Assignment 1: Welcome to TensorFlow

CS20: TensorFlow for Deep Learning Research (cs20.stanford.edu)
Due 1/31 at 11:59pm
Prepared by Chip Huyen (chiphuyen@cs.stanford.edu)

I'd like the assignments to more closely resemble what you'd encounter in the real world, so the problems are more open-ended and less structured. Problems often don't have starter code or very simple starter code. You get to implement everything from scratch!

Most problems have several options for you to choose from. Some options are easier than others, and that will be taken into account when I grade. You'll be graded on both functionality and style, e.g. how elegant your code is.

I'm more interested in the process than the results, so you might still get good grade even if you don't arrive at the results you want to, as long as you explain the difficulties you encountered along the way and how you tackled them.

I hope that through this problem set, you'd familiarize yourself with TensorFlow's official documentation.

Problem 1: Op is all you need

This first problem asks you to create simple TensorFlow ops to do certain tasks. You can see the tasks at `assignments/01/q1.py` on the GitHub repository for this class.

These tasks are simple--their purpose is to get you acquainted with the TensorFlow API.

Problem 2: Logistic regression

2a. Logistic regression with MNIST

We never got around to doing this in class but don't worry, you'll have a chance to do it at home. You can find the starter code at `examples/03_logreg_starter.py` on the GitHub repository of the class.

For the instruction, please see the lecture note 03.

2b. More logistic regression

You can choose to do one of the two tasks below.

Task 1: Improve the accuracy of logistic regression on MNIST

We got the accuracy of ~91% on our MNIST dataset with our vanilla model, which is unacceptable. The state of the art is above 99%. You have full freedom to do whatever you want here as long as your model is built in TensorFlow.

You can reuse the code from part 1, but please save your code for part 2 in a separate file and name it `q2b.py`. In the comments, explain what you decide to do, instruction on how to run your code, and report your results. I'm happy with anything above 97%.

Task 2: Logistic regression on notMNIST

MNIST is so popular as a toy dataset for computer vision tasks that the machine learning community got a little bit sick of it. Yaroslav Bulatov, a research engineer at OpenAI, created a similar dataset and literally named it notMNIST[1]. This rebel is designed to look like the classic MNIST dataset, but less 'clean' and extremely cute. The images are still 28x28 and the there are also 10 labels, representing letters 'A' to 'J'. You should save your file in `q2b.py`.

[1] Bulatov, Y. "Notmnist dataset." Google (Books/OCR), Tech. Rep.[Online]. Available: http://yaroslavvb. blogspot. it/2011/09/notmnist-dataset. html (2011).

The format of notMNIST is not the same as MNIST, but David Flanagan is very kind to publish his script to convert notMNIST to MNIST format. After converting notMNIST to MNIST format, you can use the `read_mnist` module in `utils.py` to read it in.

Once you have the data ready, you can use the exact model you built for MNIST for notMNIST. Don't freak out if the accuracy is lower for notMNIST. notMNIST is supposed to be harder to train than MNIST.

Problem 3: More on word2vec

The answers for questions 3a, 3b, and 4 should be saved in a file named a1_answers in any format of your choice.

3a. Application of word embeddings

Briefly give three examples of real world applications that use word embeddings.

3b. Visualize word embeddings and find interesting word relations

We've implemented word2vec in TensorFlow and it's pretty neat. You should visualize the word embeddings on TensorBoard. Then you can either use this visualization or write a script of your own to find the interesting word relations. For example, you can find five words closest to certain country names to see if our embeddings catch any racial bias. Another idea is that you can find word spectra:

you start with two opposite words, get the line connecting that two words, get 100 words closest to that line and arrange them by the order they appear on that line, you'll see how a word fades into another.

For inspiration, you can read about sexism in word embeddings here. I've also written a post about the inherent biases of Artificial Intelligence here.

3c. word2vec

We implemented word2vec with skip-gram model in class. Now, you can try to implement the CBOW model. Save the model in `cbow.py`.

Problem 4: Feedback

This is an opportunity for you to give me feedback on the class. Please answer some or all of the questions below:

- What parts of the class do you find useful?
- What parts do you find confusing?
- What parts do you find too easy?
- What do you want to see/do more in the class?
- What parts did you find less interesting?
- What do you think of the first assignment?
- How long did it take you to do the first assignment?
- Any recommendations for me?

Submission Instructions

Your submission should contain the following files:
q1.py
03_logreg_starter.py
q2b.py
cbow.py
a1_answers (contains your answers to questions 3a, 3b, and 4)

Compress your submission into a zip file and send it to cs20-win1718-staff@lists.stanford.edu with subject title: "[Your SUNetID]_assignment1"

Thank a lot, guys!

Assignment 2: Style Transfer

CS20: TensorFlow for Deep Learning Research (cs20.stanford.edu)

Due 2/19 at 11:59pm

Prepared by Chip Huyen (chiphuyen@cs.stanford.edu)

In this assignment, you'll be implementing the much hyped neural style transfer. Neural style is so cool even the [Twilight star co-authored a paper about it.]

Bringing Impressionism to Life with Neural Style Transfer in *Come Swim*

Bhautik J Joshi*
Research Engineer, Adobe

Kristen Stewart
Director, *Come Swim*

David Shapiro
Producer, Starlight Studios

Figure 1: *Usage of Neural Style Transfer in Come Swim; left: content image, middle: style image, right: upsampled result. Images used with permission, (c) 2017 Starlight Studios LLC & Kristen Stewart.*

For those who have been unaware of the hype, style transfer is the task of transferring the style of an image to another image. Please read the paper A Neural Algorithm of Artistic Style (Gatys et al., 2016) to understand the motivation and intuition behind this.

We want to build a system that takes two images and outputs another image whose content is closest to one image while style is closest to the style of the other.

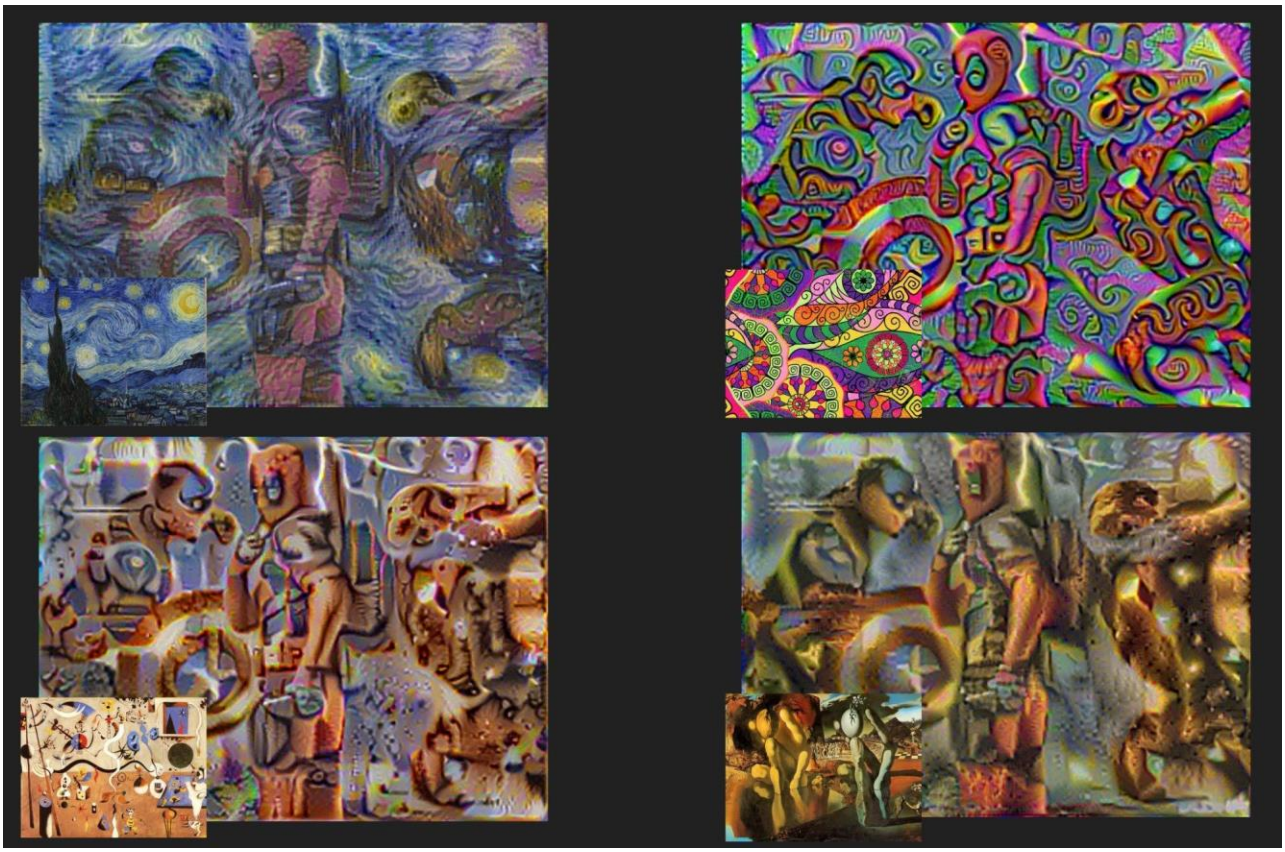
For example, take this image of Deadpool in Civil War[1] as the content image and combine it with the style of Guernica by Picasso (1937).

[1] Image source: theodysseyonline.com

After 160 iterations, you get this:

The above image is pretty big (600 x 800), 160 iterations took approximately 1.5 hours. Smaller images, for example, 250 x 333, would take around 15 minutes.

More Deadpool art



The style images, from left to right, top to bottom are:
‘The Starry Night’ by Vincent van Gogh (1889)
Pattern by Olga Drozdova (201x?)
‘The Harlequin's Carnival’ by Joan Miró (1924-1925)
‘Metamorphosis of Narcissus’ by Salvador Dalí (1937)

Or if you’re as self-indulgent as I am, you can train your model on your own images to create this artsy collage of yourself. Can you guess which paintings are used as the style images for these photos?

The above are the images I generated from the model I implemented for this assignment. If you build your model correctly, you'll be able to generate other similar but much cooler images. To speed up the computation during writing code, you might want to use lower resolution images (about 250 x 333). After you're done, you should feel free to use images with higher resolution to maximize the awesomeness.

The model is not mathematically difficult. However, the implementation is a bit involved since it's different from most of the models we've implemented in three aspects:

1. In the models that we've learned so far, we import data and use it to train variables -- we don't try to modify our original inputs. For this model, you have two fixed inputs: content image and style image, but also have a trainable input which will be trained to become the generated artwork.

2. There is not a clear distinction between the two phases of a TensorFlow program: assembling the graph and executing it. All the 3 input (content image, style image, and trainable input) have the same dimensions and act as input to the same computation to extract the same sets of features. To save us from having to assemble the same subgraph multiple times, we will use one variable for all three of them. The variable is already defined for you in the model as:

```
self.input_img = tf.get_variable('in_img',
                                 shape=([1, self.img_height, self.img_width, 3]),
                                 dtype=tf.float32,
                                 initializer=tf.zeros_initializer())
```

I know you're thinking, "What do you mean all three inputs share the same variable? How does TF know which input it's dealing with?" You remember that in TF we have the assign op for variables. When we need to do some computation that takes in the content image as the input, we first assign the content image to that variable, and so on. You'll have to do that to calculate content loss (since it depends on the content image), and style loss (since it depends on the style image).

3. In this assignment, you'll get acquainted to transfer learning: we use the weights trained for another task for this task. We will use the weights and biases already trained for the object recognition task of the model VGG-19[2] (a convolutional network with 19 layers) to extract content and style layers for style transfer. For more context on this model, you should read about it here. We'll only use their weights for the convolution layers. The paper by Gatys et al. suggested that average pooling is better than max pooling, so we'll have to do pooling ourselves.

But other than that, this is a typical model. Remember that in a model, you have several steps:
Step 1: Define inference
Step 2: Create loss functions
Step 3: Create optimizer
Step 4: Create summaries to monitor your training process
Step 5: Train your model

You'll progressively do all of those in this assignment. I suggest that you do all the tasks in that order.

There are 3 files in the starter code, which you should be able to find on GitHub.

[2] Simonyan, Karen, and Andrew Zisserman. "Very deep convolutional networks for large-scale image recognition." arXiv preprint arXiv:1409.1556 (2014).

style_transfer.py is the main file. You'll call python style_transfer.py to run your model. You'll have to modify this file.
load_vgg.py is where you load in the trained VGG variables. You'll have to modify this file.
utils.py contains utils for the assignment. You should read it to know what utilities are offered. You shouldn't have to modify this file, but you're welcome to do so if you find it necessary.

The folder **styles** contains several paintings that you can use as your style images, and the folder content contains just one image deadpool.jpg that you can use as your content image. Feel free to add more style and content images.

You can download the starter code from the class GitHub repository.

Okay, are you ready?

Step 1: Define inference

You should modify the function **conv2d_relu()** and **avgpool()** in **load_vgg.py**.
If you have problems with this part, you should refer to the convolutional model we built for MNIST.

Step 2: Create loss functions

You'll have to modify the function **_content_loss()**, **_style_loss()** and their corresponding helper functions **_single_style_loss()** and **_gram_matrix()** in **style_transfer.py**. This part is tricky, so please read the following instructions very carefully.

You have two losses and you try to minimize the combination of them. The content loss is defined as following:

$$\mathcal{L}_{content}(\vec{p}, \vec{x}, l) = \frac{1}{2} \sum_{i,j} \left(F_{ij}^{l} - P_{ij}^{l}\right)^2$$

where F is the feature representation of the generated image and P is the feature representation of the content image layer l. The paper suggests that we use the feature map from the layer **'conv4_2'**. The loss function is basically the mean squared error of F and P.

However, in practice, we've found that this function makes it really slow to converge, so people often replace the coefficient ½ with 1/(4s) in which s is the product of the dimension of P. If P has dimension [5, 5, 3] then s = 5 * 5 * 3 = 75.

The style loss is defined as following:

$$E_l = \frac{1}{4N_l^2 M_l^2} \sum_{i,j} \left(G_{ij}^l - A_{ij}^l\right)^2$$

$$\mathcal{L}_{style}(\vec{a}, \vec{x}) = \sum_{l=0}^{L} w_l E_l$$

N is the third dimension of the feature map, and M is the product of the first two dimensions of the feature map. However, remember that in TensorFlow, we have to add one extra dimension to make it 4D to make it work for the function tf.nn.conv2d, so the first dimension is actually the second, and the second is the third, and so on.

A is the Gram matrix from the original image and G is the Gram matrix of the image to be generated. To obtain the gram matrix, for example, of the style image, we first need to get the feature map of the style image at that layer, then reshape it to 2D tensor of dimension M x N, and take the dot product of 2D tensor with its transpose. You do the same thing to get the gram matrix from the feature map of the generated image. If you don't know what gram matrix does, you should look it up to understand. I recommend the Wikipedia article and this video.

The subscript l stands the layer whose feature maps we want to incorporate into the generated images. In the paper, it suggests that we use feature maps from 5 layers:

```
['conv1_1', 'conv2_1', 'conv3_1', 'conv4_1', 'conv5_1']
```

After you've calculated the tensors E's, you calculate the style loss by summing them up with their corresponding weight w's. You can tune w's, but I'd suggest that you give more emphasis to deep layers. For example, w for 'conv1_1' can be 1, then weight for 'conv2_1' can be 2, and so on.

After you've got your content loss and style loss, you should combine them with their corresponding weights to get the total loss -- and the total loss is what we'll try to minimize.

$$\mathcal{L}_{total}(\vec{p}, \vec{a}, \vec{x}) = \alpha \mathcal{L}_{content}(\vec{p}, \vec{x}) + \beta \mathcal{L}_{style}(\vec{a}, \vec{x})$$

The paper suggests that we use alpha and beta such that alpha/beta = 0.001 or 0.0001, but I've found that the ratio alpha/beta = 1/20 or 1/50 works just fine.

Step 3: Create optimizer

I suggest AdamOptimizer but you can be creative with both optimizers and learning rate to see what you find. You can find this part in the **optimize**() method in **style_transfer.py.**

Step 4: Create summaries

You need to summary ops for the values that you want to monitor through your training process in TensorBoard. You should find this in the **create_summary**() method in **style_transfer.py**

Train your model for at least 200 iterations and submit the content loss graph, style loss graph, the total loss graph, and the graph of your model. You should justify in at most 3 sentences what you see in the loss graphs and why.

Step 5: Train your model

You should modify the TO DO parts in the **train()** method of **style_transfer.py**

There are a lot of hyperparameters that you can play around with. See them in __init__ method for **StyleTransfer**. For example, you can change the weights for the content loss and the style loss, change the coefficients for content and style losses, change the learning rate, use different layers for style and content, etc.

You can see how the generated images look like after certain number of iterations in the folder **output**. The training progress looks like this:

Deliverables

1. The finished code.
2. The training curve of content loss, style loss, and the total loss. Write a few sentences about what you see.
3. The graph of your model.
4. Change at least two parameters, explain what you did and how that changed the results.
5. 3 artworks generated using at least 3 different styles.

Submission instruction

After you've finished, zip up all deliverables and name it using your SUNet ID. Send it to cs20-win1718-staff@lists.stanford.edu with subject title: "[Your SUNetID]_assignment2"

Have fun!

Assignment 3: Open Domain Dialogue System

CS20: TensorFlow for Deep Learning Research (cs20.stanford.edu)
Due 3/15 at 11:59pm
Prepared by Chip Huyen (chiphuyen@cs.stanford.edu)

There are two options for this assignment, you can choose to do any of those. *You must demo this assignment in class on 3/16 to get credit. Contact the teaching staff asap if you can't make it.*

Option 1: Open domain dialogue system

As you already know, chatbots have been all the rage. Everyone seems to want a chatbot. If you choose this option, you're given the starter code of a basic chatbot that uses a sequence to sequence model with an attention decoder. Your job is to improve on the chatbot to make it sound as human as possible. Keep in mind that this option takes a significant more training time than the others. I wouldn't recommend it unless you have access to GPUs/TPUs.

Traditionally, chatbots have been rule-based. As recent as 2014, Siri and Google Now still relied on handcrafted rules to find the most relevant answers[1]. But deep learning techniques and powerful computers have opened up new possibilities that we are still exploring.

The model

The chatbot is based on the translate model on the TensorFlow repository, with some modification to make it work for a chatbot. It's a sequence to sequence model with attention decoder. If you don't know what a sequence to sequence model is, please see the lecture 12 on the course website. The encoder is a single utterance, and the decoder is the response to that utterance. An utterance could be a sentence, more than a sentence, or even less than a sentence, anything people say in a conversation!

The chatbot is built using a wrapper function for the sequence to sequence model with bucketing. The loss function we use is sampled_softmax.

```
self.outputs, self.losses = tf.contrib.legacy_seq2seq.model_with_buckets(
                                    self.encoder_inputs,
                                    self.decoder_inputs,
                                    self.targets,
```

[1] https://www.fastcompany.com/3027067/this-cambridge-researcher-just-embarrassed-siri

```
                                        self.decoder_masks,
                                        config.BUCKETS,
                                        lambda x, y: _seq2seq_f(x, y, True),
                                        softmax_loss_function=self.softmax_loss_function)

                                        lambda x, y: _seq2seq_f(x, y, True),
                                        softmax_loss_function=self.softmax_loss_function)
```

The _seq2seq_f is defined as:

```
def _seq2seq_f(encoder_inputs, decoder_inputs, do_decode):
            return tf.nn.seq2seq.embedding_attention_seq2seq(
                    encoder_inputs, decoder_inputs, self.cell,
                    num_encoder_symbols=config.ENC_VOCAB,
                    num_decoder_symbols=config.DEC_VOCAB,
                    embedding_size=config.HIDDEN_SIZE,
                    output_projection=self.output_projection,
                    feed_previous=do_decode)
```

By default, do_decode is set to be True, which means that during training, we'll feed in the previously predicted token to help predicting the next token in the decoder even if the token was the wrong prediction. This helps approximate the training to be closer to the real environment when the chatbot has to make the prediction for the entire decoder from solely the encoder inputs.

And the softmax_loss_function is the sampled softmax to approximate the softmax.

```
def sampled_loss(inputs, labels):
            labels = tf.reshape(labels, [-1, 1])
            return tf.nn.sampled_softmax_loss(tf.transpose(w), b, inputs, labels,
                                              config.NUM_SAMPLES, config.DEC_VOCAB)
self.softmax_loss_function = sampled_loss
```

The outputs object returned by seq2seq.model_with_buckets or any pre-built seq2seq functions in TensorFlow is a list of decoder_size tensors, each of dimension 1 x decoder_vocab_size corresponding to the (more or less) probability distribution of the token at the decoder time step. I said more or less because it's not a real distribution -- the values in the tensor aren't limited to be between 0 and 1, and don't necessarily sum up to 1. However, the highest value still means the most likely token. For example, if your decoder size is 3 (which means the model should construct a decoder of 3 tokens), and your decoder vocabulary has a size of 4 corresponding to 10 tokens (a, b, c, d), then the outputs will be something like this:

```
self.outputs = [[2.3, 4.2, 3.0, 1.9], [-1.2, 0.1, 0.3, 2.0], [1.6, -1.8, 0.4, 0.5]]
```

To construct the response from an input, the starter code uses the greedy approach, which means it takes the most likely token at each time step. For example, given the outputs above, we'll get the b as the first token (corresponding to the value 4.2), d as the second token, and a as the third token. So the response will be 'b d a'.

This greedy approach works poorly, and restricts the chatbot to give one fixed answer to a input. You can improve this -- see **Your improvement** section.

Dataset

The bot comes with the script to do the pre-processing for the Cornell Movie-Dialogs Corpus, created by Cristian Danescu-Niculescu-Mizil and Lillian Lee at Cornell University. This is an extremely well-formatted dataset of dialogues from movies. It has 220,579 conversational exchanges between 10,292 pairs of movie characters, involving 9,035 characters from 617 movies with 304,713 total utterances.

The corpus comes together with the paper "Chameleons in Imagined Conversations: A new Approach to Understanding Coordination of Linguistic Style in Dialogs", which was featured on Nature.com. It is a fascinating paper that highlights several cognitive biases in conversations that will help you make your chatbot more realistic. I highly recommend that you read it.

The preprocessing is pretty basic. I consider most of the punctuations as separate tokens. I normalize all digits to '#'. I lowercase everything. I noticed that the dialogs have a lot of <u> and </u>, as well as [and], so I just get rid of those. You're welcome to experiment with other ways to pre-process your data.

Sample conversations

The bot comes with the code that writes down all the conversations the bot has on the output_convo.txt in the processed folder. Some of the conversations are sassy, but some are also pretty creepy. Some make absolutely no sense at all.

Starter code

The starter code can be found on the class GitHub repository.

In the folder **chatbot**, there are 4 main files:
model.py is where you specify the model and build the graph for your model.

data.py is the script to do all the data-related tasks, from separating the data into test set and train set, preprocessing the data, to making it ready to be fed to the model.
config.py contains configuration hyperparameters for the model.
chatbot.py is the main file that you'll run to train or to chat with your chatbot.

Please see README.md for instruction on how to run the starter code.

Your improvement

The starter bot's conversational ability is far from being satisfactory, and there are many ways you can improve the bot. You have the free range to use anything you want, even if you want to construct an entirely new architecture. Below are some of the improvements you can make.

To pass the class, you will have implement at least one of them and make it work decently. For information on how I evaluate "decently", please read the evaluation section below.

1. Train on multiple datasets

Bots are only as good as their data. If you play around with the starter bot, you'll see that the chatbot can't really hold normal conversations such as "how are you?", "what do you want to for lunch?", or "bye", and it's prone to saying dramatic things like "what about the gun?", "you're in trouble", "you're in love". The bot also tends to answer with questions. This makes sense, since Hollywood screenwriters need dramatic details and questions to advance the plot. However, training on movie dialogues makes your bot sound like a dumb version of the Terminator.

To make the bot more realistic, you can try training your bot on other datasets. Here are some of the possible datasets:

Twitter chat log (courtesy of Marsan Ma)
More movie substitles (less clean)
Every publicly available Reddit comments (1TB of data!)
Your own conversations (chat logs, text messages, emails)

You'll have to do the pre-processing yourself. Once you've had the train.dec, train.enc, test.dec, and test.enc, you can just plug the current code in to make the data ready for the model. Please see the data.py file to have a better understanding of how this is done.

2. Use more than just one utterance as the encoder

For the chatbot, the encoder is the last utterance, and the decoder is the response to that. You can see that this is problematic because you often have to use information from the previous utterances to construct an appropriate response.

You can modify the model to be able to use more than one utterance as the encoder input. This will make your model more like a summarization model in which your encoder is longer than your decoder.

To do this, you'll have to modify the bucket lengths and some of the data processing code. It will take longer to train on longer inputs.

3. Make your chatbot remember information from the previous conversation

Right now, if I tell the bot my name and ask what my name is right after, the bot will be unable to answer. This makes sense since we only use the last previous utterance as the input to predict the response without incorporating any previous information, however, this is unacceptable in real life conversation.

```
> hi
hi . what ' s your name ?
> my name is chip
nice to meet you .
> what ' s my name ?
let ' s talk about something else .
```

What you can do is to save the previous conversations you have with that user and refer to them to extract information relevant to the current conversation. This is not an easy task, but it's an exciting one.

4. Create a chatbot with personality

Right now, the chatbot is trained on the responses from thousands of characters, so you can expect the responses are rather erratic. It also can't answer to simple questions about personal information like "what's your name?" or "where are you from?" because those tokens are mostly unknown tokens due to the pre-processing phase that gets rid of rare words.

You can change this by using one of the two approaches (or another, this is a very open field).

Approach 1: At the decoder phase, inject consistent information about the bot such as name, age, hometown, current location, job.

Approach 2: Use the decoder inputs from one character only. For example: your own Sheldon Cooper bot!

There are also some pretty good Quora answers to this. Last year, my friends and I trained several bots like this and they were really fun to play with.

Michael (The Ofifce) Rude, abrupt Thinks he's the best boss 	Barney (HIMYM) Very bro-y Likes to hit on women Drinks a lot 	Sheldon (BBT) Socially awkward Thinks other people are stupid 	Joey (Friends) Not very smart but kind A player 
> hi **4_michael** have a seat . > who are you? **4_michael** i ' m just a friend . > what's your name? **4_michael** michael scott. > do you have a girlfriend? **4_michael** i have a lot . > tell me a joke **4_michael** no , i ' m not gonna tell you . > you are boring. **4_michael** you are . > where do you work? **4_michael** right here . > tell me michael. why do you work here? **4_michael** i thought i had to do something better . > bye bye michael	> who are you? **2_barney** i am the master of the most barney . > hi barney! **2_barney** hi , have a drink . > oh thanks. what drinks do you have? **2_barney** whatever what do you do ? > i'm a student. **2_barney** a . . . buddy . > yeah? **2_barney** sleep with her mom ? > lol no **2_barney** because she ' s not happy . > ha ha okay. how's work? **2_barney** okay . > just okay? **2_barney** oh , okay . > what do you think of that girl over there? **2_barney** do not even	> hi **1_sheldon** hello . > who are you? **1_sheldon** i ' m sheldon . > what's up? **1_sheldon** i see how i could do . > what can you do? **1_sheldon** i have to tell you a secret . > tell me. **1_sheldon** no . > what do you like to do for fun? **1_sheldon** i want to leave . > to go home? **1_sheldon** no . > how are you doing? **1_sheldon** i just wanted to sleep . > sleep! **1_sheldon** penny , i am getting too much for you .	> hey what's up **4_joey** hey ! > how are you? **4_joey** oh , oh , i ' m fine ! > what's your name **4_joey** joey , joey . > who are you? **4_joey** oh , joey ' s joey ! joey ? > joey joey joey **4_joey** what ? > tell me a joke. **4_joey** a woman . > a woman is not a joke! **4_joey** oh , i am very sorry .

5. Use character-level sequence to sequence model for the chatbot

We've built a character-level language model and it seems to be working pretty well, so is there any chance a character-level sequence to sequence model will work?

An obvious advantage of this model is that it uses a much smaller vocabulary so we can use full softmax instead of sampled softmax, and there will be no unknown tokens! An obvious disadvantage is that the sequence will be much longer -- it'll be approximately 4 times longer than the token-level one.

6. Non-greedy decoder

This greedy approach works poorly, and restricts the chatbot to give one fixed answer to a input. For example, if the user says "hi", the bot will always "hi" back, while in real life, people can vary their responses to "hey", "how are you?", or "hi. what's up?"

You can try to use beam search to construct the most probable response.

7. Create a feedback loop that allows users to train your chatbot

That's right, you can create a feedback loop so that users can help the bot learn the right response -- treat the bot like a baby. So when the bot says something incorrect, users can say: "That's wrong. You should have said xyz" and the bot will correct its response to xyz.

8. Use Tensor2Tensor to build your chatbot

Tensor2tensor has many off-the-shelf models that can handle encoder-decoder setup. The code is much better than my code.

9. An improvement of your choice

There is still a lot of room for improvement. Be creative!

Evaluation

The problem is that there isn't any scientific method to measure the human-like quality of speech. The matter is made even more complicated when we have humans that talk like bots.

The loss we report is the approximate softmax loss, and it means absolutely nothing in term of conversations. For example, if you convert every token to <unk> and always construct response as a series of <unk> tokens, then your loss would be 0.

So we'll try something fun with this assignment. For the last day of class, Friday March 16, we will have a demonstration of chatbots. Each person/team will have 5 minutes to talk about their work and demo their chatbots. The rest of the class can try play with their chatbots. The class will vote for their favorite chatbot!

Tips

1. Know thy data

You should know your dataset very well so that you can do the suitable data preprocessing and to see the characteristics you can expect from this dataset.

2. Adjust the learning rate

You should pay attention to the reported loss and adjust the learning rate accordingly. Please read the CS231N note on how to read your learning rate.

Keep in mind that each bucket has its own optimizer, so you can have different learning rates for different buckets. For example, buckets with a larger size might need a slightly larger learning rate.

You should feel free to experiment with other optimizers other than SGD.

3. Let your friends try the bot

You can learn a lot about how humans interact with bots when you let your friends try your bot, and you can use that information to make your bot more human-like.

4. Don't be afraid of handcrafted rules

Sometimes, you'll have to resort to handcrafted rules. For example, if the generated response is just empty, then instead of having the bot saying nothing, you can say something like: "I don't know what to say." or "I don't understand what you just said." or "Tell me about something else." This will make the conversation flows a lot more naturally.

5. Have fun!

This assignment is supposed to be fun. Don't get disheartened if your bot seems to just talk gibberish -- even famous bots made by companies with vast resources like Apple or Google give nonsensical responses most of the time.

It'll take a long time to train. For a batch of 64, it takes 1.2 - 2.2s/step on a GPU, and on a CPU it's about 4x slower with 3.8 - 7.5s/step. On a GPU, it'd take an hour to train an epoch for a train set of 100,000 samples, and you'd need to train for at least 3-4 epochs before your bot starts to make sense. Plan your time accordingly.

Deliverables

You need to submit the following:
1. Your code and instructions on how to run it
2. Demo at the class on Friday (March 16). It's going to be super fun!
For option 1:
3. Specify what dataset you use
4. output_convo.txt file
5. Detailed description of what improvement you did for the bot
Zip everything and send it to cs20-win1718-staff-owner@lists.stanford.edu

Option 2: Word embedding transformation + Language model

This option consists of two parts. We provide the data but there won't be any starter code. You get the implement everything from the scratch!

Part 1: Word embedding transformation from Spanish to English

Mikolov et al. proposed to learn an efficient linear mapping between distributed representations of words in different vector spaces. Given m pairs of words in the source language and target languages, $(x_1, y_1), \ldots, (x_m, y_m)$, and their representation in corresponding vector space S and T: $(S_{x_1}, T_{y_1}), \ldots, (S_{x_m}, T_{y_m})$, we find linear transformation from S to T such that:

$$\hat{\pi}(S_{x_i}) = W x_i + b$$

and

$$\hat{\pi} = \text{argmin}_{\pi} \sum_{i=1}^{m} ||\pi(S_{x_i}) - T_{y_m}||^2$$

Mikolov et al. suggested that you can do it with a linear transformation, e.g., we can build a model with only one fully connected layer. Please read the paper for more information.

We first need word embeddings in Spanish and English. This can be done using any of the word embedding techniques such as word2vec (Mikolov et al., 2013), GloVe (Pennington et al., 2014}, CoVe (Mccann et al., 2017). I suggest that you use the pre-trained word vectors fastText by

Bojanowski et al., 2016. fastText learns word representations while taking into account subword information. In this model, words are represented by a sum of its character n-grams. You can download the word embeddings for English and Spanish from their GitHub repo.

For training, you are given 2124 pairs of Spanish, English words.

For evaluation, you are given 218 pairs of Spanish, English words. For each Spanish word, transform its embedding into English vector space, and search for the English words closest to the transformed Spanish word embedding among the 341,696 most common English words. You use the ground truth words to evaluate the top 1 and top 5 accuracy.

a. Report the top 1 and top 5 accuracy on the entire eval vocabulary.
b. Report top 1 and top 5 closest English words to the transformed word vectors of the following Spanish words: regresar, cabra, parecer, otras, encantado, lengua, mike, hables, poder. What interesting things do you see?

Note that the fastText files you download have a lot more word embeddings than you need and might not fit in the memory. I'd recommend that you skim the files to only word embeddings that you need.

In the folder **word_transform**, you can find the following files:
train.vocab: list of 2124 pairs of Spanish, English words.
eval.vocab: list of 234 pairs of Spanish, English words.
common.en.vocab: list of 341,696 most common English words.

Part 2: Trump bot

We've trained a character-level language model on Trump tweets and even though it occasionally produces some interesting results, most of the output tweets aren't. In this exercise, we will explore Trump word embedding space and train a word-level language model on Trump tweets.

In the folder **trump_bot**, you will find the file trump_tweets.txt, which contains the same 19,469 Trump tweets that we used for the character-level models. You will need to do your own data processing and building your own vocabulary. Part 2 consists of two tasks:

a. Train a word embedding model on Trump's vocabulary. Report his vocabulary size (based on his tweets), his top 50 most frequent words. Project his word embeddings on a 3-D space using t-SNE on TensorBoard. Report 5 interesting word relations you find.

b. Train a word-language model on Trump tweets. Feel free to reduce the vocabulary size as you deem fit to speed up the training. Report 10 generated tweets by your model. Does it seem to be working? Why or why not? Note that if you train on your CPU, it might take up to 10 hours to train.

In the folder **trump_bot**, you can find the following files:
trump_tweets.txt: list of 19,469 presidential tweets.

Deliverables

You need to submit the following:
1. Your code and instructions on how to run it
2. Demo at the class on Friday (March 16). It's going to be super fun!
3. Top 1 and top 5 accuracy on the entire eval vocabulary
4. Top 1 and top 5 closest English words to regresar, cabra, parecer, otras, encantado, lengua, mike, hables, poder
5. Trump's vocabulary size
6. Trump's top 50 most frequent words
7. 5 interesting word relations based on Trump's word embedding space
8. 10 tweets generated by your language model
9. Explain why your model seems to work or why it doesn't.

Zip everything and send it to cs20-win1718-staff-owner@lists.stanford.edu