Chararacter-Level RNN, Solution

May 31, 2020

1 Character-Level LSTM in PyTorch

In this notebook, I'll construct a character-level LSTM with PyTorch. The network will train character by character on some text, then generate new text character by character. As an example, I will train on Anna Karenina. **This model will be able to generate new text based on the text from the book!**

This network is based off of Andrej Karpathy's post on RNNs and implementation in Torch. Below is the general architecture of the character-wise RNN.

First let's load in our required resources for data loading and model creation.

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

1.1 Load in Data

Then, we'll load the Anna Karenina text file and convert it into integers for our network to use.

1.1.1 Tokenization

In the second cell, below, I'm creating a couple **dictionaries** to convert the characters to and from integers. Encoding the characters as integers makes it easier to use as input in the network.

```
In [2]: # open text file and read in data as `text`
        with open('data/anna.txt', 'r') as f:
            text = f.read()
```

Now we have the text, encode it as integers.

```
In [3]: # encode the text and map each character to an integer and vice versa

        # we create two dictonaries:
        # 1. int2char, which maps integers to characters
        # 2. char2int, which maps characters to unique integers
        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])
```

Let's check out the first 100 characters, make sure everything is peachy. According to the American Book Review, this is the 6th best first line of a book ever.

```
In [17]: text[:100]
```

```
Out[17]: 'Chapter 1\n\n\nHappy families are all alike; every unhappy family is unhappy in its ow
```

And we can see those same characters encoded as integers.

```
In [18]: encoded[:100]
```

```
Out[18]: array([11, 22, 79, 57, 27, 36, 43, 40, 17, 77, 77, 77, 37, 79, 57, 57, 28,
                40, 13, 79, 10, 23, 33, 23, 36, 31, 40, 79, 43, 36, 40, 79, 33, 33,
                40, 79, 33, 23, 62, 36, 25, 40, 36, 64, 36, 43, 28, 40, 66, 21, 22,
                79, 57, 57, 28, 40, 13, 79, 10, 23, 33, 28, 40, 23, 31, 40, 66, 21,
                22, 79, 57, 57, 28, 40, 23, 21, 40, 23, 27, 31, 40,  9, 44, 21, 77,
                44, 79, 28, 81, 77, 77, 20, 64, 36, 43, 28, 27, 22, 23, 21])
```

1.2 Pre-processing the data

As you can see in our char-RNN image above, our LSTM expects an input that is **one-hot encoded** meaning that each character is converted into an intgere (via our created dictionary) and *then* converted into a column vector where only it's corresponsing integer index will have the value of 1 and the rest of the vector will be filled with 0's. Since we're one-hot encoding the data, let's make a function to do that!

```
In [4]: 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
```

1.3 Making training mini-batches

To train on this data, we also want to create mini-batches for training. Remember that we want our batches to be multiple sequences of some desired number of sequence steps. Considering a simple example, our batches would look like this:

In this example, we'll take the encoded characters (passed in as the `arr` parameter) 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.

1.3.1 Creating Batches

1. The first thing we need to do is discard some of the text so we only have completely full batches.

Each batch contains $N \times M$ characters, where N is the batch size (the number of sequences) and M is the number of steps. Then, to get the total number of batches, K, we can make from the array `arr`, you divide the length of `arr` by the number of characters per batch. Once you know the number of batches, you can get the total number of characters to keep from `arr`, $N * M * K$.

2. After that, we need to split `arr` into N sequences.

You can do this using `arr.reshape(size)` where `size` is a tuple containing the dimensions sizes of the reshaped array. We know we want N sequences, so let's make that the size of the first dimension. For the second dimension, you can use `-1` as a placeholder in the size, it'll fill up the array with the appropriate data for you. After this, you should have an array that is $N \times (M * K)$.

3. Now that we have this array, we can iterate through it to get our batches.

The idea is each batch is a $N \times M$ window on the $N \times (M * K)$ array. For each subsequent batch, the window moves over by `n_steps`. We also want to create both the input and target arrays. Remember that the targets are the inputs shifted over one character. The way I like to do this window is use `range` to take steps of size `n_steps` from 0 to `arr.shape[1]`, the total number of steps in each sequence. That way, the integers you get from `range` always point to the start of a batch, and each window is `n_steps` wide.

> **TODO:** Write the code for creating batches in the function below. The exercises in this notebook *will not be easy*. I've provided a notebook with solutions alongside this notebook. If you get stuck, checkout the solutions. The most important thing is that you don't copy and paste the code into here, **type out the solution code yourself.**

```
In [6]: def get_batches(arr, n_seqs, n_steps):
            '''Create a generator that returns batches of size
               n_seqs x n_steps from arr.

               Arguments
               ---------
               arr: Array you want to make batches from
               n_seqs: Batch size, the number of sequences per batch
               n_steps: Number of sequence steps per batch
            '''

            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
```

1.3.2 Test Your Implementation

Now I'll make some data sets and we can check out what's going on as we batch data. Here, as an example, I'm going to use a batch size of 10 and 50 sequence steps.

```
In [19]: batches = get_batches(encoded, 10, 50)
         x, y = next(batches)

In [20]: print('x\n', x[:10, :10])
         print('\ny\n', y[:10, :10])
```

```
x
 [[11 22 79 57 27 36 43 40 17 77]
 [40 79 10 40 21  9 27 40 58  9]
 [64 23 21 81 77 77 45 82 36 31]
 [21 40 80 66 43 23 21 58 40 22]
 [40 23 27 40 23 31 78 40 31 23]
 [40 51 27 40 44 79 31 77  9 21]
 [22 36 21 40 29  9 10 36 40 13]
 [25 40 39 66 27 40 21  9 44 40]
 [27 40 23 31 21 41 27 81 40 68]
 [40 31 79 23 80 40 27  9 40 22]]

y
 [[22 79 57 27 36 43 40 17 77 77]
 [79 10 40 21  9 27 40 58  9 23]
 [23 21 81 77 77 45 82 36 31 78]
 [40 80 66 43 23 21 58 40 22 23]
 [23 27 40 23 31 78 40 31 23 43]
 [51 27 40 44 79 31 77  9 21 33]
 [36 21 40 29  9 10 36 40 13  9]
 [40 39 66 27 40 21  9 44 40 31]
 [40 23 31 21 41 27 81 40 68 22]
 [31 79 23 80 40 27  9 40 22 36]]
```

If you implemented `get_batches` correctly, the above output should look something like "' x [[55 63 69 22 6 76 45 5 16 35][5 69 1 5 12 52 6 5 56 52] [48 29 12 61 35 35 8 64 76 78][12 5 24 39 45 29 12 56 5 63] [5 29 6 5 29 78 28 5 78 29][5 13 6 5 36 69 78 35 52 12] [63 76 12 5 18 52 1 76 5 58][34 5 73 39 6 5 12 52 36 5] [6 5 29 78 12 79 6 61 5 59][5 78 69 29 24 5 6 52 5 63]]

y [[63 69 22 6 76 45 5 16 35 35][69 1 5 12 52 6 5 56 52 29] [29 12 61 35 35 8 64 76 78 28][5 24 39 45 29 12 56 5 63 29] [29 6 5 29 78 28 5 78 29 45][13 6 5 36 69 78 35 52 12 43] [76 12 5 18

52 1 76 5 58 52][5 73 39 6 5 12 52 36 5 78] [5 29 78 12 79 6 61 5 59 63][78 69 29 24 5 6 52 5 63 76]] "`although the exact numbers will be different. Check to make sure the data is shifted over one step fory`'.

1.4 Defining the network with PyTorch

Below is where you'll define the network. We'll break it up into parts so it's easier to reason about each bit. Then we can connect them up into the whole network.

Next, you'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. You've also been given a method for predicting characters.

1.4.1 Model Structure

In `__init__` the suggested structure is as follows: * Create and store the necessary dictionaries (this has been done for you) * Define an LSTM layer that takes as params: an input size (the number of characters), a hidden layer size `n_hidden`, a number of layers `n_layers`, a dropout probability `drop_prob`, and a batch_first boolean (True, since we are batching) * Define a dropout layer with `dropout_prob` * Define a fully-connected layer with params: input size `n_hidden` and output size (the number of characters) * Finally, initialize the weights (again, this has been given)

Note that some parameters have been named and given in the `__init__` function, and we use them and store them by doing something like `self.drop_prob = drop_prob`.

1.4.2 LSTM Inputs/Outputs

You can create a basic LSTM cell as follows

```
self.lstm = nn.LSTM(input_size, n_hidden, n_layers,
                            dropout=drop_prob, batch_first=True)
```

where `input_size` is the number of characters this cell expects to see as sequential input, and `n_hidden` is the number of units in the hidden layers in the cell. And we can add dropout by adding a dropout parameter with a specified probability; this will automatically add dropout to the inputs or outputs. Finally, in the `forward` function, we can stack up the LSTM cells into layers using `.view`. With this, you pass in a list of cells and it will send the output of one cell into the next cell.

We also need to create an initial cell state of all zeros. This is done like so

```
self.init_weights()
```

```
In [21]: 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().__init__()
```

```
        self.drop_prob = drop_prob
        self.n_layers = n_layers
        self.n_hidden = n_hidden
        self.lr = lr

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

        ## TODO: define the LSTM
        self.lstm = nn.LSTM(len(self.chars), n_hidden, n_layers,
                            dropout=drop_prob, batch_first=True)

        ## TODO: define a dropout layer
        self.dropout = nn.Dropout(drop_prob)

        ## TODO: define the final, fully-connected output layer
        self.fc = nn.Linear(n_hidden, len(self.chars))

        # initialize the weights
        self.init_weights()


    def forward(self, x, hc):
        ''' Forward pass through the network.
            These inputs are x, and the hidden/cell state `hc`. '''

        ## TODO: Get x, and the new hidden state (h, c) from the lstm
        x, (h, c) = self.lstm(x, hc)

        ## TODO: pass x through a droupout layer
        x = self.dropout(x)

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

        ## TODO: put x through the fully-connected layer
        x = self.fc(x)

        # return x and the hidden state (h, c)
        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 = torch.from_numpy(x)
        if cuda:
            inputs = inputs.cuda()

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

        p = F.softmax(out, dim=1).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 (weight.new(self.n_layers, n_seqs, self.n_hidden).zero_(),
```

```
                      weight.new(self.n_layers, n_seqs, self.n_hidden).zero_())
```

1.4.3 A note on the `predict` function

The output of our RNN is from a fully-connected layer and it outputs a **distribution of next-character scores**.

To actually get the next character, we apply a softmax function, which gives us a *probability* distribution that we can then sample to predict the next character.

```
In [11]: def train(net, data, epochs=10, n_seqs=10, n_steps=50, lr=0.001, clip=5, val_frac=0.1,
             ''' Training 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)
inputs, targets = torch.from_numpy(x), torch.from_numpy(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([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 RNNs / L
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([each.data for each in val_h])

        inputs, targets = x, y
        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.item())

    print("Epoch: {}/{}...".format(e+1, epochs),
          "Step: {}...".format(counter),
          "Loss: {:.4f}...".format(loss.item()),
```

```
                    "Val Loss: {:.4f}".format(np.mean(val_losses)))
```

1.5 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 [12]: if 'net' in locals():
             del net
```

```
In [13]: # define and print the net
         net = CharRNN(chars, n_hidden=512, n_layers=2)
         print(net)
```

```
CharRNN(
  (dropout): Dropout(p=0.5)
  (lstm): LSTM(83, 512, num_layers=2, batch_first=True, dropout=0.5)
  (fc): Linear(in_features=512, out_features=83, bias=True)
)
```

```
In [14]: n_seqs, n_steps = 128, 100

         # you may change cuda to True if you plan on using a GPU!
         # also, if you do, please INCREASE the epochs to 25
         train(net, encoded, epochs=1, n_seqs=n_seqs, n_steps=n_steps, lr=0.001, cuda=False, pri
```

```
Epoch: 1/1... Step: 10... Loss: 3.3368... Val Loss: 3.2920
Epoch: 1/1... Step: 20... Loss: 3.1798... Val Loss: 3.1867
Epoch: 1/1... Step: 30... Loss: 3.0811... Val Loss: 3.0596
Epoch: 1/1... Step: 40... Loss: 2.8768... Val Loss: 2.8971
Epoch: 1/1... Step: 50... Loss: 2.7606... Val Loss: 2.7167
Epoch: 1/1... Step: 60... Loss: 2.6058... Val Loss: 2.6286
Epoch: 1/1... Step: 70... Loss: 2.5333... Val Loss: 2.5545
Epoch: 1/1... Step: 80... Loss: 2.4775... Val Loss: 2.5015
Epoch: 1/1... Step: 90... Loss: 2.4486... Val Loss: 2.4600
Epoch: 1/1... Step: 100... Loss: 2.3879... Val Loss: 2.4225
Epoch: 1/1... Step: 110... Loss: 2.3463... Val Loss: 2.3912
Epoch: 1/1... Step: 120... Loss: 2.2940... Val Loss: 2.3635
Epoch: 1/1... Step: 130... Loss: 2.3087... Val Loss: 2.3345
```

1.6 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.