README.md

Grammar Correction

The aim of this project is to build a Sequence-to-Sequence model which, upon taking a grammatically incorrect sentence, returns the grammatically correct sentence as output.

Dataset used

The dataset that I used for this project is the Cornell Movie Dialogs Corpus, which can be found at this link: http://www.cs.cornell.edu/~cristian/Cornell_Movie-Dialogs_Corpus.html. It contains 220,579 conversational exchanges between 10,292 pairs of movie characters, and for the most part, they are grammatically correct exchanges. For this project, I was not so much interested in the speakers themselves or the exchange of conversations, but instead more in actual dialogs. So I just pulled the dialogs from this dataset and used them for training the model. The model is trained on pairs of dialogs, where in each pair, the first sentence is the incorrect version, and the second sentence is the correct version.

The basic idea for generating the dataset works like this:

- For each type of grammatical error, decide upon a fixed number of sentences to use for that error
- Sample this fixed number of sentences from the dataset for each error type
- Apply perturbation(s) upon this sentence as per the error type
- Set this (now incorrect) sentence as the input sequence, and set the original (correct) sequence as the output.

In addition, I also selected a certain number of sentences from the dataset and used these sentences as both the input and output sequence in the pair. This was done to feed a set of correct sequence -> correct sequence mappings to the model, so that at evaluation time, it can learn to recognize if a given sequence is already correct and therefore it doesn't need to apply any 'corrections' to it.

Types of grammatical errors

For this project, I experimented with the following types of errors:

Type of error	Percentage of error	Amount of sentences
Removal of articles (a, an, the)	0.15	20,838
Removal of second part of verb contractions	0.15	20,838
Inversion of singular nouns to plural and vice versa	0.15	20,838
Correct sentences	0.1	13,892

Generating the sequence mappings

To generate the sequence mappings to constitute as training and evaluation data for the model, I had to artificially generate the incorrect sequences (their types and amounts are mentioned above). These mapped sequences are then written to a pickle file called `mapped_seqs.pkl`, which is loaded in the main script for training and evaluating the model. This all happens in the `prepare_dataset.py` script.

Removal of articles and verb contractions

For removal of articles and removal of the second part of verb contractions, I used regular expressions to identify the sentences where these perturbations can be applied. For both error types, I only applied this perturbation once per sentence, but where this gets applied in the sentence is decided randomly. So for example, if a sentence contains three articles at different locations, I selected which one to remove at random, instead of always removing the first one, for instance. Additionally, I applied both of these perturbations to a sentence simultaneously if applicable; so for example, "I've never been to a restaurant as fancy as that!" would be changed to "I never been to restaurant as fancy as that!" This was done so that the model can learn to apply multiple corrections to the same sentence if required.

Inversion of singular and plural nouns

Converting a randomly selected noun to either its singular or plural counterpart was slightly more complicated - for each sentence, I extracted its Part-Of-Speech tags by using a pre-trained English model from Spacy. This would be followed by a check to see if the sentence contains any singular and/or plural nouns (by looking for the "NN" and "NNS" tags) - if so, any one of them would be selected randomly and converted to its singular/plural form. For the conversion from singular to plural and vice versa, I used the `singularize` and `pluralize` functions from a handly Python module called `inflection`.

Correct sequences

For the correct sequence pairs, I just pick a sentence and use it as both the input and output sequence without applying any perturbations.

Examples of sequence pairs

Incorrect sequence	Correct sequence
Shit. It my Galiano.	Shit. It's my Galiano.
Spock, we on leave. You can call me Jim.	Spock, we're on leave. You can call me Jim.
They're trying to make me spend the summer here. I leaving in morning.	They're trying to make me spend the summer here. I'm leaving in the morning.
My sister's coming by to pick me up for brunches. Why don't you come, too?	My sister's coming by to pick me up for brunch. Why don't you come, too?
That my license and registration. I wanna be in compliance.	That's my license and registration. I wanna be in compliance.
I have work to do at plant.	I have work to do at the plant.
I decided I didn't want drink...I beginning to wonder.	I decided I didn't want a drink...I'm beginning to wonder.
You need a glasses?	You need a glass?
Put your fucking hand up! Don't move.	Put your fucking hands up! Don't move.
--he not my husband!	--he's not my husband!

Designing the Network

Since this problem is essentially a sequence-to-sequence mapping task, that is, the input is a sequence of variable length and the output is also a sequence of variable length, the type of model I'm using for this task is an ecoder-decoder model with attention. Since the output will be very similar to the input in most cases (if not exactly the same as in the case of correct sequences as input), I think the attention mechanism will play an important role in helping the model distinguish which parts of the sentence need correction, and what kind of correction should be applied. The model is described in more detail here.

Embeddings

The embedding layer is defined separately, outside of any network. This is done so that the same embedding layer can be used for both the encoder and decoder. This will ensure that both networks get the same representations of all words in the vocabulary. I experimented with using Glove's 100-dimensional pre-trained embeddings later. It contains embeddings for 40,000 words, along with an embedding called `<unk>` for unknown words. I initialized three random vectors each of 100 dimensions for the starting, ending and padding tokens as well (I know random vectors aren't the best way to do this, but I wanted to have a separate vector for each of these that's not the `<unk>` vector). The results of using these embeddings are documented later in the "Results" section.

Encoder

The encoder network is basically a bidirectional GRU with 2 layers. The input sequence batch is passed through the embedding layer to get vector representations of all the words in each sentence, and then this batch is packed using the `pack_padded_sequence` function, and fed into the GRU layer. THe output is the unpacked through `pad_packed_sequence`, and the bidirectional outputs are then combined through addition. Both the output and the hidden state from the GRU are returned as part of the forward pass of the GRU.

Attention

The attention network comes into play after the input sequence has gone through the encoder and we have the encoder output and the encoder hidden state, and when we are passing the output sequence through the decoder. The output from the decoder is computed step by step, and at each step, the dot product of the output from the GRU layer of the decoder (explained in more detail below) and the output from the encoder is computed, which is referred to as attention energies. This is followed by a softmax, which basically helps the model understand which parts of the input sequence to focus on for the output currently being generated by the decoder, as per my understanding.

Decoder

The decoder is perhaps the most complicated network in this configuration. It consists of a unidirectional GRU with 2 layers, two linear layers, and the attention network (since the attention model is actually initialized inside the decoder). The sequence is fed to the decoder in a word-by-word fashion. At each timestep, the hidden state from the previous timestep and the current input (which is either the target word, or the output from the previous timestep based on whether we're using teacher forcing or not) are fed into the model. Please note that the initial hidden state to the decoder for each sequence is the final hidden state from the encoder. The input is passed through the embedding layer, and then this input embedding and the hidden state are fed into the GRU. The hidden state returned from this GRU becomes the hidden state to be fed into the decoder for the next timestep. The output from the GRU is passed through the attention model along with the encoder output, which returns the attention weights for this timestep. Doing a batch matrix multiplication between these attention weights and the encoder output gives us the context vector, which is then used for generating the decoder output as follows: The output from the decoder GRU and the context vector are concatenated and then fed through a linear layer in the decoder, which reduces the size of this vector from `decoder_hidden_size * 2` to `decoder_hidden_size`. This is followed by a tanh activation function, then the second linear layer which reduces the output to the `output_size` (which is basically the vocabulary size), and finally a log softmax to get probabilities.

Loss function and optimizer

The loss function used here is the NLLLoss (when initializing this loss function, the `ignore_index` is set to the padding token, as we don't want to calculate the loss for padded values in the sequence). Two optimizers are used - one for the encoder and one for the decoder, and both of them are Adam optimizers. Their learning rates are specified in the config file.

Training the model

Preparing the data

The data preparation process is fairly standard by this point. A vocabulary class is used to get all words in the sequences, and create numeric indices for each word. Additionally, there are three other words in the vocabulary: the starting token, the ending token, and the padding token. For each sentence pair, the ending token is added to the end of each input sentence. This is so that the model can learn where a sentence ends. Similarly when generating the output from the decoder, the first output to the decoder is the starting token. The numeric representations of all sentences are generated using the vocabulary's word-to-index mapping, and then these sequences are padded to the maximum sentence length. Additionally, the maximum sentence length for the output sequences is noted, as this is later used when generating the output from the decoder. For batching, I did not use a dataloader this time; I used a simple for loop with a step size of the batch size to generate the batches. Because of this, the data is manually shuffled before each epoch.

Training process

Here is a step-by-step outline of the training process:

1. For each sentence, each word is replaced by its numeric index through the vocabulary
2. The input sequences and output sequences are both passed through the embedding layer
3. For the defined number of epochs, iterate through the dataset through a step-size of batches. Before each epoch, the input and output sequences are shuffled, and then for each batch, the input sequences are passed through the encoder to get the encoder hidden states and the encoder outputs.
4. The outputs to the decoder are passed in a word-by-word fashion. The first input for each sequence in the batch is the starting token, and the first hidden state for the each sequence in the batch is the final hidden state from the encoder. Attention weights are computed using the encoder's outputs and the output from the GRU layer of the decoder. These are then used to compute the context vector, which is followed by a batch matrix multiplication with the encoder outputs, a concatentation with the decoder GRU output, a pass through the first linear layer of the decoder and a tanh activation function, and finally the second linear layer followed by a log softmax to get the output.
5. The input for all the remaining timesteps (the amount of which is equal to the length of the maximum sequence in the output sequences) is either the actual target word, or the decoder's output (depending on whether teacher forcing is being used or not).
6. The hidden input for all the remaining timesteps is the hidden state returned from the decoder in the previous timestep.

Teacher forcing

Teacher forcing is a method used for quickly and efficiently training neural networks. The concept of teacher forcing in our sequence-to-sequence model is that during the time of training, the input fed to the decoder at each timestep is the actual target value at that timestep. This is done to help the model converge faster, and be more stable.

In contrast, the other approach is to feed the model's output from the previous timestep as input to the model at the current timestep. This is more representative of a real-world scenario, where the ground truth is not available to be fed into the decoder as input, and might help prevent the model from overfitting. However, when using this approach, the model suffers from slow convergence, and comparatively worse predictions.

I trained the decoder using both approaches - with teacher forcing all the way, and then again with no teacher forcing. All other model configurations were kept the same. The results can be found under the "Results" section.

The evaluation metric being used for this task is the BLEU score (covered in greater detail later). While the BLEU score for both versions of the models - with and without teacher forcing - was almost the same, the predictions from the model trained with no teacher forcing were noticeably worse. For instance, one issue it seem to ran into quite often is repeating the same word twice in the predicted sequence:

Input text: what are those things ?

Response: what are those those things ?

This issue was barely encountered in the model trained with teacher forcing. Moreover, while both models sometimes faced the issue of not being able to replicate the input sequence in the output (some words would be changed entirely; this usually happened with the rarer words or contexts), the model trained without teacher forcing had noticeably more trouble with this than its counterpart.

I think a good way around the decision of whether to use teacher forcing or not is to use both, with a fixed decided ratio of teacher forcing. Therefore, in each batch, it is decided randomly (with a bias) whether teacher forcing will be used or not to train the decoder in this iteration, for this batch. The teacher forcing ratio is currently kept at 0.8; that is, a random number between 0 and 1 is generated, and if this number is less than the teacher forcing ratio, then teacher forcing ratio is used in this iteration, otherwise no teacher forcing is used here.

Running the scripts

The script for training the model and testing it on the evaluation data is `main.py`. There are a lot of configurable parameters for the model, so instead of having to specify a long list of them in the command line parameters, you can specify all these parameters in the `config.py` file. It contains everything, from file paths to model parameters to number of epochs, teacher forcing ratio etc.

Please note that the repo already contains some trained models and a vocabulary, these were generated using 100% teacher forcing, not using pre-trained embeddings, and trained on 250 epochs. If you would like to generate a model using some other configuration, please specify as such in the `config.py` file, and then run the `main.py` script.

Evaluating the model

BLEU score

Since this is not a straightforward classification or regression problem, it is a little hard to come up with an evaluation metric for this problem. The BLEU (BiLingual Evaluation Understudy) score is a modified version of the precision score that is normally used as a metric for translation problems, but it is adapted for other seq2seq problems too. The BLEU approach is based on matching ngrams, and generates a score between 0 and 1, where 0 is the lowest score and 1 is the highest.

For the same configurations, this network trained with and without teacher forcing gives more or less the same BLEU score of between 0.65-0.70. This, along with other reasons, makes me believe that the BLEU score is perhaps not such a good metric for a grammar correction model. The ideal case would have been to have a test set of grammatically incorrect sentences to test the model on after training, and see how many of them it can correct - however, in some cases, there are multiple ways to correct a sentence grammatically - for instance:

Input text: Those are nice apple.

Correction 1: Those are nice apples.

Correction 2: That is a nice apple.

Moreover, the kind of errors that this model has been trained to correct do not always necessarily invalidate a sentence from a grammatical standpoint. For example, removing the article from the following sentence doesn't always make it incorrect, even if it does change the intended meaning:

Input text: It's the key.

Altered sentence (with article removed): It's key.

Same for removing verb contractions:

Input text: They've died.

Altered text (with verb cont. removed): They died.

And same for plural/singular nouns inversion:

Input text: Put your hands in the air!

Altered text (with a plural noun converted to singular): Put your hand in the air!

Therefore, for a given input sequence, if the predicted sequence does not match the target sequence, it does not necessarily mean that the predicted sequence is grammatically incorrect. Because of this, I think that the best way to evaluate this model (given the constraint on time and resources) is through human evaluation. I went through the predictions vs. target sequences for a lot of sentences in the validation data, and I made some interesting observations:

- The model faced (predictably) a lot of issues with rarer words - it would quite often generate a completely different word that it has perhaps seen more in the context so far. Seq2seq models tend to face this issue, I found, and one way to get around this is to eliminate those sentences which contain words whose occurence is below a certain threshold in the entire corpus. However, I didn't do thta, as it would have led to an OOV issue, plus I wanted to observe how often and on what kind of words the model faces this issue.
- The dataset that this model has been trained on consists of conversational dialogs between movie characters, and this has certain consequences. For one, conversations between people do not always strictly follow a grammatical structure - slang words, combining several words, shortened versions of sentences and words, dialogs being cut off, extremely variable lengths of dialogs - all means that the model is not exactly being fed a 100% ground truth of what correct grammar looks like. Perhaps it would have been better to train the model on text that is of a more formal nature, since the sentences in that would perhaps have a higher chance of being grammatically correct.
- Because of the above point, there are two further observations:
    - The model sometimes faces issues with some words that one would normally not consider as a 'rare' word - for example, the word 'incident'. This is a very common word, but the model often has trouble with it. However, it is important to think about the rarity of a word not as a whole, but in the context of conversations. The word 'incident' is more commonly used in reports/blogs/documents as compared to normal, casual conversation, and so the model has more trouble with it than say, some swear word, because the latter is found more commonly in normal conversations.
    - I noticed that the model does a better job with both correcting an incorrect sequence or letting a correct sequence remaining altered with input sequences that are closer to a conversational dialog. For example, it does just fine with correct variations of "Will you go out on a date with me?" since it's a very common piece of conversation (and I expect dialogs of this nature are found easily in this corpus), but has trouble with more formal sequences such as "The car is a very finely tuned machine."

Creating a chat service for human evaluation

To aid in the human evaluation of the model, I have created a chat service which can be launched by running the `chat_service.py` script. It's a very simple console chatbot service loads the trained models (the paths to which are specified in `config.py`), and then asks the user to enter a sequence, and returns the grammatically correct sequence. While experimenting with this service, I realized how true the statement is that researchers often only publish the very few good results in their published works and brush the much-higher-in-number mistakes their models make under the rug. For instance, this model sometimes did a great job:

Example 1

Input sequence: i not loser .

Corrected sequence: i'm not a loser .

Example 2

Input sequence: dozen times day.

Corrected sequence: a dozen times a day .

Example 3

Input sequence: hey men, how did your interviews go ?

Corrected sequence: hey man, how did your interview go ?

NOTE: The input sequence in the last example is not grammatically incorrect, but it's interesting how the model makes two corrections based on the kind of conversations it has seen.

Another interesting example was this:

Input text: i kill ya .

Actual text (correct): i 'd kill ya .

Predicted text: i 'll kill ya .

But while the model does a pretty great job on these sequences, it also does the following:

Example 1

Input sequence: that movie is awesome !

Corrected sequence: that's the movie is mugshots

Example 2

Input sequence: he great poet .

Corrected sequence: he's a great title .

Example 3

Input sequence: exactly. it time to make decisions .

Corrected sequence: exactly. it's time to make pony .

Example 4

Input sequence: what are those buildings

Corrected sequence: what are those the building

Results

Training Loss & Validation Data (for different ratios of teacher forcing)

	With teacher forcing	Without teacher forcing
No. of epochs	150	150
Batch size	300	300
Encoder learning rate	0.0005	0.0005
Decoder learning rate	0.0005	0.0005
Encoder hidden size	300	300
Decoder hidden size	300	300
Training loss	**117**	**1600**
BLEU score (val. data)	**0.69**	**0.70**

Using pre-trained Glove embeddings

Using Glove's pre-trained embeddings did not have a drastic impact on the BLEU score or the quality of the corrections. However, when the model would fail to replicate a word that it should have, the replaced word would quite sometimes make a lot more sense given the context, as opposed to a completely nonsensical word that wouldn't fit in the given context. Here are two examples below:

Example 1

Input text: something wrong . i know it . i 've heard rumors of cholera spreading south from **hamburg** .

Correct text: something 's wrong . i know it . i 've heard rumors of cholera spreading south from **hamburg** .

Predicted text: something wrong . i know it . i 've heard rumors of the garments spreading south from **berlin** .

Example 2

Input text: i can only write on a **manuals** .

Correct text: i can only write on a **manual** .

Predicted text: i can only write on a **training** .

As can be seen in both cases, the predicted sequence replaced a word altogether that it shouldn't have, but the word it chose instead is an understandable replacement; Hamburg and Berlin are both cities in Germany, and the word training and manual occur together a lot. Of course, this isn't perfect and doesn't always happen, as can be seen in the case of "cholera" being replaced by "garments". Moreover, I sometimes felt that while using the chat service, the model trained with pre-trained vectors would make more mistakes of replacing words altogether that it shouldn't.

I think using pre-trained embeddings could help the model make better predictions, but it would be better if these pre-trained embeddings are trained a little bit during the training of the actual model too.

For using the pre-trained embeddings, there's a dictionary object that I created, which contains a key-value mapping for all 40,0004 words to their vectors. This file was too large to push to the repository, so it is located in the server, at the following path:

```
/home/gusimtmu@GU.GU.SE/GrammarChecker/cornell_movie_dialogs_corpus/glove_vectors_100d.pkl
```

Final thoughts

- The model without pre-trained embeddings seems to perform a bit better, because the embeddings are trained as per the data I assume.
- The model seems to have an easier time dealing with the first two types of grammatical errors - article removal and second part of verb contraction removal. It seems to face more issues with the singular/plural inversion. Here are some examples of simple sentences:

```
Please enter a sentence: the computers
Response: the computer

Please enter a sentence: look at those cars!
Response: look at those car !

Please enter a sentence: look at car
Response: look at the car

Please enter a sentence: what are you looking at?
Response: what are you looking at ?

Please enter a sentence: did you go to office?
Response: did you go to the office ?

Please enter a sentence: do you think they done it?
Response: do you think they 've done it ?

Please enter a sentence: how're you doing?
Response: how 're you doing ?

Please enter a sentence: how you doing?
Response: how 're you doing ?
```

```
Please enter a sentence: look at that bird!
Response: look at that 's a bird
```

- The final loss value I get after training the model on 250 epochs is 28.87. Based on this, I think that this is quite a complex model that would need to be trained on a lot more data, and more number of epochs to show significant improvement in the results. Unfortunately, increasing the batch size or the hidden layer sizes would result in memory issues with the GPU, due to which I remained constricted to this configuration. Also, I had to use the quite slow learning rate as well because otherwise, the loss wouldn't converge.
- Overall, I think this has been a very interesting project to work on. This has probably been the most complicated NLP task by far that I have taken upon, and while I had a very good guideline to follow in the start, it was interesting to experiment with different model parameter configurations, different types of grammatical errors, and evaluating the model to see what types of errors it makes, as well as what it gets right.
- Using pre-trained embeddings did not help as much as I had hoped, but in retrospect I realize that it probably makes sense.
- The model doesn't capture the nuances of language as much as I had hoped, but perhaps that is too much to expect out of a model trained with this configuration. Language isn't easy for a model to understand based on text alone, but I would be interested to see if there are any improvements if the same model was to be trained over a much larger corpus, for a larger number of epochs, with a larger batch size etc.
- After having done most of the work on this project, I came to the conlusion that the Cornell Movie Dailogs was perhaps not the best dataset to train a grammar correction model on. Conversational dialogs are far from being the perfect representation of grammatically correct data, and the model sometimes makes the most basic of mistakes with simple sentences.

Future work

I would definitely like to continue working on this project in my free time, and do a bunch of different experiments on it to try and improve the model's accuracy. Here are some ideas I have in mind that I would definitely like to give a shot:

- Using a different dataset - Perhaps using a different dataset which would contain more consistent and structured text would make it a bit easier for the model to take upon this task of grammar correction. I have corpuses like Yelp Reviews, Project Gutenburg or 20 Newsgroups in mind - while they may have been designed for different tasks in this, the idea is to just load the sentences from these corpuses, introduce the same perturbations to them to generate incorrect sentences, and then train this model over them.
- If we can somehow incorporate POS tags into the training, I think that could be a very promising idea to improve the model's performance, because then we would be feeding it some information about sentence structure, rather than just feeding it text and expecting it to figure that out on its own. To achieve this, I think we could have two different recurrent layers in the encoder model; one for text tokens and the other for POS tokens; and then concatenate their outputs and hidden states. However, I'm a little confused over how the decoder would work with this; do we get predictions for both text tokens and POS tokens? And factor both into the loss? Or do we still only work with text tokens in the decoder?
- Based on a discussion with a friend, I learned that an interesting variant for encoder-decoder models is using a CRNN - a Convolutional Recurrent Neural Network. I have not looked much into it, but I think the idea is that the input text is fed first into a Convolutional layer, which extracts ngrams, and then these ngrams are fed into a recurrent layer. This way, the model learns recurrent information on ngrams. I can't say how useful it would be in the case of grammar correction, where sometimes using ngrams might not be the best idea, but it might pick up some higher level patterns - definitely worth a shot!
- For the singular/plural inversion error, I would like to somehow generate multiple correct versions of the same sentence. For example, "I like this apples" would have correct sequences of both "I like this apple" and "I like those apples". While I'm pretty sure this would end up confusing the model more, it would be interesting to see what other patterns it picks up with multiple corrections for the same incorrect sequence!
- Finally, more types of grammatical errors - specifically, those which are definitely errors, and not up for discussion. For example, spelling errors - although those might be harder to generate. But working with spelling errors could be fascinating, since if we use pre-trained embeddings, it could use some very interesting contextual information to suggest what you might actually have meant.

References

- https://www.youtube.com/watch?v=CNuI8OWsppg

- https://pytorch.org/tutorials/beginner/chatbot_tutorial.html
- https://github.com/atpaino/deep-text-corrector
- https://jeddy92.github.io/JEddy92.github.io/ts_seq2seq_intro/
- https://inflection.readthedocs.io/en/latest/
- https://spacy.io/
- https://pytorch.org/docs/stable/nn.html