Introduction to Deep Learning

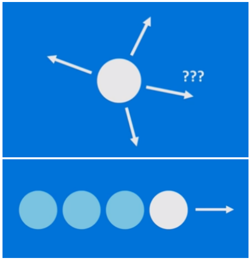
Yogesh Kulkarni

October 10, 2020

Recurrent Neural Network (RNN)

Sequences

- Need to predict when the ball would be in the next moment?
- Scenario 1: no info of previous state is given. Answer: random state.
- Scenario 2: some info of previous state is given. Answer: possible to fit trajectory and predict.
- Another example: On mobile, while typing sms you get some predictive suggestions !!!



(Ref: Recurrent Neural Network - Ava Soleimany, January 2020)

Recap: NN types: ANN, CNN, RNN

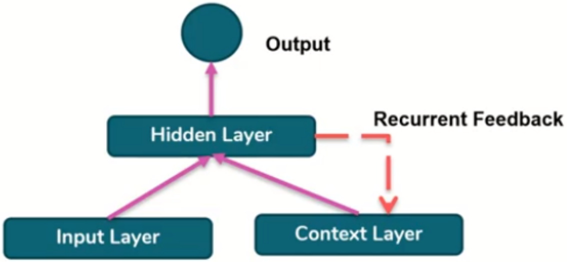
- In ANN, columns are independent of each other. None of the neighbor information is used. Shuffling columns does not have any effect on the model/results.
- In CNN, spatial (not 'special') column-neighbors are used to form features, which help in discerning the categories in the classification.
- In RNN, temporal (no, 'temporary') neighbors/column-wise are used to form features, which help in predictions.

So, Why RNN?

- Problem is: ANNs do not have memory. Meaning, any bearing to the previous samples.
- The model does not care about what came before.
- Thats not useful in case of Sequences or Time Series. Here, past is useful in deciding the present and the future.
- Solution: RNN

Background

- By Elman in 1990.
- Proposed to have input layer, hidden ayer, context layer along with recurrent feedback.



(Ref: Simple RNN Explained - Shriram Vasudevan)

Toy Example

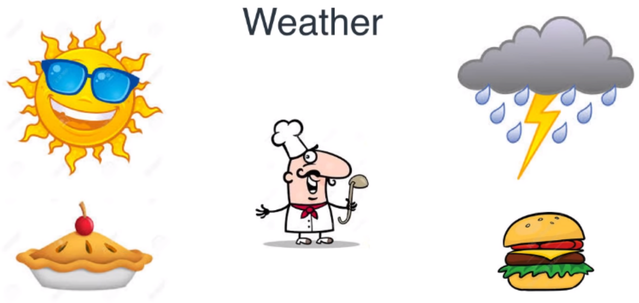
(Ref: Luis Serrano)

Toy Example: Scenario I: Cook

An ideal roommate who cooks:

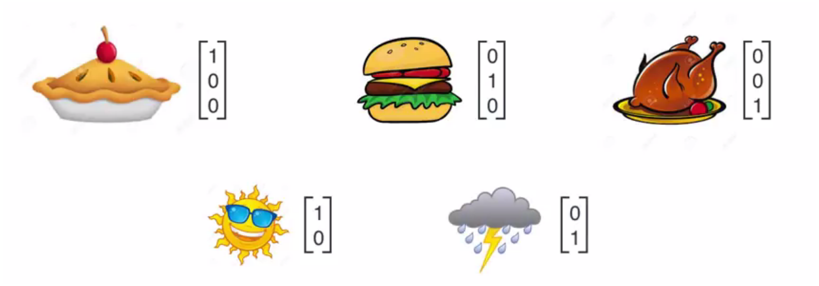
- Can cook 3 dishes: Apple Pie, Burger, Chicken
- Dish depends on the outside weather (only Sunny or Rainy)
  - Sunny: Apple Pie
  - Rainy: Burger

This can be easily modeled as Neural network with known inputs and outputs.



Toy Example: Scenario I

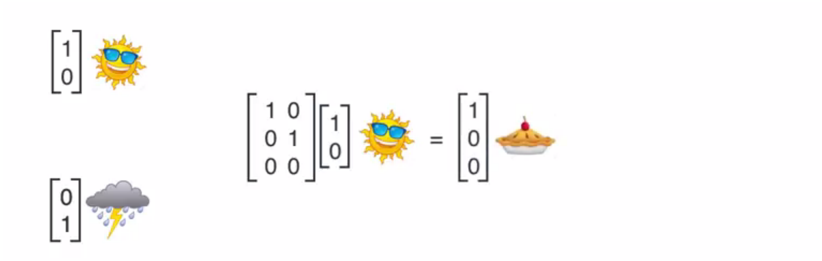
Lets model food and weather as one-hot encoding vectors



Network takes weather vector in and outputs food vector

Toy Example: Scenario I

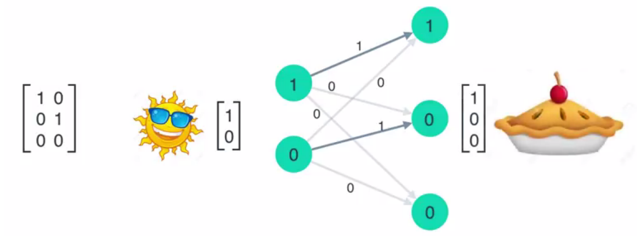
- This Neural Network is represented by a matrix (3x2)g
- Individual weather vector when post multiplies this matrix, resultant 3x1 vector represents food
- For Sunny, you get Apple Pie
- For Rainy, you get Burger.



$\begin{bmatrix}1\\0\end{bmatrix}$
$\begin{bmatrix}0\\1\end{bmatrix}$
$\begin{bmatrix}1&0\\0&1\\0&0\end{bmatrix}\begin{bmatrix}1\\0\end{bmatrix} = \begin{bmatrix}1\\0\\0\end{bmatrix}$

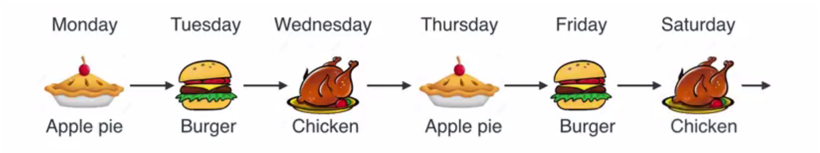
Toy Example: Scenario I

- Matrix can be seen as network as well.
- First node emits 3 rays, each to each output node: 1, 0, 1. This is nothing but first column of the matrix
- Second node emits 3 rays, each to each output node: 0, 1, 0. This is nothing but second column of the matrix
- Input weather vector values are put into input nodes, multiplication happens and then output values are generated.


$\begin{bmatrix}1&0\\0&1\\0&0\end{bmatrix}$
$\begin{bmatrix}1\\0\end{bmatrix}$
1
0
1
0
0
0
0
1
0
0
1
0
$\begin{bmatrix}1\\0\\0\end{bmatrix}$

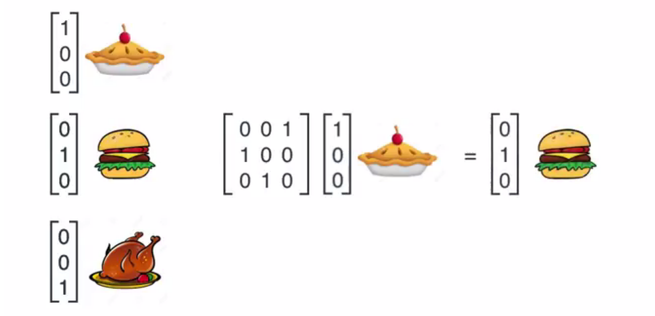
Toy Example: Scenario II

- Same roommate, but instead of weather he decides the dish based on previous day.
- One day Apple Pie, next Burger, next Chicken.



Toy Example: Scenario II

- There is no Weather input, but previous day's output is its input. Shown as loop.
- This is Time Series, like stock prices.



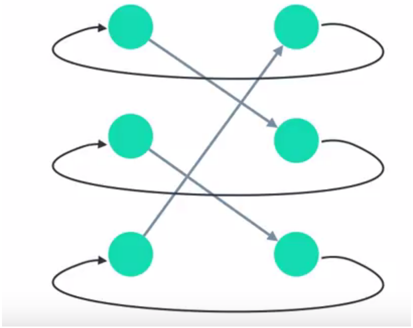
$$\begin{bmatrix}1\\0\\0\end{bmatrix}\quad\begin{bmatrix}0\\1\\0\end{bmatrix}\quad\begin{bmatrix}0\\0\\1\end{bmatrix}$$

$$\begin{bmatrix}0&0&1\\1&0&0\\0&1&0\end{bmatrix}\begin{bmatrix}1\\0\\0\end{bmatrix}=\begin{bmatrix}0\\1\\0\end{bmatrix}$$

Toy Example: Scenario II

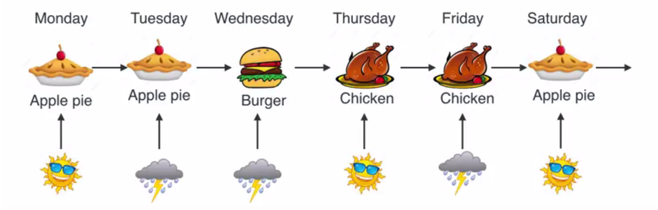
As a network:

- 3 inputs, 3 outputs
- Both representing food
- Its basically looping back. So, its Recurrent.



Toy Example: Scenario III

- Same roommate, but now the rules of cooking are dependent on both, weather as well as previous day dish
- If Sunny, he goes out, has fun, no cooking, so dish is leftover, same as yesterday
- If Rainy, he is at home, next dish on the sequence.
- Below, for Tuesday, it's Sunny weather is shown under Monday (just for illustration) as input
- So inputs are Weathers, and also the predicted output of last day ie Food.



Toy Example: Scenario III

Input Vectors

$\begin{bmatrix}1\\0\\0\end{bmatrix}$

$\begin{bmatrix}0\\1\\0\end{bmatrix}$

$\begin{bmatrix}0\\0\\1\end{bmatrix}$

$\begin{bmatrix}1\\0\end{bmatrix}$

$\begin{bmatrix}0\\1\end{bmatrix}$

Weight Matrices

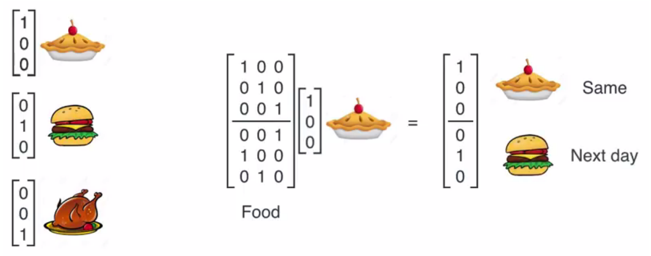
$$\underset{\text{Food}}{\begin{bmatrix}1&0&0\\0&1&0\\0&0&1\\\hline 0&0&1\\1&0&0\\0&1&0\end{bmatrix}} \underset{\text{Add}}{+} \underset{\text{Weather}}{\begin{bmatrix}1&0\\1&0\\1&0\\\hline 0&1\\0&1\\0&1\end{bmatrix}}$$

Merge

Toy Example: Scenario III

Food Matrix

- Food matrix artificially cut into two
- Top represents Sunny, bottom as Rainy
- Apple Pie comes in, top results in Apple Pie as TODAY's food (same), bottom results in next dish, ToMORROW's food.
- Nothing about Weather as of now.

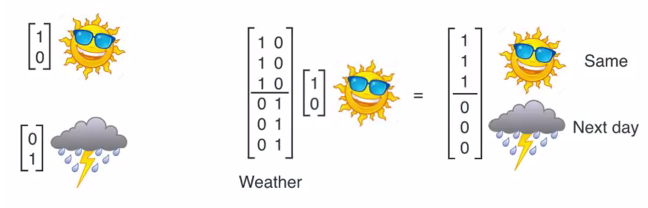


$\begin{bmatrix}1\\0\\0\end{bmatrix}$
$\begin{bmatrix}0\\1\\0\end{bmatrix}$
$\begin{bmatrix}0\\0\\1\end{bmatrix}$
$\begin{bmatrix}1&0&0\\0&1&0\\0&0&1\\ \hline 0&0&1\\1&0&0\\0&1&0\end{bmatrix}\begin{bmatrix}1\\0\\0\end{bmatrix} = \begin{bmatrix}1\\0\\0\\ \hline 0\\1\\0\end{bmatrix}$
Same
Next day
Food

Toy Example: Scenario III

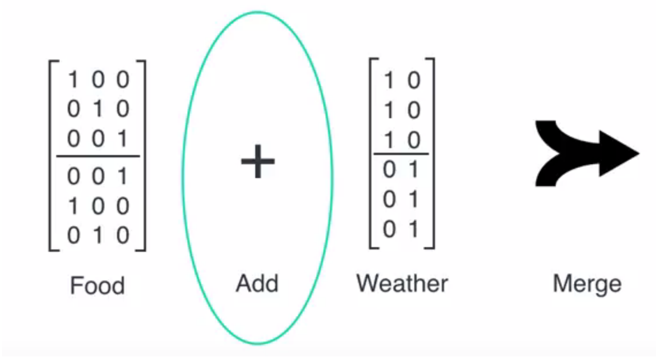
Weather Matrix

- Top results are all 1's. Same as Today, Sunny as 1 represents Sunny, 0 Rainy.
- Bottom result are all 0's. For Next day.
- Weather matrix tells, if he as to cook TODAY's food or TOMORROW's food. As the bottom results are all 1's, or YES's.


$\begin{bmatrix}1\\0\end{bmatrix}$
$\begin{bmatrix}0\\1\end{bmatrix}$
$\begin{bmatrix}1&0\\1&0\\1&0\\0&1\\0&1\\0&1\end{bmatrix}\begin{bmatrix}1\\0\end{bmatrix} = \begin{bmatrix}1\\1\\1\\0\\0\\0\end{bmatrix}$
Same
Next day
Weather

Toy Example: Scenario III

- Combining two matrices should give clear signal on what to cook.
- Food Matrix: Whats the food for today? Whats for tomorrow?
- Weather Matrix: Should I cook today? or tomorrow?

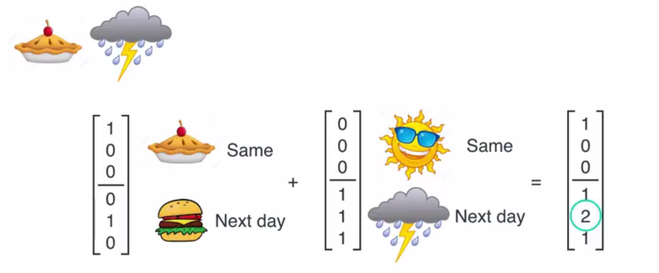


$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ \hline 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} + \begin{bmatrix} 1 & 0 \\ 1 & 0 \\ 1 & 0 \\ \hline 0 & 1 \\ 0 & 1 \\ 0 & 1 \end{bmatrix}$$

Toy Example: Scenario III

Testing

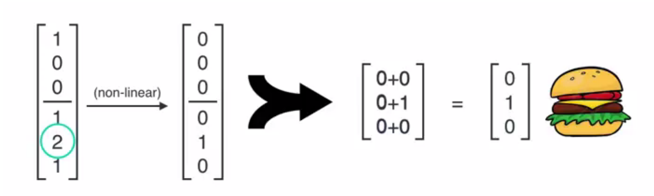
- Say, we had Apple Pie yesterday and its rainy today
- Food Matrix is computed and added to Weather Matrix



Toy Example: Scenario III

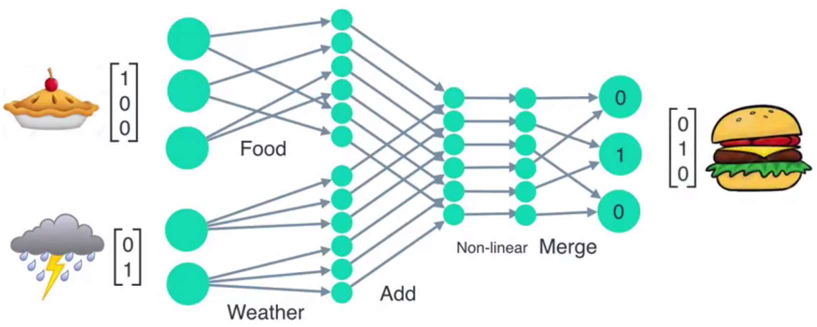
Merge

- We apply non linear function that takes input of the combined results
- Maps it to uniform one-hot like result.
- It make the LARGEST as 1, else all to 0
- Add sub-components and produces the result



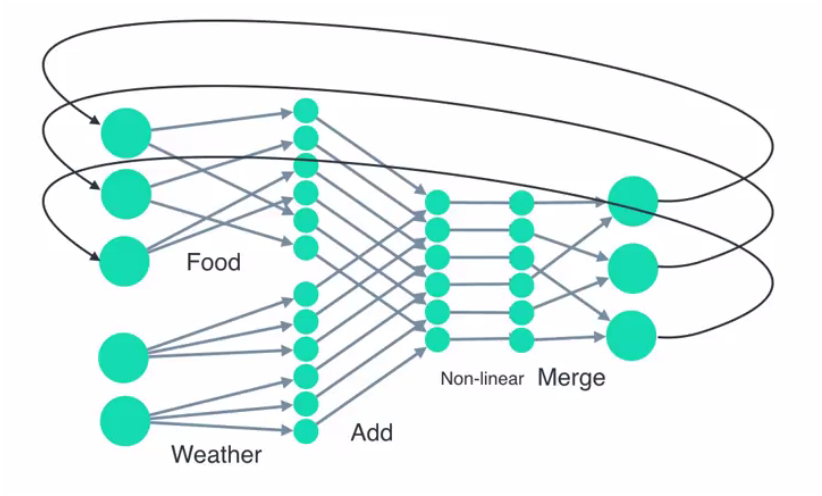
Toy Example: Scenario III

Summary (shown arrows are 1)



Toy Example: Scenario III

Recurrent

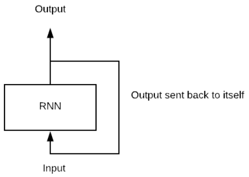


Theory of RNN

What is RNN?

A Neural network with a memory-state is added to the neurons.

- When data is sequential
- ANN: model produces the output by multiplying the input with the weight and the activation function
- RNN: output is sent back to itself and this, along with the new input is used to calculate the new output.
- The network is called 'recurrent' because it performs the same operation in each timestep.
- Example: Stock Predictions, Language Model (next word prediction), etc.



What is RNN?

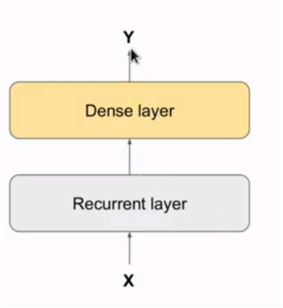
When:

- Order is important
- Variable length
- Sequential data
- Each item is processed in context

What is RNN?

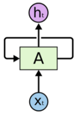
Usage of RNN layer:

- Input data points (X) one at a time.
- Simple cell is just one node and an activation function, thats it.
- GRU (Gated Recurrent Units) and LSTM (Long Short Term Memory) are two other types of RNN cells.
- RNN layer will process sequential data. Output is flattened vector
- Dense Layer is normal ANN, for prediction or classification (Y)

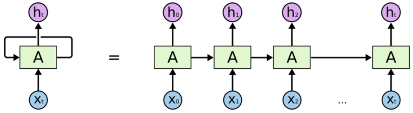


What is RNN?

A recurrent neural network can be thought of as multiple copies of the same network, each passing a message to a successor. The diagram above shows what happens if we unroll the loop.



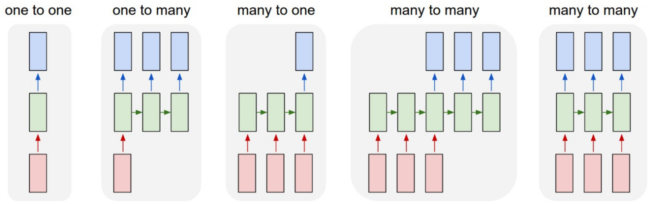
Recurrent Neural Networks have loops.



An unrolled recurrent neural network.

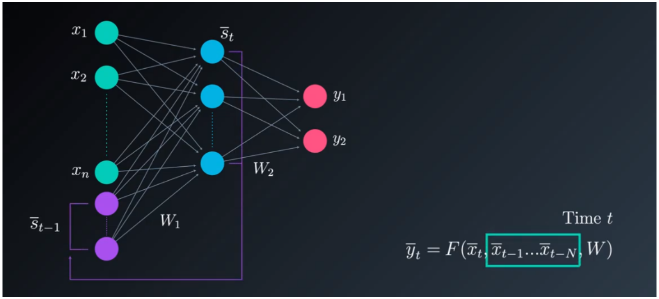
What is RNN?

RNN allow us to operate over sequences of vectors. Many types



Each rectangle is a vector and arrows represent functions (e.g. matrix multiply). Input vectors are in red, output vectors are in blue and green vectors hold the RNN's state (hidden layer nodes are called 'state' here as it has memory)

What is RNN?


x_1
x_2
x_n
$\overline{s}_t$
$\overline{s}_{t-1}$
y_1
y_2
W_1
W_2
Time t
$\overline{y}_t = F(\overline{x}_t, \boxed{\overline{x}_{t-1} \ldots \overline{x}_{t-N}}, W)$

(Ref: RNN - Udacity)

RNN can be imagined to be a usual ANN, with additional nodes from next layer back into inputs.

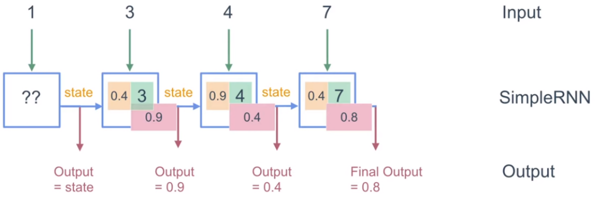
Memory in RNN

What is Neural Memory?

- Neural networks have hidden layers. Normally, the state of your hidden layer is based ONLY on your input data.
- So, normally a neural network's information flow would look like this: *input* $\rightarrow$ *hidden* $\rightarrow$ *output*
- This is straightforward. Certain types of input create certain types of hidden layers.
- Certain types of hidden layers create certain types of output layers.
- It's kind-of a closed system. Memory changes this.

What is Neural Memory?

- Memory means that the hidden layer is a combination of your input data at the current time-step and the hidden layer of the previous timestep. $(input + prev\_hidden) \rightarrow hidden \rightarrow output$
- Why the hidden layer? Well, we could technically do this. $(input + prev\_input) \rightarrow hidden \rightarrow output$

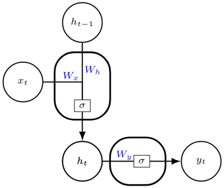


(Ref: Weights and Biases : Introduction to RNNs)

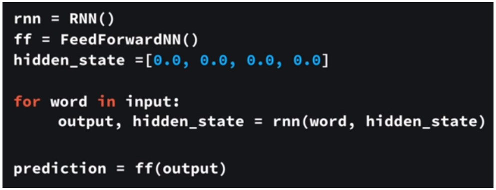
What is Neural Memory?

Each cell

- $h_t = tanh(W_{hh}h_{t-1} + W_{xh}X_t + b_h)$
- $y_t = tanh(W_{hy}h_t + b_y)$
- Back-propagation will adjust all the weights for all unrolled time-steps.



(Ref: RNN with Keras: Understanding computations - Alexis Huet)

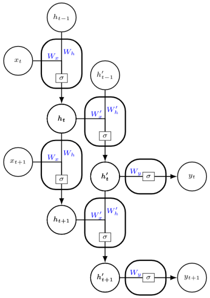


```
rnn = RNN()
ff = FeedForwardNN()
hidden_state =[0.0, 0.0, 0.0, 0.0]

for word in input:
    output, hidden_state = rnn(word, hidden_state)

prediction = ff(output)
```

RNN with 2 hidden layers or units

Each cell

- $h'_t = tanh(W_{hh}h_{t-1} + W_{xh}X_t + b_h)$
- $h'_t = tanh(W'_{hh}h'_{t-1} + W'_{xh}X_t + b'_h)$
- $y_t = tanh(W_{hy}h'_t + b_y)$
- Back-propagation will adjust all the weights for all unrolled time-steps.



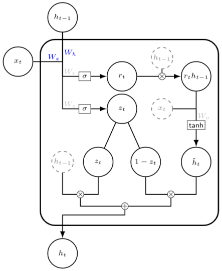
(Ref: RNN with Keras: Understanding computations - Alexis Huet)

GRU cell

Each cell

$$
\begin{aligned}
z_t &= \sigma(W_{zx}x_t + W_{zh}h_{t-1} + b_z) \\
r_t &= \sigma(W_{rx}x_t + W_{rh}h_{t-1} + b_r) \\
\tilde{h}_t &= \tanh(W_{ox}x_t + W_{oh}r_t h_{t-1} + b_o)
\end{aligned}
$$

$$
h_t = z_t h_{t-1} + (1 - z_t)\tilde{h}_t
$$



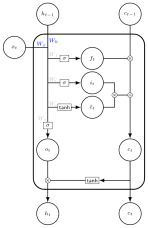
(Ref: RNN with Keras: Understanding computations - Alexis Huet)

LSTM cell

Each cell

$$
\begin{aligned}
i_t &= \sigma(W_{ix}x_t + W_{ih}h_{t-1} + b_i)\\
f_t &= \sigma(W_{fx}x_t + W_{fh}h_{t-1} + b_f)\\
\tilde{c}_t &= \tanh(W_{cx}x_t + W_{ch}h_{t-1} + b_c)\\
o_t &= \sigma(W_{ox}x_t + W_{oh}h_{t-1} + b_o)
\end{aligned}
$$

$$
\begin{aligned}
c_t &= f_t c_{t-1} + i_t\tilde{c}_t\\
h_t &= o_t \tanh(c_t)
\end{aligned}
$$



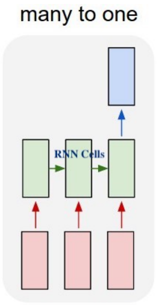
(Ref: RNN with Keras: Understanding computations - Alexis Huet)

Implementation of RNN

RNN Inputs Specification

FIXED sequence length case: Say,

- Dataset has 10000 rows and 64 features/columns
- Need to generate classification model, binary, so labels are 0 or 1.
- How does RNN handle this matrix (i.e., (1000,64))? Does it input each column something like this figure?
- Should SimpleRNN() units always be equal to the number of features?



RNN Inputs Specification

- batch_input_shape to be with shape: `(batch_size, X.shape[1], 1)`
- So batch size* 1000 examples will go at a time. Total, 10 such batches.
- Each of the 1000 example contains `X.shape[1]` time-stamps (number of pink boxes in your image, same as number of features, ie 64) and each time-stamp is shape 1 (scalar). Dimensionality of that entity is 1. It can be a vector as well.
- So input shape is `(1000,64,1)`
- The larger a batch, the more representative the computed gradient is for the whole data set, but larger memory requirements

(More info: https://stats.stackexchange.com/questions/153531/what-is-batch-size-in-neural-network)

RNN Inputs Specification: Batch Size, Num Steps

- 'batch size' pertains to the amount of training samples to consider at a time for updating your network weights. Training can be erratic if weights are updated for each row. After a training pass of size batch_size, the gradient of your loss function with respect to each of the network parameters is computed and your weights updated.
- Till the batch is over, losses per training sample are either averaged or accumulated, with that values, weights are updated. (Ref: https://stats.stackexchange.com/questions/174708/how-are-weights-updated-in-the-batch-learning-method-in-neural-networks)
- In RNN, it is computationally very, very expensive to calculate the gradients of the loss function with respect to network parameters if you have to consider back-propagation through all states since the creation of your network, there is a neat little trick to speed up your computation: approximate your gradients with a subset of historical network states num_steps.

*

(More info: https://stackoverflow.com/questions/44381450/doubts-regarding-batch-size-and-time-steps-in-rnn)

RNN Inputs Specification: Batch Size, Num Steps

Example: data size of 100; batch size of 5, for 30 network parameter updates during each epoch.

- Propagates the first 5 training examples, updates its parameters based on the optimization method provided, then takes the next 5, until it made a full pass over the data.
- num_steps determines the amount of cells you unroll and hence the amount of data used in gradient computation.
- As each cell/layer shares parameters, this does not result in an increase in parameters to optimize, but it enables context learning, which is why you'd want RNNs in the first place

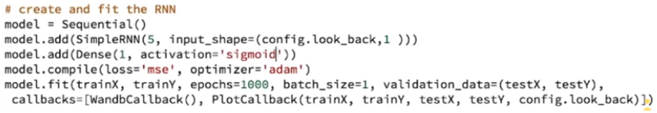
So, NUM STEPS is the number of 'features' you LOOK BACK (in case of text-words input) or consider while computing gradients and updating the weights.
*

(More info: https://stackoverflow.com/questions/44381450/doubts-regarding-batch-size-and-time-steps-in-rnn)

RNN LAYER Specification

RNN Layer can have multiple hidden states, physically but internally to the cell. Each physical RNN cell (called units) unrolls time-step wise.

- Do we have to have RNN units, ie green, boxes, sames as number of columns or features, ie 64.
- No! units will be your output dim. Thats inside direction, its about number of internal state values, which are given as outputs as well as hidden states passed to the next time-step.
- This is '. . . to MANY" architecture, for predicting next n outputs.
- Units will be the shape of the RNN's internal states.



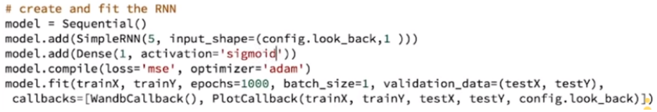
```
# create and fit the RNN
model = Sequential()
model.add(SimpleRNN(5, input_shape=(config.look_back,1 )))
model.add(Dense(1, activation='sigmoid'))
model.compile(loss='mse', optimizer='adam')
model.fit(trainX, trainY, epochs=1000, batch_size=1, validation_data=(testX, testY),
 callbacks=[WandbCallback(), PlotCallback(trainX, trainY, testX, testY, config.look_back)])
```

(Ref: Weights and Biases : Introduction to RNNs)

RNN LAYER Specification

RNN Layer can have multiple hidden states, physically but internally to the cell. Each physical RNN cell (called units) unrolls time-step wise.

- So, if you declare units=2000 your output will be (1000,2000).)
- Each weight matrix will be num features x num outputs.
- Each output will get tuned, optimized as per its given output label.
- More units, is better to model the function. In case you want singular output you can still have more units and then later, add a dense layer to reduce the dimension to 1.



```
# create and fit the RNN
model = Sequential()
model.add(SimpleRNN(5, input_shape=(config.look_back,1 )))
model.add(Dense(1, activation='sigmoid'))
model.compile(loss='mse', optimizer='adam')
model.fit(trainX, trainY, epochs=1000, batch_size=1, validation_data=(testX, testY),
 callbacks=[WandbCallback(), PlotCallback(trainX, trainY, testX, testY, config.look_back)])
```

(Ref: Weights and Biases : Introduction to RNNs)

RNN Inputs Specification

VARIABLE sequence length case: Say,

- Can either PAD the sequence with '0's and make it FIXED size, for each batch (or overall), else specify 'None' for that variable for time steps in Keras. If you don't want '0' padding, then you can specify sample weights.
- Now, suppose length of training examples in

$$batch - 1 = 10,$$
$$batch - 2 = 15.$$
$$batch - 3 = 18 \ldots$$

- Now while training batch-1 , RNN will be unrolled/looped 10 times. For batch-2, 15 times and so on. The number of weights do not change, or even the network architecture does not change, but just the times same architecture is looped, changes in each batch.

(Ref: https://datascience.stackexchange.com/questions/26366/training-an-rnn-with-examples-of-different-lengths-in-keras).

RNN Inputs Specification

- But, within a batch, do we need same fixed length of all sequences?
- Because Tensorflow (or any other framework) works on tensors/arrays and those can't be "ragged" like $[[1, 2, 3], [4, 5]]$.
- They need to be rectangular. Thats why within a batch, it needs to be fixed size.
- Anyway achieved by num steps parameter, it extends or truncates.
- The reason a fixed length is used in keras, is because it greatly improves performance by creating tensors of fixed shapes.
- But that's only for training. After training, you'll have learned the right weights for your task.

(Ref: https://datascience.stackexchange.com/questions/26366/training-an-rnn-with-examples-of-different-lengths-in-keras).

RNN Model

Code:

```
X = X.reshape((X.shape[0], X.shape[1], 1)) # IT HAS TO BE 3D, so fake 1 dim
     added
tr_X, ts_X, tr_y, ts_y = train_test_split(X, y, train_size=.8)
batch_size = 1000

model = Sequential()
model.add(SimpleRNN(64, activation='relu', batch_input_shape=(batch_size,
     X.shape[1], 1)))
model.add(Dense(1, activation='relu'))

model.compile(loss='binary_crossentropy', optimizer='adam',
     metrics=['accuracy'])

model.fit(tr_X, tr_y,
          batch_size=batch_size, epochs=1,
          shuffle=True, validation_data=(ts_X, ts_y))
```

RNN Parameters calculations

- num_units = equals the number of units in the RNN
- num_features = equals the number features of your input
- input_weights = num_features*num_units
- recurrent_weights = num_units*num_units
- biases = num_units*1
- total params = recurrent_weights + input_weights + biases

RNNs Model Summary

```
Layer (type)                 Output Shape              Param #
=================================================================
simple_rnn_1 (SimpleRNN)     (1000, 64)                4224
_________________________________________________________________
dense_1 (Dense)              (1000, 1)                 65
=================================================================
Total params: 4,289
Trainable params: 4,289
Non-trainable params: 0
_________________________________________________________________
```

Conclusions

RNNs advantages

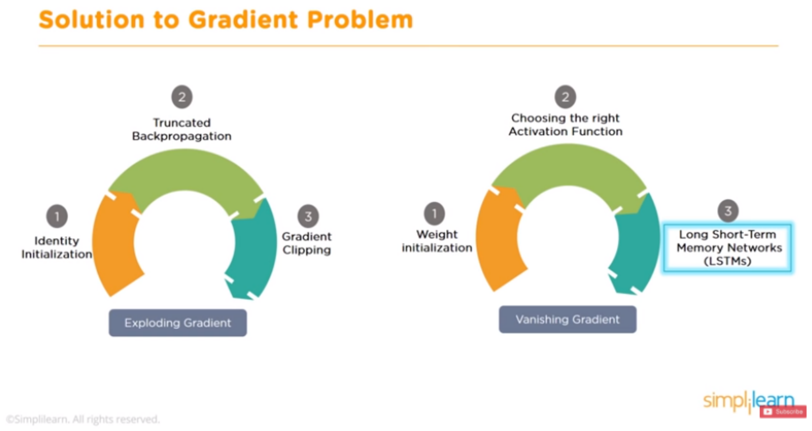
- The applications of standard Neural Networks (and also Convolutional Networks) are limited due to:
- They only accepted a fixed-size vector as input (e.g., an image) and produce a fixed-size vector as output (e.g., probabilities of different classes).
- These models use a fixed amount of computational steps (e.g. the number of layers in the model).

RNNs advantages

- Recurrent Neural Networks are unique as they allow us to operate over sequences of vectors.
- Sequences in the input, the output, or in the most general case both

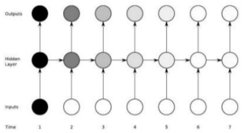
RNN Shortcomings

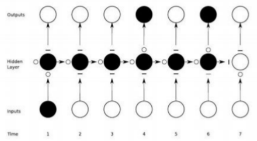
- RNN does not work in some situations
- Especially when long past words need to be accounted for
- Due to Vanishing Gradient problem.


Solution to Gradient Problem
2
Truncated Backpropagation
1
Identity Initialization
3
Gradient Clipping
Exploding Gradient
2
Choosing the right Activation Function
1
Weight initialization
3
Long Short-Term Memory Networks (LSTMs)
Vanishing Gradient
simplilearn
©Simplilearn. All rights reserved.

RNN vanishing gradients

- Vanilla RNNs suffers at vanishing gradients problems.
- A modest recurrent neural network may have 200-to-400 input time steps, resulting conceptually in a very deep network.
- problem: The derivative the activation nonlinearities, sigmoid or tanh, is smaller than 1. So, after some timesteps, very minute gradient remains to make any weight changes. In further layers, weights do not change. So, NO TRAINING.
- Solution: the hidden transfer function should be a linear function. Relu (rectilinear activation function) looks and acts like a linear function, making it easier to train and less likely to saturate, but is, in fact, a nonlinear function. The ReLU derivative is a constant of either 0 or 1, so it isn't as likely to suffer from vanishing gradients.





How LSTM solves Vanishing Gradient

- LSTM increases memory network, handles more information, longer memory.
- LSTM network has three gates that update and control the cell states.
- So if we want Gradient not to vanish, our network needs to increase the likelihood that at least some of these sub gradients will not vanish.
- Forget gate sub-gradient does not behave similarly with other sub-gradient, so they all do not converge to zero.

(Ref: How LSTM networks solve the problem of vanishing gradients - Nir Arbel)

Summary

- RNNs suffer from vanishing gradients and caused by long series of multiplications of small values, diminishing the gradients and causing the learning process to become degenerate.
- LSTMs solve the problem using a unique additive gradient structure that includes direct access to the forget gate's activations, enabling the network to encourage desired behaviour from the error gradient using frequent gates update on every time step of the learning process.

(Ref: How LSTM networks solve the problem of vanishing gradients - Nir Arbel)

RNN with TensorFlow

Toy Example: Predict sum of 3 numbers

Problem Description

- Aim: to predict the sum of 3 numbers with RNN
- Inputs: samples like $[x_0, x_1, x_2][x_0, x_1, x_2]$
- Outputs: like $y = x_0 + x_1 + x_2$

```python
import numpy as np
from tf.keras.models import Model
from tf.keras.layers import Input, SimpleRNN
# Data and model parameters
seq_len = 3   #Length of each sequence
rnn_size = 1  #Output shape of RNN
input_size = 10000 #Numbers of instances
```

Creating Data

```
all_feat = np.random.randint(low=0, high=10, size=(input_size,3,1))
print(all_feat[:2, :])  # top 2, all values

array([[[9],
        [0],
        [1]],
       [[7],
        [6],
        [4]]])

all_label = np.apply_along_axis(func1d=np.sum, axis=1, arr=all_feat)
print(all_label[:2])

array([[10],
       [17]])
```

Define model

- Using only a Simple RNN
- Expectation with RNN is that it will learn to pass the input as it is to next layer and do the SUM of the sequence.
- LINEAR activation is used (obviously)

```
x = Input(shape=(3,1,), name='Input_Layer') # 1, as each number is single and
     not a vector
y = SimpleRNN(rnn_size, activation='linear', name='RNN_Layer')(x)
model = Model(inputs=x, outputs=y)
model.summary()

_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
Input_Layer (InputLayer)     (None, 3, 1)              0
_________________________________________________________________
RNN_Layer (SimpleRNN)        (None, 1)                 3
=================================================================
Total params: 3.0
Trainable params: 3.0
Non-trainable params: 0.0
_________________________________________________________________
```

Fit model

```
model.compile(optimizer='adam', loss='mean_squared_error', metrics=['acc'])
history = model.fit(x=all_feat, y=all_label, batch_size=4, epochs=5,
      validation_split=0.2, verbose=1)

Train on 8000 samples, validate on 2000 samples
Epoch 1/5
8000/8000 [==============================] - 9s - loss: 52.2708 - acc: 0.1204
      - val_loss: 2.8959 - val_acc: 0.2050
Epoch 2/5
8000/8000 [==============================] - 8s - loss: 2.3057 - acc: 0.2139 -
      val_loss: 1.5666 - val_acc: 0.2775
Epoch 3/5
8000/8000 [==============================] - 8s - loss: 0.9501 - acc: 0.3466 -
      val_loss: 0.4068 - val_acc: 0.5105
Epoch 4/5
8000/8000 [==============================] - 9s - loss: 0.1705 - acc: 0.7825 -
      val_loss: 0.0324 - val_acc: 1.0000
Epoch 5/5
8000/8000 [==============================] - 9s - loss: 0.0084 - acc: 1.0000 -
      val_loss: 1.4700e-04 - val_acc: 1.0000
```

Prediction

```
print('\nInput features: \n', all_feat[-2:,:])
print('\nLabels: \n', all_label[-2:,:])
print('\nPredictions: \n', model.predict(all_feat[-2:,:]))

Input features:
 [[[0]
  [9]
  [3]]
 [[4]
  [2]
  [5]]]
Labels:
 [[12]
 [11]]
Predictions:
 [[ 12.00395012]
 [ 11.0082655 ]]
```

Deep Dive into RNN

- W: Input to RNN weight Matrix
- U: RNN to RNN (or hidden layer to RNN) weight Matrix
- b: Bias matrix

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

[array([[ 0.99675155]], dtype=float32),
 array([[ 1.00106668]], dtype=float32),
 array([ 0.01110852], dtype=float32)]
```

RNN equation is $h_t = f(X \times W + h_{t-1} \times U + b)$.
With linear f, it becomes $h_t = X \times W + h_{t-1} \times U + b$
So, $W = 1, U = 1, b = 0$ looks more or less correct.

References

Many publicly available resources have been refereed for making this presentation. Some of the notable ones are:

- TensorFlow 2 Tutorial: Get Started in Deep Learning With tf.keras - Jason Brownlee
- Deep Learning using Keras- Alyosamah
- Introduction to Keras - Francois Chollet
- Michael Nielsen's Neural Networks and Deep Learning: `http://neuralnetworksanddeeplearning.com/`

Thanks ... yogeshkulkarni@yahoo.com