BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE

NLP LAB - 6

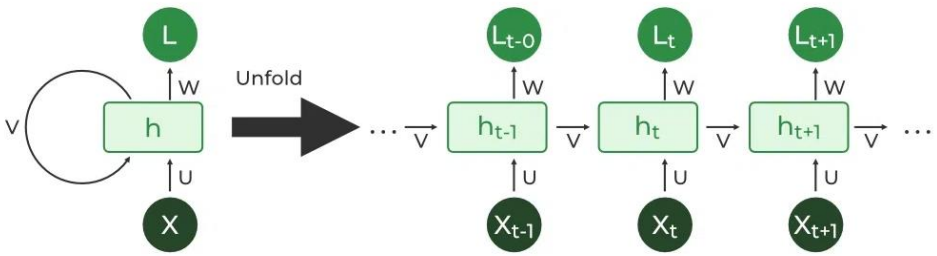
[RNN & SENTIMENT ANALYSIS USING RNN]

DATE: 20/10/2023 – 22/10/2023
TIME: 02 Hours

This lab-sheet demonstrates Recurrent Neural Networks (RNNs) and an example problem statement of sentiment analysis using RNN.

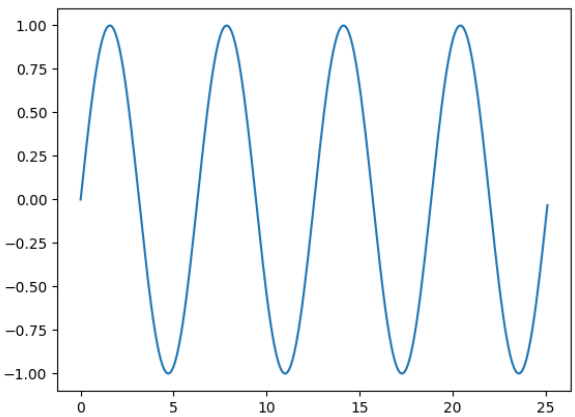
RNN

A typical RNN looks like this:



Sequence Prediction using RNN

1. Now, we will learn a sequence prediction problem using RNN. One of the simplest tasks for this is sine wave prediction. This is what a sine wave looks like:



2. We will first devise a recurrent neural network from scratch to solve this problem.
3. We will formulate our problem like this – given a sequence of 50 numbers belonging to a sine wave, predict the 51st number in the series.

Data Preparation

What does our network model expect the data to be like?
It would accept a single sequence of length 50 as input. So the shape of the input data will be:

`(number_of_records x length_of_sequence x types_of_sequences)`

Here, types_of_sequences is 1, because we have only one type of sequence – the sine wave. On the other hand, the output would have only one value for each record. This will of course be the 51st value in the input sequence. So, it's shape would be:
`(number_of_records x types_of_sequences)`

```
# Import libraries
import numpy as np
import matplotlib.pyplot as plt
import math
```

To create a sine wave like data, we will use the sine function from Python's *math* library:

```
sin_wave = np.array([math.sin(x) for x in np.arange(200)])

# Visualizing the sine wave we've just generated:
plt.plot(sin_wave[:50])  # plot the sine wave
```

We will create the data as shown in the code snippet below:

```
X = []
Y = []
seq_len = 50
num_records = len(sin_wave) - seq_len
for i in range(num_records - 50):
    X.append(sin_wave[i:i+seq_len])
    Y.append(sin_wave[i+seq_len])

X = np.array(X)
X = np.expand_dims(X, axis=2)
Y = np.array(Y)
Y = np.expand_dims(Y, axis=1)
```

```
# Print the shape of the data:
X.shape, Y.shape  # ((100, 50, 1), (100, 1))
```

```
# Note that we looped for (num_records – 50),
# because we want to set aside 50 records as our validation data.
# We can create this validation data now:
X_val = []
Y_val = []

for i in range(num_records - 50, num_records):
    X_val.append(sin_wave[i:i+seq_len])
    Y_val.append(sin_wave[i+seq_len])

X_val = np.array(X_val)
X_val = np.expand_dims(X_val, axis=2)
Y_val = np.array(Y_val)
Y_val = np.expand_dims(Y_val, axis=1)
```

Create the Architecture for our RNN model

Our next task is defining all the necessary variables and functions which we will use in the RNN model. Our model will take in the input sequence, process it through a hidden layer of 100 units, and produce a single valued output:

```
learning_rate = 0.0001
nepoch = 2
T = 50 # length of sequence
hidden_dim = 100
output_dim = 1
bptt_truncate = 5
min_clip_value = -10
max_clip_value = 10

# We will then define the weights of the network:
U = np.random.uniform(0, 1, (hidden_dim, T))
W = np.random.uniform(0, 1, (hidden_dim, hidden_dim))
V = np.random.uniform(0, 1, (output_dim, hidden_dim))
```

Here,

U is the weight matrix for weights between input and hidden layers

V is the weight matrix for weights between hidden and output layers

W is the weight matrix for shared weights in the RNN layer (hidden layer)

Finally, we will define the activation function, sigmoid, to be used in the hidden layer:

Train the Model

Now that we have defined our model, we can finally move on with training it on our sequence data. We can subdivide the training process into smaller steps, namely:

Step 2.1 : Check the loss on training data
- Step 2.1.1 : Forward Pass
- Step 2.1.2 : Calculate Error

Step 2.2 : Check the loss on validation data
- Step 2.2.1 : Forward Pass
- Step 2.2.2 : Calculate Error

Step 2.3 : Start actual training
- Step 2.3.1 : Forward Pass
- Step 2.3.2 : Backpropagate Error
- Step 2.3.3 : Update weights

We need to repeat these steps until convergence.

Step 2.1: Check the loss on training data

We will do a forward pass through our RNN model and calculate the squared error for the predictions for all records in order to get the loss value.

```
for epoch in range(nepoch):
    # check loss on train
    loss = 0.0

    # do a forward pass to get prediction
    for i in range(Y.shape[0]):
        x, y = X[i], Y[i]                    # get input, output values of
each record
        prev_s = np.zeros((hidden_dim, 1))   # here, prev-s is the value of
the previous activation of hidden layer; which is initialized as all zeroes
        for t in range(T):
            new_input = np.zeros(x.shape)    # we then do a forward pass for
every timestep in the sequence
            new_input[t] = x[t]              # for this, we define a single
input for that timestep
            mulu = np.dot(U, new_input)
            mulw = np.dot(W, prev_s)
            add = mulw + mulu
            s = sigmoid(add)
            mulv = np.dot(V, s)
            prev_s = s

        # calculate error
        loss_per_record = (y - mulv)**2 / 2
        loss += loss_per_record
    loss = loss / float(y.shape[0])
```

Step 2.2: Check the loss on validation data

We will do the same thing for calculating the loss on validation data (in the same loop):

```
    # check loss on val
    val_loss = 0.0
    for i in range(Y_val.shape[0]):
        x, y = X_val[i], Y_val[i]
        prev_s = np.zeros((hidden_dim, 1))
        for t in range(T):
            new_input = np.zeros(x.shape)
            new_input[t] = x[t]
            mulu = np.dot(U, new_input)
            mulw = np.dot(W, prev_s)
            add = mulw + mulu
            s = sigmoid(add)
            mulv = np.dot(V, s)
            prev_s = s
        loss_per_record = (y - mulv)**2 / 2
        val_loss += loss_per_record
    val_loss = val_loss / float(y.shape[0])

    print('Epoch: ', epoch + 1, ', Loss: ', loss, ', Val Loss: ', val_loss)
```

Step 2.3: Start actual training

We will now start with the actual training of the network. In this, we will first do a forward pass to calculate the errors and a backward pass to calculate the gradients and update them.

Step 2.3.1: Forward Pass

In the forward pass:

- We first multiply the input with the weights between input and hidden layers
- Add this with the multiplication of weights in the RNN layer. This is because we want to capture the knowledge of the previous timestep
- Pass it through a sigmoid activation function
- Multiply this with the weights between hidden and output layers
- At the output layer, we have a linear activation of the values so we do not explicitly pass the value through an activation layer
- Save the state at the current layer and also the state at the previous time step in a dictionary

Here is the code for doing a forward pass (note that it is in continuation of the above loop):

```
    # train model
    for i in range(Y.shape[0]):
        x, y = X[i], Y[i]

        layers = []
        prev_s = np.zeros((hidden_dim, 1))
        dU = np.zeros(U.shape)
        dV = np.zeros(V.shape)
        dW = np.zeros(W.shape)

        dU_t = np.zeros(U.shape)
        dV_t = np.zeros(V.shape)
        dW_t = np.zeros(W.shape)

        dU_i = np.zeros(U.shape)
        dW_i = np.zeros(W.shape)

        # forward pass
        for t in range(T):
            new_input = np.zeros(x.shape)
            new_input[t] = x[t]
            mulu = np.dot(U, new_input)
            mulw = np.dot(W, prev_s)
            add = mulw + mulu
            s = sigmoid(add)
            mulv = np.dot(V, s)
            layers.append({'s':s, 'prev_s':prev_s})
            prev_s = s
```

Step 2.3.2 : Backpropagate Error

After the forward propagation step, we calculate the gradients at each layer, and backpropagate the errors. We will use truncated back propagation through time (TBPTT), instead of vanilla backprop. It may sound complex but its actually pretty straight forward.

The core difference in BPTT versus backprop is that the backpropagation step is done for all the time steps in the RNN layer. So if our sequence length is 50, we will backpropagate for all the timesteps previous to the current timestep.

If you have guessed correctly, BPTT seems very computationally expensive. So instead of backpropagating through all previous timestep, we backpropagate till x timesteps to save computations. Consider this ideologically similar to stochastic gradient descent, where we include a batch of data points instead of all the data points.

Here is the code for backpropagating the errors:

```
        # derivative of pred
        dmulv = (mulv - y)

        # backward pass
        for t in range(T):
            dV_t = np.dot(dmulv, np.transpose(layers[t]['s']))
            dsv = np.dot(np.transpose(V), dmulv)

            ds = dsv
            dadd = add * (1 - add) * ds

            dmulw = dadd * np.ones_like(mulw)

            dprev_s = np.dot(np.transpose(W), dmulw)


            for i in range(t-1, max(-1, t-bptt_truncate-1), -1):
                ds = dsv + dprev_s
                dadd = add * (1 - add) * ds

                dmulw = dadd * np.ones_like(mulw)
                dmulu = dadd * np.ones_like(mulu)

                dW_i = np.dot(W, layers[t]['prev_s'])
                dprev_s = np.dot(np.transpose(W), dmulw)

                new_input = np.zeros(x.shape)
                new_input[t] = x[t]
                dU_i = np.dot(U, new_input)
                dx = np.dot(np.transpose(U), dmulu)

                dU_t += dU_i
                dW_t += dW_i

            dV += dV_t
            dU += dU_t
            dW += dW_t
```

Step 2.3.3 : Update weights

Lastly, we update the weights with the gradients of weights calculated. One thing we have to keep in mind that the gradients tend to explode if you don't keep them in check. This is a fundamental issue in training neural networks, called the exploding gradient problem. So we have to clamp them in a range so that they dont explode. We can do it like this

```
            if dU.max() > max_clip_value:
                dU[dU > max_clip_value] = max_clip_value
            if dV.max() > max_clip_value:
                dV[dV > max_clip_value] = max_clip_value
            if dW.max() > max_clip_value:
                dW[dW > max_clip_value] = max_clip_value


            if dU.min() < min_clip_value:
                dU[dU < min_clip_value] = min_clip_value
            if dV.min() < min_clip_value:
                dV[dV < min_clip_value] = min_clip_value
            if dW.min() < min_clip_value:
                dW[dW < min_clip_value] = min_clip_value

        # update
        U -= learning_rate * dU
        V -= learning_rate * dV
        W -= learning_rate * dW
```

Step 3: Get predictions

We will do a forward pass through the trained weights to get our predictions:

```
preds = []
for i in range(Y.shape[0]):
    x, y = X[i], Y[i]
    prev_s = np.zeros((hidden_dim, 1))
    # Forward pass
    for t in range(T):
        mulu = np.dot(U, x)
        mulw = np.dot(W, prev_s)
        add = mulw + mulu
        s = sigmoid(add)
        mulv = np.dot(V, s)
        prev_s = s

    preds.append(mulv)

preds = np.array(preds)
```

Plotting these predictions alongside the actual values:

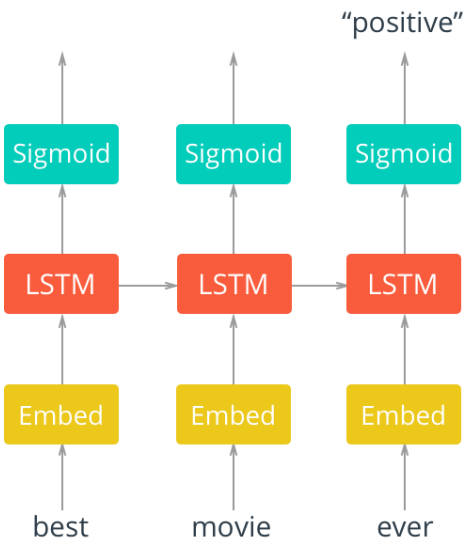
```
plt.plot(preds[:, 0, 0], 'g')
plt.plot(Y[:, 0], 'r')
plt.show()
```

This was on the training data. How do we know if our model didn't overfit? This is where the validation set, which we created earlier, comes into play:

```
preds = []
for i in range(Y_val.shape[0]):
    x, y = X_val[i], Y_val[i]
    prev_s = np.zeros((hidden_dim, 1))
    # For each time step...
    for t in range(T):
        mulu = np.dot(U, x)
        mulw = np.dot(W, prev_s)
        add = mulw + mulu
        s = sigmoid(add)
        mulv = np.dot(V, s)
        prev_s = s

    preds.append(mulv)

preds = np.array(preds)

plt.plot(preds[:, 0, 0], 'g')
plt.plot(Y_val[:, 0], 'r')
plt.show()
```

Sentiment Analysis using RNNs

In this notebook, we will implement a RNN for the example problem statement of sentiment analysis. Using an RNN rather than a feedfoward network (FFNN) gives us higher accuracy, as the information about the *sequence* of words is captured. Here we'll use a dataset of movie reviews, accompanied by labels.
Model architecture is shown below:



Here, we'll pass in words to an embedding layer. We need an embedding layer because we have tens of thousands of words, so we'll need a more efficient representation for our input data than one-hot encoded vectors. You should have seen this before from the word2vec labsheet. You can actually train up an embedding with word2vec and use it here. But it's good enough to just have an embedding layer and let the network learn the embedding table on it's own.
From the embedding layer, the new representations will be passed to LSTM cells. These will add recurrent connections to the network so we can include information about the sequence of words in the data. Finally, the LSTM cells will go to a sigmoid output layer here. We're using the sigmoid because we're trying to predict if this text has positive or negative sentiment. The output layer will just be a single unit then, with a sigmoid activation function.
We don't care about the sigmoid outputs except for the very last one, we can ignore the rest. We'll calculate the cost from the output of the last step and the training label.

```
import numpy as np
import tensorflow as tf
```

```
!unzip data.zip
!rm data.zip
```

```
with open('reviews.txt', 'r') as f:
    reviews = f.read()
with open('labels.txt', 'r') as f:
    labels = f.read()
```

```
reviews[:2000]
```

```
bromwell high is a cartoon comedy . it ran at the same time as some
other programs about school life such as teachers . my years in the
teaching profession lead me to believe that bromwell high s satire is
much closer to reality than is teachers . the scramble to survive
financially the insightful students who can see right through their
pathetic teachers pomp the pettiness of the whole situation all remind
me of the schools i knew and their students . when i saw the episode in
which a student repeatedly tried to burn down the school i immediately
recalled . . . . . at . . . . . high . a classic line
inspector i m here to sack one of your teachers . student welcome to
bromwell high . i expect that many adults of my age think that bromwell
high is far fetched . what a pity that it isn t \nstory of a man who
has unnatural feelings for a pig . starts out with a opening scene that
is a terrific example of absurd comedy . a formal orchestra audience is
tur...
```

Data Preprocessing

The first step when building a neural network model is getting your data into the proper form to feed into the network. Since we're using embedding layers, we'll need to encode each word with an integer. We'll also want to clean it up a bit.

You can see an example of the reviews data above. We'll want to get rid of those periods. Also, you might notice that the reviews are delimited with newlines `\n`. To deal with those, we are going to split the text into each review using `\n` as the delimiter. Then we can combined all the reviews back together into one big string.
First, let's remove all punctuation. Then get all the text without the newlines and split it into individual words.

```
from string import punctuation
all_text = ''.join([c for c in reviews if c not in punctuation])
reviews = all_text.split('\n')

all_text = ' '.join(reviews)
words = all_text.split()
```

```
all_text[:2000]
```

```
'bromwell high is a cartoon comedy it ran at the same time as some other programs about school life such as teachers my years in the teaching profession lead me to believe that bromwell high s satire is much closer to reality than is teachers the scramble to survive financially the insightful students who can see right through their pathetic teachers pomp the pettiness of the whole situation all remind me of the schools i knew and their students when i saw the episode in which a student repeatedly tried to burn down the school i immediately recalled at high a classic line inspector i m here to sack one of your teachers student welcome to bromwell high i expect that many adults of my age think that bromwell high is far fetched what a pity that it isn t story of a man who has unnatural feelings for a pig starts out with a opening scene that is a terrific example of absurd comedy a formal orchestra audience is turned into an insane violent mo...
```

```
words[:100]
```

```
['bromwell', 'high', 'is', 'a', 'cartoon', 'comedy', 'it', 'ran', 'at',
'the', 'same', 'time', 'as', 'some', 'other', 'programs', 'about',
'school', 'life', 'such', 'as', 'teachers', 'my', 'years', 'in', 'the',
'teaching', 'profession', 'lead', 'me', 'to', 'believe', 'that',
'bromwell', 'high', 's', 'satire', 'is', 'much', 'closer', 'to',
'reality', 'than', 'is', 'teachers', 'the', 'scramble', 'to',
'survive', 'financially', 'the', 'insightful', 'students', 'who',
'can', 'see', 'right', 'through', 'their', 'pathetic', 'teachers',
'pomp', 'the', 'pettiness', 'of', 'the', 'whole', 'situation', 'all',
'remind', 'me', 'of', 'the', 'schools', 'i', 'knew', 'and', 'their',
'students', 'when', 'i', 'saw', 'the', 'episode', 'in', 'which', 'a',
'student', 'repeatedly', 'tried', 'to', 'burn', 'down', 'the',
'school', 'i', 'immediately', 'recalled', 'at', 'high']
```

```
print(type(reviews), len(reviews), reviews[0],'last review:',
reviews[25000])
```

```
<class 'list'> 25001 bromwell ... last review:
```

Encoding the words

The embedding lookup requires that we pass in integers to our network. The easiest way to do this is to create dictionaries that map the words in the vocabulary to integers. Then we can convert each of our reviews into integers so they can be passed into the network.

TODO: Now you're going to encode the words with integers. Build a dictionary that maps words to integers. Later we're going to pad our input vectors with zeros, so make sure the integers **start at 1, not 0**. Also, convert the reviews to integers and store the reviews in a new list called reviews_ints.

```
from collections import Counter
counts = ...
vocab = ...
vocab_to_int = {...} # HINT: Use list comprehension

reviews_ints = []
for each in reviews:
    reviews_ints.append([...]) # HINT: Use list comprehension
```

Encoding the labels

Our labels are "positive" or "negative". To use these labels in our network, we need to convert them to 0 and 1.

TODO: Convert labels from positive and negative to 1 and 0, respectively.

```
labels = labels.split('\n')
labels = np.array(...)
```

```
review_lens = Counter([len(x) for x in reviews_ints])
print("Zero-length reviews: {}".format(review_lens[0]))
print("Maximum review length: {}".format(max(review_lens)))
```

```
Zero-length reviews: 1
Maximum review length: 2514
```

Okay, a couple issues here.
We seem to have one review with zero length. And, the maximum review length is way too many steps for our RNN. Let's truncate to 200 steps.

- For reviews shorter than 200, we'll pad with 0s.
- For reviews longer than 200, we can truncate them to the first 200 characters.

TODO: First, remove the review with zero length from the reviews_ints list.

```
# Filter out that review with 0 length
reviews_ints = [...] # TODO: All reviews of positive length
```

TODO: Now, create an array `features` that contains the data we'll pass to the network. The data should come from `review_ints`, since we want to feed integers to the network. Each row should be 200 elements long. For reviews shorter than 200 words, left pad with 0s. That is, if the review is `['best', 'movie', 'ever']`, `[117, 18, 128]` as `integers`, the row will look like `[0, 0, 0, ..., 0, 117, 18, 128]`. For reviews longer than 200, use on the first 200 words as the feature vector.

This isn't trivial and there are a bunch of ways to do this. But, if you're going to be building your own deep learning networks, you're going to have to get used to preparing your data.

```
seq_len = 200
features = ...
```

```
features[:10,:100]
```

```
array([

[ 0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0, 21025,  308,  6,  3, 1050,  207,  8,
 2138,  32,  1,  171,  57,  15,  49,  81, 5785,  44,  382,  110,  140,  15, 5194,
   60,  154,  9,  1, 4975, 5852,  475,  71,  5,  260,  12, 21025,  308,  13, 1978,
    6,  74, 2395],

[ 0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  63,  4,  3,  125,  36,  47,
 7472, 1395,  16,  3, 4181,  505,  45,  17],

[22382,  42, 46418,  15,  706, 17139, 3389,  47,  77,  35, 1819,  16,  154,  19,
  114,  3, 1305,  5,  336,  147,  22,  1,  857,  12,  70,  281, 1168,  399,  36,  120,
  283,  38,  169,  5,  382,  158,  42, 2269,  16,  1,  541,  90,  78,  102,  4,  1,
 3244,  15,  43,  3,  407, 1068,  136, 8055,  44,  182,  140,  15, 3043,  1,  320,
   22, 4818, 26224,  346,  5, 3090, 2092,  1, 18839, 17939,  42, 8055,  46,  33,
  236,  29,  370,  5,  130,  56,  22,  1, 1928,  7,  7,  19,  48,  46,  21,  70,  344,
    3, 2099,  5,  408,  22,  1, 1928,  16],

...

[ 54,  10,  14,  116,  60,  798,  552,  71,  364,  5,  1,  730,  5,  66, 8057,  8,
  14,  30,  4,  109,  99,  10,  293,  17,  60,  798,  19,  11,  14,  1,  64,  30,  69,
 2500,  45,  4,  234,  93,  10,  68,  114,  108, 8057,  363,  43, 1009,  2,  10,  97,
  28, 1431,  45,  1,  357,  4,  60,  110,  205,  8,  48,  3, 1929, 10880,  2, 2124,
  354,  412,  4,  13, 6609,  2, 2974, 5148, 2125, 1366,  6,  30,  4,  60,  502,
  876,  19, 8057,  6,  34,  227,  1,  247,  412,  4,  582,  4,  27,  599,  9,  1,
13586,  396,  4, 14047]

])
```

Training, Validation, Test

With our data in nice shape, we'll split it into training, validation, and test sets.

TODO: Create the training, validation, and test sets here. You'll need to create sets for the features and the `labels`, `train_x` and `train_y` for example. Define a split fraction, `split_frac` as the fraction of data to keep in the training set. Usually this is set to `0.8` or `0.9`. The rest of the data will be split in half to create the validation and testing data.

```
split_frac = 0.8
train_x, val_x = ...
train_y, val_y = ...

val_x, test_x = ...
val_y, test_y = ...

print("\t\t\tFeature Shapes:")
print("Train set: \t\t{}".format(train_x.shape),
      "\nValidation set: \t{}".format(val_x.shape),
      "\nTest set: \t\t{}".format(test_x.shape))
```

```
                Feature Shapes:
Train set:      (20000, 200)
Validation set: (2500, 200)
Test set:       (2501, 200)
```

Build the Graph

Here, we'll build the graph. First up, defining the hyperparameters.

- `lstm_size`: Number of units in the hidden layers in the LSTM cells. Usually larger is better performance wise. Common values are 128, 256, 512, etc.
- `lstm_layers`: Number of LSTM layers in the network. I'd start with 1, then add more if I'm underfitting.
- `batch_size`: The number of reviews to feed the network in one training pass. Typically this should be set as high as you can go without running out of memory.
- `learning_rate`: Learning rate

```
lstm_size = 256
lstm_layers = 1
batch_size = 500
learning_rate = 0.001
```

For the network itself, we'll be passing in our 200 element long review vectors. Each batch will be `batch_size` vectors. We'll also be using dropout on the LSTM layer, so we'll make a placeholder for the keep probability.

TODO: Create the `inputs_`, `labels_`, and drop out `keep_prob` placeholders. `labels_` needs to be two-dimensional to work with some functions later.

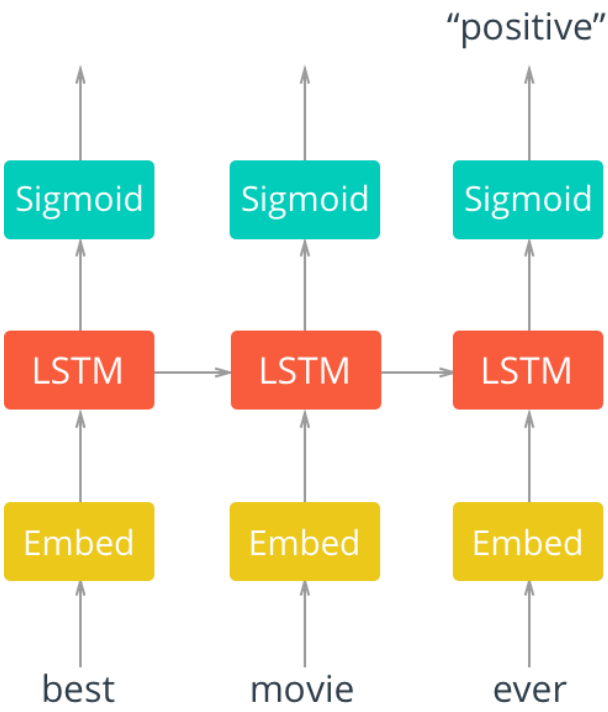
Embedding

Now we'll add an embedding layer. We need to do this because there are 74000 words in our vocabulary. It is massively inefficient to one-hot encode our classes here. You should remember dealing with this problem from the word2vec lesson. Instead of one-hot encoding, we can have an embedding layer and use that layer as a lookup table. You could train an embedding layer using word2vec, then load it here. But, it's fine to just make a new layer and let the network learn the weights.

TODO: Create the embedding lookup matrix as a `tf.Variable`. Use that embedding matrix to get the embedded vectors to pass to the LSTM cell with `tf.nn.embedding_lookup`. This function takes the embedding matrix and an input tensor, such as the review vectors. Then, it'll return another tensor with the embedded vectors. So, if the embedding layer as 200 units, the function will return a tensor with size [batch_size, 200].

```
# Size of the embedding vectors (number of units in the embedding layer)
embed_size = 300

with graph.as_default():
    embedding = ...
    embed = ...
```

LSTM Cell



Next, we'll create our LSTM cells to use in the recurrent network. Here we are just defining what the cells look like. This isn't actually building the graph, just defining the type of cells we want in our graph.
The final cell you're using in the network is actually multiple (or just one) LSTM cells with dropout. But it all works the same from an achitectural viewpoint, just a more complicated graph in the cell.

```
# In the graph
    # Your basic LSTM cell
    lstm = # TODO: BasicLSTMCell(lstm_size)

    # Add dropout to the cell
    drop = # TODO: DropoutWrapper(lstm, output_keep_prob=keep_prob)

    # Stack up multiple LSTM layers, for deep learning
    cell = # TODO: MultiRNNCell([drop] * lstm_layers)

    # Getting an initial state of all zeros
    initial_state = cell.zero_state(batch_size, tf.float32)
```

Now we need to actually run the data through the RNN nodes. You'd pass in the RNN cell you created (our multiple layered LSTM `cell` for instance), and the inputs to the network.

TODO: Add forward pass through the RNN. Remember that we're actually passing in vectors from the embedding layer, `embed`

Output

We only care about the final output, we'll be using that as our sentiment prediction. So we need to grab the last output with `outputs[:, -1]`, the calculate the cost from that and `labels_`.

```
    predictions = ... # Sigmoid Activation
    cost = ... # Mean squared error loss
    optimizer = ... # Adam
```

Validation Accuracy

Here add nodes to calculate the accuracy which we'll use in the validation pass.

Batching

This is a simple function for returning batches from our data. First it removes data such that we only have full batches. Then it iterates through the `x` and `y` arrays and returns slices out of those arrays with size `[batch_size]`.

```
def get_batches(x, y, batch_size=100):

    n_batches = len(x)//batch_size
    x, y = x[:n_batches*batch_size], y[:n_batches*batch_size]
    for ii in range(0, len(x), batch_size):
        yield x[ii:ii+batch_size], y[ii:ii+batch_size]
```

Training

Below is the typical training code. If you want to do this yourself, feel free to delete all this code and implement it yourself. Before you run this, make sure the `checkpoints` directory exists.

```
epochs = 10

with graph.as_default():
    saver = tf.compat.v1.train.Saver()

with tf.compat.v1.Session(graph=graph) as sess:
    sess.run(tf.compat.v1.global_variables_initializer())
    iteration = 1
    for e in range(epochs):
        state = sess.run(initial_state)

        for ii, (x, y) in enumerate(get_batches(train_x, train_y,
batch_size), 1):
            feed = {inputs_: x,
                    labels_: y[:, None],
                    keep_prob: 0.5,
                    initial_state: state}
            loss, state, _ = sess.run([cost, final_state, optimizer],
feed_dict=feed)

            if iteration%5==0:
                print("Epoch: {}/{}".format(e, epochs),
                      "Iteration: {}".format(iteration),
                      "Train loss: {:.3f}".format(loss))

            if iteration%25==0:
                val_acc = []
                val_state = sess.run(cell.zero_state(batch_size,
tf.float32))
                for x, y in get_batches(val_x, val_y, batch_size):
                    feed = {inputs_: x,
                            labels_: y[:, None],
                            keep_prob: 1,
                            initial_state: val_state}
                    batch_acc, val_state = sess.run([accuracy,
final_state], feed_dict=feed)
                    val_acc.append(batch_acc)
                print("Val acc: {:.3f}".format(np.mean(val_acc)))
            iteration +=1
    saver.save(sess, "checkpoints/sentiment.ckpt")
```

Test Accuracy

Feel free to delete the code and implement it yourself.

```
test_acc = []
with tf.compat.v1.Session(graph=graph) as sess:
    saver.restore(sess,
tf.train.latest_checkpoint('/output/checkpoints'))
    test_state = sess.run(cell.zero_state(batch_size, tf.float32))
    for ii, (x, y) in enumerate(get_batches(test_x, test_y,
batch_size), 1):
        feed = {inputs_: x,
                labels_: y[:, None],
                keep_prob: 1,
                initial_state: test_state}
        batch_acc, test_state = sess.run([accuracy, final_state],
feed_dict=feed)
        test_acc.append(batch_acc)
    print("Test accuracy: {:.3f}".format(np.mean(test_acc)))
```