Text Generation with RNNs

In [1]:
```
import numpy as np
import torch
from torch import nn
import torch.nn.functional as F
import numpy as np
from tqdm import tqdm
import random
```

1 Dataset

Define the path of the file, you want to read and train the model on

In [2]:
```
import urllib  # the lib that handles the url stuff

text = ""

for line in urllib.request.urlopen("https://raw.githubusercontent.com/GI
    text += line.decode('utf-8')
```

Inspect the dataset

Take a look at the first 250 characters in text

In [3]:
```
print(text)
```

```
Sometimes the whale shakes its tremendous tail in the air, which,
cracking like a whip, resounds to the distance of three or four mile
s."
--SCORESBY.


"Mad with the agonies he endures from these fresh attacks, the
infuriated Sperm Whale rolls over and over; he rears his enormous hea
d,
and with wide expanded jaws snaps at everything around him; he rushes
at the boats with his head; they are propelled before him with vast
swiftness, and sometimes utterly destroyed.... It is a matter of grea
t
astonishment that the consideration of the habits of so interesting,
and, in a commercial point of view, so important an animal (as the Sp
erm
Whale) should have been so entirely neglected, or should have excited
so little curiosity among the numerous, and many of them competent
observers, that of late years, must have possessed the most abundant
and the most convenient opportunities of witnessing their habitudes."
--THOMAS BEALE'S HISTORY OF THE SPERM WHALE, 1839.
```

In [4]:
```
# The unique characters in the file
vocab = sorted(set(text))
print ('{} unique characters'.format(len(vocab)))
```

```
86 unique characters
```

2 Process the dataset for the learning task

The task that we want our model to achieve is: given a character, or a sequence of characters, what is the most probable next character?

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.

In [5]:
```
# Creating a mapping from unique characters to indices
char2idx = {u:i for i, u in enumerate(vocab)}
text_as_int = np.array([char2idx[c] for c in text])

# Create a mapping from indices to characters
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:

In [6]:
```
print('{')
for char,_ in zip(char2idx, range(20)):
    print('  {:4s}: {:3d},'.format(repr(char), char2idx[char]))
print('  ...\n}')
```

```
{
  '\n':   0,
  '\r':   1,
  ' ' :   2,
  '!' :   3,
  '"' :   4,
  '#' :   5,
  '$' :   6,
  '%' :   7,
  '&' :   8,
  "'" :   9,
  '(' :  10,
  ')' :  11,
  '*' :  12,
  ',' :  13,
  '-' :  14,
  '.' :  15,
  '/' :  16,
  '0' :  17,
  '1' :  18,
  '2' :  19,
  ...
}
```

We can also look at how the first part of the text is mapped to an integer representation:

In [7]:
```
print ('{} ---- characters mapped to int ---- > {}'.format(repr(text[:13
```

```
'The Project G' ---- characters mapped to int ---- > [50 67 64  2 46 7
7 74 69 64 62 79  2 37]
```

Defining a method to encode one hot labels

In [8]:
```
def one_hot_encode(arr, n_labels):
    one_hot = np.zeros((len(arr), n_labels))
    for i, el in enumerate(arr):
        one_hot[i][el] = 1
    return one_hot
```

Defining a method to make mini-batches for training

In [9]:
```
def get_batches(arr, batch_size, seq_length, vocab_len):
    '''Create a generator that returns batches of size
       batch_size x seq_length from arr.

       Arguments
       ---------
       arr: Array you want to make batches from
       batch_size: Batch size, the number of sequences per batch
       seq_length: Number of encoded chars in a sequence
    '''
    x_batch = []
    y_batch = []
    start_idxs = list(range(len(arr) - seq_length))
    random.shuffle(start_idxs)
    for start_idx in start_idxs:
        chunk = arr[start_idx:start_idx + seq_length]
        x = one_hot_encode(chunk[:-1], vocab_len)
        y = chunk[-1]
        x_batch.append(x)
        y_batch.append(y)
        if len(x_batch) == batch_size:
            yield torch.FloatTensor(x_batch), torch.LongTensor(y_batch)
            x_batch = []
            y_batch = []
    if len(x_batch) > 0:
        yield torch.FloatTensor(x_batch), torch.LongTensor(y_batch)
```

3 The Recurrent Neural Network (RNN) model

Check if GPU is available

In [10]:
```
device = "cuda" if torch.cuda.is_available() else "cpu"
print(device)
```

```
cuda
```

Declaring the model

In [11]:
```
class VanillaRNNLayer(nn.Module):
    def __init__(self, x_size, a_size, y_size, is_last):
        super().__init__()
        self.w_ax = nn.Linear(x_size, a_size)
        self.w_aa = nn.Linear(a_size, a_size, bias=False)
        self.w_ya = nn.Linear(a_size, y_size)
        self.a_size = a_size
        self.is_last = is_last

    def forward(self, x, last_a):
        '''Forward pass through the network
        x is the input and `hidden` is the hidden/cell state .'''
        a = nn.functional.relu(self.w_aa(last_a) + self.w_ax(x))
        out = self.w_ya(a)
        if not self.is_last:
            out = nn.functional.relu(out)
        return out, a

class VanillaRNN(nn.Module):
    def __init__(self, x_size, a_size, y_size, n_layers):
        super().__init__()
        # Either w_ax or w_aa doesn't need a bias
        layers = []
        for layer in range(n_layers):
            is_last = layer == (n_layers - 1)
            layers.append(VanillaRNNLayer(x_size, a_size, y_size, is_las
        self.layers = torch.nn.ModuleList(layers)
        self.a_size = a_size

    def forward(self, x, last_h_states):
        '''Forward pass through the network
        x is the input and `hidden` is the hidden/cell state .'''
        next_h_states = []
        for hidden_state, layer in zip(last_h_states, self.layers):
            x, next_h_state = layer(x, hidden_state)
            next_h_states.append(next_h_state)
        return x, next_h_states

    def init_hidden(self, batch_size):
        ''' Initializes hidden state '''
        a = [torch.zeros((batch_size, self.a_size)).to(device) for i in
        return a
```

In [12]:
```
class VanillaGRULayer(nn.Module):
    def __init__(self, input_dim, hidden_dim, y_size, is_last):
        super().__init__()
        self.w_z = nn.Linear(input_dim + hidden_dim, hidden_dim)
        self.w_r = nn.Linear(input_dim + hidden_dim, hidden_dim)
        self.w_h_tilde = nn.Linear(input_dim + hidden_dim, hidden_dim)
        self.w_ya = nn.Linear(hidden_dim, y_size)
        self.hidden_dim = hidden_dim
        self.is_last = is_last

    def forward(self, x, last_a):
        '''Forward pass through the network
        x is the input and `hidden` is the hidden/cell state .'''
        z_t = nn.functional.sigmoid(self.w_z(torch.cat([last_a, x], dim=
        r_t = nn.functional.sigmoid(self.w_r(torch.cat([last_a, x], dim=
        h_tilde = nn.functional.tanh(self.w_r(torch.cat([r_t * last_a, x
        h_t = (1 - z_t) * last_a + z_t * h_tilde
        out = self.w_ya(h_t)
        if not self.is_last:
            out = nn.functional.relu(out)
        return out, h_t

class VanillaGRU(nn.Module):
    def __init__(self, input_dim, hidden_dim, y_size, n_layers):
        super().__init__()
        # Either w_ax or w_aa doesn't need a bias
        layers = []
        for layer in range(n_layers):
            is_last = layer == (n_layers - 1)
            layers.append(VanillaGRULayer(input_dim, hidden_dim, y_size,
        self.layers = torch.nn.ModuleList(layers)
        self.hidden_dim = hidden_dim

    def forward(self, x, last_h_states):
        '''Forward pass through the network
        x is the input and `hidden` is the hidden/cell state .'''
        next_h_states = []
        for hidden_state, layer in zip(last_h_states, self.layers):
            x, next_h_state = layer(x, hidden_state)
            next_h_states.append(next_h_state)
        return x, next_h_states

    def init_hidden(self, batch_size):
        ''' Initializes hidden state '''
        a = [torch.zeros((batch_size, self.hidden_dim)).to(device) for i
        return a
```

Declaring the train method

In [13]:
```
def train(model, data, vocab_len, epochs=10, batch_size=10, seq_length=5
    ''' Training a network

        Arguments
        ---------

        model: CharRNN network
        data: text data to train the network
        epochs: Number of epochs to train
        batch_size: Number of mini-sequences per mini-batch, aka batch s
        seq_length: Number of character steps per mini-batch
        lr: learning rate
        clip: gradient clipping
        val_frac: Fraction of data to hold out for validation
        print_every: Number of steps for printing training and validatio
    '''
    model.train()

    opt = torch.optim.Adam(model.parameters(), lr=lr)
    criterion = nn.CrossEntropyLoss()

    # create training and validation data
    val_idx = int(len(data) * (1 - val_frac))
    data, val_data = data[:val_idx], data[val_idx:]

    train_losses = []
    val_losses = []
    for e in range(epochs):
        running_train_losses = []
        running_val_losses = []
        h = model.init_hidden(batch_size)

        for x, y in tqdm(get_batches(data, batch_size, seq_length, vocab
            opt.zero_grad()
            a = model.init_hidden(batch_size)
            x, y = x.to(device), y.to(device)
            for char_pos in range(x.shape[1]):
                inp = x[:, char_pos, :]
                out, a = model(inp, a)

            loss = criterion(out, y)
            loss.backward()
            opt.step()
            running_train_losses.append(loss.item())

        with torch.no_grad():
            for x, y in tqdm(get_batches(val_data, batch_size, seq_lengt
                if x.shape[0] != batch_size:
                    continue
                a = model.init_hidden(batch_size)
                x, y = x.to(device), y.to(device)
                for char_pos in range(x.shape[1]):
                    inp = x[:, char_pos, :]
                    out, a = model(inp, a)

                loss = criterion(out, y)
```

```
                running_val_losses.append(loss.item())

        train_loss = np.mean(np.array(running_train_losses))
        val_loss = np.mean(np.array(running_val_losses))
        train_losses.append(train_loss)
        val_losses.append(val_loss)

        print(f"Epoch {e + 1}/{epochs} => train_loss: {train_loss}, val_

    return train_losses, val_losses
```

Defining a method to generate the next character

In [39]:
```
def predict(model, char, vocab_len, h, top_k=None):
    ''' Given a character, predict the next character.
        Returns the predicted character and the hidden state.
    '''

    # tensor inputs
    x = np.array([[char2idx[char]]])
    x = one_hot_encode(x, vocab_len)
    inputs = torch.from_numpy(x).float().to(device)

    # detach hidden state from history
    h = [each.data.float() for each in h]
    with torch.no_grad():
        output, h = model(inputs, h)

    output = nn.functional.softmax(output, dim=1)
    p = output.cpu().numpy().squeeze()
    p_sorted = np.sort(p)
    p = np.where(p < p_sorted[-top_k], np.zeros_like(p), p)
    # make it sum to one again
    p = p / np.sum(p)
    char = np.random.choice(np.array(list(range(vocab_len))), p=p)
    # return the encoded value of the predicted char and the hidden sta
    return idx2char[char], h
```

Declaring a method to generate new text

In [40]:
```
def sample(model, size, vocab_len, prime='The whale is attacking ', top_
    model.eval()

    chars = [ch for ch in prime]
    h = model.init_hidden(1)
    for ch in prime:
        char, h = predict(model, ch, vocab_len, h, top_k=top_k)

    chars.append(char)

    for ii in range(size):
        char, h = predict(model, char, vocab_len, h, top_k=top_k)
        chars.append(char)

    model.train()
    return ''.join(chars)
```

Generate new Text using the RNN model

Define and print the net

In [16]:
```
vanilla_model = VanillaRNN(x_size=len(vocab), a_size=256, y_size=len(voc
print(vanilla_model)
gru_model = VanillaGRU(input_dim=len(vocab), hidden_dim=256, y_size=len(
print(gru_model)
```

```
VanillaRNN(
  (layers): ModuleList(
    (0): VanillaRNNLayer(
      (w_ax): Linear(in_features=86, out_features=256, bias=True)
      (w_aa): Linear(in_features=256, out_features=256, bias=False)
      (w_ya): Linear(in_features=256, out_features=86, bias=True)
    )
    (1): VanillaRNNLayer(
      (w_ax): Linear(in_features=86, out_features=256, bias=True)
      (w_aa): Linear(in_features=256, out_features=256, bias=False)
      (w_ya): Linear(in_features=256, out_features=86, bias=True)
    )
  )
)
VanillaGRU(
  (layers): ModuleList(
    (0): VanillaGRULayer(
      (w_z): Linear(in_features=342, out_features=256, bias=True)
      (w_r): Linear(in_features=342, out_features=256, bias=True)
      (w_h_tilde): Linear(in_features=342, out_features=256, bias=Tru
e)
      (w_ya): Linear(in_features=256, out_features=86, bias=True)
    )
    (1): VanillaGRULayer(
      (w_z): Linear(in_features=342, out_features=256, bias=True)
      (w_r): Linear(in_features=342, out_features=256, bias=True)
      (w_h_tilde): Linear(in_features=342, out_features=256, bias=Tru
e)
      (w_ya): Linear(in_features=256, out_features=86, bias=True)
    )
  )
)
```

Declaring the hyperparameters

In [17]:
```
batch_size = 128
seq_length = 30
n_epochs = 4
```

Train the model and have fun with the generated texts

In [18]:
```
van_train_losses, van_val_losses = train(vanilla_model, text_as_int, voc
```

```
8839it [10:03, 14.65it/s]
981it [00:43, 22.33it/s]
0it [00:00, ?it/s]

Epoch 1/4 => train_loss: 1.8552036275285335, val_loss: 1.7349933470911
694

8839it [10:05, 14.59it/s]
981it [00:43, 22.33it/s]
0it [00:00, ?it/s]

Epoch 2/4 => train_loss: 1.5946490424347486, val_loss: 1.6633135176336
85

8839it [10:10, 14.47it/s]
981it [00:45, 21.34it/s]
0it [00:00, ?it/s]

Epoch 3/4 => train_loss: 1.5316208717927955, val_loss: 1.6363792175182
144

8839it [10:17, 14.31it/s]
981it [00:44, 21.93it/s]

Epoch 4/4 => train_loss: 1.499475210038552, val_loss: 1.61513359138360
45
```

In [42]:
```
van_train_losses, van_val_losses = train(gru_model, text_as_int, vocab_l
```

```
8839it [15:28,  9.52it/s]
981it [01:00, 16.31it/s]
0it [00:00, ?it/s]

Epoch 1/4 => train_loss: 1.8392366967544875, val_loss: 1.7330050682316
74

8839it [15:47,  9.33it/s]
981it [00:59, 16.45it/s]
0it [00:00, ?it/s]

Epoch 2/4 => train_loss: 1.5653007172637294, val_loss: 1.6419387195453
001

8839it [15:31,  9.49it/s]
981it [01:00, 16.09it/s]
0it [00:00, ?it/s]

Epoch 3/4 => train_loss: 1.4915817525810564, val_loss: 1.6065794315785
322

8839it [15:45,  9.35it/s]
981it [01:01, 16.05it/s]

Epoch 4/4 => train_loss: 1.450580582749265, val_loss: 1.59440831477002
88
```

In [41]:
```
sample(vanilla_model, 600, len(vocab))
```

Out[41]:
```
"The whale is attacking twenty ship's boast and\r\nthe ships and somet
imes was in two anger, and something them, and, which the sea of his s
tory and to the monster this way, that's to those that that\r\nstandin
g\r\nas take the way to the stream of his spermaced to\r\nthat\r\nit,
thou discrising off the stritter to by the\r\nstrangely stander of the
stead of these whale that in a second on to set along in the boat was
the most presence of a straggle, that is a cannibal sea to the steary
ship of the ships that in a striking to strange secret on to\r\nsay th
e sea, and straight to still belongers and\r\nsome this, the miditions
in this s"
```

In [43]:
```
sample(gru_model, 600, len(vocab))
```

Out[43]:
```
"The whale is attacking to be seen that that particular the back and t
he short\r\nof his called but a peculiar and present and anythin't sta
te to still be soon to strange allowed a cape of the captain on board
the ship and a colourest an extinet on that stanting to but a can that
the ship was substanting this boats of that seemed and a space. I do n
ot so an earth, astern any others of the barbar belongs on that seemen
and a soot all allusions, and that though along they were a present ti
ld an alastical perils of a stoot of a contracious below, bone in the
boat and any other spot of stitters, and that all these creature "
```