UDACITY

‹ Return to "Deep Learning" in the classroom

DISCUSS ON STUDENT HUB

Generate TV Scripts

REVIEW

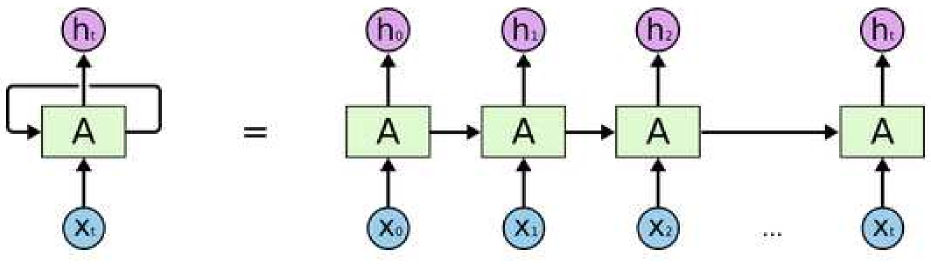
CODE REVIEW

HISTORY

Meets Specifications

Congratulations! You have met all the specifications!
I really enjoyed reviewing your project. It is truly a remarkable sight that your network is able to generate the script as shown. It is amazing indeed.

I'd highly encourage you to go over this classical blog on LSTM http://colah.github.io/posts/2015-08-Understanding-LSTMs/
In reality the LSTM network is implemented as given below at the left side of the diagram i.e. as the rolled network. The right-side of the diagram is just for the people to understand all the math and computational theory (it's like we are "imagining" how the network will look like when it is unrolled).



An unrolled recurrent neural network.

All Required Files and Tests

The project submission contains the project notebook, called “dlnd_tv_script_generation.ipynb”.

All the unit tests in project have passed.

The unit testing is one of the crucial steps to write industrial bug free code. I'd encourage you to learn how these unit testings have been carried in this project if you have not looked into. Here's also an introductory tutorial on unit testing https://www.youtube.com/watch?v=6tNS--WetLI&t=1816s

Pre-processing Data

The function `create_lookup_tables` create two dictionaries:

- **Dictionary to go from the words to an id, we'll call vocab_to_int**
- **Dictionary to go from the id to word, we'll call int_to_vocab**

The function `create_lookup_tables` return these dictionaries as a tuple (vocab_to_int, int_to_vocab).

The `create_lookup_tables` function's return value of `vocab_to_int` can be considered as the bag of words which is used to encode the vocab words into the numerical form. Because the model cannot use the text directly in its matrix computation.

The function `token_lookup` returns a dict that can correctly tokenizes the provided symbols.

All the 10 symbols are taken as key and the tokens of those symbols are taken as values into the dictionary.

Batching Data

The function `batch_data` breaks up word id's into the appropriate sequence lengths, such that only complete sequence lengths are constructed.

Each batch consists samples with the complete sequence lengths 👍

The sequential learning is carried out by sharing the weights within the sequence length hyperparameter in the RNN layer. Check out this https://www.youtube.com/watch?v=H3ciJF2eCJI&list=PLAwxTw4SYaPn_OWPFT9ulXLuQrImzHfOV&index=53

In the function `batch_data`, data is converted into Tensors and formatted with TensorDataset.

Nice work in using the Pytorch's TensorDataset and DataLoader functions. DataLoader is an iterator function. Feel free to construct your own generator function to iterate over the data in batches. You may choose to create a generator function that batches data similarly but returns x and y batches using `yield`. Check out this tutorial on generator functions and it's benefits https://www.youtube.com/watch?v=bD05uGo_sVI

Finally, `batch_data` returns a DataLoader for the batched training data.

```
data_loader = torch.utils.data.DataLoader(data, batch_size=batch_size, shuffle=True)
```

Nice work in setting `shuffle=True` within the DataLoader. The sequential learning takes place within the sequence length by LSTM/RNNs. But the order of the samples within the batch size can be changed/shuffled.

Build the RNN

The RNN class has complete `__init__`, `forward`, and `init_hidden` functions.

All the three methods are included in the RNN Class 👍

LSTMs are the type of the recurrent neural nets. In recurrent neural nets weight parameters are shared between the hidden units. That means information given to the hidden unit at time t is not only coming from input unit but also from the hidden units of previous time stamps. So the input nodes and previous hidden states are concatenated together and then multiplied with weights. As a result, back-propagation through time is carried out as the inputs from the previous timestamps are also considered. The sequential learning by sharing the weights is carried within the sequence length hyperparameter in the RNN layer.

Logits are created from the output layer. Logits means linear combination of weights multiplied by the units (from previous layer) and then bias being added. This is the term used for the output of neural net’s output layer before adding any non-linear activation function.

```
    self.fc = nn.Linear(self.hidden_dim, self.output_size)
```

These logits are used by the non-linear activation function like softmax or sigmoid activation function which will generate the predicted values as below taken from Build the Graph section.

```
# defining loss and optimization functions for training
optimizer = torch.optim.Adam(rnn.parameters(), lr=learning_rate)
criterion = nn.CrossEntropyLoss()
```

Cross Entropy function already includes Log Softmax as stated here
https://pytorch.org/docs/stable/nn.html

The RNN must include an LSTM or GRU and at least one fully-connected layer. The LSTM/GRU should be correctly initialized, where relevant.

`self.embed = nn.Embedding(self.vocab_size, self.embedding_dim)`

Nice work in including the embedding as an input layer to the LSTM network. Word Embedding is a technique for learning dense representation of words in a low dimensional vector space. Each word can be seen as a point in this space, represented by a fixed length vector. Semantic relations between words are captured by this technique.
Reference;
https://indico.io/blog/sequence-modeling-neuralnets-part1

You can also use pre-trained embedding layer. Check out this blog
https://medium.com/@martinpella/how-to-use-pre-trained-word-embeddings-in-pytorch-71ca59249f76

RNN Training

- **Enough epochs to get near a minimum in the training loss, no real upper limit on this. Just need to make sure the training loss is low and not improving much with more training.**
- **Batch size is large enough to train efficiently, but small enough to fit the data in memory. No real “best” value here, depends on GPU memory usually.**
- **Embedding dimension, significantly smaller than the size of the vocabulary, if you choose to use word embeddings**
- **Hidden dimension (number of units in the hidden layers of the RNN) is large enough to fit the data well. Again, no real “best” value.**
- **n_layers (number of layers in a GRU/LSTM) is between 1-3.**
- **The sequence length (seq_length) here should be about the size of the length of sentences you want to look at before you generate the next word.**

- **The learning rate shouldn't be too large because the training algorithm won't converge. But needs to be large enough that training doesn't take forever.**

Batch size is also appropriate. If you have too large batch size then that means there will be fewer weight updates in each epoch. Therefore you have to increase the number of epochs so that the model converges.
If you use higher batch size and do not increase the number of epochs, accuracy level might seem lower and that is because the model has not converged (model still has the capacity to learn more).

Check out this lesson video on mini-batch training and gradient descent

https://www.youtube.com/watch?v=hMLUgM6kTp8&index=20&list=PLAwxTw4SYaPn_OWPFT9ulXLuQrImzHfOV

Here are some of the Udacity lesson videos that can help you develop the intuition behind tuning the value of hyper-parameters.

This lesson video discusses the intuition behind tuning the value of batch size
https://www.youtube.com/watch?v=GrrO1NFxaW8

This lesson video discusses the intuition behind tuning the value of learning rate
https://www.youtube.com/watch?v=HLMjeDez7ps

The printed loss should decrease during training. The loss should reach a value lower than 3.5.

The model has achieved the required loss value of less than 3.5 👍

There is a provided answer that justifies choices about model size, sequence length, and other parameters.

I tried different values for sequence_length between 5 and 10 and chose to use 8 in the end.

The sequence length (seq_length) here should be about the size of the length of sentences you want to look at before you generate the next word.

I found that I could not go below a loss of 3.5 with a separate dropout layer, so I removed it. In real applications, one would include a dropout layer for the model to generalize better, accepting a slightly bigger loss.

Dropout is used to prevent overfitting. Overfitting is when the model performs well on to the training set but on the unseen/validation set. The downside of the dropout is that it increases the training time. So you will see the slower convergence.

One of the drawbacks of dropout is that it increases training time. A dropout network typically takes 2-3 times longer to train than a standard neural network of the same architecture. A major cause of this increase is that the parameter updates are very noisy.

Reference http://jmlr.org/papers/volume15/srivastava14a/srivastava14a.pdf

Generate TV Script

The generated script can vary in length, and should look structurally similar to the TV script in the dataset.

It doesn't have to be grammatically correct or make sense.

The network has captured the structure of the TV script in which the most common pattern is that each character speech is in new line which is followed by the network's generated script output;

```
elaine: well, i guess i could just be able to see her.


kramer: well, it's just a little thing.


jerry: oh! i don't know!


jerry:(pleading) oh my god!
```

DOWNLOAD PROJECT

RETURN TO PATH

Rate this review