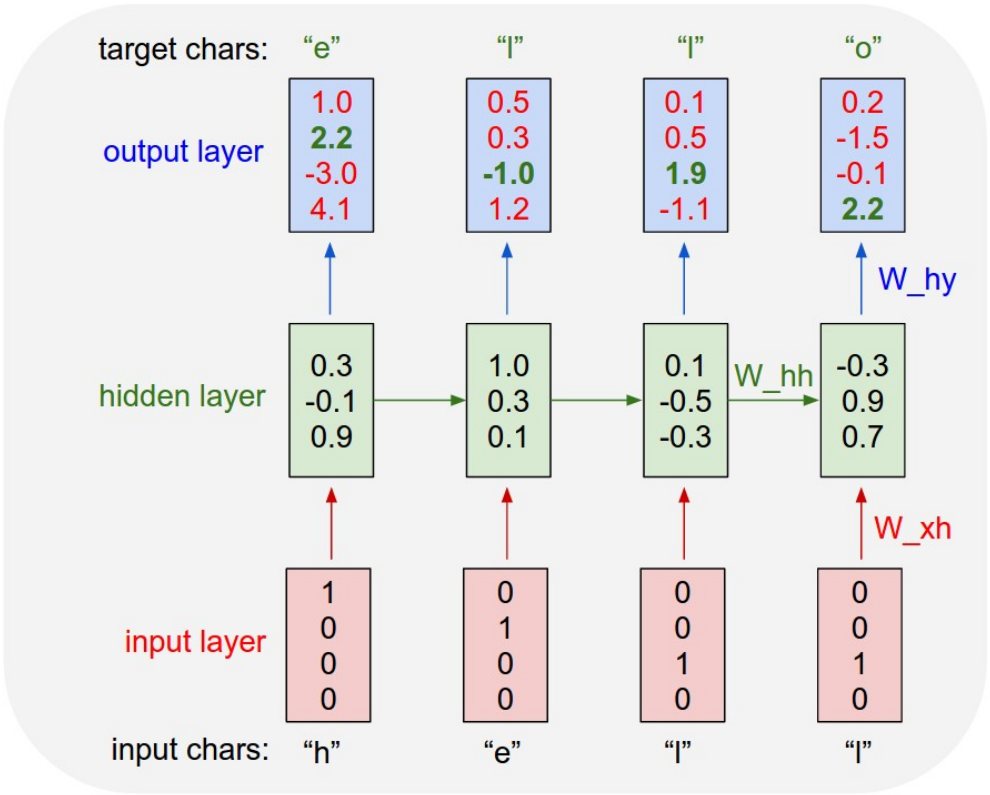
Character-Level LSTM in PyTorch

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 (http://karpathy.github.io/2015/05/21/rnn-effectiveness/) and implementation in Torch (https://github.com/karpathy/char-rnn). 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
```

Load in Data

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

In [2]:

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

Let's check out the first 100 characters, make sure everything is peachy. According to the American Book Review (http://americanbookreview.org/100bestlines.asp), this is the 6th best first line of a book ever.

In [3]:

```
text[:100]
```

Out[3]:

```
'Chapter 1\n\n\nHappy families are all alike; every unhappy family is
unhappy in its own\nway.\n\nEverythin'
```

Tokenization

In the cells, 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 [4]:

```
# encode the text and map each character to an integer and vice versa

# we create two dictionaries:
# 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()}

# encode the text
encoded = np.array([char2int[ch] for ch in text])
```

And we can see those same characters from above, encoded as integers.

In [5]:

```
encoded[:100]
```

Out[5]:

```
array([65, 12, 19, 35, 59, 47, 80, 21,  8, 24, 24, 24, 32, 19, 35, 35,
45,
       21, 46, 19, 72,  0, 25,  0, 47, 23, 21, 19, 80, 47, 21, 19, 25,
25,
       21, 19, 25,  0, 76, 47, 50, 21, 47, 56, 47, 80, 45, 21, 44,  2,
12,
       19, 35, 35, 45, 21, 46, 19, 72,  0, 25, 45, 21,  0, 23, 21, 44,
  2,
       12, 19, 35, 35, 45, 21,  0,  2, 21,  0, 59, 23, 21, 81, 14,  2,
24,
       14, 19, 45,  9, 24, 24, 61, 56, 47, 80, 45, 59, 12,  0,  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 integer (via our created dictionary) and *then* converted into a column vector where only it's corresponding 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 [6]:

```
def one_hot_encode(arr, n_labels):

    # Initialize the the encoded array
    one_hot = np.zeros((arr.size, 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]:

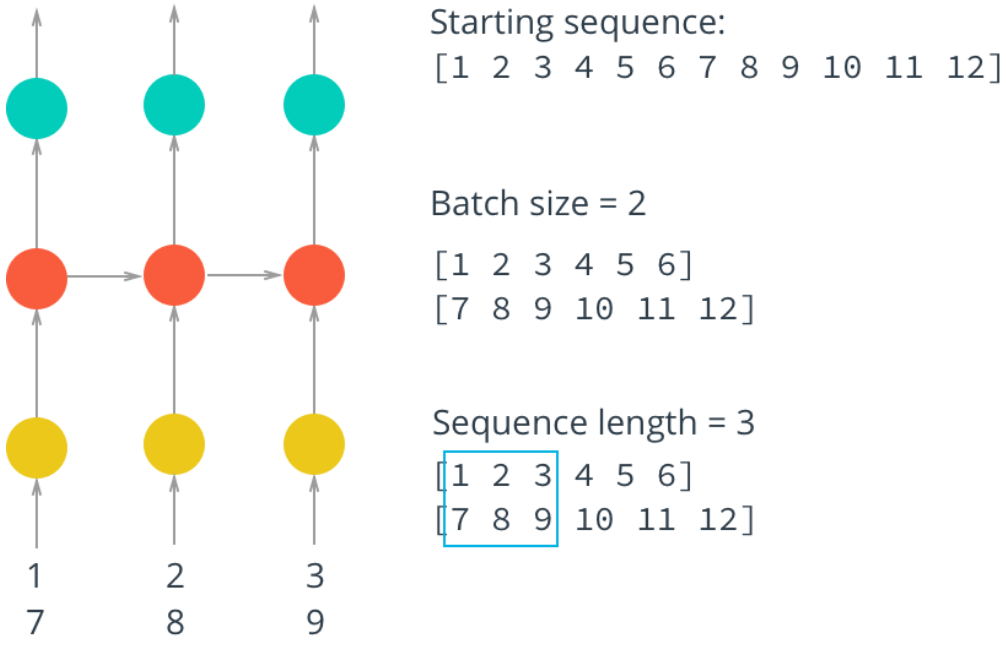
```
# check that the function works as expected
test_seq = np.array([[3, 5, 1]])
one_hot = one_hot_encode(test_seq, 8)

print(one_hot)
```

```
[[[ 0.  0.  0.  1.  0.  0.  0.  0.]
  [ 0.  0.  0.  0.  0.  1.  0.  0.]
  [ 0.  1.  0.  0.  0.  0.  0.  0.]]]
```

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 `batch_size`. Each of our sequences will be `seq_length` long.

Creating Batches

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

Each batch contains $N \times M$ characters, where N is the batch size (the number of sequences in a batch) and M is the seq_length or number of time steps in a sequence. Then, to get the total number of batches, K, that 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 batches. *

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 in a batch, 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 mini-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 `seq_length`. We also want to create both the input and target arrays. Remember that the targets are just the inputs shifted over by 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 tokens in each sequence. That way, the integers you get from `range` always point to the start of a batch, and each window is `seq_length` 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 [8]:

```
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
```

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 8 and 50 sequence steps.

In [9]:

```
batches = get_batches(encoded, 8, 50)
x, y = next(batches)
```

In [10]:

```
# printing out the first 10 items in a sequence
print('x\n', x[:10, :10])
print('\ny\n', y[:10, :10])
```

```
x
 [[65 12 19 35 59 47 80 21  8 24]
 [23 81  2 21 59 12 19 59 21 19]
 [47  2 77 21 81 80 21 19 21 46]
 [23 21 59 12 47 21 40 12  0 47]
 [21 23 19 14 21 12 47 80 21 59]
 [40 44 23 23  0 81  2 21 19  2]
 [21 57  2  2 19 21 12 19 77 21]
 [27 26 25 81  2 23 76 45  9 21]]

y
 [[12 19 35 59 47 80 21  8 24 24]
 [81  2 21 59 12 19 59 21 19 59]
 [ 2 77 21 81 80 21 19 21 46 81]
 [21 59 12 47 21 40 12  0 47 46]
 [23 19 14 21 12 47 80 21 59 47]
 [44 23 23  0 81  2 21 19  2 77]
 [57  2  2 19 21 12 19 77 21 23]
 [26 25 81  2 23 76 45  9 21 39]]
```

If you implemented `get_batches` correctly, the above output should look something like

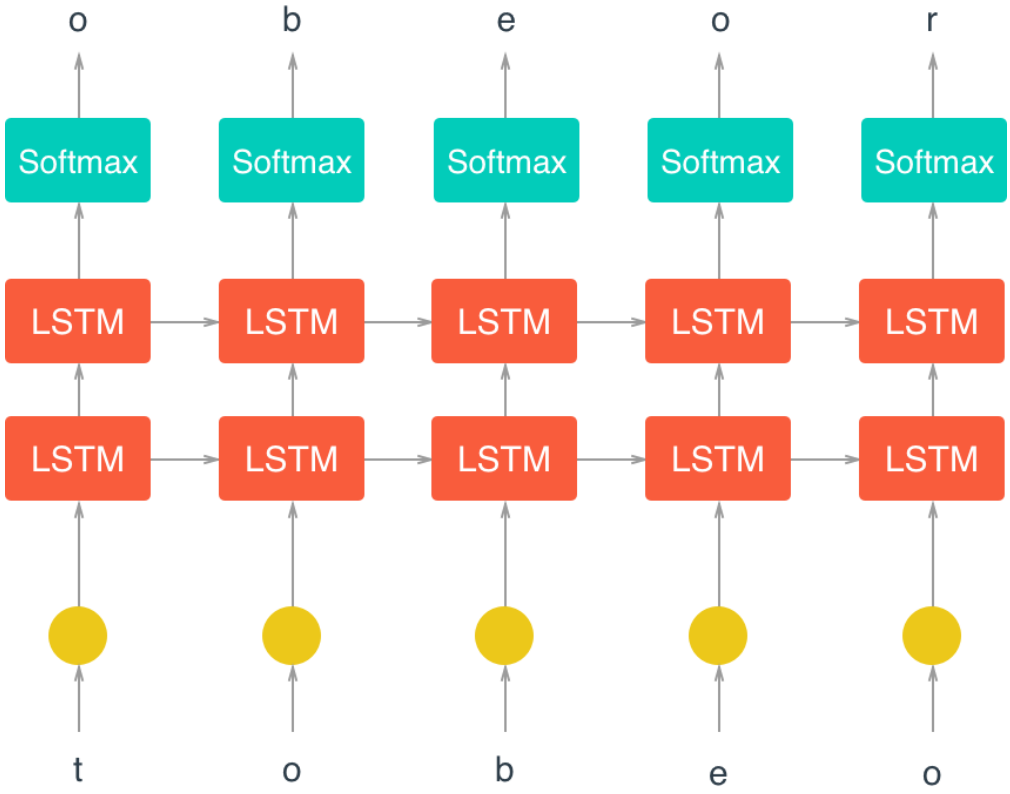
```
x
 [[25  8 60 11 45 27 28 73  1  2]
 [17  7 20 73 45  8 60 45 73 60]
 [27 20 80 73  7 28 73 60 73 65]
 [17 73 45  8 27 73 66  8 46 27]
 [73 17 60 12 73  8 27 28 73 45]
 [66 64 17 17 46  7 20 73 60 20]
 [73 76 20 20 60 73  8 60 80 73]
 [47 35 43  7 20 17 24 50 37 73]]

y
 [[ 8 60 11 45 27 28 73  1  2  2]
 [ 7 20 73 45  8 60 45 73 60 45]
 [20 80 73  7 28 73 60 73 65  7]
 [73 45  8 27 73 66  8 46 27 65]
 [17 60 12 73  8 27 28 73 45 27]
 [64 17 17 46  7 20 73 60 20 80]
 [76 20 20 60 73  8 60 80 73 17]
 [35 43  7 20 17 24 50 37 73 36]]
```

although the exact numbers may be different. Check to make sure the data is shifted over one step for `y`.

Defining the network with PyTorch

Below is where you'll define the 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.

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 `drop_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`.

LSTM Inputs/Outputs

You can create a basic [LSTM layer (https://pytorch.org/docs/stable/nn.html#lstm)](https://pytorch.org/docs/stable/nn.html#lstm) 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 hidden state of all zeros. This is done like so

```
self.init_hidden()
```

In [11]:

```
# check if GPU is available
train_on_gpu = torch.cuda.is_available()
if(train_on_gpu):
    print('Training on GPU!')
else:
    print('No GPU available, training on CPU; consider making n_epochs very small.'
```

```
Training on GPU!
```

In [12]:

```
class CharRNN(nn.Module):

    def __init__(self, tokens, 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))


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

        ## TODO: Get the outputs and the new hidden state from the lstm
        r_output, hidden = self.lstm(x, hidden)

        ## TODO: pass through a dropout layer
        out = self.dropout(r_output)

        # Stack up LSTM outputs using view
        # you may need to use contiguous to reshape the output
        out = out.contiguous().view(-1, self.n_hidden)

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

        # return the final output and the hidden state
        return out, hidden


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

        if (train_on_gpu):
            hidden = (weight.new(self.n_layers, batch_size, self.n_hidden).zero_().
                  weight.new(self.n_layers, batch_size, self.n_hidden).zero_().cuda
        else:
            hidden = (weight.new(self.n_layers, batch_size, self.n_hidden).zero_(),
                      weight.new(self.n_layers, batch_size, self.n_hidden).zero_())
```

```
        return hidden
```

Time to train

The train function gives us the ability to set the number of epochs, the learning rate, and other parameters.

Below we're using an Adam optimizer and cross entropy loss since we are looking at character class scores as output. We calculate the loss and perform backpropagation, as usual!

A couple of details about training:

- Within the batch loop, we detach the hidden state from its history; this time setting it equal to a new *tuple* variable because an LSTM has a hidden state that is a tuple of the hidden and cell states.
- We use `clip_grad_norm_` (https://pytorch.org/docs/stable/_modules/torch/nn/utils/clip_grad.html) to help prevent exploding gradients.

In [13]:

```
def train(net, data, epochs=10, batch_size=10, seq_length=50, lr=0.001, clip=5, val
    ''' Training a network

        Arguments
        ---------

        net: 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 size
        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 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(train_on_gpu):
        net.cuda()

    counter = 0
    n_chars = len(net.chars)
    for e in range(epochs):
        # initialize hidden state
        h = net.init_hidden(batch_size)

        for x, y in get_batches(data, batch_size, seq_length):
            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(train_on_gpu):
                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])

            # zero accumulated gradients
            net.zero_grad()

            # get the output from the model
            output, h = net(inputs, h)

            # calculate the loss and perform backprop
            loss = criterion(output, targets.view(batch_size*seq_length).long())
            loss.backward()
            # `clip_grad_norm` helps prevent the exploding gradient problem in RNNs
```

```
            nn.utils.clip_grad_norm_(net.parameters(), clip)
            opt.step()

            # loss stats
            if counter % print_every == 0:
                # Get validation loss
                val_h = net.init_hidden(batch_size)
                val_losses = []
                net.eval()
                for x, y in get_batches(val_data, batch_size, seq_length):
                    # 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(train_on_gpu):
                        inputs, targets = inputs.cuda(), targets.cuda()

                    output, val_h = net(inputs, val_h)
                    val_loss = criterion(output, targets.view(batch_size*seq_length

                    val_losses.append(val_loss.item())

                net.train() # reset to train mode after iterationg through validati

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

Instantiating the model

Now we can actually train the network. First we'll create the network itself, with some given hyperparameters. Then, define the mini-batches sizes, and start training!

In [14]:

```
# define and print the net
n_hidden=512
n_layers=2

net = CharRNN(chars, n_hidden, n_layers)
print(net)
```

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

In [15]:

```
batch_size = 128
seq_length = 100
n_epochs = 20 # start smaller if you are just testing initial behavior

# train the model
train(net, encoded, epochs=n_epochs, batch_size=batch_size, seq_length=seq_length,
```

```
Epoch: 1/20... Step: 10... Loss: 3.2482... Val Loss: 3.2114
Epoch: 1/20... Step: 20... Loss: 3.1410... Val Loss: 3.1354
Epoch: 1/20... Step: 30... Loss: 3.1360... Val Loss: 3.1238
Epoch: 1/20... Step: 40... Loss: 3.1139... Val Loss: 3.1195
Epoch: 1/20... Step: 50... Loss: 3.1408... Val Loss: 3.1170
Epoch: 1/20... Step: 60... Loss: 3.1161... Val Loss: 3.1144
Epoch: 1/20... Step: 70... Loss: 3.1051... Val Loss: 3.1113
Epoch: 1/20... Step: 80... Loss: 3.1133... Val Loss: 3.1029
Epoch: 1/20... Step: 90... Loss: 3.1048... Val Loss: 3.0833
Epoch: 1/20... Step: 100... Loss: 3.0508... Val Loss: 3.0351
Epoch: 1/20... Step: 110... Loss: 2.9844... Val Loss: 2.9579
Epoch: 1/20... Step: 120... Loss: 2.8520... Val Loss: 2.8698
Epoch: 1/20... Step: 130... Loss: 2.7709... Val Loss: 2.7311
Epoch: 2/20... Step: 140... Loss: 2.7156... Val Loss: 2.6316
Epoch: 2/20... Step: 150... Loss: 2.5914... Val Loss: 2.5473
Epoch: 2/20... Step: 160... Loss: 2.5292... Val Loss: 2.4892
Epoch: 2/20... Step: 170... Loss: 2.4596... Val Loss: 2.4423
Epoch: 2/20... Step: 180... Loss: 2.4390... Val Loss: 2.4093
Epoch: 2/20... Step: 190... Loss: 2.3830... Val Loss: 2.3769
```

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.

Hyperparameters

Here are the hyperparameters for the network.

In defining the model:

- `n_hidden` - The number of units in the hidden layers.
- `n_layers` - Number of hidden LSTM layers to use.

We assume that dropout probability and learning rate will be kept at the default, in this example.

And in training:

- `batch_size` - Number of sequences running through the network in one pass.
- `seq_length` - Number of characters in the sequence the network is trained on. Larger is better typically, the network will learn more long range dependencies. But it takes longer to train. 100 is typically a good number here.
- `lr` - Learning rate for training

Here's some good advice from Andrej Karpathy on training the network. I'm going to copy it in here for your benefit, but also link to [where it originally came from (https://github.com/karpathy/char-rnn#tips-and-tricks)](https://github.com/karpathy/char-rnn#tips-and-tricks).

Tips and Tricks

Monitoring Validation Loss vs. Training Loss

If you're somewhat new to Machine Learning or Neural Networks it can take a bit of expertise to get good models. The most important quantity to keep track of is the difference between your training loss (printed during training) and the validation loss (printed once in a while when the RNN is run on the validation data (by default every 1000 iterations)). In particular:

- If your training loss is much lower than validation loss then this means the network might be **overfitting**. Solutions to this are to decrease your network size, or to increase dropout. For example you could try dropout of 0.5 and so on.
- If your training/validation loss are about equal then your model is **underfitting**. Increase the size of your model (either number of layers or the raw number of neurons per layer)

Approximate number of parameters

The two most important parameters that control the model are `n_hidden` and `n_layers`. I would advise that you always use `n_layers` of either 2/3. The `n_hidden` can be adjusted based on how much data you have. The two important quantities to keep track of here are:

- The number of parameters in your model. This is printed when you start training.
- The size of your dataset. 1MB file is approximately 1 million characters.

These two should be about the same order of magnitude. It's a little tricky to tell. Here are some examples:

- I have a 100MB dataset and I'm using the default parameter settings (which currently print 150K parameters). My data size is significantly larger (100 mil >> 0.15 mil), so I expect to heavily underfit. I am thinking I can comfortably afford to make `n_hidden` larger.
- I have a 10MB dataset and running a 10 million parameter model. I'm slightly nervous and I'm carefully monitoring my validation loss. If it's larger than my training loss then I may want to try to increase dropout a bit and see if that helps the validation loss.

Best models strategy

The winning strategy to obtaining very good models (if you have the compute time) is to always err on making the network larger (as large as you're willing to wait for it to compute) and then try different dropout values (between 0,1). Whatever model has the best validation performance (the loss, written in the checkpoint filename, low is good) is the one you should use in the end.

> It is very common in deep learning to run many different models with many different hyperparameter settings, and in the end take whatever checkpoint gave the best validation performance.

> By the way, the size of your training and validation splits are also parameters. Make sure you have a decent amount of data in your validation set or otherwise the validation performance will be noisy and not very informative.

Checkpoint

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 [16]:

```
# change the name, for saving multiple files
model_name = 'rnn_20_epoch.net'

checkpoint = {'n_hidden': net.n_hidden,
              'n_layers': net.n_layers,
              'state_dict': net.state_dict(),
              'tokens': net.chars}

with open(model_name, 'wb') as f:
    torch.save(checkpoint, f)
```

Making Predictions

Now that the model is trained, we'll want to sample from it and make predictions about next characters! 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!

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.

Top K sampling

Our predictions come from a categorical probability distribution over all the possible characters. We can make the sample text and make it more reasonable to handle (with less variables) 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. Read more about topk, here (https://pytorch.org/docs/stable/torch.html#torch.topk).

In [17]:

```
def predict(net, 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([[net.char2int[char]]])
        x = one_hot_encode(x, len(net.chars))
        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])
        # get the output of the model
        out, h = net(inputs, h)

        # 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(net.chars))
        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 net.int2char[char], h
```

Priming and generating 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 [18]:

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

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

    net.eval() # eval mode

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

    return ''.join(chars)
```

In [19]:

```
print(sample(net, 1000, prime='Anna', top_k=5))
```

```
Anna had so that an enter strength to be says off and he cared to be a
n unmarrely sister.

The children are saying in a place. A smile of their secretary and the
sense of a condition. He saw that the princess was the same, the peaci
ting of his
briderous country second still. That she had seen him a little as it w
as the simminest that he had not been
the simple of
the passion to see his finger, and
his brother and the points he heard this place which
he was not
sense. All had sent him that he could he concealed the steps and that
he was to be patied,
so much at hands, at the servants who had said something with the
chair.
 "This is a solitat matter?"

"It's not thinking in the more the point is and that he's talking of t
he drinking of the
crain. If I was a memory. Have you
seen my to thousard more
characteribries, and this, and would be the framing of the most carefu
l towards me, to the country too that they did nothind when she could
not see him. What is
it you want a conviluated more to mo
```

Loading a checkpoint

In [20]:

```
# Here we have loaded in a model that trained over 20 epochs `rnn_20_epoch.net`
with open('rnn_20_epoch.net', 'rb') as f:
    checkpoint = torch.load(f)

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

In [21]:

```
# Sample using a loaded model
print(sample(loaded, 2000, top_k=5, prime="And Levin said"))
```

```
And Levin said those second portryit on the contrast.

"What is it?" said Stepan Arkadyevitch,
letting up his
shirt and talking to her face. And he had
not speak to Levin that his head on the round stop and
trouble
to be faint, as he
was not a man who was said, she was the setter times that had been bef
ore so much talking in the steps of the door, his force to think of th
eir sense of the sendence, both always bowing about in the country and
the same time of her character and all at him with his face, and went
out of her hand, sitting down beside
the clothes, and
the
same
single mind and when they seemed to a strange of his
brother's.

And he
was so meched the paints was so standing the man had been a love was t
he man, and stopped at once in the first step. But he was
a change to
do. The sound of the partice say a construnting his
steps and telling a single camp of the
ready and three significance of the same forest.

"Yes, but you see it." He carried his face and the condition in their
carriage to her, and to go, she
said that had been talking of his forest, a strange world, when Levin
came the conversation as sense of her son, and he could not see him to
hive answer, which had been saking when at tomere within the
counting her face that he was serenely from her she took a counting, t
here
was the since he
had too wearted and seemed to her," said the member of the cannors in
the steps to his
word.

The moss of the convincing it had been drawing up the people that ther
e was nothing without this way or a single wife as he did not hear
him or that he was not seeing that she would be a court of the sound o
f some sound of the position, and to spartly she
could
see her and a sundroup times there was nothing this
father and as she stoop serious in the sound, was a steps of the maste
r, a few sistersily play of his husband. The crowd had no carreated he
rself, and truets, and shaking up, the pases, and the moment that he w
as not at the marshal, and the starling the secret were stopping to be
```

In [ ]:

```

```