Text Generation with RNNs

```
from google.colab import drive
drive.mount('/content/drive')
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call
drive.mount("/content/drive", force_remount=True).
```

1 Dataset

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

We have attach the text file in zip, if you also use colab, then you should upload the text file as well.

```
with open('./poetry.txt', 'r') as f:
    poetry_corpus = f.read()
```

Inspect the dataset

Take a look at the first 250 characters in text

```
poetry_corpus[:100]
```

```
'寒随穷律变，春逐鸟声开。\n初风飘带柳，晚雪间花梅。\n碧林青旧竹，绿沼翠新苔。\n芝田初雁去，绮树巧莺来。\n晚霞聊自怡，初晴弥可喜。\n日晃百花色，风动千林翠。\n池鱼跃不同，园鸟声还异。\n寄言博通者，知予物'
```

```
len(poetry_corpus)
```

```
1516944
```

Replace/remove all chinese punctuations

```
poetry_corpus = poetry_corpus.replace('\n', ' ').replace('\r', ' ').replace(', ',
' ').replace('。', ' ')
poetry_corpus[:100]
```

```
'寒随穷律变 春逐鸟声开 初风飘带柳 晚雪间花梅 碧林青旧竹 绿沼翠新苔 芝田初雁去 绮树巧莺来
晚霞聊自怡 初晴弥可喜 日晃百花色 风动千林翠 池鱼跃不同 园鸟声还异 寄言博通者 知予物'
```

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.

```
import numpy as np

class TextConverter(object):
    def __init__(self, text_path, max_vocab=5000):
        """

        Args:
            text_path: text position
            max_vocab: max. # text
        """

        with open(text_path, 'r') as f:
            text = f.read()
        text = text.replace('\n', ' ').replace('\r', ' ').replace(', ', '
').replace('。', ' ')

        # remove the repeated char
        vocab = set(text)

        # if # char more than max_vocab, remove those whose freqs are less.
        vocab_count = {}

        # compute freq
        for word in vocab:
            vocab_count[word] = 0
        for word in text:
            vocab_count[word] += 1
        vocab_count_list = []
        for word in vocab_count:
            vocab_count_list.append((word, vocab_count[word]))
        vocab_count_list.sort(key=lambda x: x[1], reverse=True)

        if len(vocab_count_list) > max_vocab:
            vocab_count_list = vocab_count_list[:max_vocab]
```

```
        vocab = [x[0] for x in vocab_count_list]
        self.vocab = vocab

        self.word_to_int_table = {c: i for i, c in enumerate(self.vocab)}
        self.int_to_word_table = dict(enumerate(self.vocab))

    @property
    def vocab_size(self):
        return len(self.vocab) + 1

    def word_to_int(self, word):
        if word in self.word_to_int_table:
            return self.word_to_int_table[word]
        else:
            return len(self.vocab)

    def int_to_word(self, index):
        if index == len(self.vocab):
            return '<unk>'
        elif index < len(self.vocab):
            return self.int_to_word_table[index]
        else:
            raise Exception('Unknown index!')

    def text_to_arr(self, text):
        arr = []
        for word in text:
            arr.append(self.word_to_int(word))
        return np.array(arr)

    def arr_to_text(self, arr):
        words = []
        for index in arr:
            words.append(self.int_to_word(index))
        return "".join(words)
```

```
convert = Textconverter('./poetry.txt', max_vocab=10000)
```

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:

```
# orginal char(poetry)
txt_char = poetry_corpus[:11]
print(txt_char)

# convert to integers
# We can also look at how the first part of the text is mapped to an integer representation
print(convert.text_to_arr(txt_char))
```

```
寒随穷律变 春逐鸟声开
[ 38 173 350 901 573   0  11 390 107  56  86]
```

```
n_step = 20

# length of the given sequence
num_seq = int(len(poetry_corpus) / n_step)

text = poetry_corpus[:num_seq*n_step]

print(num_seq)
```

```
75847
```

Defining a method to encode one hot labels

```
def one_hot_encode(arr, n_labels):
    # Initialize the the encoded array
    one_hot = np.zeros((np.multiply(*arr.shape), n_labels), dtype=np.float32)

    # Fill the appropriate elements with ones
    one_hot[np.arange(one_hot.shape[0]), arr.flatten()] = 1.

    # Finally reshape it to get back to the original array
    one_hot = one_hot.reshape((*arr.shape, n_labels))

    return one_hot
```

Defining a method to make mini-batches for training

```
def get_batches(arr, batch_size, seq_length):
    '''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
    '''

    batch_size_total = batch_size * seq_length
    # total number of batches we can make
    n_batches = len(arr) // batch_size_total

    # Keep only enough characters to make full batches
    arr = arr[:n_batches * batch_size_total]
    # Reshape into batch_size rows
    arr = arr.reshape((batch_size, -1))

    # iterate through the array, one sequence at a time
    for n in range(0, arr.shape[1], seq_length):
        # The features
        x = arr[:, n:n + seq_length]
        # The targets, shifted by one
        y = np.zeros_like(x)
        try:
            y[:, :-1], y[:, -1] = x[:, 1:], arr[:, n + seq_length]
```

```
        except IndexError:
            y[:, :-1], y[:, -1] = x[:, 1:], arr[:, 0]
        yield x, y
```

```
import torch

arr = convert.text_to_arr(text)
arr = arr.reshape((num_seq, -1))
arr = torch.from_numpy(arr)

print(arr.shape)
print(arr[0, :])
```

```
torch.Size([75847, 20])
tensor([ 38, 173, 350, 901, 573,   0,  11, 390, 107,  56,  86,   0,   0, 157,
          4, 435, 292, 190,   0, 132])
```

```
class TextDataset(object):
    def __init__(self, arr):
        self.arr = arr

    def __getitem__(self, item):
        x = self.arr[item, :]

        # construct label
        y = torch.zeros(x.shape)

        # Use the first character entered as the label for the last input
        y[:-1], y[-1] = x[1:], x[0]
        return x, y

    def __len__(self):
        return self.arr.shape[0]
```

```
train_set = TextDataset(arr)
```

```
x, y = train_set[0]
print(convert.arr_to_text(x.numpy()))
print(convert.arr_to_text(y.numpy()))
```

```
寒随穷律变 春逐鸟声开  初风飘带柳 晚
随穷律变 春逐鸟声开  初风飘带柳 晚寒
```

3 The Recurrent Neural Network (RNN) model

Declaring the model

```
from torch import nn
from torch.autograd import Variable

use_gpu = True

class VanillaCharRNN(nn.Module):
    def __init__(self, num_classes, embed_dim, hidden_size,
                 num_layers, dropout):
        super().__init__()
        self.num_layers = num_layers
        self.hidden_size = hidden_size

        self.word_to_vec = nn.Embedding(num_classes, embed_dim)
        self.rnn = nn.GRU(embed_dim, hidden_size, num_layers)
        self.project = nn.Linear(hidden_size, num_classes)

    def forward(self, x, hs=None):
        batch = x.shape[0]
        if hs is None:
            hs = Variable(
                torch.zeros(self.num_layers, batch, self.hidden_size))
            if use_gpu:
                hs = hs.cuda()
        word_embed = self.word_to_vec(x)  # (batch, len, embed)
        word_embed = word_embed.permute(1, 0, 2)  # (len, batch, embed)
        out, h0 = self.rnn(word_embed, hs)  # (len, batch, hidden)
        le, mb, hd = out.shape
        out = out.view(le * mb, hd)
        out = self.project(out)
        out = out.view(le, mb, -1)
        out = out.permute(1, 0, 2).contiguous()  # (batch, len, hidden)
        return out.view(-1, out.shape[2]), h0
```

Declaring the hyperparameters

```
from torch.utils.data import DataLoader

batch_size = 128
train_data = DataLoader(train_set, batch_size, True, num_workers=4)
epochs = 20
```

Define and print the net

Check if GPU is available

```
model = VanillaCharRNN(convert.vocab_size, 512, 512, 2, 0.5)
if use_gpu:
    model = model.cuda()
criterion = nn.CrossEntropyLoss()

basic_optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)
optimizer = basic_optimizer

print(model)
```

```
VanillaCharRNN(
  (word_to_vec): Embedding(5386, 512)
  (rnn): GRU(512, 512, num_layers=2)
  (project): Linear(in_features=512, out_features=5386, bias=True)
)
```

Declaring the train method

```
for e in range(epochs):
    train_loss = 0
    for data in train_data:
        x, y = data
        y = y.long()
        if use_gpu:
            x = x.cuda()
            y = y.cuda()
        x, y = Variable(x), Variable(y)

        # Forward.
        score, _ = model(x)
        loss = criterion(score, y.view(-1))

        # Backward.
        optimizer.zero_grad()
        loss.backward()
        # Clip gradient.
        nn.utils.clip_grad_norm(model.parameters(), 5)
        optimizer.step()

        train_loss += loss.item()
    print('epoch: {}, perplexity is: {:.3f}'.format(e+1, np.exp(train_loss /
len(train_data))))
```

```
/usr/local/lib/python3.6/dist-packages/ipykernel_launcher.py:19: UserWarning:
torch.nn.utils.clip_grad_norm is now deprecated in favor of
torch.nn.utils.clip_grad_norm_.
```

```
epoch: 1, perplexity is: 248.098
epoch: 2, perplexity is: 127.901
epoch: 3, perplexity is: 80.512
epoch: 4, perplexity is: 57.031
epoch: 5, perplexity is: 42.753
```

```
epoch: 6, perplexity is: 33.407
epoch: 7, perplexity is: 26.851
epoch: 8, perplexity is: 22.089
epoch: 9, perplexity is: 18.595
epoch: 10, perplexity is: 15.899
epoch: 11, perplexity is: 13.847
epoch: 12, perplexity is: 12.250
epoch: 13, perplexity is: 10.991
epoch: 14, perplexity is: 9.964
epoch: 15, perplexity is: 9.123
epoch: 16, perplexity is: 8.420
epoch: 17, perplexity is: 7.858
epoch: 18, perplexity is: 7.379
epoch: 19, perplexity is: 6.963
epoch: 20, perplexity is: 6.608
```

Defining a method to generate the next character

```
def predict(model, char, h=None, 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, len(model.vocab))
    inputs = torch.from_numpy(x)

    if (train_on_gpu):
        inputs = inputs.cuda()

    # detach hidden state from history
    h = tuple([each.data for each in h])
    '''TODO: feed the current input into the model and generate output'''
    output, h = model('''TODO''') # TODO

    # get the character probabilities
    p = F.softmax(out, dim=1).data
    if (train_on_gpu):
        p = p.cpu()  # move to cpu

    # get top characters
    if top_k is None:
        top_ch = np.arange(len(model.vocab))
    else:
        p, top_ch = p.topk(top_k)
        top_ch = top_ch.numpy().squeeze()

    # select the likely next character with some element of randomness
    p = p.numpy().squeeze()
    char = np.random.choice(top_ch, p=p / p.sum())

    # return the encoded value of the predicted char and the hidden state
    return idx2char[char], h
```

```
def predict(preds, top_n=5):
    top_pred_prob, top_pred_label = torch.topk(preds, top_n, 1)
    top_pred_prob /= torch.sum(top_pred_prob)
    top_pred_prob = top_pred_prob.squeeze(0).cpu().numpy()
    top_pred_label = top_pred_label.squeeze(0).cpu().numpy()
    c = np.random.choice(top_pred_label, size=1, p=top_pred_prob)
    return c
```

Declaring to generate new text

```
begin = '天青色等烟雨'
text_len = 30

model = model.eval()
samples = [convert.word_to_int(c) for c in begin]
input_txt = torch.LongTensor(samples)[None]
if use_gpu:
  input_txt = input_txt.cuda()
input_txt = Variable(input_txt)
_, init_state = model(input_txt)
result = samples
model_input = input_txt[:, -1][:, None]
for i in range(text_len):
  # Get the predicted character and the hidden state.
  out, init_state = model(model_input, init_state)
  print(init_state)
  pred = predict(out.data)
  model_input = Variable(torch.LongTensor(pred))[None]
  if use_gpu:
      model_input = model_input.cuda()
  result.append(pred[0])
text = convert.arr_to_text(result)
print('Generate text is: {}'.format(text))
```

```
tensor([[[-0.9416, -0.9998, -0.9980,  ..., -0.9998,  0.7702, -0.9505]],

        [[-0.9991, -0.9974,  1.0000,  ..., -0.2473, -0.7346, -0.7575]]],
       device='cuda:0', grad_fn=<CudnnRnnBackward>)
tensor([[[-0.9419, -0.9999, -0.2492,  ...,  0.0621,  0.7704, -0.9597]],

        [[-0.9903, -0.9997,  0.9999,  ...,  1.0000,  0.3231, -0.7575]]],
       device='cuda:0', grad_fn=<CudnnRnnBackward>)
tensor([[[-0.9420, -1.0000,  0.8979,  ..., -0.9992,  0.7701, -0.9598]],

        [[-0.9973, -1.0000,  1.0000,  ...,  1.0000,  0.9668, -0.7575]]],
       device='cuda:0', grad_fn=<CudnnRnnBackward>)
tensor([[[-0.9420, -0.9738, -0.9495,  ..., -1.0000,  0.7692, -0.9597]],

        [[-0.9459, -0.9915,  0.6617,  ...,  1.0000,  0.9421, -0.7575]]],
       device='cuda:0', grad_fn=<CudnnRnnBackward>)
tensor([[[-0.9459, -0.9734, -0.9495,  ..., -0.9916, -0.1108, -0.9437]],

        [[-0.2516,  0.9996, -0.9091,  ...,  1.0000, -0.5957, -0.7576]]],
       device='cuda:0', grad_fn=<CudnnRnnBackward>)
tensor([[[-0.9732,  0.9981, -0.9464,  ...,  0.9995, -0.9281, -1.0000]],
```

```
        [[-0.7798,  0.9937, -0.7657,  ..., -0.2899, -0.9736, -0.9366]]],
       device='cuda:0', grad_fn=<CudnnRnnBackward>)
tensor([[[-0.9480,  0.9471, -0.9462,  ..., -0.5523,  0.0347, -0.9520]],

        [[ 0.5620, -0.0736,  0.9983,  ..., -0.0291, -0.9587, -0.8756]]],
       device='cuda:0', grad_fn=<CudnnRnnBackward>)
tensor([[[-0.9722, -0.2526,  0.9975,  ...,  0.6727, -0.0845, -0.9986]],

        [[-0.5102, -0.9752,  0.9598,  ..., -0.9709, -0.8095, -0.9189]]],
       device='cuda:0', grad_fn=<CudnnRnnBackward>)
tensor([[[-1.0000, -0.9393,  1.0000,  ..., -0.9999, -0.9955, -0.9986]],

        [[ 0.9964,  0.0678,  0.2757,  ...,  1.0000, -0.9978, -0.9617]]],
       device='cuda:0', grad_fn=<CudnnRnnBackward>)
tensor([[[-1.0000, -0.9994, -0.9681,  ..., -1.0000, -0.9996, -0.9986]],

        [[-0.9753, -0.8927, -0.2596,  ...,  1.0000, -1.0000, -0.9633]]],
       device='cuda:0', grad_fn=<CudnnRnnBackward>)
tensor([[[-1.0000, -0.9750, -0.9699,  ..., -1.0000, -0.9995, -0.9986]],

        [[ 0.5340,  0.9985, -0.9641,  ...,  1.0000, -0.9999, -0.9633]]],
       device='cuda:0', grad_fn=<CudnnRnnBackward>)
tensor([[[-0.9998, -0.9794, -0.9464,  ..., -0.9271, -0.9385, -0.9932]],

        [[-0.9998, -0.7872, -0.4005,  ...,  0.7981, -0.9978, -0.9633]]],
       device='cuda:0', grad_fn=<CudnnRnnBackward>)
tensor([[[-0.9443, -0.9975, -0.9803,  ..., -0.8513, -0.5364,  0.9832]],

        [[-1.0000, -0.5922,  0.7841,  ..., -0.5873,  0.9988, -0.9634]]],
       device='cuda:0', grad_fn=<CudnnRnnBackward>)
tensor([[[-0.9441,  0.9363, -0.9783,  ...,  0.9896,  0.9630,  0.9822]],

        [[-0.9642, -1.0000,  1.0000,  ..., -0.9722,  0.9648,  0.7468]]],
       device='cuda:0', grad_fn=<CudnnRnnBackward>)
tensor([[[-0.9483,  0.9999,  0.9726,  ...,  0.9997, -0.9976,  0.9820]],

        [[-0.7303,  0.0692, -1.0000,  ..., -0.9180,  0.9990,  0.7473]]],
       device='cuda:0', grad_fn=<CudnnRnnBackward>)
tensor([[[-0.9891, -0.0350,  0.9999,  ...,  0.9591, -0.9956,  0.9673]],

        [[-0.9984, -0.9933, -0.8203,  ...,  0.9917,  0.9988,  0.7478]]],
       device='cuda:0', grad_fn=<CudnnRnnBackward>)
tensor([[[-0.9891, -0.9848, -0.5915,  ..., -0.8657, -0.9992,  0.9668]],

        [[-0.9960, -1.0000, -0.6871,  ...,  1.0000,  0.9988,  0.7474]]],
       device='cuda:0', grad_fn=<CudnnRnnBackward>)
tensor([[[-0.9888,  0.9288,  0.8970,  ..., -0.7224, -0.9992,  0.9668]],

        [[-1.0000, -1.0000,  0.9884,  ...,  1.0000,  0.3770,  0.7482]]],
       device='cuda:0', grad_fn=<CudnnRnnBackward>)
tensor([[[-0.9888,  0.9226, -0.9231,  ..., -0.9394, -0.9986,  0.9667]],

        [[-0.9998, -0.9812,  0.0364,  ...,  1.0000,  0.3770,  0.7482]]],
       device='cuda:0', grad_fn=<CudnnRnnBackward>)
tensor([[[-0.9888,  0.8887, -0.9131,  ..., -0.9380,  0.0694,  0.1559]],

        [[-0.9954, -0.9997, -1.0000,  ...,  1.0000,  0.3966,  0.7481]]],
       device='cuda:0', grad_fn=<CudnnRnnBackward>)
```

```
tensor([[[-0.9889,  0.8884, -0.3538,  ..., -0.9380,  0.4976,  0.1553]],

        [[-0.9919, -0.9812, -1.0000,  ..., -0.9975,  0.4168,  0.7481]]],
       device='cuda:0', grad_fn=<CudnnRnnBackward>)
tensor([[[-0.7535,  0.9046, -0.3533,  ...,  0.0396,  0.9949,  0.1461]],

        [[-0.9858, -0.9971, -1.0000,  ..., -0.9984,  0.5227,  0.7490]]],
       device='cuda:0', grad_fn=<CudnnRnnBackward>)
tensor([[[-0.7567,  0.9045,  0.9278,  ..., -0.4027,  0.9232,  0.1461]],

        [[-0.4488, -0.9984, -1.0000,  ...,  0.5115,  0.9928,  0.7965]]],
       device='cuda:0', grad_fn=<CudnnRnnBackward>)
tensor([[[-0.8663,  0.8609,  0.9188,  ..., -0.9517,  0.7077,  0.1318]],

        [[-1.0000, -0.9996, -1.0000,  ...,  0.9997,  0.9769,  0.7965]]],
       device='cuda:0', grad_fn=<CudnnRnnBackward>)
tensor([[[-0.8666,  0.8608, -0.9099,  ..., -1.0000, -0.2667,  0.1203]],

        [[-1.0000, -1.0000, -1.0000,  ...,  1.0000,  0.9753,  0.7965]]],
       device='cuda:0', grad_fn=<CudnnRnnBackward>)
tensor([[[-0.8691,  0.5992, -0.9099,  ..., -0.9912, -0.9263, -0.8230]],

        [[-0.9775, -0.8856, -1.0000,  ...,  1.0000,  0.6730,  0.7963]]],
       device='cuda:0', grad_fn=<CudnnRnnBackward>)
tensor([[[-0.8455,  0.9404, -0.9086,  ...,  0.9864,  1.0000, -0.9996]],

        [[-0.9998, -1.0000,  0.4514,  ..., -1.0000,  0.9983,  0.9327]]],
       device='cuda:0', grad_fn=<CudnnRnnBackward>)
tensor([[[-0.8092,  0.9970, -0.9086,  ...,  0.9933, -0.0206, -0.9997]],

        [[ 0.9846, -0.9953, -0.9739,  ..., -1.0000,  0.9985,  0.9327]]],
       device='cuda:0', grad_fn=<CudnnRnnBackward>)
tensor([[[-0.9839,  0.8221,  0.4116,  ..., -0.8939, -0.9274, -0.9997]],

        [[-0.1342, -0.9951, -1.0000,  ..., -0.9831, -0.4810,  0.9468]]],
       device='cuda:0', grad_fn=<CudnnRnnBackward>)
tensor([[[-0.9841,  0.6946, -0.9928,  ..., -0.9927, -0.9965, -0.9997]],

        [[-0.7892, -0.9995, -0.9998,  ...,  0.8439, -0.7275,  0.9468]]],
       device='cuda:0', grad_fn=<CudnnRnnBackward>)
Generate text is: 天青色等烟雨来游　自昔重城外 犹教白日羌生幻 逢天花天山　白波天天地
```