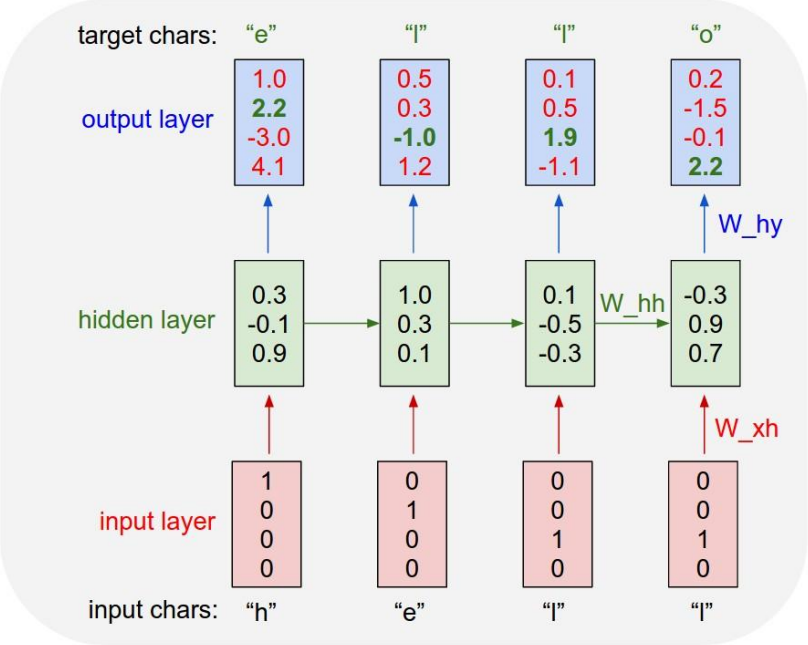
Anna KaRNNa

In this notebook, I'll build a character-wise RNN trained on Anna Karenina, one of my all-time favorite books. It'll 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. Also, some information here at r2rtand from Sherjil Ozair on GitHub. Below is the general architecture of the character-wise RNN.



In [1]:

```
import time
from collections import namedtuple
import numpy as np
import tensorflow as tf
```

First we'll load the text file and convert it into integers for our network to use. Here 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]:

```
with open('anna.txt', 'r') as f:
    text=f.read()
vocab = set(text)
vocab_to_int = {c: i for i, c in enumerate(vocab)}
int_to_vocab = dict(enumerate(vocab))
chars = np.array([vocab_to_int[c] for c in text], dtype=np.int32)
```

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 [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'
```

And we can see the characters encoded as integers.

In [4]:

```
chars[:100]
```

Out[4]:

```
array([46, 19, 18, 44, 49, 58,  6, 75, 60, 61, 61, 61, 47, 18, 44, 44, 80,
       75, 23, 18, 50, 25, 57, 25, 58,  9, 75, 18,  6, 58, 75, 18, 57, 57,
       75, 18, 57, 25, 13, 58, 42, 75, 58, 62, 58,  6, 80, 75, 76, 65, 19,
       18, 44, 44, 80, 75, 23, 18, 50, 25, 57, 80, 75, 25,  9, 75, 76, 65,
       19, 18, 44, 44, 80, 75, 25, 65, 75, 25, 49,  9, 75, 10, 28, 65, 61,
       28, 18, 80, 37, 61, 61, 26, 62, 58,  6, 80, 49, 19, 25, 65], dtype=int32)
```

Since the network is working with individual characters, it's similar to a classification problem in which we are trying to predict the next character from the previous text. Here's how many 'classes' our network has to pick from.

In [5]:

```
np.max(chars)+1
```

Out[5]:

```
83
```

Making training and validation batches

Now I need to split up the data into batches, and into training and validation sets. I should be making a test set here, but I'm not going to worry about that. My test will be if the network can generate new text.

Here I'll make both input and target arrays. The targets are the same as the inputs, except shifted one character over. I'll also drop the last bit of data so that I'll only have completely full batches.

The idea here is to make a 2D matrix where the number of rows is equal to the batch size. Each row will be one long concatenated string from the character data. We'll split this data into a training set and validation set using the `split_frac`keyword. This will keep 90% of the batches in the training set, the other 10% in the validation set.

In [6]:

```
def split_data(chars, batch_size, num_steps, split_frac=0.9):
    """
    Split character data into training and validation sets, inputs and targets for each set.

    Arguments
    ---------
    chars: character array
    batch_size: Size of examples in each of batch
    num_steps: Number of sequence steps to keep in the input and pass to the network
    split_frac: Fraction of batches to keep in the training set

    Returns train_x, train_y, val_x, val_y
    """


    slice_size = batch_size * num_steps
    n_batches = int(len(chars) / slice_size)

    # Drop the last few characters to make only full batches
    x = chars[: n_batches*slice_size]
    y = chars[1: n_batches*slice_size + 1]

    # Split the data into batch_size slices, then stack them into a 2D matrix
    x = np.stack(np.split(x, batch_size))
    y = np.stack(np.split(y, batch_size))
```

```
    # Now x and y are arrays with dimensions batch_size x n_batches*num_steps

    # Split into training and validation sets, keep the first split_frac
batches for training
    split_idx = int(n_batches*split_frac)
    train_x, train_y= x[:, :split_idx*num_steps], y[:, :split_idx*num_steps]
    val_x, val_y = x[:, split_idx*num_steps:], y[:, split_idx*num_steps:]

    return train_x, train_y, val_x, val_y
```

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

In [7]:

```
train_x, train_y, val_x, val_y = split_data(chars, 10, 50)
```

In [8]:

```
train_x.shape
```

Out[8]:

```
(10, 178650)
```

Looking at the size of this array, we see that we have rows equal to the batch size. When we want to get a batch out of here, we can grab a subset of this array that contains all the rows but has a width equal to the number of steps in the sequence. The first batch looks like this:

In [9]:

```
train_x[:,:50]
```

Out[9]:

```
array([[46, 19, 18, 44, 49, 58,  6, 75, 60, 61, 61, 61, 47, 18, 44, 44, 80,
        75, 23, 18, 50, 25, 57, 25, 58,  9, 75, 18,  6, 58, 75, 18, 57, 57,
        75, 18, 57, 25, 13, 58, 42, 75, 58, 62, 58,  6, 80, 75, 76, 65],
…,
       [75,  9, 18, 25, 77, 75, 49, 10, 75, 19, 58,  6,  9, 58, 57, 23, 41,
        75, 18, 65, 77, 75, 29, 58, 27, 18, 65, 75, 18, 27, 18, 25, 65, 75,
        23,  6, 10, 50, 75, 49, 19, 58, 75, 29, 58, 27, 25, 65, 65, 25]],
dtype=int32)
```

I'll write another function to grab batches out of the arrays made by `split_data`. Here each batch will be a sliding window on these arrays with size `batch_size X num_steps`. For example, if we want our network to train on a sequence of 100 characters, `num_steps = 100`. For the next batch, we'll shift this window the next sequence of `num_steps` characters. In this way we can feed batches to the network and the cell states will continue through on each batch.

In [10]:

```
def get_batch(arrs, num_steps):
    batch_size, slice_size = arrs[0].shape

    n_batches = int(slice_size/num_steps)
    for b in range(n_batches):
        yield [x[:, b*num_steps: (b+1)*num_steps] for x in arrs]
```

Building the model

Below is a function where I build the graph for the network.

In [11]:

```
def build_rnn(num_classes, batch_size=50, num_steps=50, lstm_size=128,
num_layers=2,
              learning_rate=0.001, grad_clip=5, sampling=False):
```

```
    # When we're using this network for sampling later, we'll be passing in
    # one character at a time, so providing an option for that
    if sampling == True:
        batch_size, num_steps = 1, 1

    tf.reset_default_graph()

    # Declare placeholders we'll feed into the graph
    inputs = tf.placeholder(tf.int32, [batch_size, num_steps], name='inputs')
    targets = tf.placeholder(tf.int32, [batch_size, num_steps], name='targets')

    # Keep probability placeholder for drop out layers
    keep_prob = tf.placeholder(tf.float32, name='keep_prob')

    # One-hot encoding the input and target characters
    x_one_hot = tf.one_hot(inputs, num_classes)
    y_one_hot = tf.one_hot(targets, num_classes)

    ### Build the RNN layers
    # Use a basic LSTM cell
    lstm = tf.contrib.rnn.BasicLSTMCell(lstm_size)

    # Add dropout to the cell
    drop = tf.contrib.rnn.DropoutWrapper(lstm, output_keep_prob=keep_prob)

    # Stack up multiple LSTM layers, for deep learning
    cell = tf.contrib.rnn.MultiRNNCell([drop] * num_layers)
    initial_state = cell.zero_state(batch_size, tf.float32)

    ### Run the data through the RNN layers
    # Run each sequence step through the RNN and collect the outputs
    outputs, state = tf.nn.dynamic_rnn(cell, x_one_hot,
initial_state=initial_state)
    final_state = state

    # Reshape output so it's a bunch of rows, one output row for each step for
each batch
    seq_output = tf.concat(outputs, axis=1)
    output = tf.reshape(seq_output, [-1, lstm_size])

    # Now connect the RNN outputs to a softmax layer
    with tf.variable_scope('softmax'):
        softmax_w = tf.Variable(tf.truncated_normal((lstm_size, num_classes),
stddev=0.1))
        softmax_b = tf.Variable(tf.zeros(num_classes))

    # Since output is a bunch of rows of RNN cell outputs, logits will be a
    # bunch of rows of logit outputs, one for each step and batch
    logits = tf.matmul(output, softmax_w) + softmax_b

    # Use softmax to get the probabilities for predicted characters
    preds = tf.nn.softmax(logits, name='predictions')
```

```
    # Reshape the targets to match the logits
    y_reshaped = tf.reshape(y_one_hot, [-1, num_classes])
    loss = tf.nn.softmax_cross_entropy_with_logits(logits=logits,
labels=y_reshaped)
    cost = tf.reduce_mean(loss)

    # Optimizer for training, using gradient clipping to control exploding
gradients
    tvars = tf.trainable_variables()
    grads, _ = tf.clip_by_global_norm(tf.gradients(cost, tvars), grad_clip)
    train_op = tf.train.AdamOptimizer(learning_rate)
    optimizer = train_op.apply_gradients(zip(grads, tvars))

    # Export the nodes
    # NOTE: I'm using a namedtuple here because I think they are cool
    export_nodes = ['inputs', 'targets', 'initial_state', 'final_state',
                    'keep_prob', 'cost', 'preds', 'optimizer']
    Graph = namedtuple('Graph', export_nodes)
    local_dict = locals()
    graph = Graph(*[local_dict[each] for each in export_nodes])

    return graph
```

Hyperparameters

Here I'm defining the hyperparameters for the network.

- `batch_size` - Number of sequences running through the network in one pass.
- `num_steps` - 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.
- `lstm_size` - The number of units in the hidden layers.
- `num_layers` - Number of hidden LSTM layers to use
- `learning_rate` - Learning rate for training
- `keep_prob` - The dropout keep probability when training. If you're network is overfitting, try decreasing this.

Here's some good advice from Andrej Karpathy on training the network. I'm going to write it in here for your benefit, but also link to where it originally came from.

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 `lstm_size` and `num_layers`. I would advise that you always use `num_layers` of either 2/3. The `lstm_size` 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 `lstm_size` 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.

In [12]:

```
batch_size = 10
num_steps = 100
lstm_size = 512
num_layers = 2
learning_rate = 0.001
keep_prob = 0.5
```

Training

Time for training which is pretty straightforward. Here I pass in some data, and get an LSTM state back. Then I pass that state back in to the network so the next batch can continue the state from the previous batch. And every so often (set by `save_every_n`) I calculate the validation loss and save a checkpoint.

Here I'm saving checkpoints with the format
`i{iteration number}_l{# hidden layer units}_v{validation loss}.ckpt`

In [13]:

```
epochs = 20
# Save every N iterations
save_every_n = 200
train_x, train_y, val_x, val_y = split_data(chars, batch_size, num_steps)

model = build_rnn(len(vocab),
                  batch_size=batch_size,
                  num_steps=num_steps,
                  learning_rate=learning_rate,
                  lstm_size=lstm_size,
                  num_layers=num_layers)
saver = tf.train.Saver(max_to_keep=100)
```

```
with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())

    # Use the line below to load a checkpoint and resume training
    #saver.restore(sess, 'checkpoints/______.ckpt')

    n_batches = int(train_x.shape[1]/num_steps)
    iterations = n_batches * epochs
    for e in range(epochs):

        # Train network
        new_state = sess.run(model.initial_state)
        loss = 0
        for b, (x, y) in enumerate(get_batch([train_x,train_y], num_steps), 1):
            iteration = e*n_batches + b
            start = time.time()
            feed = {model.inputs: x,
                    model.targets: y,
                    model.keep_prob: keep_prob,
                    model.initial_state: new_state}
            batch_loss, new_state, _ = sess.run([model.cost, model.final_state,
model.optimizer], feed_dict=feed)
            loss += batch_loss
            end = time.time()
            print('Epoch {}/{} '.format(e+1, epochs),
                  'Iteration {}/{}'.format(iteration, iterations),
                  'Training loss: {:.4f}'.format(loss/b),
                  '{:.4f} sec/batch'.format((end-start)))


            if (iteration%save_every_n == 0) or (iteration == iterations):
                # Check performance, notice dropout has been set to 1
                val_loss = []
                new_state = sess.run(model.initial_state)
                for x, y in get_batch([val_x, val_y], num_steps):
                    feed = {model.inputs: x,
                            model.targets: y,
                            model.keep_prob: 1.,
                            model.initial_state: new_state}
                    batch_loss, new_state = sess.run([model.cost,
model.final_state], feed_dict=feed)
                    val_loss.append(batch_loss)

                print('Validation loss:', np.mean(val_loss),
                      'Saving checkpoint!')
                saver.save(sess,
"checkpoints/i{}_l{}_v{:.3f}.ckpt".format(iteration, lstm_size,
np.mean(val_loss)))
```

```
Epoch 1/20  Iteration 1/35720 Training loss: 4.4195 2.4569 sec/batch
Epoch 1/20  Iteration 2/35720 Training loss: 4.3782 1.9192 sec/batch

.........
```

Saved checkpoints

Read up on saving and loading checkpoints here: https://www.tensorflow.org/programmers_guide/variables

In [ ]:

```
tf.train.get_checkpoint_state('checkpoints')
```

Sampling

Now that the network is trained, we'll can use it to generate new text. The idea is that we pass in a character, then the network will predict the next character. We can use the new one, to predict the next one. And we keep doing this to generate all new text. I also included some functionality to prime the network with some text by passing in a string and building up a state from that.

The network gives us predictions for each character. To reduce noise and make things a little less random, I'm going to only choose a new character from the top N most likely characters.

In [17]:

```
def pick_top_n(preds, vocab_size, top_n=5):
    p = np.squeeze(preds)
    p[np.argsort(p)[:-top_n]] = 0
    p = p / np.sum(p)
    c = np.random.choice(vocab_size, 1, p=p)[0]
    return c
```

In [41]:

```
def sample(checkpoint, n_samples, lstm_size, vocab_size, prime="The "):
    samples = [c for c in prime]
    model = build_rnn(vocab_size, lstm_size=lstm_size, sampling=True)
    saver = tf.train.Saver()
    with tf.Session() as sess:
        saver.restore(sess, checkpoint)
        new_state = sess.run(model.initial_state)
        for c in prime:
            x = np.zeros((1, 1))
            x[0,0] = vocab_to_int[c]
            feed = {model.inputs: x,
                    model.keep_prob: 1.,
                    model.initial_state: new_state}
            preds, new_state = sess.run([model.preds, model.final_state],
                                        feed_dict=feed)

        c = pick_top_n(preds, len(vocab))
        samples.append(int_to_vocab[c])

        for i in range(n_samples):
            x[0,0] = c
            feed = {model.inputs: x,
                    model.keep_prob: 1.,
                    model.initial_state: new_state}
            preds, new_state = sess.run([model.preds, model.final_state],
                                        feed_dict=feed)

            c = pick_top_n(preds, len(vocab))
            samples.append(int_to_vocab[c])
    return ''.join(samples)
```

Here, pass in the path to a checkpoint and sample from the network.

In [ ]:

```
checkpoint = "checkpoints/____.ckpt"
samp = sample(checkpoint, 2000, lstm_size, len(vocab), prime="Far")
print(samp)
```