Generating Shakespeare with a Character-Level RNN

In this part, we'll turn from traditional n-gram based language models to a more advanced form of language modeling using a Recurrent Neural Network. Specifically, we'll be setting up a character-level recurrent neural network (char-rnn) for short.

Andrej Karpathy, a researcher at OpenAI, has written an excellent blog post about using RNNs for language models, which you should read before beginning this assignment. The title of his blog post is The Unreasonable Effectiveness of Recurrent Neural Networks (http://karpathy.github.io/2015/05/21/rnn-effectiveness/).

Karpathy shows how char-rnns can be used to generate texts for several fun domains:

- Shakespeare plays
- Essays about economics
- LaTeX documents
- Linux source code
- Baby names

Recommended Reading

You should install PyTorch, know Python, and understand Tensors:

- http://pytorch.org/ (http://pytorch.org/) For installation instructions
- Deep Learning with PyTorch: A 60-minute Blitz (https://github.com/pytorch/tutorials/blob/master/Deep%20Learning%20with%20PyTorch.ipynb) to get started with PyTorch in general
- jcjohnson's PyTorch examples (https://github.com/jcjohnson/pytorch-examples) for an in depth overview
- Introduction to PyTorch for former Torchies (https://github.com/pytorch/tutorials/blob/master/Introduction%20to%20PyTorch%20for%20former%20Torchies.ipynb) if you are former Lua Torch user

It would also be useful to know about RNNs and how they work:

- The Unreasonable Effectiveness of Recurrent Neural Networks (http://karpathy.github.io/2015/05/21/rnn-effectiveness/) shows a bunch of real life examples
- Understanding LSTM Networks (http://colah.github.io/posts/2015-08-Understanding-LSTMs/) is about LSTMs specifically but also informative about RNNs in general

Also see these related tutorials from the series:

- Classifying Names with a Character-Level RNN (https://github.com/spro/practical-pytorch/blob/master/char-rnn-classification/char-rnn-classification.ipynb) uses an RNN for classification
- Generating Names with a Conditional Character-Level RNN (https://github.com/spro/practical-pytorch/blob/master/conditional-char-rnn/conditional-char-rnn.ipynb) builds on this model to add a category as input

You can also set up Pytorch in Google Colab

Pytorch is one of the most popular deep learning frameworks in both industry and academia, and learning its use will be invaluable should you choose a career in deep learning.

Setup

Using Google Colab (recommended)

1. Upload this notebook on Colab (https://colab.research.google.com/notebooks/welcome.ipynb).
2. Set hardware accelerator to `GPU` under `notebook settings` in the `Edit` menu.
3. Run the first cell to set up the environment.

Note

Prepare data

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`).

In [ ]:
```
import unidecode
import string
import random
import re
import torch

all_characters = string.printable
n_characters = len(all_characters)

file = unidecode.unidecode(open('shakespeare_data/shakespeare_inp
ut.txt').read())
file_len = len(file)
print('file_len =', file_len)
```

```
file_len = 4573338
```

To make inputs out of this big string of data, we will be splitting it into chunks.

In [ ]:
```
chunk_len = 200

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

print(random_chunk())
```

```
thank
God I have as little patience as another man; and
therefore I can be quiet.

DON ADRIANO DE ARMADO:
I do affect the very ground, which is base, where
her shoe, which is baser, guided by her foot,
```

Build the Model

This model will take as input the character for step t_{-1} and is expected to output the next character t. There are three layers - one linear layer that encodes the input character into an internal state, one GRU layer (which may itself have multiple layers) that operates on that internal state and a hidden state, and a decoder layer that outputs the probability distribution. You need to finish the forward method. (Refer to Pytorch GRU Documentation (https://pytorch.org/docs/stable/nn.html#gru))

In [ ]:
```
import torch
import torch.nn as nn

device = torch.device("cuda" if torch.cuda.is_available() else "c
pu")

class RNN(nn.Module):
    def __init__(self, input_size, hidden_size, output_size, n_la
yers=1):
        super(RNN, self).__init__()
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.output_size = output_size
        self.n_layers = n_layers

        self.encoder = nn.Embedding(input_size, hidden_size)
        self.gru = nn.GRU(hidden_size, hidden_size, n_layers)
        self.decoder = nn.Linear(hidden_size, output_size)

    def forward(self, input, hidden):
        #Input input: torch Tensor of shape (1,)
        #hidden: torch Tensor of shape (self.n_layers, 1, self.hi
dden_size)
        #Return output: torch Tensor of shape (1, self.output_siz
e)
        #and hidden: torch Tensor of shape (self.n_layers, 1, sel
f.hidden_size)
        encoded = self.encoder(input)
        # encoded = encoded.squeeze(0)
        # print(encoded.view())
        # print(hidden.view())
        encoded = encoded.unsqueeze(0)
        encoded = encoded.unsqueeze(0)
        output, hidden = self.gru(encoded,hidden)
        # hidden = hidden.view(n_layers, n_directions=1, batch_si
ze=1, hidden_dim=self.hidden_size)
        # hidden = hidden[-1]
        # output = output.squeeze(1)
        # hidden_forward, hidden_backward = hidden[0], hidden[1]
        out = self.decoder(output)
        out = out.squeeze(1)
        return out, hidden


    def init_hidden(self):
        return torch.zeros(self.n_layers, 1, self.hidden_size).to
(device)
```

Inputs and Targets

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`.

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

print(char_tensor('abcDEF'))
```

```
tensor([10, 11, 12, 39, 40, 41], device='cuda:0')
```

Finally we can assemble a pair of input and target tensors for training, from a random chunk. The input will be all characters *up to the last*, and the target will be all characters *from the first*. So if our chunk is "abc" the input will correspond to "ab" while the target is "bc".

In [ ]:
```
def random_training_set():
    chunk = random_chunk()
    inp = char_tensor(chunk[:-1])
    target = char_tensor(chunk[1:])
    return inp, target
```

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.

In [ ]:
```
def evaluate(prime_str='A', predict_len=100, temperature=0.8):
    hidden = decoder.init_hidden()
    prime_input = char_tensor(prime_str)
    predicted = prime_str

    # Use priming string to "build up" hidden state
    for p in range(len(prime_str) - 1):
        _, hidden = decoder(prime_input[p], hidden)
    inp = prime_input[-1]

    for p in range(predict_len):
        output, hidden = decoder(inp, 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)[0]


    return predicted
```

Training

A helper to print the amount of time passed:

In [ ]:
```
import time, math

def time_since(since):
    s = time.time() - since
    m = math.floor(s / 60)
    s -= m * 60
    return '%dm %ds' % (m, s)
```

The main training function

In [ ]:
```
def train(inp, target):
    hidden = decoder.init_hidden()
    decoder.zero_grad()
    loss = 0

    for c in range(chunk_len):
        output, hidden = decoder(inp[c], hidden)
        loss += criterion(output, target.unsqueeze(1)[c])

    loss.backward()
    decoder_optimizer.step()

    return loss.item() / chunk_len
```

Then we define the training parameters, instantiate the model, and start training:

In [ ]:
```
n_epochs = 2000
print_every = 100
plot_every = 10
hidden_size = 100
n_layers = 2
lr = 0.001

decoder = RNN(n_characters, hidden_size, n_characters, n_layers).
to(device)
decoder_optimizer = torch.optim.Adam(decoder.parameters(), lr=lr)
criterion = nn.CrossEntropyLoss()

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

for epoch in range(1, n_epochs + 1):
    loss = train(*random_training_set())
    loss_avg += loss

    if epoch % print_every == 0:
        print('[%s (%d %d%%) %.4f]' % (time_since(start), epoch,
epoch / n_epochs * 100, loss))
        print(evaluate('Wh', 100), '\n')

    if epoch % plot_every == 0:
        all_losses.append(loss_avg / plot_every)
        loss_avg = 0
```

```
[0m 13s (100 5%) 2.8362]
Wht at Biton iw oac, aco reast for hathrers mhe to, ahm fn iched
th hof at ut:
To nont hoth
tot,
Tht l

[0m 27s (200 10%) 2.4487]
Whot ot hhe no the,
Dasd sare id sire fher cit en?

EACMCEAp:, ansdam, not me nor,, she me yo we, you

[0m 41s (300 15%) 2.2000]
Whe this, thou?

MEAREU:

For; in louns she of a end a the welor, and I the wery and ho Ay
a me to is

[0m 55s (400 20%) 2.1136]
Wher correse in at'tay greertringen:
Tor eostes wothhave mathes is thester, mods corluy he thir cowen

[1m 8s (500 25%) 2.0062]
Whan ere to you ent foan to ta in.

CEAN:
Mat old senth lemy fore, and the of me all aping they the th

[1m 22s (600 30%) 2.2487]
Wher thousan cleas you wive fare is yould the to sale hare ham.

HTINTO:
Prost that befastent more be

[1m 36s (700 35%) 1.9990]
Whelker thy thear yould old
Wir will lown thus gord ould she me storoar that of mebless bolde
res mun a

[1m 49s (800 40%) 2.0068]
Wher that I the with thage.

OTINA:
She the that we lord bofuad shet in have alse and thy for-wheran:


[2m 3s (900 45%) 2.0764]
What it your mast that even, is houndpries will theman tame whoo'
s,
He damy, what thead, of hike,
Whel

[2m 17s (1000 50%) 2.0023]
Whal,
Thy hith our shut-sent gorther with to drentand wourd,
Won! hen groito for dristrien, would it b
```

```
[2m 30s (1100 55%) 2.0064]
Wheld by Romovere his the costder to mise:
Sif ye cise eremeading so, we bettere,
But wand I lestats s

[2m 44s (1200 60%) 1.9589]
Whelby!
Nart thy croy's brod, forth then not lords,
And the were casins colth and comuch well'd
Prow t

[2m 57s (1300 65%) 1.8900]
Whair fruath,.
Net there entlest it hath and to the father hast to my ince have
his do my to my have p

[3m 11s (1400 70%) 1.7337]
Wher scans or theee frows there:
You; it with my his an shill you whoy, of mosing all not doobk sh
all

[3m 25s (1500 75%) 1.7519]
Wher for the all mist the and
And of is brot, hand hatring,
Aring he chave wouls it shall and you the

[3m 38s (1600 80%) 1.8131]
Whan too dount of you swarrus and this westal
Hood his greed thus: I'll my lean.

DOBEL:
Caistor him-p

[3m 52s (1700 85%) 1.8427]
Why good dane?

Servion furt the stay sons,
The his culso more fall of that lide.

LANGHIRA:
Were he f

[4m 6s (1800 90%) 1.8984]
Where shall of it told with the chanry now on conting
shall did you see-thing brow he man:
whither too

[4m 20s (1900 95%) 1.8849]
Where there same, send of
not to the mone thee not daused do my efrech?
Therefore hou duck 'to shourse

[4m 33s (2000 100%) 1.7411]
Whine his feart I have cains fried?

SALGO:
And what you
have with shall be of the way.

ACBECLESS:
```

```
Sh
```

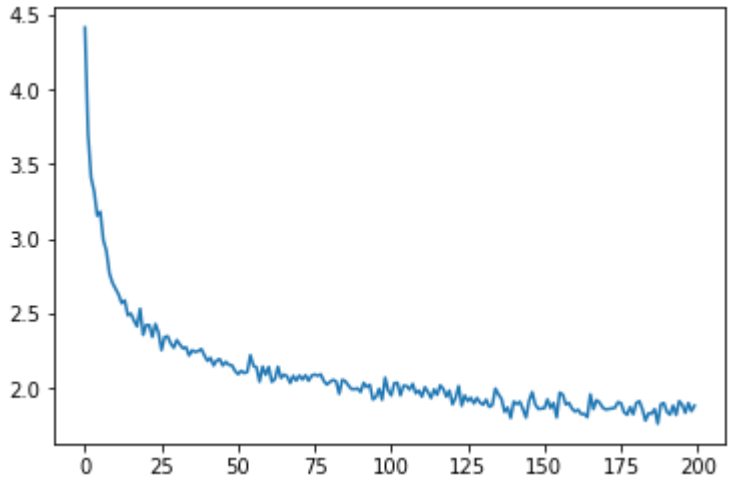
Plotting the Training Losses

Plotting the historical loss from all_losses shows the network learning:

In [ ]:
```
import matplotlib.pyplot as plt
import matplotlib.ticker as ticker
%matplotlib inline

plt.figure()
plt.plot(all_losses)
```

Out[ ]:
```
[<matplotlib.lines.Line2D at 0x7f4503019400>]
```



Evaluating at different "temperatures"

In the `evaluate` function above, every time a prediction is made the outputs are divided by the "temperature" argument passed. Using a higher number makes all actions more equally likely, and thus gives us "more random" outputs. Using a lower value (less than 1) makes high probabilities contribute more. As we turn the temperature towards zero we are choosing only the most likely outputs.

We can see the effects of this by adjusting the `temperature` argument.

In [ ]:
```
print(evaluate('Th', 200, temperature=0.8))
```

```
Ther hape, in that trut would
What uncuralm. I wall the malion.

CIRLUS:
I should us didst for not to that the drowht.

AUNIL:
If the have scallow so nou wild thou werick so man,
The drangent: thou prow
```

Lower temperatures are less varied, choosing only the more probable outputs:

In [ ]:
```
print(evaluate('Th', 200, temperature=0.2))
```

```
There a prainter the will the brown the will the will and the for
the some to the such the could the are the are and the prince and
the some to the come
And the should the say the lord the sentle the co
```

Higher temperatures more varied, choosing less probable outputs:

In [ ]:
```
print(evaluate('Th', 200, temperature=1.4))
```

```
Thelect of. wOHLCIAN:
Arfoild,
As spited tnog, soullque not britalh,
Thou rin yamethou was vare surbWhom hand.

MOFY FABlAnd,, O witherhart for now this
Sapinemes's ome joget a fly oone son
furmpy.
T.'
```

In [ ]:
```
import torch.nn.functional as F
def perp(testfile):
    inp = char_tensor(testfile[:-1]).to(device)
    target = char_tensor(testfile[1:]).to(device)
    test_len=len(testfile)
    hidden = decoder.init_hidden()
    decoder.zero_grad()
    perplexity=torch.tensor(0.0).to(device)

    for c in range(test_len-1):
        output, hidden = decoder(inp[c], hidden)
        perplexity -=F.log_softmax(output,dim=1)[0][target[c]]

    return (perplexity/test_len).exp().item()

testfile = unidecode.unidecode(open('shakespeare_data/shakespeare
_sonnets.txt').read())
print('Perplexity:',perp(testfile))
```

```
Perplexity: 6.900490760803223
```

FAQs

I'm unfamiliar with PyTorch. How do I get started?

If you are new to the paradigm of computational graphs and functional programming, please have a look at this tutorial (https://hackernoon.com/linear-regression-in-x-minutes-using-pytorch-8eec49f6a0e2) before getting started.

How do I speed up training?

Send the model and the input, output tensors to the GPU using `.to(device)` . Refer the PyTorch docs (https://pytorch.org/docs/stable/notes/cuda.html) for further information.