## Data

In this part, the book of Alice's Adventures in Wonderland by Lewis Carrol is used as the dataset. The book's Plain Text UTF-8 can be downloaded at https://www.gutenberg.org/ebooks/11/

# Generating Text with an RNN

```
!pip install unidecode
```

```
Collecting unidecode
  Downloading Unidecode-1.3.2-py3-none-any.whl (235 kB)
     |████████████████████████████████| 235 kB 3.9 MB/s
Installing collected packages: unidecode
Successfully installed unidecode-1.3.2
```

```
import unidecode
import string
import random
import re
import time

import torch
import torch.nn as nn

%matplotlib inline

%load_ext autoreload
%autoreload 2
```

```
from rnn.model_part2 import RNN
from rnn.helpers import time_since
from rnn.generate_part2 import generate
```

```
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
```

## Data Processing

The file we are using is a plain text file. We turn any potential unicode characters into plain ASCII by using the `unidecode` package (which you can install via `pip` or `conda`).

```
all_characters = string.printable
n_characters = len(all_characters)
print(n_characters)

file path = 'data/wonderland.txt'
```

✓ 0s completed at 3:25 PM

```
print('file_len =', file_len)

# we will leave the last 1/10th of text as test
split = int(0.9*file_len)
train_text = file[:split]
test_text = file[split:]

print('train len: ', len(train_text))
print('test len: ', len(test_text))
```

```
100
file_len = 164307
train len:  147876
test len:  16431
```

```
chunk_len = 200

def random_chunk(text):
    start_index = random.randint(0, len(text) - chunk_len)
    end_index = start_index + chunk_len + 1
    return text[start_index:end_index]

print(random_chunk(train_text))
```

```
 she felt very
lonely and low-spirited. In a little while, however, she again heard a
little pattering of footsteps in the distance, and she looked up
eagerly, half hoping that the Mouse had changed hi
```

## Input and Target data

To make training samples out of the large string of text data, we will be splitting the text into chunks.

Each chunk will be turned into a tensor, specifically a `LongTensor` (used for integer values), by looping through the characters of the string and looking up the index of each character in `all_characters`.

```
# Turn string into list of longs
def char_tensor(string):
    tensor = torch.zeros(len(string), requires_grad=True).long()
    for c in range(len(string)):
        tensor[c] = all_characters.index(string[c])
    return tensor
```

The following function loads a batch of input and target tensors for training. Each sample comes from a random chunk of text. A sample input will consist of all characters *except the last*, while the target wil contain all characters *following the first*. For example: if random_chunk='abc', then input='ab' and target='bc'

```
        target = torch.zeros(batch_size, chunk_len).long().to(device)
        for i in range(batch_size):
            start_index = random.randint(0, len(text) - chunk_len - 1)
            end_index = start_index + chunk_len + 1
            chunk = text[start_index:end_index]
            input_data[i] = char_tensor(chunk[:-1])
            target[i] = char_tensor(chunk[1:])
        return input_data, target
```

# Implement model

Your RNN model will take as input the character for step $t_{-1}$ and output a prediction for the next character $t$. The model should consiste of three layers - a linear layer that encodes the input character into an embedded state, an RNN layer (which may itself have multiple layers) that operates on that embedded state and a hidden state, and a decoder layer that outputs the predicted character scores distribution.

You must implement your model in the `rnn/model.py` file. You should use a `nn.Embedding` object for the encoding layer, a RNN model like `nn.RNN` or `nn.LSTM`, and a `nn.Linear` layer for the final a predicted character score decoding layer.

**TODO:** Implement the model in RNN `rnn/model.py`

# Evaluating

To evaluate the network we will feed one character at a time, use the outputs of the network as a probability distribution for the next character, and repeat. To start generation we pass a priming string to start building up the hidden state, from which we then generate one character at a time.

Note that in the `evaluate` function, every time a prediction is made the outputs are divided by the "temperature" argument. Higher temperature values make actions more equally likely giving more "random" outputs. Lower temperature values (less than 1) high likelihood options contribute more. A temperature near 0 outputs only the most likely outputs.

You may check different temperature values yourself, but we have provided a default which should work well.

```
def evaluate(rnn, prime_str='Ab', predict_len=100, temperature=0.85):
    hidden = rnn.init_hidden(1, device=device)
    prime_input = char_tensor(prime_str)
    predicted = prime_str

    # Use priming string to "build up" hidden state
```

```
    for p in range(predict_len):
        output, hidden = rnn(inp.to(device), hidden)

        # # Sample from the network as a multinomial distribution
        output_dist = output.data.view(-1).div(temperature).exp()
        top_i = torch.multinomial(output_dist, 1)[0]

        # # Add predicted character to string and use as next input
        predicted_char = all_characters[top_i]
        predicted += predicted_char
        inp = char_tensor(predicted_char)

    return predicted
```

# Train RNN

```
batch_size = 1
n_epochs = 6000
hidden_size = 150
n_layers = 5
learning_rate = 0.0025
model_type = 'rnn'
print_every = 100
plot_every = 40
```

```
def eval_test(rnn, inp, target):
    with torch.no_grad():
        hidden = rnn.init_hidden(batch_size, device=device)
        loss = 0
        for c in range(chunk_len):
            output, hidden = rnn(inp[:,c], hidden)
            loss += criterion(output.view(batch_size, -1), target[:,c])

    return loss.data.item() / chunk_len
```

## Train function

**TODO**: Fill in the train function. You should initialize a hidden layer representation using your RNN's `init_hidden` function, set the model gradients to zero, and loop over each time step (character) in the input tensor. For each time step compute the output of the of the RNN and compute the loss over the output and the corresponding ground truth time step in `target`. The loss should be averaged over all time steps. Lastly, call backward on the averaged loss and take an optimizer step.

```
    - optimizer: rnn model optimizer
    - criterion: loss function
    Returns:
    - loss: computed loss value as python float
    """
    loss = 0
    ####################################
    #          YOUR CODE HERE          #
    ####################################
    hidden = rnn.init_hidden(batch_size, device=device)
    rnn.zero_grad()
    # input = input.to(device)
    # target = target.to(device)
    for c in range(chunk_len):
      output, hidden = rnn(input[:,c], hidden)
      loss += criterion(output, target[:,c])

    loss.backward()
    optimizer.step()
    loss = loss.item() / chunk_len
    ##########       END      ##########


    return loss
    # return loss.data.item() / chunk_len


```

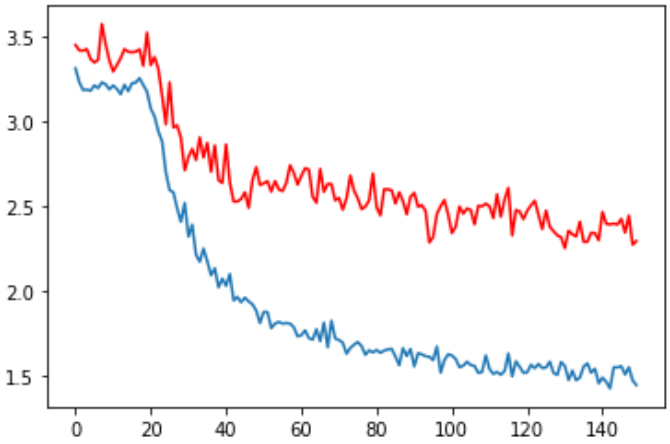
```
rnn = RNN(n_characters, hidden_size, n_characters, model_type=model_type, n_layers=n_
rnn_optimizer = torch.optim.Adam(rnn.parameters(), lr=learning_rate)
criterion = nn.CrossEntropyLoss()

start = time.time()
all_losses = []
test_losses = []
loss_avg = 0
test_loss_avg = 0


print("Training for %d epochs..." % n_epochs)
for epoch in range(1, n_epochs + 1):
    loss = train(rnn, *load_random_batch(train_text, chunk_len, batch_size), rnn_optim
    loss_avg += loss

    test_loss = eval_test(rnn, *load_random_batch(test_text, chunk_len, batch_size))
    test_loss_avg += test_loss

    if epoch % print_every == 0:
```

```
Training for 6000 epochs...
[0m 42s (100 1%) train loss: 3.1471, test_loss: 3.4471]
Wh rmfhhaatt n  wtbottly  e "yourhee  o ;p d e la s aCeh sri s otseoemr itoa
t eo  y telhuh   dg na  a

[1m 24s (200 3%) train loss: 3.2787, test_loss: 3.6182]
Wh rraAh trua,d h n  nhd
 dn  sonedo  u ee

oettt" shotha weeg   f  dwiefsyld odh cy eteke n

 hint

[2m 7s (300 5%) train loss: 3.2668, test_loss: 3.1902]
Whdhl h ctini nssunobs nbhte m  byr ltnh lt  nhetots ottuia sae rreeste  dfd seoeaure

[2m 49s (400 6%) train loss: 3.1903, test_loss: 3.4914]
Whn an owds  et"rnee drraanm sbhtns
os duontavnte "eton ak l oehord t see ieli  'hdt nb,uo" t  sp  iie

[3m 31s (500 8%) train loss: 3.3044, test_loss: 3.4585]
Whdt o Hhne  l ydt,posn
iyga
i tyaaw lfniWsd a psalop  ean s  k aw ha
aeurtImulq ialtac etaelg leec t

[4m 13s (600 10%) train loss: 3.2154, test_loss: 3.2450]
Whh !  set ec t   hft
mehwno p hsbd t
c p dn   et a sei tth"s sesdsi  ttd anh   n! sh    beg" de onac

[4m 55s (700 11%) train loss: 3.2915, test_loss: 3.0946]
WhhAhntrs oolrT
s ucei
etoecinH a eh!o hieiftreGaan  skraolsa otastr t cete c .bsso e r deo o  eMnt" u

[5m 37s (800 13%) train loss: 3.4424, test_loss: 3.0419]
Whdo hn ee co eibendthdal ndfdeletidorindtas l  s"l tne  hesgu  sgige tf ganetuhgthi

[6m 18s (900 15%) train loss: 2.8964, test_loss: 3.2031]
Whe teen oaise ceeytauoL le hiine
 OdapoisonA
ase ) ncoue aoiwat", moufhad nd, ohT nac vlahio ng sh os

[7m 0s (1000 16%) train loss: 2.5150, test_loss: 3.2930]
Whe atg shot" mde_ Ane t
the Ane th,e Konow t
lituued.y haed wore the, thec on.ee here sot the Therovt

[7m 42s (1100 18%) train loss: 2.5269, test_loss: 2.8505]
```

# Plot the Training and Test Losses

```
1  import matplotlib.pyplot as plt
2  import matplotlib.ticker as ticker
3
4  plt.figure()
5  plt.plot(all_losses)
6  plt.plot(test_losses, color='r')
```

```
[<matplotlib.lines.Line2D at 0x7f249c1f0e90>]
```




# Evaluate text generation

Check what the outputted text looks like

```
1  print(evaluate(rnn, prime_str='Th', predict_len=1000))
```

```
The trouse! I the Mouse, she said the Duchess was now!" she said, and their to
Alice.

"Why," said the Mock Turtle, and what goept but afrat knew had growice: "Rtlely remar

"Oh!ver peeping any didn't know," said Alice, "have his gause.!" said Alicen, "and at
moo?;
"I'm I had a lying!" she growing replied done. I mutter to not."
```

Some things you should try to improve your network performance are:

- Different RNN types. Switch the basic RNN network in your model to a GRU and LSTM to compare all three.
- Try adding 1 or two more layers
- Increase the hidden layer size
- Changing the learning rate

**TODO:** Try changing the RNN type and hyperparameters. Record your results.

```
1
```