NLP Fundamentals

# Lesson 1: Introduction to NLP

## NLP and Pipelines

<https://www.youtube.com/watch?time_continue=37&v=UQBxJzoCp-I&feature=emb_logo>

## How NLP Pipelines work

<https://www.youtube.com/watch?time_continue=1&v=vJx6oKlu_MM&feature=emb_logo>

## Text Preprocessing

<https://www.youtube.com/watch?v=pqheVyctkNQ&feature=emb_logo>

## Feature Extraction

<https://www.youtube.com/watch?v=Bd6TJB8eVLQ&feature=emb_logo>

## Bag of Words

<https://www.youtube.com/watch?v=A7M1z8yLl0w&feature=emb_logo>

## TF-IDF

<https://www.youtube.com/watch?v=XZBiBIRcACE&feature=emb_logo>

## One-Hot Encoding

<https://www.youtube.com/watch?v=a0j1CDXFYZI&feature=emb_logo>

## Word Embeddings

<https://www.youtube.com/watch?v=4mM_S9L2_JQ&feature=emb_logo>

## Word2Vec

<https://www.youtube.com/watch?v=7jjappzGRe0&feature=emb_logo>

## GloVe

<https://www.youtube.com/watch?time_continue=1&v=KK3PMIiIn8o&feature=emb_logo>

## Embeddings for Deep Learning

<https://www.youtube.com/watch?time_continue=2&v=gj8u1KG0H2w&feature=emb_logo>

## Modeling

<https://www.youtube.com/watch?time_continue=1&v=P4w_2rkxBvE&feature=emb_logo>

# Lesson 2: Implementation of LSTM and RNN

## Implementing RNNs

<https://www.youtube.com/watch?time_continue=2&v=BHoiwB61ays&feature=emb_logo>

## Time-Series Prediction

<https://www.youtube.com/watch?time_continue=1&v=xV5jHLFfJbQ&feature=emb_logo>

### Code Walkthrough & Repository

The below video is a walkthrough of code that you can find in our public Github repository, if you navigate to recurrent-neural-networks > time-series and [**the Simple\_RNN.ipynb notebook**](https://github.com/udacity/deep-learning-v2-pytorch/blob/master/recurrent-neural-networks/time-series/Simple_RNN.ipynb). Feel free to go through this code on your own, locally.

This example is meant to give you an idea of how PyTorch represents RNNs and how you might represent memory in code. Later, you'll be given more complex exercise and solution notebooks, in-classroom.

## Training & Memory

<https://www.youtube.com/watch?time_continue=1&v=sx7T_KP5v9I&feature=emb_logo>

### Recurrent Layers

Here is the documentation for the main types of [**recurrent layers in PyTorch**](https://pytorch.org/docs/stable/nn.html#recurrent-layers). Take a look and read about the three main types: RNN, LSTM, and GRU.

## Character-Wise RNNs

<https://www.youtube.com/watch?time_continue=3&v=dXl3eWCGLdU&feature=emb_logo>

## Sequence Batching

<https://www.youtube.com/watch?time_continue=1&v=Z4OiyU0Cldg&feature=emb_logo>

## Notebook: Character-Level RNN

Now you have all the information you need to implement an RNN of our own. The next few videos will be all about character-level text prediction with an LSTM!

**It's suggested that you open the notebook in a new, working tab and continue working on it as you go through the instructional videos in this tab.** This way you can toggle between learning new skills and coding/applying new skills.

To open this notebook, you have two options:

* Go to the next page in the classroom (recommended).
* Clone the repo from [**Github**](https://github.com/udacity/deep-learning-v2-pytorch) and open the notebook **Character\_Level\_RNN\_Exercise.ipynb** in the **recurrent-neural-networks > char-rnn** folder. You can either download the repository with git clone https://github.com/udacity/deep-learning-v2-pytorch.git, or download it as an archive file from [**this link**](https://github.com/udacity/deep-learning-v2-pytorch/archive/master.zip).

### Instructions

* Load in text data
* Pre-process that data, encoding characters as integers and creating one-hot input vectors
* Define an RNN that predicts the next character when given an input sequence
* Train the RNN and use it to generate new text

This is a self-assessed lab. If you need any help or want to check your answers, feel free to check out the solutions notebook in the same folder, or by clicking [**here**](https://github.com/udacity/deep-learning-v2-pytorch/blob/master/recurrent-neural-networks/char-rnn/Character_Level_RNN_Solution.ipynb).

### GPU Workspaces

The next workspace is **GPU-enabled**, which means you can select to train on a GPU instance. The recommendation is this:

* Load in data, test functions and models (checking parameters and doing a short training loop) while in CPU (non-enabled) mode
* When you're ready to extensively train and test your model, **enable** GPU to quickly train the model!

All models and data they see as input will have to be moved to the GPU device, so take note of the relevant movement code in the model creation and training process.

## Implementing a Char-RNN

<https://www.youtube.com/watch?time_continue=4&v=MMtgZXzFB10&feature=emb_logo>

## Batching Data, Solution

<https://www.youtube.com/watch?time_continue=22&v=9Eg0wf3eW-k&feature=emb_logo>

## Defining the Model

<https://www.youtube.com/watch?time_continue=22&v=_LWzyqq4hCY&feature=emb_logo>

## Char-RNN, Solution

<https://www.youtube.com/watch?v=ed33qePHrJM&feature=emb_logo>

### Representing Memory

You’ve learned that RNN’s work well for sequences of data because they have a kind of memory. This memory is represented by something called the **hidden state**.

In the character-level LSTM example, each LSTM cell, in addition to accepting a character as input and generating an output character, also has some hidden state, and each cell will pass along its hidden state to the next cell.

This connection creates a kind of memory by which a series of cells can remember which characters they’ve just seen and use that information to inform the next prediction!

For example, if a cell has just generated the character a it likely will not generate another a, right after that!

#### net.eval()

There is an omission in the above code: including net.eval() !

net.eval() will set all the layers in your model to evaluation mode. This affects layers like dropout layers that turn "off" nodes during training with some probability, but should allow every node to be "on" for evaluation. So, you should set your model to evaluation mode **before testing or validating your model**, and before, for example, sampling and making predictions about the likely next character in a given sequence. I'll set net.train()` (training mode) only during the training loop.

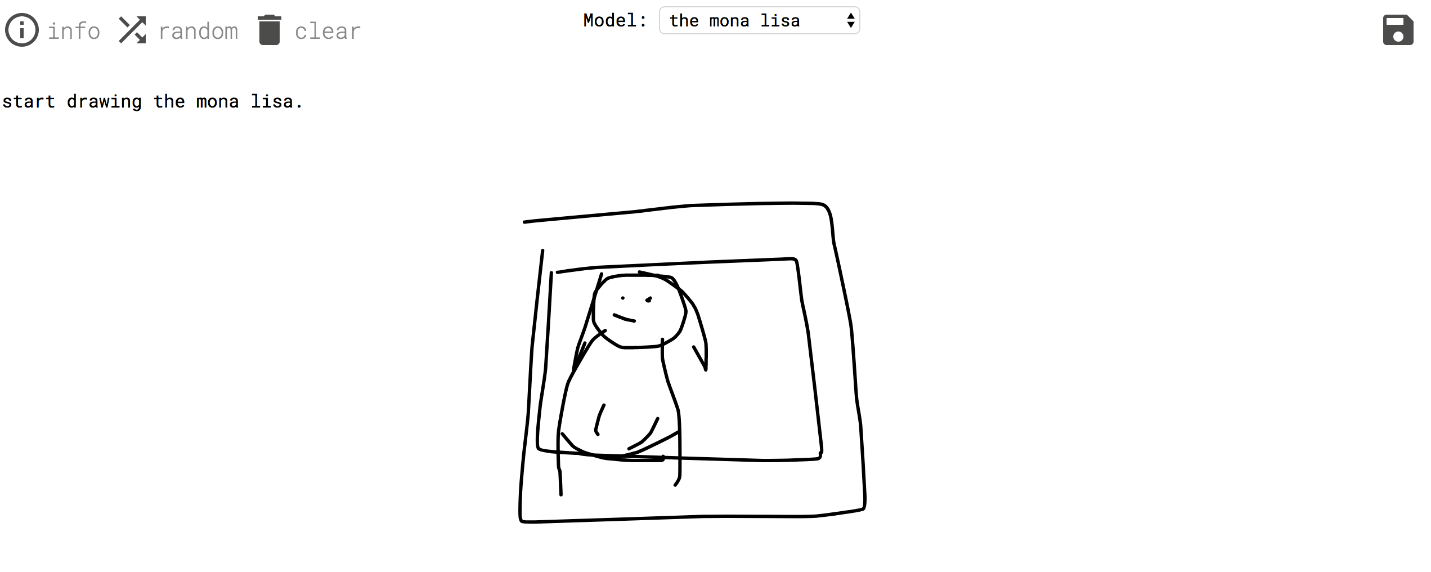
This is reflected in the previous notebook code and in our **[Github repository](https://github.com/udacity/deep-learning-v2-pytorch/blob/master/recurrent-neural-networks/char-rnn" \t "_blank)**.

## Making Predictions

<https://www.youtube.com/watch?time_continue=1&v=BhrpV3kwATo&feature=emb_logo>

### Examples of RNNs

Take a look at one of my favorite examples of RNNs making predictions based on some user-generated input dat: the [**sketch-rnn by Magenta**](https://magenta.tensorflow.org/assets/sketch_rnn_demo/index.html). This RNN takes as input a starting sketch, drawn by you, and then tries to comp ete your sketch using a particular model. For example, it can learn to complete a sketch of a pineapple or the mona lisa!



Example sketch-rnn output of the mona lisa.

# Lesson 3: Sentiment RNN Prediction

## Sentiment RNN, Introduction

<https://www.youtube.com/watch?time_continue=2&v=bQWUuaMc9ZI&feature=emb_logo>

## Pre-Notebook: Sentiment RNN

### Notebook: Sentiment RNN

The next few videos will be all about implementing a complete RNN that can classify the sentiment of movie reviews (positive or negative).

**It's suggested that you open the notebook in a new, working tab and continue working on it as you go through the instructional videos in this tab.** This way you can toggle between learning new skills and coding/applying new skills.

To open this notebook, you have two options:

* Go to the next page in the classroom (recommended).
* Clone the repo from [**Github**](https://github.com/udacity/deep-learning-v2-pytorch) and open the notebook **Sentiment\_RNN\_Exercise.ipynb** in the **sentiment-rnn** folder. You can either download the repository with git clone https://github.com/udacity/deep-learning-v2-pytorch.git, or download it as an archive file from [**this link**](https://github.com/udacity/deep-learning-v2-pytorch/archive/master.zip).

### Instructions

* Load in text data
* Pre-process that data, encoding characters as integers
* Pad the data such that each review is a standard sequence length
* Define an RNN with embedding and hidden LSTM layers that predicts the sentiment of a given review
* Train the RNN
* See how it performs on test data

This is a self-assessed lab. If you need any help or want to check your answers, feel free to check out the solutions notebook in the same folder, or by clicking [**here**](https://github.com/udacity/deep-learning-v2-pytorch/blob/master/sentiment-rnn/Sentiment_RNN_Solution.ipynb).

### GPU Workspaces

The next workspace is **GPU-enabled**, which means you can select to train on a GPU instance. The recommendation is this:

* Load in data, test functions and models (checking parameters and doing a short training loop) while in CPU (non-enabled) mode
* When you're ready to extensively train and test your model, **enable** GPU to quickly train the model!

All models and data they see as input will have to be moved to the GPU device, so take note of the relevant movement code in the model creation and training process.

## Data PreProcessing

<https://www.youtube.com/watch?time_continue=1&v=Xw1MWmql7no&feature=emb_logo>

## Encoding Words, Solution

<https://www.youtube.com/watch?time_continue=22&v=4RYyn3zv1Hg&feature=emb_logo>

## Getting Rid of Zero-Length

<https://www.youtube.com/watch?time_continue=17&v=Hs6ithuvDJg&feature=emb_logo>

## Cleaning And Padding Data

<https://www.youtube.com/watch?time_continue=3&v=UgPo1_cq-0g&feature=emb_logo>

## Padded Features, Solution

<https://www.youtube.com/watch?v=sYOd1IDmep8&feature=emb_logo>

## TensorDataset & Batching Data

<https://www.youtube.com/watch?time_continue=3&v=Oxuf2QIPjj4&feature=emb_logo>

### Omission: shuffling data

Make sure to shuffle your data, so that your model doesn't learn anything about the ordering of the data, and instead can focus on the content. We can do this with a DataLoader by setting shuffle=True. You'll find this updated code in the exercise and solution notebooks.

# make sure to SHUFFLE your data

train\_loader = DataLoader(train\_data, shuffle=True, batch\_size=batch\_size)

valid\_loader = DataLoader(valid\_data, shuffle=True, batch\_size=batch\_size)

test\_loader = DataLoader(test\_data, shuffle=True, batch\_size=batch\_size)

### TensorDataset

Take a look at the source code for [**the TensorDataset class**](https://github.com/pytorch/tnt/blob/master/torchnet/dataset/tensordataset.py), you can see that it's "purpose" is to provide an easy way to create a dataset out of standard data structures.

## Defining the Model

<https://www.youtube.com/watch?time_continue=1&v=SpvIZl1YQRI&feature=emb_logo>

## Complete Sentiment RNN

### Consult the Solution Code

To take a closer look at this solution, feel free to check out the solution workspace or click [**here**](https://github.com/udacity/deep-learning-v2-pytorch/blob/master/sentiment-rnn/Sentiment_RNN_Solution.ipynb) to see it as a webpage.

### Complete RNN Class

I hope you tried out defining this model on your own and got it to work! Below, is how I completed this model.

I know I want an embedding layer, a recurrent layer, and a final, linear layer with a sigmoid applied; I defined all of those in the \_\_init\_\_ function, according to passed in parameters.

**def** **\_\_init\_\_**(self, vocab\_size, output\_size, embedding\_dim, hidden\_dim, n\_layers, drop\_prob=0.5):

"""

Initialize the model by setting up the layers.

"""

super(SentimentRNN, self).\_\_init\_\_()

self.output\_size = output\_size

self.n\_layers = n\_layers

self.hidden\_dim = hidden\_dim

*# embedding and LSTM layers*

self.embedding = nn.Embedding(vocab\_size, embedding\_dim)

self.lstm = nn.LSTM(embedding\_dim, hidden\_dim, n\_layers,

dropout=drop\_prob, batch\_first=**True**)

*# dropout layer*

self.dropout = nn.Dropout(0.3)

*# linear and sigmoid layers*

self.fc = nn.Linear(hidden\_dim, output\_size)

self.sig = nn.Sigmoid()

#### \_\_init\_\_ explanation

First I have an **embedding layer**, which should take in the size of our vocabulary (our number of integer tokens) and produce an embedding of embedding\_dim size. So, as this model trains, this is going to create and embedding lookup table that has as many rows as we have word integers, and as many columns as the embedding dimension.

Then, I have an **LSTM layer**, which takes in inputs of embedding\_dim size. So, it's accepting embeddings as inputs, and producing an output and hidden state of a hidden size. I am also specifying a number of layers, and a dropout value, and finally, I’m setting batch\_first to True because we are using DataLoaders to batch our data like that!

Then, the LSTM outputs are passed to a dropout layer and then a fully-connected, linear layer that will produce output\_size number of outputs. And finally, I’ve defined a sigmoid layer to convert the output to a value between 0-1.

### Feedforward behavior

Moving on to the forward function, which takes in an input x and a hidden state, I am going to pass an input through these layers in sequence.

**def** **forward**(self, x, hidden):

"""

Perform a forward pass of our model on some input and hidden state.

"""

batch\_size = x.size(0)

*# embeddings and lstm\_out*

embeds = self.embedding(x)

lstm\_out, hidden = self.lstm(embeds, hidden)

*# stack up lstm outputs*

lstm\_out = lstm\_out.contiguous().view(-1, self.hidden\_dim)

*# dropout and fully-connected layer*

out = self.dropout(lstm\_out)

out = self.fc(out)

*# sigmoid function*

sig\_out = self.sig(out)

*# reshape to be batch\_size first*

sig\_out = sig\_out.view(batch\_size, -1)

sig\_out = sig\_out[:, -1] *# get last batch of labels*

*# return last sigmoid output and hidden state*

**return** sig\_out, hidden

#### forward explanation

So, first, I'm getting the batch\_size of my input x, which I’ll use for shaping my data. Then, I'm passing x through the embedding layer first, to get my embeddings as output

These embeddings are passed to my lstm layer, alongside a hidden state, and this returns an lstm\_output and a new hidden state! Then I'm going to stack up the outputs of my LSTM to pass to my last linear layer.

Then I keep going, passing the reshaped lstm\_output to a dropout layer and my linear layer, which should return a specified number of outputs that I will pass to my sigmoid activation function.

Now, I want to make sure that I’m returning only the **last** of these sigmoid outputs for a batch of input data, so, I’m going to shape these outputs into a shape that is batch\_size first. Then I'm getting the last bacth by called `sig\_out[:, -1], and that’s going to give me the batch of last labels that I want!

Finally, I am returning that output and the hidden state produced by the LSTM layer.

#### init\_hidden

That completes my forward function and then I have one more: init\_hidden and this is just the same as you’ve seen before. The hidden and cell states of an LSTM are a tuple of values and each of these is size (n\_layers by batch\_size, by hidden\_dim). I’m initializing these hidden weights to all zeros, and moving to a gpu if available.

**def** **init\_hidden**(self, batch\_size):

''' Initializes hidden state '''

*# Create two new tensors with sizes n\_layers x batch\_size x hidden\_dim,*

*# 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.hidden\_dim).zero\_().cuda(),

weight.new(self.n\_layers, batch\_size, self.hidden\_dim).zero\_().cuda())

**else**:

hidden = (weight.new(self.n\_layers, batch\_size, self.hidden\_dim).zero\_(),

weight.new(self.n\_layers, batch\_size, self.hidden\_dim).zero\_())

**return** hidden

After this, I’m ready to instantiate and train this model, you should see if you can decide on good hyperparameters of your own, and then check out the solution code, next!

## Training the Model

### Hyperparameters

After defining my model, next I should instantiate it with some hyperparameters.

*# Instantiate the model w/ hyperparams*

vocab\_size = len(vocab\_to\_int)+1 *# +1 for the 0 padding + our word tokens*

output\_size = 1

embedding\_dim = 400

hidden\_dim = 256

n\_layers = 2

net = SentimentRNN(vocab\_size, output\_size, embedding\_dim, hidden\_dim, n\_layers)

print(net)

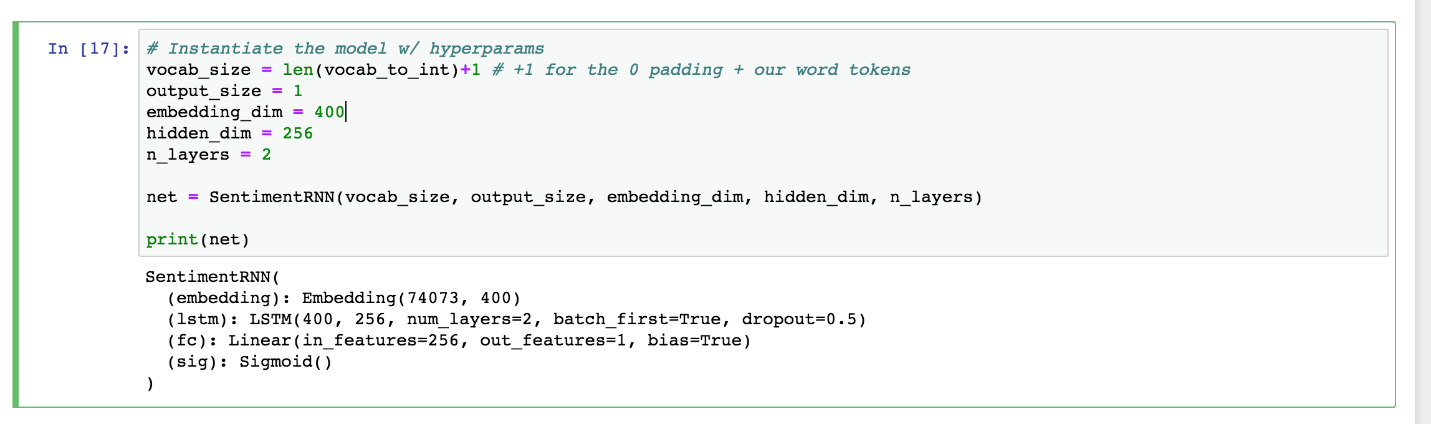
This should look familiar, but the main thing to note here is our vocab\_size.

This is actually the length of our vocab\_to\_int dictionary (all our unique words) **plus one** to account for the 0-token that we added, when we padded our input features. So, if you do data pre-processing, you may end up with one or two extra, special tokens that you’ll need to account for, in this parameter!

Then, I want my output\_size to be 1; this will be a sigmoid value between 0 and 1, indicating whether a review is positive or negative.

Then I have my embedding and hidden dimension. The embedding dimension is just a smaller representation of my vocabulary of 70k words and I think any value between like 200 and 500 or so would work, here. I’ve chosen 400. Similarly, for our hidden dimension, I think 256 hidden features should be enough to distinguish between positive and negative reviews.

I’m also choosing to make a 2 layer LSTM. Finally, I’m instantiating my model and printing it out to make sure everything looks good.



Model hyperparameters

### Training and Optimization

The training code, should look pretty familiar. One new detail is that, we'll be using a new kind of cross entropy loss that is designed to work with a single Sigmoid output.

[**BCELoss**](https://pytorch.org/docs/stable/nn.html#bceloss), or **Binary Cross Entropy Loss**, applies cross entropy loss to a single value between 0 and 1.

We'll define an Adam optimizer, as usual.

*# loss and optimization functions*

lr=0.001

criterion = nn.BCELoss()

optimizer = torch.optim.Adam(net.parameters(), lr=lr)

#### Output, target format

You should also notice that, in the training loop, we are making sure that our outputs are squeezed so that they do not have an empty dimension output.squeeze() and the labels are float tensors, labels.float(). Then we perform backpropagation as usual.

#### Train and eval mode

Below, you can also see that we switch between train and evaluation mode when the model is training versus when it is being evaluated on validation data!

#### Training Loop

Below, you’ll see a usual training loop.

I’m actually only going to do four epochs of training because that's about when I noticed the validation loss stop decreasing.

* You can see that I am initializing my hidden state before entering the batch loop then have my usual detachment from history for the hidden state and backpropagation steps.
* I’m getting my input and label data from my train\_dataloader. Then applying my model to the inputs and comparing the outputs and the true labels.
* I also have some code that checks performance on my validation set, which, if you want, may be a great thing to use to decide when to stop training or which best model to save!

*# training params*

epochs = 4 *# 3-4 is approx where I noticed the validation loss stop decreasing*

counter = 0

print\_every = 100

clip=5 *# gradient clipping*

*# move model to GPU, if available*

**if**(train\_on\_gpu):

net.cuda()

net.train()

*# train for some number of epochs*

**for** e **in** range(epochs):

*# initialize hidden state*

h = net.init\_hidden(batch\_size)

*# batch loop*

**for** inputs, labels **in** train\_loader:

counter += 1

**if**(train\_on\_gpu):

inputs, labels = inputs.cuda(), labels.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.squeeze(), labels.float())

loss.backward()

*# `clip\_grad\_norm` helps prevent the exploding gradient problem in RNNs / LSTMs.*

nn.utils.clip\_grad\_norm\_(net.parameters(), clip)

optimizer.step()

*# loss stats*

**if** counter % print\_every == 0:

*# Get validation loss*

val\_h = net.init\_hidden(batch\_size)

val\_losses = []

net.eval()

**for** inputs, labels **in** valid\_loader:

*# 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])

**if**(train\_on\_gpu):

inputs, labels = inputs.cuda(), labels.cuda()

output, val\_h = net(inputs, val\_h)

val\_loss = criterion(output.squeeze(), labels.float())

val\_losses.append(val\_loss.item())

net.train()

print("Epoch: {}/{}...".format(e+1, epochs),

"Step: {}...".format(counter),

"Loss: {:.6f}...".format(loss.item()),

"Val Loss: {:.6f}".format(np.mean(val\_losses)))

Make sure to take a look at how training **and** validation loss decrease during training! Then, once you're satisfied with your trained model, you can test it out in a couple ways to see how it behaves on new data!

### Consult the Solution Code

To take a closer look at this solution, feel free to check out the solution workspace or click [**here**](https://github.com/udacity/deep-learning-v2-pytorch/blob/master/sentiment-rnn/Sentiment_RNN_Solution.ipynb) to see it as a webpage.

## Testing

### Testing the Trained Model

I want to show you two great ways to test: using test data and using inference. The first is similar to what you’ve seen in our CNN lessons. I am iterating through the test data in the test\_loader, recording the test loss and calculating the accuracy based on how many labels this model got correct!

I’m doing this by looking at the **rounded value** of our output. Recall that this is a sigmoid output between 0-1 and so rounding this value will give us an integer that is the most likely label: 0 or 1. Then I’m comparing that predicted label to the true label; if it matches, I record that as a correctly-labeled test review.

*# Get test data loss and accuracy*

test\_losses = [] *# track loss*

num\_correct = 0

*# init hidden state*

h = net.init\_hidden(batch\_size)

net.eval()

*# iterate over test data*

**for** inputs, labels **in** test\_loader:

*# Creating new variables for the hidden state, otherwise*

*# we'd backprop through the entire training history*

h = tuple([each.data **for** each **in** h])

**if**(train\_on\_gpu):

inputs, labels = inputs.cuda(), labels.cuda()

*# get predicted outputs*

output, h = net(inputs, h)

*# calculate loss*

test\_loss = criterion(output.squeeze(), labels.float())

test\_losses.append(test\_loss.item())

*# convert output probabilities to predicted class (0 or 1)*

pred = torch.round(output.squeeze()) *# rounds to the nearest integer*

*# compare predictions to true label*

correct\_tensor = pred.eq(labels.float().view\_as(pred))

correct = np.squeeze(correct\_tensor.numpy()) **if** **not** train\_on\_gpu **else** np.squeeze(correct\_tensor.cpu().numpy())

num\_correct += np.sum(correct)

*# -- stats! -- ##*

*# avg test loss*

print("Test loss: {:.3f}".format(np.mean(test\_losses)))

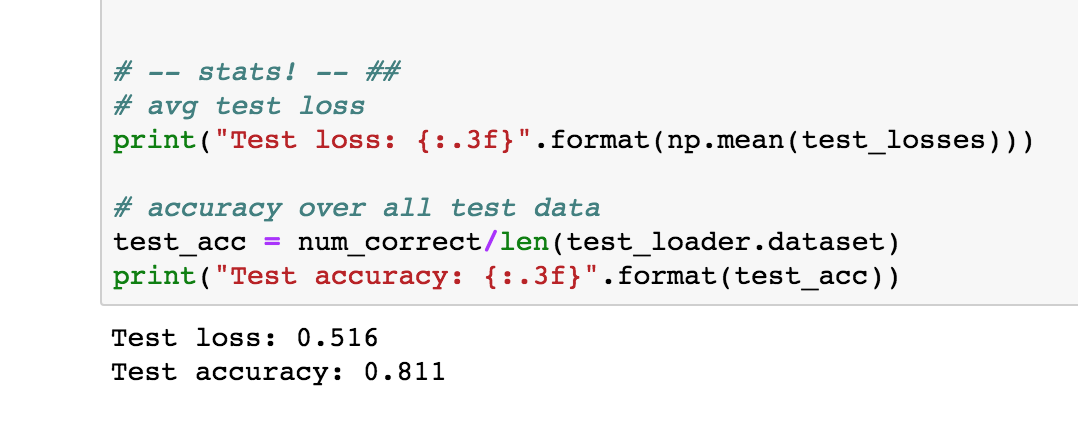
*# accuracy over all test data*

test\_acc = num\_correct/len(test\_loader.dataset)

print("Test accuracy: {:.3f}".format(test\_acc))

Below, I’m printing out the average test loss and the accuracy, which is just the number of correctly classified items divided by the number of pieces of test data,total.

We can see that the test loss is 0.516 and the accuracy is about **81.1%** !



Test results

Next, you're ready for your last task! Which is to define a predict function to perform inference on any given text review!

### Exercise: Inference on a test review

You can change this test\_review to any text that you want. Read it and think: is it pos or neg? Then see if your model predicts correctly!

**Exercise:** Write a predict function that takes in a trained net, a plain text\_review, and a sequence length, and prints out a custom statement for a positive or negative review!

* You can use any functions that you've already defined or define any helper functions you want to complete predict, but it should just take in a trained net, a text review, and a sequence length.

**def** **predict**(net, test\_review, sequence\_length=200):

''' Prints out whether a give review is predicted to be

positive or negative in sentiment, using a trained model.

params:

net - A trained net

test\_review - a review made of normal text and punctuation

sequence\_length - the padded length of a review

'''

*# print custom response based on whether test\_review is pos/neg*

Try to solve this task on your own, then check out the solution, next!

## Inference, Solution

### Inference

Let's put all these pieces together! One of the coolest ways to test a model like this is to give it user-generated data, without any true label, and see what happens. So, in this case, that data will just be a single string: a review that you can write and here’s just one test\_reviewas an example:

*# negative test review*

test\_review\_neg = 'The worst movie I have seen; acting was terrible and I want my money back. This movie had bad acting and the dialogue was slow.'

We can see that this review is a negative one, but let's see if our model can identify it's sentiment correctly!

Our task is to write a predict function that takes in a trained model, a test\_review like this one that is just normal text and punctuation, a sequence\_length for padding.

The process by which you make predictions based on user data, is called **inference**.

#### Pre-process the test\_review

The first thing we'll have to do it to process the test\_review, so that it is converted into a tensor that our model can see as input. In fact, this involves quite a lot of pre-processing, but nothing that you haven't seen before!

I broke this down into a series of steps.

I have a helper function tokenize\_review that is responsible for doing some data processing on my test\_review.

It takes in my test\_review, and then does a couple of things:

1. First, I convert my test\_review to lowercase, and remove any punctuation, so I’m left with all text.
2. Then I breaks it into individual words with split(), and I’m left with a list of words in the review.
3. I encode those words using the vocab\_to\_int dictionary that we already defined, near the start of this lesson.

Now, I am assuming a few things here, including: this review is one review, not a batch, and that this review only includes words already in our dictionary, and in this case that will be true, but you can add code to handle unknown characters, I just didn’t do that in my model.

**from** string **import** punctuation

**def** **tokenize\_review**(test\_review):

test\_review = test\_review.lower() *# lowercase*

*# get rid of punctuation*

test\_text = ''.join([c **for** c **in** test\_review **if** c **not** **in** punctuation])

*# splitting by spaces*

test\_words = test\_text.split()

*# tokens*

test\_ints = []

test\_ints.append([vocab\_to\_int[word] **for** word **in** test\_words])

**return** test\_ints

Okay, so this tokenize function returns a list of integers; my tokenized review!

#### Padding and converting into a Tensor

For my next couple of steps, I’m going to pad the ints, returned by the tokenize\_review function and shape them into our sequence\_length size; since our model was trained on sequence lengths of 200, I’m going to use the same length, here. I'll pad it using the pad\_features function that we defined earlier.

Finally, I’m going to convert the padded result into a Tensor. So, these are all the steps, and I’m going to wrap this all up in my predict function.

**def** **predict**(net, test\_review, sequence\_length=200):

net.eval()

*# tokenize review*

test\_ints = tokenize\_review(test\_review)

*# pad tokenized sequence*

seq\_length=sequence\_length

features = pad\_features(test\_ints, seq\_length)

*# convert to tensor to pass into your model*

feature\_tensor = torch.from\_numpy(features)

batch\_size = feature\_tensor.size(0)

*# initialize hidden state*

h = net.init\_hidden(batch\_size)

**if**(train\_on\_gpu):

feature\_tensor = feature\_tensor.cuda()

*# get the output from the model*

output, h = net(feature\_tensor, h)

*# convert output probabilities to predicted class (0 or 1)*

pred = torch.round(output.squeeze())

*# printing output value, before rounding*

print('Prediction value, pre-rounding: {:.6f}'.format(output.item()))

*# print custom response*

**if**(pred.item()==1):

print("Positive review detected!")

**else**:

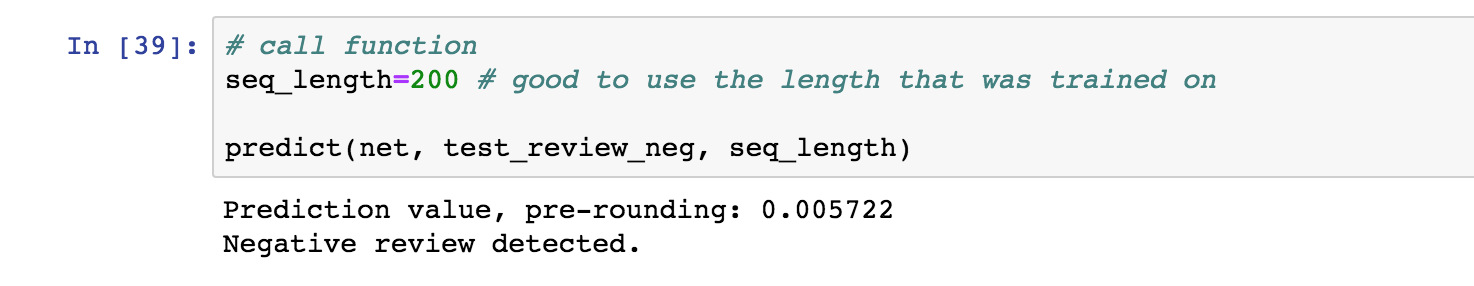
print("Negative review detected.")

So, using the passed in arguments, I’m tokenizing my review using my helper function, then padding it using my pad function, and converting it into a Tensor that can be seen by my model.

Then, I’m passing this tensor into my trained net which will return an output of length one. With this output, I can grab the most likely class, which will be the rounded value 0 or 1; this is my prediction!

Lastly, I want to print out a custom message for a positive or negative detected review, and I’m doing that at the bottom of the above function!

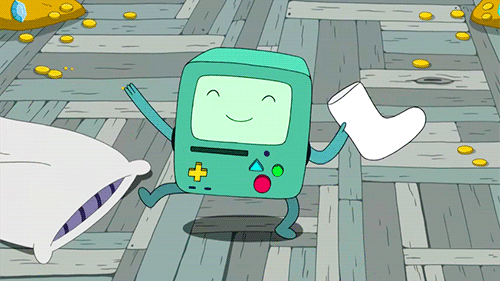
**You can test this out on sample positive and negative text reviews to see how this trained model behaves!** Below, you can see how it identifies our negative test review correctly.



#### Identifies negative review

### Conclusion

Now that you have a trained model and a predict function, you can pass in any kind of text and this model will predict whether the text has a positive or negative sentiment. You can use this to try to find what words it associates with positive or negative sentiment.



Dancing Beemo from [**Adventure Time**](https://en.wikipedia.org/wiki/Adventure_Time) to celebrate!

Later, you'll learn how to deploy a model like this to a production environment so that it can respond to any kind of user data put into a web app!

**For now, great job implementing so many kinds of recurrent neural networks!!**