5. Recurrent Neural Network

5.1 What is RNN?

A recurrent neural network (RNN) has looped, or recurrent, connections which allow the network to hold information across inputs. These connections can be thought of as similar to memory. RNNs are particularly useful for learning sequential data like music.

In TensorFlow, the recurrent connections in a graph are unrolled into an equivalent feed-forward network. That network is then trained using a gradient descent technique called backpropagation through time (BPTT).

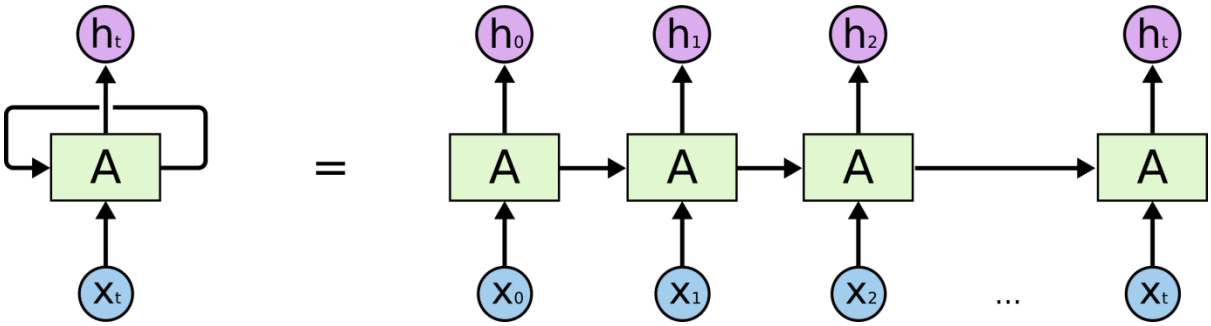


Figure 8 RNN

There are endless ways that an RNN can connect back to itself with recurrent connections. People typically stick to a few common patterns, the most common being Long Short-Term Memory (LSTM) cells and Gated Recurrent Units (GRU).

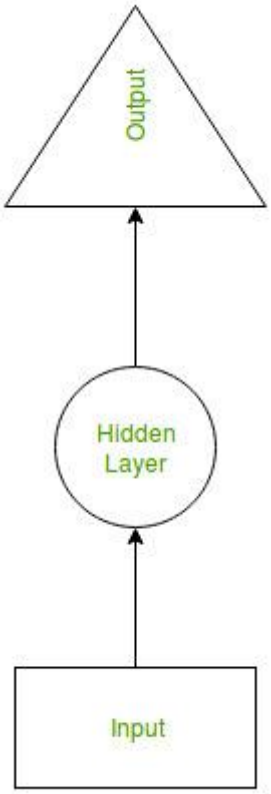
These both have multiplicative gates that protect their internal memory from being overwritten too easily, allowing them to handle longer sequences. We use LSTMs in this model.

A recurrent neural network is a class of artificial neural networks that make use of sequential information.

They are called recurrent because they perform the same function for every single element of a sequence, with the result being dependent on previous computations. Whereas outputs are independent of previous computations in traditional neural networks.

5.2 Recurrent Neural Network Architecture

Recurrent Neural Network(RNN) are a type of Neural Network where the **output from previous step are fed as input to the current step**. In traditional neural networks, all the inputs and outputs are independent of each other, but in cases like when it is required to predict the next word of a sentence, the previous words are required and hence there is a need to remember the previous words. Thus RNN came into existence, which solved this issue with the help of a Hidden Layer. The main and most important feature of RNN is **Hidden state**, which remembers some information about a sequence.



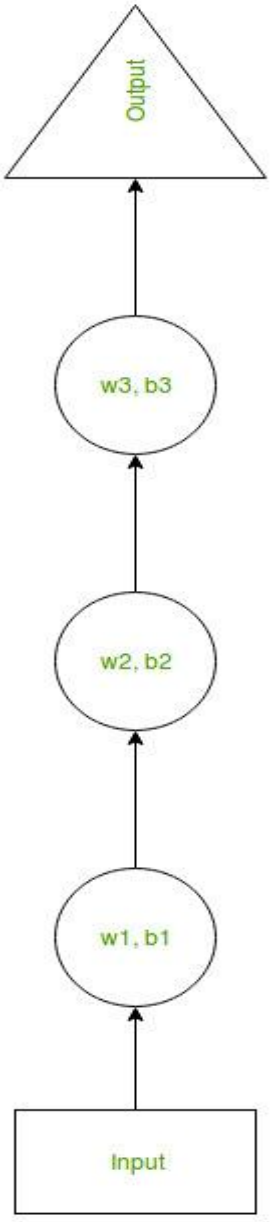
RNN have a **"memory"** which remembers all information about what has been calculated. It uses the same parameters for each input as it performs the same task on all the inputs or hidden layers to produce the output. This reduces the complexity of parameters, unlike other neural networks.

5.3 How RNN works?

The working of a RNN can be understood with the help of below example:

Example:

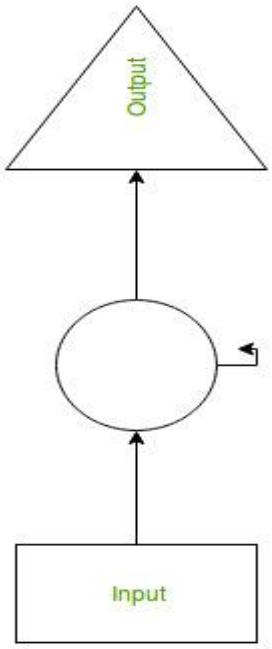
Suppose there is a deeper network with one input layer, three hidden layers and one output layer. Then like other neural networks, each hidden layer will have its own set of weights and biases, let's say, for hidden layer 1 the weights and biases are (w1, b1), (w2, b2) for second hidden layer and (w3, b3) for third hidden layer. This means that each of these layers are independent of each other, i.e. they do not memorize the previous outputs.



Now the RNN will do the following:

- RNN converts the independent activations into dependent activations by providing the same weights and biases to all the layers, thus reducing the complexity of increasing parameters and memorizing each previous outputs by giving each output as input to the next hidden layer.

Hence these three layers can be joined together such that the weights and bias of all the hidden layers is the same, into a single recurrent layer.



- **Formula for calculating current state:**

$$h_t = f(h_{t-1}, x_t)$$

- **Formula for applying Activation function(tanh):**

$$h_t = \tanh (W_{hh}h_{t-1} + W_{xh}x_t)$$

- **Formula for calculating output:**

$$y_t = W_{hy}h_t$$

5.4 Training through RNN

1. A single time step of the input is provided to the network.
2. Then calculate its current state using set of current input and the previous state.
3. The current ht becomes ht-1 for the next time step.
4. One can go as many time steps according to the problem and join the information from all the previous states.
5. Once all the time steps are completed the final current state is used to calculate the output.
6. The output is then compared to the actual output i.e the target output and the error is generated.
7. The error is then back-propagated to the network to update the weights and hence the network (RNN) is trained.

5.5 Advantages of Recurrent Neural Network

1. An RNN remembers each and every information through time. It is useful in time series prediction only because of the feature to remember previous inputs as well. This is called Long Short Term Memory.
2. Recurrent neural network are even used with convolutional layers to extend the effective pixel neighborhood.

5.6 Disadvantages of Recurrent Neural Network

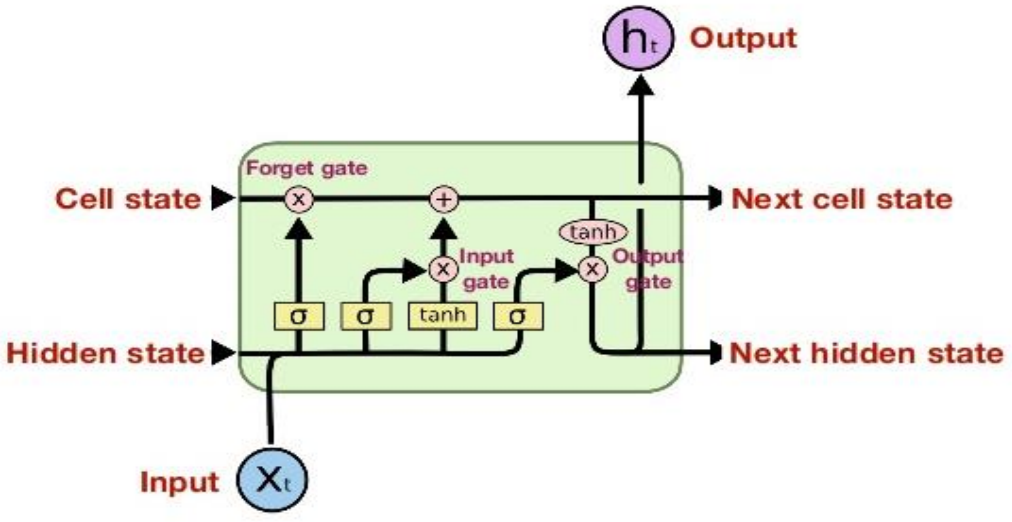
1. Gradient vanishing and exploding problems.
2. Training an RNN is a very difficult task.
3. It cannot process very long sequences if using tanh or relu as an activation function.

5.7 Applications of RNN

RNNs are powerful machine learning models and have found use in a wide range of areas. It is distinctly different from CNN models like GoogleNet. In this article, we have explored the different applications of RNNs in detail. The main focus of RNNs is to use sequential data.

RNNs are widely used in the following domains/ applications:

- Prediction problems
- Language Modelling and Generating Text
- Machine Translation
- Speech Recognition
- Generating Image Descriptions
- Video Tagging
- Text Summarization
- Call Center Analysis
- Face detection, OCR Applications as Image Recognition
- Other applications like Music composition



6. Long Short Term Memory (LSTM)

6.1 Introduction

Sequence prediction problems have been around for a long time. They are considered as one of the hardest problems to solve in the data science industry.

These include a wide range of problems; from predicting sales to finding patterns in stock markets' data, from understanding movie plots to recognizing your way of speech, from language translations to predicting your next word on your iPhone's keyboard.

With the recent breakthroughs that have been happening in data science, it is found that for almost all of these sequence prediction problems, Long short Term Memory networks, a.k.a LSTMs have been observed as the most effective solution.

LSTMs have an edge over conventional feed-forward neural networks and RNN in many ways. This is because of their property of selectively remembering patterns for long durations of time. The purpose of this article is to explain LSTM and enable you to use it in real life problems.

6.2 Architecture of LSTMs

The functioning of LSTM can be visualized by understanding the functioning of a news channel's team covering a murder story. Now, a news story is built around facts, evidence and statements of many people. Whenever a new event occurs you take either of the three steps.

Let's say, we were assuming that the murder was done by 'poisoning' the victim, but the autopsy report that just came in said that the cause of death was 'an impact on the head'. Being a part of this news team what do you do? You immediately **forget** the previous cause of death and all stories that were woven around this fact.

Now, this is nowhere close to the simplified version which we saw before, but let me walk you through it. A typical LSTM network is comprised of different memory blocks called **cells**

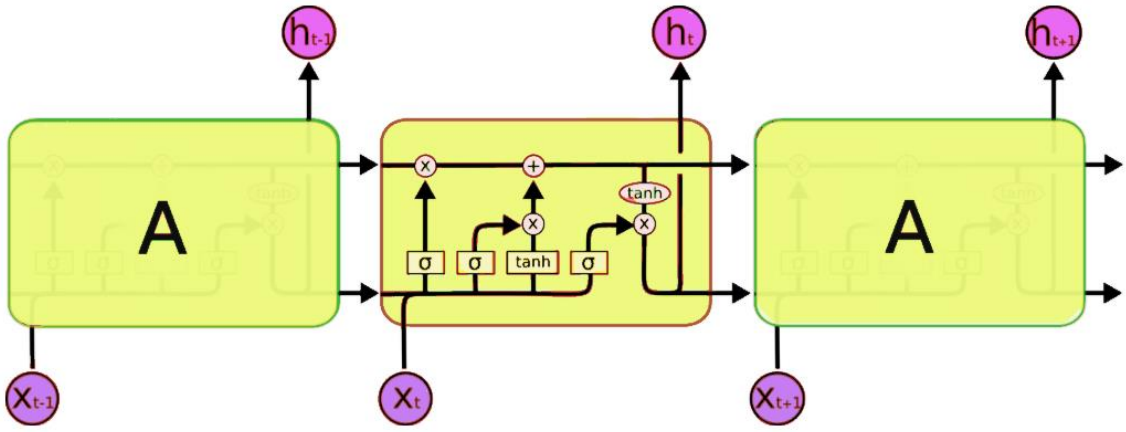
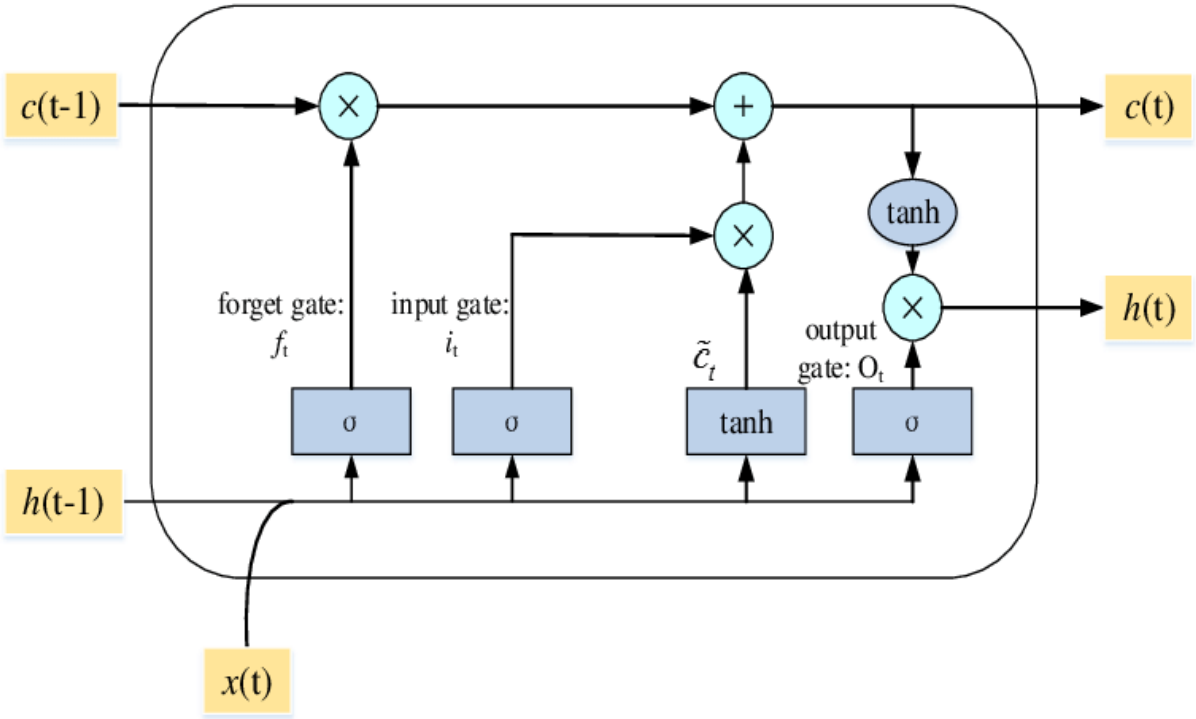


Figure 9 Architecture of LSTM

Now all these broken pieces of information cannot be served on mainstream media. So, after a certain time interval, you need to summarize this information and **output** the relevant things to your audience. Maybe in the form of "*XYZ turns out to be the prime suspect.*".



7. ABC Notation

The **ABC notation** is a text-based (ASCII) musical notation format commonly used for folk and traditional music. It is an alphabetical shorthand, using the letters A to G to represent notes and other numbers and symbols to represent note length, sharps, flats, ornaments, etc.

While alphabetical notation systems have been used informally for a number of years, the current standard is based on work done by Chris Walshaw in the early 1990s and later revised and extended by a number of other users. The current standard is v2.1 (December 2011), with a draft v2.1.1 standard published in February 2013.

7.1 Structure

Every ABC file must start with the string "%abc", which may optionally be followed by a version identifier e.g. "%abc-2.1". In order to maintain backwards-compatibility with files created according to older standards, there are two ways in which ABC files may be interpreted. **Strict interpretation** applies to files with a version number of 2.1 or greater, with errors reported to the user. **Loose interpretation** applies to files with either a version number of 2.0 or lower, or to files with no version number.

ABC files consist of three basic parts - the file header, the tune header(s) and the tune body / bodies. The file header contains processing information for the entire file e.g. typesetting and stylesheets, while the tune header contains metadata and other information such as the key, meter and standard note length of the following tune. The tune body consists primarily of ABC music code, but possibly also contains certain information fields.

7.2 File Header

The file header contains processing information for the entire file. While many of the available information fields can be used in the file header, we have covered those mainly in the Tune Header section. However, we will look here at the **I** (instruction) field which has a number of uses.

I:abc-charset can be used to indicate the character encoding used by the file. Legal values are iso-8859-1 to iso-8859-10, us-ascii and utf-8. This cannot be changed further on in the file.

I:abc-version can be used to indicate tunes conforming to a different standard than the one indicated at the start of the file. Presumably this appears most commonly in tune headers - while it can be used in the file header, it is hard to imagine a scenario in which this might be appropriate.

I:abc-include can be used to import information kept in a separate header (**.abh**) into the file or tune header.

I:abc-creator should contain the name and version number of the software used to create the ABC file (if software was used)

I:linebreak can be used to specify which symbol (or set of symbols) should be used to indicate score line-breaks (i.e. those line-breaks which occur in the typeset / printed score rather than those which occur in the raw code). There are four possible options for this field - $, !, <EOL> (use code line-breaks), <none> (carry out line-breaking automatically). Note that these can be combined - "I:linebreak % !" is a legal option. The default option is "I:linebreak <EOL> $"

I:decoration can be used to specify which of two symbols should be used to delimit decorations. The default symbol is !, but it can be set to + with this field. Note that "I:decoration +" is automatically invoked if "I:linebreak !" is set.

7.3 Tune Header

The header of an ABC file consists of a number of fields defining metadata and other information. A complete header using every field might look like this:

```
X:1
T:Sample Song
C:A. Composer
O:America; Nebraska; Power Cable.
A:Area
M:6/8
```

```
L:1/4
Q:"Allegro" 1/4=120
P:ABAB
Z:Tran Scriber, <t.s@notarealemail.com>
N:Notes
G:Group
H:This song was traditionally played
+:at the Power Cable, Nebraska Boot Festival
R:Polka
B:Tunes of Power Cable, Nebraska vol. I
D:Traditional Folk Music of Nebraska
F:this.is.not.a.real.url/song.abc
S:Source
K:D minor
```

X is the reference number. As ABC files can contain a number of tunes, this is used to identify the first, second, third, etc. tunes in the file. Thus, the first tune would be X:1, the second X:2, and so on.

T is the song title. As you would expect, this is used to store the title of the song.

C is the composer. Again, as expected, this contains the name of the composer (or, e.g. "Trad." for traditional songs where the original composer's name is lost)

O is the geographic origin of a tune. The data is entered in a hierarchical manner with a semi-colon to separate elements[

- **A** is the area field. This is now deprecated and it is recommended that specific information about a tune's area of origin be entered in the **O** field
- **M** is the meter. Information about the meter can be entered in one of three ways. The first is standard - M:6/8 or M:3/4, for example. There are also special symbols M:C and M:C| to represent common time and cut time respectively. Finally, complex meters can be specified in the format M:(2+3+2)/8 - note that the parentheses are optional.
- **L** is the unit note length i.e. what length of note a single letter represents in an ABC file. This uses standard values for note length - L:1/4 is a quarter note or crotchet, L:1/8 is an eighth note or quaver, etc. Note values L:1/64 and lower are optional -

they may not be supported by all ABC software. If no note field is specified, the note length is calculated based on the meter.

- **Q** is the tempo, in terms of beats per minute. In the simplest form, this would be something like Q:1/2=120 (120 half-note beats per minute), but the definition can contain up to four beats e.g. Q:1/4 3/8 1/4 3/8=40. Tempo definitions may also contain a string, enclosed by quotes e.g. Q:"Allegro" 1/4=120. It is also acceptable to provide a string with no explicit tempo indication e.g. Q:"Slowly"
- **P** is the parts field. This is currently used in two places - in the header, to indicate the order in which parts of tunes are played (P:ABAB), and within the tune body, to mark each part (P:A). The syntax for the parts field can become relatively complex. The simplest form is to list each part of the tune in order, e.g. P:ABABCDABE. It is also possible to use numbers to indicate number of times to repeat a part - P:A3 is equivalent to P:AAA. Where things get tricky is with the introduction of parentheses. These can be use to indicate number of times to repeat a section - P:(AB)2 is equivalent to P:ABAB - and can also be nested, making it possible to condense fairly complex sequences of repetition - P:((AB)3(CD)3E)2 would expand to P:ABABABCDCDCDEABABABCDCDCDE. Full stops / periods can be used to separate individual elements for legibility.
- **Z** is the transcription field, containing information about the transcribers and editors of the tune. While there is no formal syntax for this field, the convention appears to be to use it to replace the earlier, deprecated fields **%%abc-copyright** and **%%abc-edited-by**. Note that, by convention, Z:abc-copyright will refer to the copyright on the transcription, not the original song.
- **N** is the notes fields. As the name suggests, this field is for general information about the tune or transcription not pertinent to any other field.
- **G** is the *group* field. It may be used to group together sets of tune, but there no current standard for use of this field
- **H** is the *history* field. It is used to contain anecdotes, historical information, etc. about the tune. Note that multi-line input should use **+** fields following on from the initial **H**, not more **H** fields (although multiple **H** fields are acceptable where one wishes to record multiple, distinct anecdotes)
- **R** is the rhythm field. This is used to give musicians an indication of the type of tune represented (e.g. polka)

- **B**, **D**, **F** and **S** contain background information about the tune and where it can be found - B:book, D:discography, F:file url, S:source
- **K** is the key. **The first occurrence of this field always marks the end of the header**. The basic structure of the key field is as follows: capital letter between A and G (key signature), # or b to indicate sharp or flat respectively (optional), mode (if none is specified, major is assumed). Note that for modes, the capitalisation is ignored and only the first three letters are parsed - K:F#MIX is equivalent to K:F# mixolydian. The key field also supports a number of more advanced parameters allowing for the specification of accidentals, clef type, etc. Note that it is possible to specify no key signature by using either an empty **K** field or K:none.

7.4 Tune Body

The "tune code" used by ABC is too complex to allow complete coverage of every element here. However, the table below will provide a brief overview of the various elements used.

Element	Usage
A-G	Upper-case letters represent notes on the bottom octave. To group notes together under one beam, they should be grouped together without spaces (A B C D will produce separate single notes, ABCD will produce beamed notes)
a-g	Lower-case letters represent notes on the top octave
,	Appearing after notes, lowers the note by one octave. Multiple commas may be used to lower multiple octaves e.g. A, B,, C,,,

'	Appearing after notes, raises the note by one octave. Behaves as commas
^	Appearing before notes, represents a sharp
=	Appearing before notes, represents a natural
_	Appearing before notes, represents a flat
1,2,3...	Used to create notes whose lengths are multiples of the length set in field L:, e.g. if note length is set to a sixteenth note, **A** represents a sixteenth note, **A2** an eighth note, **A4** a quarter note, etc.
/	Used to create notes whose lengths are fractions of the length set in field L: e.g. if note length is set to an eight note, **A/2** represents a sixteenth note. Note that these can be used either with numbers or independently - **A/** is equivalent to **A/2**, **A//** to **A/4**, etc.
>	One of two symbols used to denote broken rhythm. This is

	used in the form a>b to indicate that the note preceding it is dotted and the note succeeding it halved. This can be extended - a>>b, a>>>b, etc. - to indicate double-dotted / quartered, triple-dotted / divided by eight and so on
<	The second symbol used to denote broken rhythm. Used in the form a<b to indicate that the note preceding it is halved and the note succeeding it is dotted. Can be extended in the same form as **>**
z, Z, x, X	All of these denote rests and all can be lengthened / shortened in the same form as notes. "z" rests are printed in sheet music produced from the file, "x" rests are invisible. Capital letters are used to denote multi-measure rests (Z4 and z4|z4|z4|z4 are musically equivalent but will be printed differently)
`	An unprocessed mark which may be placed between notes to be beamed to increase legibility

|	Bar line
|]	Thin-thick double bar line
||	Thin-thin double bar line
[|	Thick-thin double bar line
|:	Start of repeated section. This can be extended - |:: means repeat a section three times, |::: four, etc.
:|	End of repeated section - can be extended as above
::	Start and end of two repeated sections. Functionally equivalent to :|: or :||:
.|	Dotted bar line
[|]	Invisible bar line
[1, [2	First and second repeats. Used in combinations with field P: (parts) this can be extended to denote multiple variant endings for a repeated section. In this situation, [may be followed by any list of numbers / ranges e.g. [1,3,5-7 <notes> will play <notes> on the 1st, 3rd, 5th, 6th

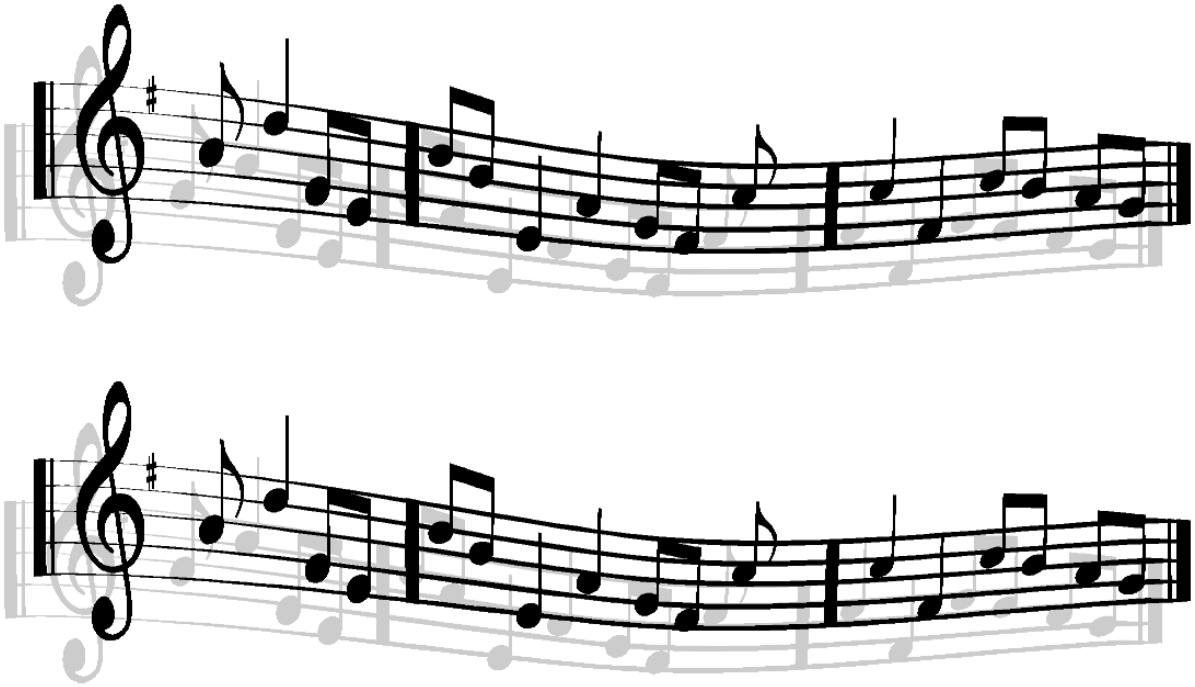
	and 7th repeats.
-	Used to tie two notes of the same pitch together
()	Denotes that the notes contained within are slurred. These can be nested e.g. (AB(ABCD)CD) and can also start and end on the same note - (AB(C)DE) denotes two separate slurs, the first ending and the second starting on C.
{}	Denotes the notes contained within are grace notes. Acciaccaturas (I had to look it up too) are notated with a forward slash immediately following the opening brace e.g. {/g}C
(2, (3 ... (9	Used to signify x-tuplets e.g. (3abc, (4abcd, etc.
(p:q:r	A more general form of the tuplets syntax, meaning put *p* notes into *q* time for the next *r* notes
.	Staccato mark
~, !roll!	Irish roll

H	Fermata
L	Accident or emphasis
M, !lowermordent!, !mordent!	Lowermordent
O, !coda!	Coda
P, !uppermordent!, !pralltriller!	Uppermordent
S, !segno!	Segno
T, !trill!	Trill. **!trill(!** is used for the start of an extended trill and **!trill)!** for the end
u, !upbow!	up-bow
v, !downbow!	down-bow
!turn!	turn mark / grupetto
!turnx!	as above with a line through it
!invertedturn!	inverted turn mark
!invertedturnx!	as above with a line through it
!arpeggio!	vertical squiqqle
!>!, !accent!, !emphasis!	> mark
!fermata!	hold / fermata
!invertedfermata!	inverted fermata

!tenuto!	horizontal line, indicates that note should be held for full duration
!0! - !5!	Fingerings
!+!, !plus!	left-hand pizzicato / rasp for French horn
!snap!	snap-pizzicato
!slide!	Slide
!wedge!	wedge mark
!open!	open string / harmonic
!thumb!	cello thumb symbol
!breath!	breath mark
!pppp!, !ppp!, !pp!, !p!, !mp!, !mf!, !f!, !ff!, !fff!, !ffff!, !sfz!	dynamics marks
!<(!, !crescendo(!	start of a crescendo mark
!<)!, !crescendo)!	end of a crescendo mark
!>(!, !diminuendo(!	start of a diminuendo mark
!>)!, !diminuendo)!	end of a diminuendo mark
!D.S.!	"D.S." (Da Segno)
!D.C.!	"D.C." (Da Coda / Da Capo)

!dacoda!	Da Coda
!dacapo!	Da Capo
!fine!	"fine"
!shortphrase!	vertical line on upper part of staff
!mediumphrase!	as above, extending to centre line
!longphrase!	as above, extending to 3/4 down
s:	Denotes that a symbol line should be used. These can be stacked.
H-W, h-w, ~	User-assignable symbols, using the U: field
[]	Denotes that the notes contained within are part of a chord
e.g. "Am7"	Chord symbols - these follow the syntax <note><accidental><type></bass > and should be contained within double quotes
e.g. "<note"	Textual annotations. These should be contained within double quotes and the string must be preceded by one of five

	symbols: ^, _, <, >, @ which control where the annotation is placed. These are, respectively: above, below, left or right of the following note, with "@" leaving the placement up to the interpreting program



8. Music Generation with RNNs

8.1 Dependencies

First, let's download the course repository, install dependencies, and import the relevant packages I'll need for this project.

```
import tensorflow as tf

!pip install mitdeeplearning
import mitdeeplearning as mdl

import numpy as np
import os
import time
import functools
from IPython import display as ipythondisplay
from tqdm import tqdm
!apt-get install abcmidi timidity > /dev/null 2>&1
```

8.2 Datasets

MIT have gathered a dataset of thousands of Irish folk songs, represented in the ABC notation.

```
songs = mdl.lab1.load_training_data()

example_song = songs[0]
print("\nExample song: ")
print(example_song)
```

Example song:

```
X:2
T:An Buachaill Dreoite
Z: id:dc-hornpipe-2
```

M:C|

L:1/8

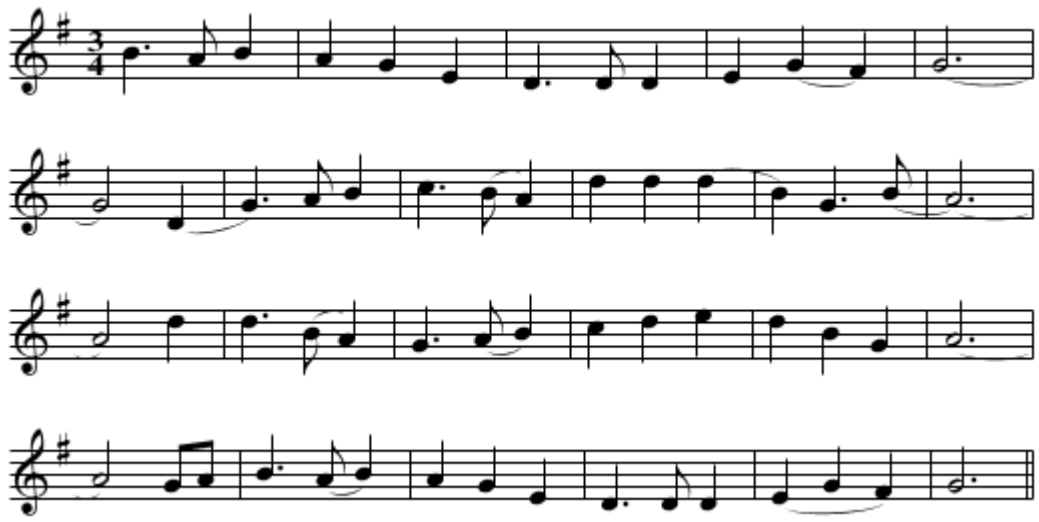
K:G Major

GF|DGGB d2GB|d2GF Gc (3AGF|DGGB d2GB|dBcA F2GF|!

DGGB d2GF|DGGF G2Ge|fgaf gbag|fdcA G2:|!

GA|B2BG c2cA|d2GF G2GA|B2BG c2cA|d2DE F2GA|!

B2BG c2cA|d^cde f2 (3def|g2gf gbag|fdcA G2:|!



We can easily convert a song in ABC notation to an audio waveform and play it back. It can take some time.

```
mdl.lab1.play_song(example_song)
songs_joined = "\n\n".join(songs)
vocab = sorted(set(songs_joined))
print("There are", len(vocab), "unique characters in the dataset")
```

8.3 Process the dataset for the learning task

Let's take a step back and consider our prediction task. We're trying to train a RNN model to learn patterns in ABC music, and then use this model to generate (i.e., predict) a new piece of music based on this learned information.

Breaking this down, what we're really asking the model is: given a character, or a sequence of characters, what is the most probable next character? We'll train the model to perform this task.

To achieve this, we will input a sequence of characters to the model, and train the model to predict the output, that is, the following character at each time step. RNNs maintain an internal state that depends on previously seen elements, so information about all characters seen up until a given moment will be taken into account in generating the prediction.

Vectorize the text

Before we begin training our RNN model, we'll need to create a numerical representation of our text-based dataset. To do this, we'll generate two lookup tables: one that maps characters to numbers, and a second that maps numbers back to characters. Recall that we just identified the unique characters present in the text.

```
char2idx = {u:i for i, u in enumerate(vocab)}
idx2char = np.array(vocab)
```

This gives us an integer representation for each character. Observe that the unique characters (i.e., our vocabulary) in the text are mapped as indices from 0 to len(unique). Let's take a peek at this numerical representation of our dataset:

```
print('{')
for char,_ in zip(char2idx, range(20)):
    print('  {:4s}: {:3d},'.format(repr(char), char2idx[char]))
print('  ...\n}')
def vectorize_string(string):
  vectorized_output = np.array([char2idx[char] for char in string])
  return vectorized_output
vectorized_songs = vectorize_string(songs_joined)
print ('{} ---- characters mapped to int ----
> {}'.format(repr(songs_joined[:10]), vectorized_songs[:10]))
assert isinstance(vectorized_songs, np.ndarray)
```

8.4 Create Training Examples and Targets

Our next step is to actually divide the text into example sequences that we'll use during training. Each input sequence that we feed into our RNN will contain seq_length characters from the text. We'll also need to define a target sequence for each input sequence, which will be used in training the RNN to predict the next character. For each input, the corresponding target will contain the same length of text, except shifted one character to the right.

To do this, we'll break the text into chunks of seq_length+1. Suppose seq_length is 4 and our text is "Hello". Then, our input sequence is "Hell" and the target sequence is "ello".

The batch method will then let us convert this stream of character indices to sequences of the desired size.

```
def get_batch(vectorized_songs, seq_length, batch_size):
  n = vectorized_songs.shape[0] - 1
  idx = np.random.choice(n-seq_length, batch_size)
input_batch = [vectorized_songs[i : i+seq_length] for i in idx]

  output_batch = [vectorized_songs[i+1 : i+seq_length+1] for i in idx]

  x_batch = np.reshape(input_batch, [batch_size, seq_length])
  y_batch = np.reshape(output_batch, [batch_size, seq_length])
  return x_batch, y_batch

test_args = (vectorized_songs, 10, 2)
if not mdl.lab1.test_batch_func_types(get_batch, test_args) or \
   not mdl.lab1.test_batch_func_shapes(get_batch, test_args) or \
   not mdl.lab1.test_batch_func_next_step(get_batch, test_args):
   print("======\n[FAIL] could not pass tests")
else:
   print("======\n[PASS] passed all tests!")
x_batch, y_batch = get_batch(vectorized_songs, seq_length=5, batch_size=1)

for i, (input_idx, target_idx) in enumerate(zip(np.squeeze(x_batch), np.squeeze(y_batch))):
```

```
print("Step {:3d}".format(i))
print("  input: {} ({:s})".format(input_idx, repr(idx2char[input_idx])))
print("  expected output: {} ({:s})".format(target_idx, repr(idx2char[target_idx])))
```

8.5 The Recurrent Neural Network (RNN) Model

Now we're ready to define and train a RNN model on our ABC music dataset, and then use that trained model to generate a new song. We'll train our RNN using batches of song snippets from our dataset, which we generated in the previous section.

The model is based off the LSTM architecture, where we use a state vector to maintain information about the temporal relationships between consecutive characters.

The final output of the LSTM is then fed into a fully connected Dense layer where we'll output a softmax over each character in the vocabulary, and then sample from this distribution to predict the next character.

As we introduced in the first portion of this lab, we'll be using the Keras API, specifically, tf.keras.Sequential, to define the model. Three layers are used to define the model:

- tf.keras.layers.Embedding: This is the input layer, consisting of a trainable lookup table that maps the numbers of each character to a vector with embedding_dim dimensions.
- tf.keras.layers.LSTM: Our LSTM network, with size units=rnn_units.
- tf.keras.layers.Dense: The output layer, with vocab_size outputs.

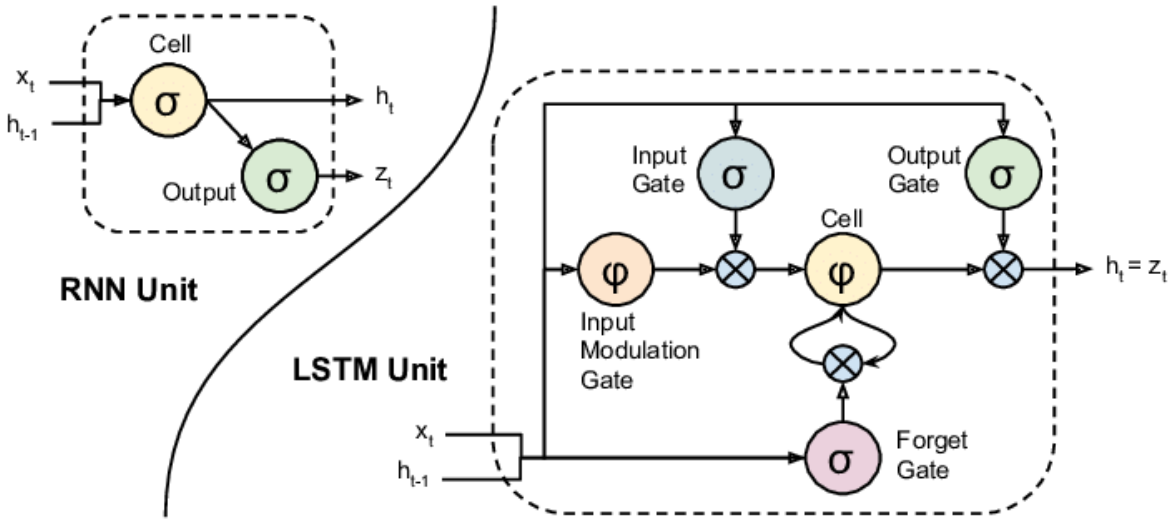


Figure 10 RNN Model

8.6 Define the RNN model

```
def LSTM(rnn_units):
  return tf.keras.layers.LSTM(
    rnn_units,
    return_sequences=True,
    recurrent_initializer='glorot_uniform',
    recurrent_activation='sigmoid',
    stateful=True,
  )
```

8.7 Test out the RNN model

It's always a good idea to run a few simple checks on our model to see that it behaves as expected.

First, we can use the Model.summary function to print out a summary of our model's internal workings. Here we can check the layers in the model, the shape of the output of each of the layers, the batch size, etc.

CodeText

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
```

```
embedding (Embedding)        (32, None, 256)           21248
_________________________________________________________________
lstm (LSTM)                  (32, None, 1024)          5246976
_________________________________________________________________
dense (Dense)                (32, None, 83)            85075
=================================================================
Total params: 5,353,299
Trainable params: 5,353,299
Non-trainable params: 0
```

8.8 Training the model :Loss and training operations

At this point, we can think of our next character prediction problem as a standard classification problem. Given the previous state of the RNN, as well as the input at a given time step, we want to predict the class of the next character -- that is, to actually predict the next character.

To train our model on this classification task, we can use a form of the crossentropy loss (negative log likelihood loss). Specifically, we will use the sparse_categorical_crossentropy loss, as it utilizes integer targets for categorical classification tasks.

We will want to compute the loss using the true targets -- the labels -- and the predicted targets -- the logits.

Let's first compute the loss using our example predictions from the untrained model:

```
def compute_loss(labels, logits):
  loss = tf.keras.losses.sparse_categorical_crossentropy(labels, logits, from_logits=True)
  return loss
example_batch_loss = compute_loss(y, pred)
print("Prediction shape: ", pred.shape, " # (batch_size, sequence_length, vocab_size)")
print("scalar_loss:      ", example_batch_loss.numpy().mean())
```

8.8 Predictions from the untrained Model

To get actual predictions from the model, we sample from the output distribution, which is defined by a softmax over our character vocabulary. This will give us actual character indices. This means we are using a categorical distribution to sample over the example prediction. This gives a prediction of the next character (specifically its index) at each timestep.

```
sampled_indices = tf.random.categorical(pred[0], num_samples=1)
sampled_indices = tf.squeeze(sampled_indices,axis=-1).numpy()
sampled_indices
print("Input: \n", repr("".join(idx2char[x[0]])))
print()
print("Next Char Predictions: \n", repr("".join(idx2char[sampled_indices])))
num_training_iterations = 2000  # Increase this to train longer
batch_size = 4  # Experiment between 1 and 64
seq_length = 100  # Experiment between 50 and 500
learning_rate = 5e-3  # Experiment between 1e-5 and 1e-1
vocab_size = len(vocab)
embedding_dim = 256
rnn_units = 1024  # Experiment between 1 and 2048
checkpoint_dir = './training_checkpoints'
checkpoint_prefix = os.path.join(checkpoint_dir, "my_ckpt")
```

8.9 Generate music using the RNN model

We can use our trained RNN model to generate some music! When generating music, we'll have to feed the model some sort of seed to get it started (because it can't predict anything without something to start with!).

Once we have a generated seed, we can then iteratively predict each successive character (remember, we are using the ABC representation for our music) using our trained RNN. More specifically, recall that our RNN outputs a softmax over possible successive characters.

For inference, we iteratively sample from these distributions, and then use our samples to encode a generated song in the ABC format.

Then, all we have to do is write it to a file and listen!

The prediction procedure

Now, we're ready to write the code to generate text in the ABC music format:

- Initialize a "seed" start string and the RNN state, and set the number of characters we want to generate.
- Use the start string and the RNN state to obtain the probability distribution over the next predicted character.
- Sample from multinomial distribution to calculate the index of the predicted character. This predicted character is then used as the next input to the model.
- At each time step, the updated RNN state is fed back into the model, so that it now has more context in making the next prediction. After predicting the next character, the updated RNN states are again fed back into the model, which is how it learns sequence dependencies in the data, as it gets more information from the previous predictions.

```
model.reset_states()

 tqdm._instances.clear()

for i in tqdm(range(generation_length))

    predictions = model(input_eval)

        predictions = tf.squeeze(predictions, 0)

    predicted_id = tf.random.categorical(predictions, num_samples=1)[-1,0].numpy()

        input_eval = tf.expand_dims([predicted_id], 0)

    return (start_string + ''.join(text_generated))
```

8.10 Playback the Generated Music

We can now call a function to convert the ABC format text to an audio file, and then play that back to check out our generated music! Try training longer if the resulting song is not long enough, or re-generating the song!

```
generated_songs = mdl.lab1.extract_song_snippet(generated_text)

for i, song in enumerate(generated_songs):

  waveform = mdl.lab1.play_song(song)

  if waveform:
    print("Generated song", i)

    ipythondisplay.display(waveform)
```

9. Hardware and Software Requirements

9.1 Hardware Requirement

- Sun-Fire-V240
- Pentium 233-megahertz (MHz) processor or faster (300 MHz is recommended)
- At least 64 megabytes (MB) of RAM (128 MB is recommended)
- At least 1.5 gigabytes (GB) of available space on the hard disk
- CD-ROM or DVD-ROM drive
- Keyboard and a Microsoft Mouse.
- Monitor with Super VGA (800 x 600) or higher resolution.

9.2 Software Requirement

- Windows 7
- Linux OS
- Internet Explorer/Google Chrome
- Anaconda
- Spyder
- Google Collaboratry

10. Conclusion and Future Scope

We have generated a good quality music, but there is a huge scope of improvement in it.

First, starting and ending music can be added in every new generated tune to give a tune a better start and better ending. By doing this, our generated music will become melodious.

Second, the model can be trained with more tunes. Here, we have trained our model with only 405 musical tunes. By training the model with more musical tunes, our model will not only expose to more variety of music but the number of classes will also increase. By this more melodious and at the same time more variety of music can be generated through the model.

Third, model can also be trained with multi-instrument tunes. As of now, the music generated is of only one piece of instrument. It would be interesting to listen what music the model will produce if it is trained on multi-instrument music.

Finally, a method can be added into the model which can handle unknown notes in the music. By filtering unknown notes and replacing them with known notes, model can generate more robust quality music.

With the advancements in artificial intelligence and machine learning, computers are now able to do tasks that were believed impossible before. And in many of these "impossible" fields, machines learn from scratch, without the knowledge of the governing rules. We hope that our project will be a small step in the right direction in the realm of machine music composition. Without having any prior concept of music, our models were able to learn to compose music that were both syntactically correct and also able to fool humans. We were also able to show that the flexibility of our model allows for exploring different parts of music composition, such as composing duet songs.