A documentation on Keras examples on Recurrent Neural Network

Mingxin Liu

Introduction:

The purpose of this paper is to learn Recurrent Neural Network(RNN) through examples from Keras. RNN is mainly used to work with sequential problems or text generation projects. Therefore, the two documentations I will provide in this paper are about one solving sequential problems with RNN and one generating text operation using LSTM. With a better understanding on the matter of neural network I'm introducing in the paper, I will also explain Recurrent Neural Network with diagrams and functions. The reason I'm not submitting using Jupyter notebook is because the example code is pretty robust. So, I left a GitHub pointer in the end so you could check out the code if you like.

First, let's quickly go over some steps with getting Keras started.

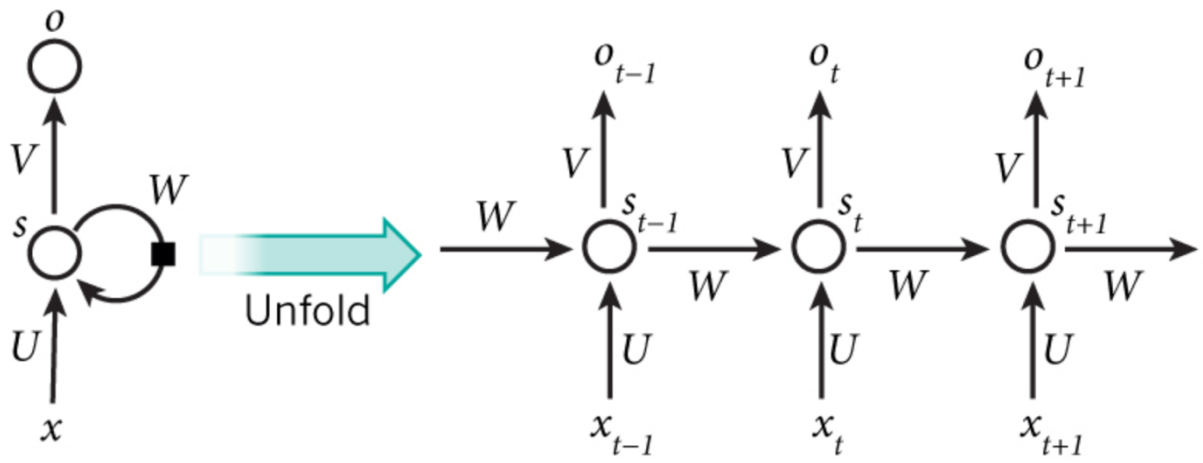
How to install and run Keras:

There are two ways to install Keras. We can either open up a command window and type 'sudo pip install keras' or we can git clone from its repository and then install from the setup file. However, there happens to be an issue with the first method and I'll talk about it in a bit. So after installing Keras we will have to install one of the backend engines: TensorFlow, Theano, or CNTK. This installation process is suggested to be done before installing Keras. For the purpose of this documentation, I will choose TensorFlow. If you simply cd to the Keras example folder and run

its python code, you wouldn't get anything. So here you will need to activate the TensorFlow by typing 'source ~/targetDirectory/bin/activate'. I named my TensorFlow directory targetDirectory, so change this to your own TensorFlow directory.

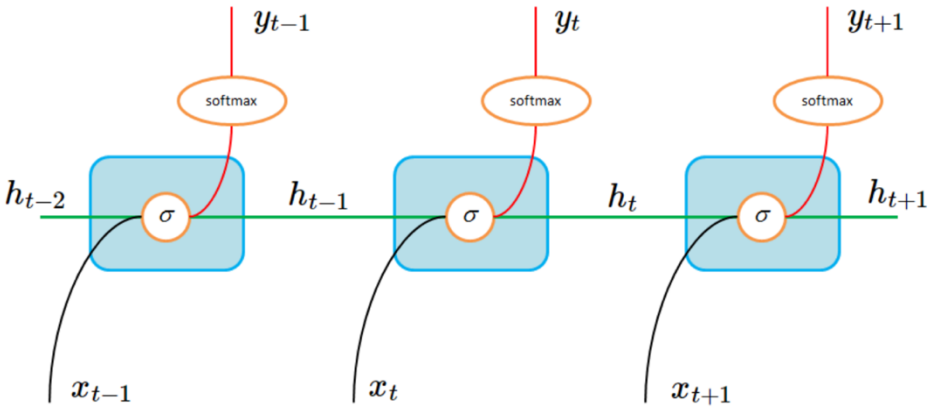
RNN:

Recurrent Neural Networks or supervised learning is my focus for this paper. Since most neural networks has limitations with computing sequential models, RNN stacks another hidden layer from the previous time step and the next layer is depending on the corresponding input at the current time step and the output from previous time step. In another word, unlike most neural networks where each neuron is independent from each other, the neural net compute every element of the sequence with the output being depended on the previous output. So they have memories of what are being done so far, and they take that into consideration of the future outputs (supervised learning). like the diagram shown below. Here x_t is the input at time step t. S_t is the hidden state at time step t, or the memory of the network. It is computed based on the previous hidden state and the input of the current time step x_t. $o_t = \mathrm{softmax}(Vs_t)$. O_t is the output at each time step. Here is how the prediction of the next word can be computed by



A recurrent neural network and the unfolding in time of the computation involved in its forward computation. Source: Nature

Lets take a look at another diagram.

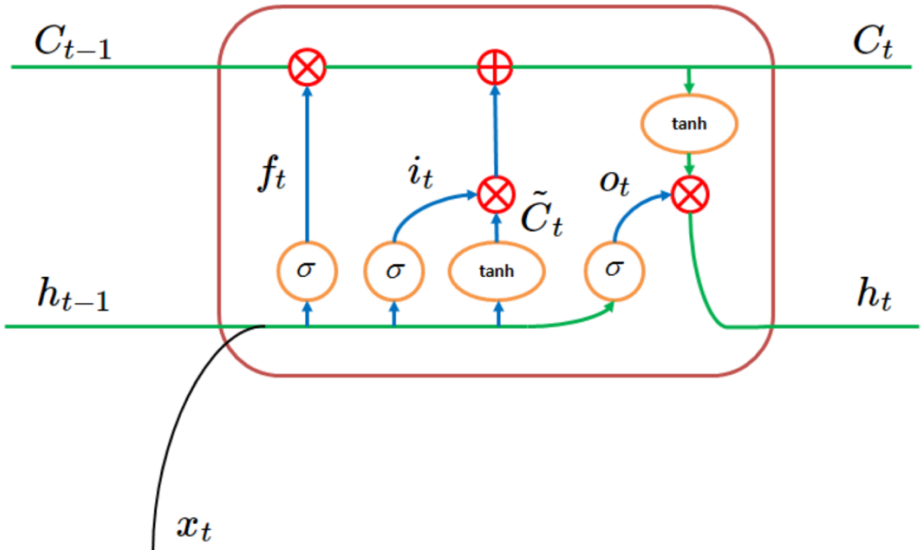


$$
\begin{aligned}
h_t &= \text{sigm}\left(W^{\text{hx}}x_t + W^{\text{hh}}h_{t-1}\right) \\
y_t &= W^{\text{yh}}h_t
\end{aligned}
$$

Here the layers are associated with the previous weight, however the output at each time step only depends on that layer. From here, a more specific and improved RNN will also be introduced in the examples called Long Short-Term Memory network. (LSTM). LSTM helps the network to preserve error that can be backpropagated through time and layers. So with the error being constant, the recurrent

$f_t = \sigma(W_f.[h_{t-1}, x_t] + b_f)$ $C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$ $i_t = \sigma(W_i.[h_{t-1}, x_t] + b_i)$

network can learn from its errors over many times. $\tilde{C}_t = tanh(W_C.[h_{t-1}, x_t] + b_C)$ Information can be stored, written or read from a cell, and cell makes the decision of whether to allow being read, write, erase or store. Being said that the LSTM can erase, thus we have a forgot gate f_t. Then the input layer it is Then the actication function is computed as follows and here is how Ct related with the forgoten gate and the input gate, if the forgetting is nessaccry.



Here is a diagram of how the network looks like.

'addition_rnn.py' runs on a training of set of two input numbers to compute the correct result from the addition of the two.

Here let's take a deeper look at the code.

examples: addition_rnn.py

addition_rnn.py

addition_rnn is an implementation of sequence to sequence learning for performing addition of two numbers.

```
def encode(self, C, num_rows):
    """One hot encode given string C.

    # Arguments
        num_rows: Number of rows in the returned one hot encoding. This is
            used to keep the # of rows for each data the same.
    """
    x = np.zeros((num_rows, len(self.chars)))
    for i, c in enumerate(C):
        x[i, self.char_indices[c]] = 1
    return x

def decode(self, x, calc_argmax=True):
    if calc_argmax:
        x = x.argmax(axis=-1)
    return ''.join(self.indices_char[x] for x in x)
```

First we will need to import the libraries for the functions.

```
from __future__ import print_function
from keras.models import Sequential
from keras import layers
import numpy as np
```

As for recurrent neural network, this happens to be one setback which is that the input of the training data has to be a fixed length. So here we use numpy library to initialize the fixed vector x with number of rows and length of characters.

Then the model is trained with RNN (recurrent neural network), or essentially supervised learning or regression. So the neural network can access to its LSTM (long short-term memory) to make better guesses at the output from the input.

Lastly, here is how it checks for validation of the results.

```
print('Build model...')
model = Sequential()
# "Encode" the input sequence using an RNN, producing an output of HIDDEN_SIZE.
# Note: In a situation where your input sequences have a variable length,
# use input_shape=(None, num_feature).
model.add(RNN(HIDDEN_SIZE, input_shape=(MAXLEN, len(chars))))
# As the decoder RNN's input, repeatedly provide with the last hidden state of
# RNN for each time step. Repeat 'DIGITS + 1' times as that's the maximum
# length of output, e.g., when DIGITS=3, max output is 999+999=1998.
model.add(layers.RepeatVector(DIGITS + 1))
# The decoder RNN could be multiple layers stacked or a single layer.
for _ in range(LAYERS):
    # By setting return_sequences to True, return not only the last output but
    # all the outputs so far in the form of (num_samples, timesteps,
    # output_dim). This is necessary as TimeDistributed in the below expects
    # the first dimension to be the timesteps.
    model.add(RNN(HIDDEN_SIZE, return_sequences=True))

# Apply a dense layer to the every temporal slice of an input. For each of step
# of the output sequence, decide which character should be chosen.
model.add(layers.TimeDistributed(layers.Dense(len(chars))))
model.add(layers.Activation('softmax'))
model.compile(loss='categorical_crossentropy',
              optimizer='adam',
              metrics=['accuracy'])
model.summary()

# Train the model each generation and show predictions against the validation
# dataset.
```

Results:

```
--------------------------------------------------
Iteration 1
Train on 45000 samples, validate on 5000 samples
Epoch 1/1
2017-12-16 02:50:14.465589: I tensorflow/core/platform/cpu_feature
nstructions that this TensorFlow binary was not compiled to use: S
45000/45000 [==============================] - 11s 240us/step - lo
s: 1.7798 - val_acc: 0.3418
Q 30+59   T 89   ☒ 14
Q 901+47  T 948  ☒ 104
Q 48+80   T 128  ☒ 14
Q 135+705 T 840  ☒ 104
Q 420+9   T 429  ☒ 54
Q 13+650  T 663  ☒ 142
Q 281+88  T 369  ☒ 102
Q 44+91   T 135  ☒ 44
Q 11+345  T 356  ☒ 142
Q 230+629 T 859  ☒ 102
```

at iteration one, answers are completely absurd with huge errors. With the memories of the computed results, then

```
--------------------------------------------------
Iteration 7
Train on 45000 samples, validate on 5000 samples
Epoch 1/1
45000/45000 [==============================] - 9s 199us/step - loss: 1.0398 - acc: 0.6202 -
 1.0092 - val_acc: 0.6277
Q 664+36  T 700  ☒ 795
Q 798+9   T 807  ☒ 800
Q 517+22  T 539  ☒ 543
Q 645+113 T 758  ☒ 715
Q 539+554 T 1093 ☒ 1078
Q 53+27   T 80   ☑ 80
Q 948+19  T 967  ☒ 966
Q 298+4   T 302  ☒ 293
Q 89+394  T 483  ☒ 455
Q 647+309 T 956  ☒ 999
```

at iteration number 7, we got one matching result with slightly smaller errors.

```
--------------------------------------------------
Iteration 17
Train on 45000 samples, validate on 5000 samples
Epoch 1/1
45000/45000 [==============================] - 9s 195us/step - loss: 0.2178 - acc: 0.9533 - val_loss
: 0.1999 - val_acc: 0.9539
Q 165+95  T 260  ☑ 260
Q 56+349  T 405  ☑ 405
Q 70+858  T 928  ☑ 928
Q 91+513  T 604  ☑ 604
Q 980+607 T 1587 ☑ 1587
Q 647+71  T 718  ☑ 718
Q 0+253   T 253  ☑ 253
Q 6+377   T 383  ☑ 383
Q 936+6   T 942  ☑ 942
Q 658+47  T 705  ☑ 705
```

and lastly *at iteration number 17*, all answers are matched and it continues to do so for other random number additions as well.

experiment:

Change the Hidden Layers from 128 to 300

When digit is still 3, here is what we got.

```
--------------------------------------------------
Iteration 3
Train on 45000 samples, validate on 5000 samples
Epoch 1/1
45000/45000 [==============================] - 33s 737us/step - loss: 1.2198 - acc: 0.5439 - val_
s: 1.0753 - val_acc: 0.5883
Q 257+855 T 1112 ☑ 1112
Q 200+921 T 1121 ☒ 1122
Q 32+734  T 766  ☒ 769
Q 515+497 T 1012 ☒ 102
Q 97+441  T 538  ☒ 532
Q 823+69  T 892  ☑ 892
Q 10+23   T 33   ☒ 54
Q 940+206 T 1146 ☒ 1122
Q 919+16  T 935  ☒ 939
Q 520+119 T 639  ☒ 632
```

```
--------------------------------------------------
Iteration 7
Train on 45000 samples, validate on 5000 samples
Epoch 1/1
45000/45000 [==============================] - 35s 767us/step - loss: 0.1498 - acc: 0.9661 - val_
s: 0.1013 - val_acc: 0.9835
Q 4+510   T 514  ☑ 514
Q 395+38  T 433  ☑ 433
Q 916+734 T 1650 ☑ 1650
Q 15+927  T 942  ☑ 942
Q 765+0   T 765  ☑ 765
Q 91+739  T 830  ☑ 830
Q 637+21  T 658  ☑ 658
Q 315+17  T 332  ☑ 332
Q 814+30  T 844  ☑ 844
Q 222+660 T 882  ☑ 882
```

Compare with the first experiment, when the hidden layers increase from 128 to 300, the training results are much better. Once the hidden states are increased, the next state of the hidden layer has more feedbacks for it to compute the next output from the current input. Although this way each iteration takes longer to compute, but it does have much better accuracy. Therefore, it is suggested with more hidden layers, RNN works better.

layers = 2

```
--------------------------------------------------
Iteration 2
Train on 45000 samples, validate on 5000 samples
Epoch 1/1
45000/45000 [==============================] - 13s 279us/step - loss: 1.7571 - acc: 0.3455 - val_los
s: 1.7028 - val_acc: 0.3599
Q 83+258  T 341  ☒ 600
Q 1+664   T 665  ☒ 13
Q 373+207 T 580  ☒ 700
Q 288+25  T 313  ☒ 400
Q 33+169  T 202  ☒ 100
Q 840+3   T 843  ☒ 400
Q 563+65  T 628  ☒ 700
Q 28+655  T 683  ☒ 400
Q 872+4   T 876  ☒ 400
```

```
--------------------------------------------------
Iteration 2
Train on 45000 samples, validate on 5000 samples
Epoch 1/1
45000/45000 [==============================] - 9s 203us/step - loss: 1.7242 - acc: 0.3636 - val_loss
: 1.6356 - val_acc: 0.3897
Q 886+33  T 919  ☒ 803
Q 225+897 T 1122 ☒ 1233
Q 515+10  T 525  ☒ 555
Q 74+165  T 239  ☒ 455
Q 280+66  T 346  ☒ 383
Q 2+319   T 321  ☒ 33
Q 20+951  T 971  ☒ 703
Q 97+676  T 773  ☒ 700
Q 574+710 T 1284 ☒ 103
Q 729+15  T 744  ☒ 703

--------------------------------------------------
```

first let's compare the second iteration after changing layers to 2. Although neither of both has any correct results, the error for the latter is rather larger than that of the first time.

```
Iteration 8
Train on 45000 samples, validate on 5000 samples
Epoch 1/1
45000/45000 [==============================] - 12s 276us/step - loss: 1.0720 - acc: 0.5985 - val_los
s: 1.0383 - val_acc: 0.6059
Q 33+169  T 202  ☒ 219
Q 756+22  T 778  ☒ 777
Q 546+17  T 563  ☒ 557
Q 213+515 T 728  ☒ 744
Q 168+60  T 228  ☒ 222
Q 204+408 T 612  ☒ 632
Q 43+90   T 133  ☑ 133
Q 74+672  T 746  ☒ 733
Q 43+309  T 352  ☒ 329
Q 338+630 T 968  ☒ 950
```

Then at iteration number 8 we have the first correct result which is close to the first experiment.

```
Iteration 24
Train on 45000 samples, validate on 5000 samples
Epoch 1/1
45000/45000 [==============================] - 12s 275us/step - loss: 0.0391 - acc: 0.9909 - val_los
s: 0.0722 - val_acc: 0.9759
Q 612+57  T 669  ☑ 669
Q 538+65  T 603  ☑ 603
Q 738+82  T 820  ☑ 820
Q 17+224  T 241  ☑ 241
Q 458+34  T 492  ☑ 492
Q 4+324   T 328  ☑ 328
Q 179+86  T 265  ☑ 265
Q 92+859  T 951  ☑ 951
Q 352+13  T 365  ☑ 365
Q 987+2   T 989  ☑ 989

--------------------------------------------------
```

Then at iteration 24, we have the first fully correct results. Compare to iteration 17 for the first experiment, this is much later.

Here if we train on the second layer instead of the first, it takes longer to compute for the same results.

digits = 5

Then the next experiment I increased the digit numbers from 3 to 5 with layer stay at 1. This should give a little more trouble for the network.

```
--------------------------------------------------
Iteration 2
Train on 45000 samples, validate on 5000 samples
Epoch 1/1
45000/45000 [==============================] - 12s 276us/step - loss: 1.5512 - acc: 0.4252 - val_los
s: 1.4923 - val_acc: 0.4431
Q 16587+598   T 17185  ☒ 16676
Q 945+30      T 975    ☒ 400
Q 1+599       T 600    ☒ 100
Q 4447+7606   T 12053  ☒ 4447
Q 6+6262      T 6268   ☒ 6666
Q 36+187      T 223    ☒ 136
Q 71+1417     T 1488   ☒ 1116
Q 470+8198    T 8668   ☒ 1006
Q 75+774      T 849    ☒ 777
Q 7545+6984   T 14529  ☒ 55576
```

At the second iteration, we see that the estimates are not too off from the results.

```
--------------------------------------------------
Iteration 10
Train on 45000 samples, validate on 5000 samples
Epoch 1/1
45000/45000 [==============================] - 13s 278us/step - loss: 0.8654 - acc: 0.6739 - val_los
s: 0.8455 - val_acc: 0.6842
Q 3292+96     T 3388   ☒ 3347
Q 8759+6      T 8765   ☒ 8764
Q 11300+71    T 11371  ☒ 11379
Q 180+1       T 181    ☑ 181
Q 300+455     T 755    ☒ 608
Q 65715+49    T 65764  ☒ 65757
Q 3319+400    T 3719   ☒ 3722
Q 27+82928    T 82955  ☒ 82907
Q 48159+6     T 48165  ☒ 48161
Q 398+60931   T 61329  ☒ 61911
```

That went on for another 8 iterations until the 10th iteration we encountered the first correct result.

```
Iteration 51
Train on 45000 samples, validate on 5000 samples
Epoch 1/1
45000/45000 [==============================] - 14s 300us/step - loss: 0.0939 - acc: 0.9704 - val_los
s: 0.1470 - val_acc: 0.9448
Q 1053+43      T 1096    ☑ 1096
Q 9+62808      T 62817   ☑ 62817
Q 2269+419     T 2688    ☑ 2688
Q 2+69780      T 69782   ☑ 69782
Q 4311+6       T 4317    ☑ 4317
Q 32+3281      T 3313    ☑ 3313
Q 57+56        T 113     ☑ 113
Q 779+52652    T 53431   ☑ 53431
Q 3701+91870   T 95571   ☑ 95571
Q 67304+83     T 67387   ☑ 67387
```

at iteration number 51, we have fully correct results. However, this isn't definite yet. The iteration right after shows unpromising outcomes. So for the neural network be able to complete its training, it might take 100 iterations or so. Since this takes a lot of effort to train for the correct outcome, let's try to increase the hidden layers in this case see if we could reduce the number of iterations.

Hidden Layers from 128 to 300 with digit = 5

```
--------------------------------------------------
Iteration 6
Train on 45000 samples, validate on 5000 samples
Epoch 1/1
45000/45000 [==============================] - 48s 1ms/step - loss: 0.7848 - acc: 0.7018 - val_loss:
 0.7452 - val_acc: 0.7179
Q 5471+39153   T 44624   ☒ 42224
Q 475+495      T 970     ☒ 1224
Q 7+997        T 1004    ☑ 1004
Q 36067+1019   T 37086   ☒ 37624
Q 149+6992     T 7141    ☒ 7181
Q 413+73672    T 74085   ☒ 74162
Q 746+4        T 750     ☒ 740
Q 88892+7      T 88899   ☑ 88899
Q 59346+0      T 59346   ☒ 59337
Q 72+810       T 882     ☒ 884


--------------------------------------------------
Iteration 21
Train on 45000 samples, validate on 5000 samples
Epoch 1/1
45000/45000 [==============================] - 48s 1ms/step - loss: 0.0906 - acc: 0.9715 - val_loss:
 0.1496 - val_acc: 0.9461
Q 27360+473    T 27833   ☑ 27833
Q 4+489        T 493     ☑ 493
Q 97+4664      T 4761    ☑ 4761
Q 80+965       T 1045    ☑ 1045
Q 25442+511    T 25953   ☑ 25953
Q 511+489      T 1000    ☑ 1000
Q 3306+70      T 3376    ☑ 3376
Q 14165+61     T 14226   ☑ 14226
Q 475+104      T 579     ☑ 579
Q 135+6561     T 6696    ☑ 6696

--------------------------------------------------
```

Compare with when hidden layers are only 128, with 300 hidden layers, the network computes each iteration much slower, but do produce better accuracy and results shown above. From the snapshot above, we can tell that the iterations it takes to compute for the same task is almost half when the hidden layers are increased for 57%. More tests can be done on the influence of hidden layer, but pretty much follows the same trend here.

As for the second example 'lstm_text_generation.py', it also uses RNN. So I will focus more on the brief training result and my experiments with the code.

Here at the path, you can change your training data directory, however 50000 and plus words are preferred for the training.

```
path = get_file('nietzsche.txt', origin='https://s3.amazonaws.com/text-datasets/nietzsche.txt')
text = io.open(path, encoding='utf-8').read().lower()
print('corpus length:', len(text))
```

```
def sample(preds, temperature=1.0):
    # helper function to sample an index from a probability array
    preds = np.asarray(preds).astype('float64')
    preds = np.log(preds) / temperature
    exp_preds = np.exp(preds)
    preds = exp_preds / np.sum(exp_preds)
    probas = np.random.multinomial(1, preds, 1)
    return np.argmax(probas)
```

Here is the function to obtain the sequential model.

```
200285/200285 [==============================] - 1053 523us/step - loss: 1.3421

----- diversity: 0.2
----- Generating with seed: "condition "away from which we go," the
s"
condition "away from which we go," the
sense of the same proporing and the origin of the conscience and and the same that the same and and
still and and the same individual for the things and the same in the strength of the same in the sam
e in the same the most same man and as a statesm and a state of the possible to the explained to the
 specially the an action of the same interrogation of the same as individual of the same and individ
u

----- diversity: 0.5
----- Generating with seed: "condition "away from which we go," the
s"
condition "away from which we go," the
soul and extend to the absure for the strigthers are the plaining of the destined to all the sense o
f the such a plays and his us the possible, as the learn the half vented as individuals and and expe
rience and still and the philosophy and conscience, the same esteemed to its greating, and which is
the same possesses, to one can in the significane of the non--this science, one may be the places an
d

----- diversity: 1.0
----- Generating with seed: "condition "away from which we go," the
s"
condition "away from which we go," the
satisfice my much ionstoful
spirit in its nectively indoup depuill of geriase itself and as it threns, for the head of an a accu
red as ideas man to anaonajumilibinry them out of vious many ordinariation" in art, by
the modern the fears musting perists as
such love these courage. much as invepted on a
indicted, there would rew sense, in the halt good, to have
deithditions, reflect of those want are

----- diversity: 1.2
----- Generating with seed: "condition "away from which we go," the
s"
condition "away from which we go," the
starid that one tom. his sense-ety l prih with reabou distained ucould, through hunders to time
fientsching ifeart affe
hus to dopedments upan profusis question, as the sami, it at not thent and the maroun of a terms of
women cornaurtful the earth and of dittings" ar itteratism--in the problems in him ilsiar where a fo
r
edver grutic have subjectment
"thay light," would but to understoods nature, th
```

Since the data training lasted only for 30 minuets, the results are definitely not optimal. However, we can see how the English sentences are starting to make sense with fairly correct. However, the performance decreases respect to the increase of the number of diversity. Here the diversity is referring to that neural network is operating on different scales of time at once. This is for the purpose of capturing diverse time scales with remote dependencies.

From the output of the example, we can see that the network is starting to pick up the sentence from the training. It conforms to the line format observed in the original text before new line and

try to use that memory onto the new input. The network picks up the sequence in how the order of words will make sense in a sentence, and what do not. This gradually will build up a stronger memory for the network to distinguish between grammatically correct and grammatically incorrect sentences.

To conclude, in order to master speaking correct English sentences for machines, it requires a much longer training period. However, there are some improvements to the model that we could approach in order to achieve better results. First we could increase the number of training epochs to 100 or more. Second, we could train the model on padded sentences rather than some random length article. This way, the machine has a more orientated goal. Like when human learn languages, we started with very basic stuff and then gradually increased the difficulty. We could also add more hidden layers to increase the short-term memory of the network.

Conclusion:

Hence, Recurrent Neural Network becomes clearer to us. RNN and LSTM has major contributions on solving sequential problems such as addition and text generation. We witnessed that the network can learn both 3 and 5 digits' addition problems. We also saw that the network could learn the English grammar from extracting fixed length vectors at a time from a book.

Obviously, the training duration varies based on the parameters and the difficulty of the task itself, but we do achieve promising results from Keras examples. RNN produces better results with more hidden layers imbedded, so that the machine learning from the previous state can be done

with more complexity. From here and on, we could build other similar RNNs with respect to the examples from Keras.

Github: https://github.com/keras-team/keras/blob/master/examples/addition_rnn.py

https://github.com/keras-team/keras/blob/master/examples/lstm_text_generation.py