Building an RNN in PyTorch

In this notebook, I'll construct a character-level RNN with PyTorch. The network will train character by character on some text, then generate new text character by character. [Ref Article (http://karpathy.github.io/2015/05/21/rnn-effectiveness/)](http://karpathy.github.io/2015/05/21/rnn-effectiveness/)

In [1]:

```
#Import all libs
import numpy as np
import torch
from torch import nn
import torch.nn.functional as F
from torch.autograd import Variable
```

In [2]:

```
with open('data/anna.txt', 'r') as f:
    text = f.read()
```

In [3]:

```
charss = set(text)
print(len(charss))
```

```
83
```

Now we have the text, encode it as integers.

In [4]:

```
chars = tuple(set(text))
int2char = dict(enumerate(chars))
char2int = {ch: ii for ii, ch in int2char.items()}
encoded = np.array([char2int[ch] for ch in text])
```

In [5]:

```
print(chars)
```

```
('u', 't', 'x', '1', '0', '2', '\n', 'q', 'W', 'Q', 'k', '"', '5',
':', 'K', 'I', 'v', '_', 'E', '6', 'X', '4', 'S', ' ', '.', 'Y',
'p', '3', '*', 'm', '&', ',', '@', 'f', 'l', 'j', 'g', "'", 'r',
'Z', 'y', 'M', 'G', 'i', '7', 'O', 'A', '$', 'H', 'c', '?', 'D',
'(', '9', 'T', ')', 'C', '8', 'n', '/', '-', 'd', 'J', 'P', '%',
'V', 'N', 'b', 'R', 'w', ';', '!', 'z', 'o', 'h', 's', 'e', 'B',
'F', 'a', 'U', '`', 'L')
```

Processing the data

We're one-hot encoding the data, so we will make a function to do that.

we will also create mini-batches for training. We'll take the encoded characters and split them into multiple sequences, given by `n_seqs` (also refered to as "batch size" in other places). Each of those sequences will be `n_steps` long.

In [6]:

```
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
```

In [7]:

```
def get_batches(arr, n_seqs, n_steps):
    '''Create a generator that returns mini-batches of size
       n_seqs x n_steps from arr.
    '''

    batch_size = n_seqs * n_steps
    n_batches = len(arr)//batch_size

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

    for n in range(0, arr.shape[1], n_steps):
        # The features
        x = arr[:, n:n+n_steps]
        # The targets, shifted by one
        y = np.zeros_like(x)
        try:
            y[:, :-1], y[:, -1] = x[:, 1:], arr[:, n+n_steps]
        except IndexError:
            y[:, :-1], y[:, -1] = x[:, 1:], arr[:, 0]
        yield x, y
```

Defining the network with PyTorch

Here I'll use PyTorch to define the architecture of the network. We start by defining the layers and operations we want. Then, define a method for the forward pass. I'm also going to write a method for predicting characters.

In [8]:

```
class CharRNN(nn.Module):
    def __init__(self, tokens, n_steps=100, n_hidden=256, n_layers=2,
                               drop_prob=0.5, lr=0.001):
        super(CharRNN,self).__init__()
        self.drop_prob = drop_prob
        self.n_layers = n_layers
        self.n_hidden = n_hidden
        self.lr = lr

        self.chars = tokens
        self.int2char = dict(enumerate(self.chars))
        self.char2int = {ch: ii for ii, ch in self.int2char.items()}

        self.dropout = nn.Dropout(drop_prob)
        self.lstm = nn.LSTM(len(self.chars), n_hidden, n_layers,
                            dropout=drop_prob, batch_first=True)
        self.fc = nn.Linear(n_hidden, len(self.chars))

        self.init_weights()

    def forward(self, x, hc):
        ''' Forward pass through the network '''

        x, (h, c) = self.lstm(x, hc)
        x = self.dropout(x)

        # Stack up LSTM outputs
        x = x.view(x.size()[0]*x.size()[1], self.n_hidden)

        x = self.fc(x)

        return x, (h, c)

    def predict(self, char, h=None, cuda=False, top_k=None):
        ''' Given a character, predict the next character.

            Returns the predicted character and the hidden state.
        '''
        if cuda:
            self.cuda()
        else:
            self.cpu()

        if h is None:
            h = self.init_hidden(1)

        x = np.array([[self.char2int[char]]])
        x = one_hot_encode(x, len(self.chars))
        inputs = Variable(torch.from_numpy(x), volatile=True)
        if cuda:
            inputs = inputs.cuda()

        h = tuple([Variable(each.data, volatile=True) for each in h])
        out, h = self.forward(inputs, h)

        p = F.softmax(out).data
        if cuda:
            p = p.cpu()
```

```
        if top_k is None:
            top_ch = np.arange(len(self.chars))
        else:
            p, top_ch = p.topk(top_k)
            top_ch = top_ch.numpy().squeeze()

        p = p.numpy().squeeze()
        char = np.random.choice(top_ch, p=p/p.sum())

        return self.int2char[char], h

    def init_weights(self):
        ''' Initialize weights for fully connected layer '''
        initrange = 0.1

        # Set bias tensor to all zeros
        self.fc.bias.data.fill_(0)
        # FC weights as random uniform
        self.fc.weight.data.uniform_(-1, 1)

    def init_hidden(self, n_seqs):
        ''' Initializes hidden state '''
        # Create two new tensors with sizes n_layers x n_seqs x n_hidden,
        # initialized to zero, for hidden state and cell state of LSTM
        weight = next(self.parameters()).data
        return (Variable(weight.new(self.n_layers, n_seqs, self.n_hidden).zero_
()),
                Variable(weight.new(self.n_layers, n_seqs, self.n_hidden).zero_
()))
```

In [9]:

```
def train(net, data, epochs=10, n_seqs=10, n_steps=50, lr=0.001, clip=5, val_fra
c=0.1, cuda=False, print_every=10):
    ''' Traing a network

        Arguments
        ---------

        net: CharRNN network
        data: text data to train the network
        epochs: Number of epochs to train
        n_seqs: Number of mini-sequences per mini-batch, aka batch size
        n_steps: Number of character steps per mini-batch
        lr: learning rate
        clip: gradient clipping
        val_frac: Fraction of data to hold out for validation
        cuda: Train with CUDA on a GPU
        print_every: Number of steps for printing training and validation loss

    '''

    net.train()
    opt = torch.optim.Adam(net.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:]

    if cuda:
        net.cuda()

    counter = 0
    n_chars = len(net.chars)
    for e in range(epochs):
        h = net.init_hidden(n_seqs)
        for x, y in get_batches(data, n_seqs, n_steps):
            counter += 1

            # One-hot encode our data and make them Torch tensors
            x = one_hot_encode(x, n_chars)
            x, y = torch.from_numpy(x), torch.from_numpy(y)

            inputs, targets = Variable(x), Variable(y)
            if cuda:
                inputs, targets = inputs.cuda(), targets.cuda()

            # Creating new variables for the hidden state, otherwise
            # we'd backprop through the entire training history
            h = tuple([Variable(each.data) for each in h])

            net.zero_grad()

            output, h = net.forward(inputs, h)
            loss = criterion(output, targets.view(n_seqs*n_steps))

            loss.backward()

            # `clip_grad_norm` helps prevent the exploding gradient problem in R
NNs / LSTMs.
```

```
            nn.utils.clip_grad_norm(net.parameters(), clip)

            opt.step()

            if counter % print_every == 0:

                # Get validation loss
                val_h = net.init_hidden(n_seqs)
                val_losses = []
                for x, y in get_batches(val_data, n_seqs, n_steps):
                    # One-hot encode our data and make them Torch tensors
                    x = one_hot_encode(x, n_chars)
                    x, y = torch.from_numpy(x), torch.from_numpy(y)

                    # Creating new variables for the hidden state, otherwise
                    # we'd backprop through the entire training history
                    val_h = tuple([Variable(each.data, volatile=True) for each i
n val_h])

                    inputs, targets = Variable(x, volatile=True), Variable(y, vo
latile=True)
                    if cuda:
                        inputs, targets = inputs.cuda(), targets.cuda()

                    output, val_h = net.forward(inputs, val_h)
                    val_loss = criterion(output, targets.view(n_seqs*n_steps))

                    val_losses.append(val_loss.data[0])

                print("Epoch: {}/{}...".format(e+1, epochs),
                      "Step: {}...".format(counter),
                      "Loss: {:.4f}...".format(loss.data[0]),
                      "Val Loss: {:.4f}".format(np.mean(val_losses)))
```

Time to train

Now we can actually train the network. First we'll create the network itself, with some given hyperparameters. Then, define the mini-batches sizes (number of sequences and number of steps), and start the training. With the train function, we can set the number of epochs, the learning rate, and other parameters. Also, we can run the training on a GPU by setting `cuda=True`.

In [10]:

```
if 'net' in locals():
    del net
```

In [11]:

```
net = CharRNN(chars, n_hidden=512, n_layers=2)
```

In [12]:

```
n_seqs, n_steps = 128, 100
train(net, encoded, epochs=2, n_seqs=n_seqs, n_steps=n_steps, lr=0.001, cuda=True, print_every=10)
```

```
/users/gpu/anupriy/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:58: UserWarning: torch.nn.utils.clip_grad_norm is now deprecated in favor of torch.nn.utils.clip_grad_norm_.
/users/gpu/anupriy/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:74: UserWarning: volatile was removed and now has no effect. Use `with torch.no_grad():` instead.
/users/gpu/anupriy/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:76: UserWarning: volatile was removed and now has no effect. Use `with torch.no_grad():` instead.
/users/gpu/anupriy/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:83: UserWarning: invalid index of a 0-dim tensor. This will be an error in PyTorch 0.5. Use tensor.item() to convert a 0-dim tensor to a Python number
/users/gpu/anupriy/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:87: UserWarning: invalid index of a 0-dim tensor. This will be an error in PyTorch 0.5. Use tensor.item() to convert a 0-dim tensor to a Python number

Epoch: 1/2... Step: 10... Loss: 3.3130... Val Loss: 3.3000
Epoch: 1/2... Step: 20... Loss: 3.1633... Val Loss: 3.1923
Epoch: 1/2... Step: 30... Loss: 3.0914... Val Loss: 3.0759
Epoch: 1/2... Step: 40... Loss: 2.9071... Val Loss: 2.9167
Epoch: 1/2... Step: 50... Loss: 2.7734... Val Loss: 2.7651
Epoch: 1/2... Step: 60... Loss: 2.6146... Val Loss: 2.6330
Epoch: 1/2... Step: 70... Loss: 2.5515... Val Loss: 2.5624
Epoch: 1/2... Step: 80... Loss: 2.4771... Val Loss: 2.5053
Epoch: 1/2... Step: 90... Loss: 2.4504... Val Loss: 2.4624
Epoch: 1/2... Step: 100... Loss: 2.3860... Val Loss: 2.4245
Epoch: 1/2... Step: 110... Loss: 2.3510... Val Loss: 2.3908
Epoch: 1/2... Step: 120... Loss: 2.2904... Val Loss: 2.3660
Epoch: 1/2... Step: 130... Loss: 2.3140... Val Loss: 2.3344
Epoch: 2/2... Step: 140... Loss: 2.2809... Val Loss: 2.3130
Epoch: 2/2... Step: 150... Loss: 2.2499... Val Loss: 2.2947
Epoch: 2/2... Step: 160... Loss: 2.2279... Val Loss: 2.2600
Epoch: 2/2... Step: 170... Loss: 2.1881... Val Loss: 2.2499
Epoch: 2/2... Step: 180... Loss: 2.1471... Val Loss: 2.2251
Epoch: 2/2... Step: 190... Loss: 2.0920... Val Loss: 2.2013
Epoch: 2/2... Step: 200... Loss: 2.1030... Val Loss: 2.1710
Epoch: 2/2... Step: 210... Loss: 2.1045... Val Loss: 2.1453
Epoch: 2/2... Step: 220... Loss: 2.0518... Val Loss: 2.1335
Epoch: 2/2... Step: 230... Loss: 2.0606... Val Loss: 2.1150
Epoch: 2/2... Step: 240... Loss: 2.0451... Val Loss: 2.1078
Epoch: 2/2... Step: 250... Loss: 1.9938... Val Loss: 2.0795
Epoch: 2/2... Step: 260... Loss: 1.9563... Val Loss: 2.0665
Epoch: 2/2... Step: 270... Loss: 1.9850... Val Loss: 2.0472
```

Getting the best model

To set your hyperparameters to get the best performance, you'll want to watch the training and validation losses. If your training loss is much lower than the validation loss, you're overfitting. Increase regularization (more dropout) or use a smaller network. If the training and validation losses are close, you're underfitting so you can increase the size of the network.

After training, we'll save the model so we can load it again later if we need too. Here I'm saving the parameters needed to create the same architecture, the hidden layer hyperparameters and the text characters.

In [13]:

```
checkpoint = {'n_hidden': net.n_hidden,
              'n_layers': net.n_layers,
              'state_dict': net.state_dict(),
              'tokens': net.chars}
with open('rnn.net', 'wb') as f:
    torch.save(checkpoint, f)
```

Sampling

Now that the model is trained, we'll want to sample from it. To sample, we pass in a character and have the network predict the next character. Then we take that character, pass it back in, and get another predicted character. Just keep doing this and you'll generate a bunch of text!

Top K sampling

Our predictions come from a categorcial probability distribution over all the possible characters. We can make the sampled text more reasonable but less variable by only considering some K most probable characters. This will prevent the network from giving us completely absurd characters while allowing it to introduce some noise and randomness into the sampled text.

Typically you'll want to prime the network so you can build up a hidden state. Otherwise the network will start out generating characters at random. In general the first bunch of characters will be a little rough since it hasn't built up a long history of characters to predict from.

In [14]:

```
def sample(net, size, prime='The', top_k=None, cuda=False):

    if cuda:
        net.cuda()
    else:
        net.cpu()

    net.eval()

    # First off, run through the prime characters
    chars = [ch for ch in prime]
    h = net.init_hidden(1)
    for ch in prime:
        char, h = net.predict(ch, h, cuda=cuda, top_k=top_k)

    chars.append(char)

    # Now pass in the previous character and get a new one
    for ii in range(size):
        char, h = net.predict(chars[-1], h, cuda=cuda, top_k=top_k)
        chars.append(char)

    return ''.join(chars)
```

In [15]:

```
print(sample(net, 50, prime='Anna', top_k=5, cuda=False))
```

```
Anna tremeding
that in to sele the righat, and saint,
a
```

```
/users/gpu/anupriy/anaconda3/lib/python3.7/site-packages/ipykernel_l
auncher.py:49: UserWarning: volatile was removed and now has no effe
ct. Use `with torch.no_grad():` instead.
/users/gpu/anupriy/anaconda3/lib/python3.7/site-packages/ipykernel_l
auncher.py:53: UserWarning: volatile was removed and now has no effe
ct. Use `with torch.no_grad():` instead.
/users/gpu/anupriy/anaconda3/lib/python3.7/site-packages/ipykernel_l
auncher.py:56: UserWarning: Implicit dimension choice for softmax ha
s been deprecated. Change the call to include dim=X as an argument.
```

Loading a checkpoint

In [16]:

```
with open('rnn.net', 'rb') as f:
    checkpoint = torch.load(f)

loaded = CharRNN(checkpoint['tokens'], n_hidden=checkpoint['n_hidden'], n_layers
=checkpoint['n_layers'])
loaded.load_state_dict(checkpoint['state_dict'])
```

In [17]:

```
print(sample(loaded, 100, cuda=True, top_k=5, prime="AI"))
```

```
AI't to shis what it the conters wat to that ho have to dithen said
a the sinct," the pariens ond the m
```

```
/users/gpu/anupriy/anaconda3/lib/python3.7/site-packages/ipykernel_l
auncher.py:49: UserWarning: volatile was removed and now has no effe
ct. Use `with torch.no_grad():` instead.
/users/gpu/anupriy/anaconda3/lib/python3.7/site-packages/ipykernel_l
auncher.py:53: UserWarning: volatile was removed and now has no effe
ct. Use `with torch.no_grad():` instead.
/users/gpu/anupriy/anaconda3/lib/python3.7/site-packages/ipykernel_l
auncher.py:56: UserWarning: Implicit dimension choice for softmax ha
s been deprecated. Change the call to include dim=X as an argument.
```