - loss: 2.2050e-05 -
acc: 1.0000 - val_loss: 1.4364e-05 - val_acc: 1.0000
Epoch 4/4
8000/8000 [==============================] - 4s - loss: 9.6034e-06 -
acc: 1.0000 - val_loss: 6.8176e-06 - val_acc: 1.0000
```

In:

```
print('\nInput features: \n', all_feat[-10:,:])
print('\nLabels: \n', all_label[-10:])
print('\nPredictions: \n', model.predict(all_feat[-10:,:]))
```

Out:

```
Input features:
 [[5 6 9]
 [9 9 6]
 [5 8 4]
 [5 8 1]
```

```
 [6 8 1]
 [9 2 7]
 [6 0 0]
 [9 7 8]
 [8 7 9]
 [0 2 7]]

Labels:
 [0 0 1 0 0 0 0 0 0 0]

Predictions:
 [[  4.59608827e-06]
 [  4.54348674e-06]
 [  9.99956727e-01]
 [  4.26225279e-06]
 [  4.60571482e-06]
 [  4.42924829e-06]
 [  4.36698019e-06]
 [  4.22405401e-06]
 [  4.26433053e-06]
 [  4.23319989e-06]]
```

In :

```
embd_layer = model.get_layer('Embedding_Layer')
embd_mats = embd_layer.get_weights()

wgt_layer = model.get_layer('RNN_Layer')
wgts_mats = wgt_layer.get_weights()
```

In :

```
embd_mats
```

Out:

```
[array([[ 0.28512904,  0.25177249, -0.27367339,  0.30789083],
        [ 0.25492424,  0.26012757, -0.29675287,  0.30629158],
        [ 0.28270748,  0.28677672, -0.26648894,  0.25294301],
        [ 0.29766184,  0.26947719, -0.26931292,  0.28334641],
        [-1.50977004, -1.4418962 ,  1.47527599, -1.76327348],
        [ 0.33177933,  0.24144135, -0.26087469,  0.27504072],
        [ 0.25945908,  0.24145941, -0.25259978,  0.34160569],
        [ 0.29079974,  0.26891741, -0.2553148 ,  0.30429697],
        [ 0.29276261,  0.23101816, -0.28032303,  0.29925823],
```

```
        [ 0.27864024,  0.23305847, -0.24671097,  0.35506517]],
dtype=float32)]
```

This time embeddings seem more intuitive. We can see bigger and negative numbers at index 4.

In:

```
print('\n W_embd * W + b: \n', np.matmul(embd_mats[0], wgts_mats[0])
+ wgts_mats[2])
print('\nU: \n', wgts_mats[1])
```

Out:

```
W_embd * W + b:
 [[ -3.31143618]
 [ -3.31706262]
 [ -3.2351234 ]
 [ -3.3211081 ]
 [ 19.09644699]
 [ -3.28206587]
 [ -3.23133922]
 [ -3.31450295]
 [ -3.26382184]
 [ -3.28180361]]

U:
 [[ 1.23604953]]
```

This looks similar to previous model. So lower embedding works!

. . .

Minimal embedding

Let's get more intuitive. It's a classification problem. So why we need 10 or 4 embedding. We should learn only 1 number: high value for 4, low value for others.

In :

```
del model

input_1 = Input(shape=(3,), name='Input_Layer')
x = Embedding(input_dim=10, output_dim=1, name='Embedding_Layer')
(input_1)
x = SimpleRNN(rnn_size, activation='linear', name='RNN_Layer')(x)
y = Activation('sigmoid', name='Activation_Layer')(x)

model = Model(inputs=input_1, outputs=y)

print(model.summary())

model.compile(optimizer=Adam(0.02), loss='binary_crossentropy',
metrics=['acc'])
history = model.fit(x=all_feat, y=all_label, batch_size=8, epochs=4,
validation_split=0.2, verbose=1)
```

Out:

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
Input_Layer (InputLayer)     (None, 3)                 0
_________________________________________________________________
Embedding_Layer (Embedding)  (None, 3, 1)              10
_________________________________________________________________
RNN_Layer (SimpleRNN)        (None, 1)                 3
_________________________________________________________________
Activation_Layer (Activation (None, 1)                 0
=================================================================
Total params: 13.0
Trainable params: 13.0
Non-trainable params: 0.0
_________________________________________________________________
None
Train on 8000 samples, validate on 2000 samples
Epoch 1/4
8000/8000 [==============================] - 4s - loss: 0.0248 -
acc: 0.9902 - val_loss: 4.7527e-04 - val_acc: 1.0000
Epoch 2/4
8000/8000 [==============================] - 4s - loss: 2.2767e-04 -
acc: 1.0000 - val_loss: 1.1516e-04 - val_acc: 1.0000
Epoch 3/4
8000/8000 [==============================] - 4s - loss: 6.9932e-05 -
acc: 1.0000 - val_loss: 4.4882e-05 - val_acc: 1.0000
Epoch 4/4
8000/8000 [==============================] - 4s - loss: 2.9992e-05 -
acc: 1.0000 - val_loss: 2.1167e-05 - val_acc: 1.0000
```

In:

```
embd_layer = model.get_layer('Embedding_Layer')
embd_mats = embd_layer.get_weights()

wgt_layer = model.get_layer('RNN_Layer')
wgts_mats = wgt_layer.get_weights()

print('\nEmbedding weights: \n',embd_mats[0])
print('\nRNN weights: \n', wgts_mats[0], wgts_mats[1], wgts_mats[2]
)

print('\n W_embd * W + b: \n', np.matmul(embd_mats[0], wgts_mats[0])
+ wgts_mats[2])
print('\nU: \n', wgts_mats[1])
```

Out:

```
Embedding weights:
 [[-0.43760461]
 [-0.43051854]
 [-0.43027946]
 [-0.44076917]
 [ 2.7104342 ]
 [-0.44030327]
 [-0.42853352]
 [-0.43567708]
 [-0.43628559]
 [-0.43209222]]

RNN weights:
 [[ 6.39243031]] [[ 1.37941635]] [ 0.15867162]

 W_embd * W + b:
 [[ -2.63868523]
 [ -2.59338808]
 [ -2.59185982]
 [ -2.65891457]
 [ 17.48493385]
 [ -2.65593624]
 [ -2.58069897]
 [ -2.62636375]
 [ -2.63025355]
 [ -2.60344768]]

U:
 [[ 1.37941635]]
```

This time we can observe that both embedding and RNN weights look sensible. Transformed weights are still similar to previous models.

Lesson:

When choosing embedding size,go with your the number that you think should be enough to capture number of feature required to help model reach a decision.

. . .

Getting closer to real-world models

We have consistently been using ***Linear*** activation i.e. no change in input to make things easier to understand. This approach will fail for longer sequences. Why?

Remember the value of **U**; it was somewhat higher than 1. So a big value coming from the start of sequence will keep getting multiplied by **U** causing a blowup. A low value at the start will vanish.
If the value of $\mathbf{U} < 1$, the the reverse will happen.

Consider different complex variations of input over longer sequences and you will get the idea of the problems.

To deal with it we will use a non-activation. Genrally it's ***tanh***, but as I want my probability, I will use ***sigmoid*** (***tanh*** output number in range (-1,1)).

In :

```
del model

input_1 = Input(shape=(3,), name='Input_Layer')
x = Embedding(input_dim=10, output_dim=1, name='Embedding_Layer')
(input_1)
y = SimpleRNN(rnn_size, activation='sigmoid', name='RNN_Layer')(x)

model = Model(inputs=input_1, outputs=y)

print(model.summary())

model.compile(optimizer=Adam(0.03), loss='binary_crossentropy',
metrics=['acc'])
history = model.fit(x=all_feat, y=all_label, batch_size=8, epochs=4,
validation_split=0.2, verbose=1)
```

Out:

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
Input_Layer (InputLayer)     (None, 3)                 0
_________________________________________________________________
Embedding_Layer (Embedding)  (None, 3, 1)              10
_________________________________________________________________
RNN_Layer (SimpleRNN)        (None, 1)                 3
=================================================================
Total params: 13.0
Trainable params: 13.0
Non-trainable params: 0.0
_________________________________________________________________
None
Train on 8000 samples, validate on 2000 samples
Epoch 1/4
8000/8000 [==============================] - 5s - loss: 0.1316 -
acc: 0.9599 - val_loss: 0.0094 - val_acc: 1.0000
Epoch 2/4
8000/8000 [==============================] - 4s - loss: 0.0054 -
acc: 1.0000 - val_loss: 0.0029 - val_acc: 1.0000
Epoch 3/4
8000/8000 [==============================] - 4s - loss: 0.0020 -
acc: 1.0000 - val_loss: 0.0013 - val_acc: 1.0000
Epoch 4/4
8000/8000 [==============================] - 4s - loss: 9.8492e-04 -
acc: 1.0000 - val_loss: 6.7974e-04 - val_acc: 1.0000
```

In:

```
print('\nInput features: \n', all_feat[-10:,:])
print('\nLabels: \n', all_label[-10:])
print('\nPredictions: \n', model.predict(all_feat[-10:,:]))
```

Out:

```
Input features:
 [[5 6 9]
 [9 9 6]
 [5 8 4]
 [5 8 1]
 [6 8 1]
 [9 2 7]
 [6 0 0]
 [9 7 8]
 [8 7 9]
 [0 2 7]]
```

```
Labels:
 [0 0 1 0 0 0 0 0 0 0]

Predictions:
 [[  4.83995711e-04]
 [  5.38199791e-04]
 [  9.99984384e-01]
 [  5.32271166e-04]
 [  5.32272446e-04]
 [  5.16529719e-04]
 [  4.35839931e-04]
 [  5.51217643e-04]
 [  4.83845797e-04]
 [  5.16526750e-04]]
```

In:

```
embd_layer = model.get_layer('Embedding_Layer')
embd_mats = embd_layer.get_weights()

wgt_layer = model.get_layer('RNN_Layer')
wgts_mats = wgt_layer.get_weights()

print('\nEmbedding weights: \n',embd_mats[0])
print('\nRNN weights: \n', wgts_mats[0], wgts_mats[1], wgts_mats[2]
)

print('\n W_embd * W + b: \n', np.matmul(embd_mats[0], wgts_mats[0])
+ wgts_mats[2])
print('\nU: \n', wgts_mats[1])
```

Out:

```
Embedding weights:
 [[ 1.22580421]
 [ 1.1842196 ]
 [ 1.15448904]
 [ 1.20168781]
 [-2.71523046]
 [ 1.19090605]
 [ 1.18169999]
 [ 1.19069457]
 [ 1.17678034]
 [ 1.20412242]]

RNN weights:
 [[-4.77058172]] [[ 13.87013912]] [-1.89605367]
```

```
W_embd * W + b:
[[ -7.74385309]
 [ -7.54547024]
 [ -7.40363789]
 [ -7.62880373]
 [ 11.05717564]
 [ -7.57736826]
 [ -7.53345013]
 [ -7.57635975]
 [ -7.50998068]
 [ -7.64041805]]

U:
 [[ 13.87013912]]
```

Weights still make sense. We can use non-linear activations.

Following is the link to the notebook for this part:

VishnuDuttSharma/DeepLearning

DeepLearning - This repository contains my submissions for Deep Learning course taught at IIT Kgp. The script of choice...

github.com

Machine Learning Keras Embedding Rnn Deep Learning