UDACITY

‹ Return to Classroom

DISCUSS ON STUDENT HUB ›

Generate TV Scripts

REVIEW

CODE REVIEW

HISTORY

Meets Specifications

I am really impressed with the amount of effort you've put into the project. You deserve applaud for your hardwork!

🎉 Finally, Congratulations on completing this project. You are one step closer to finishing your Nanodegree.

Wishing you good luck for all future projects 🎉

Some general suggestions -

Use of assertions and Logging:

- Consider using Python assertions for sanity testing - assertions are great for catching bugs. This is especially true of a dynamically type-checked language like Python where a wrong variable type or shape can cause errors at runtime
- Logging is important for long-running applications. Logging done right produces a report that can be analyzed to debug errors and find crucial information. There could be different levels of logging or logging tags that can be used to filter messages most relevant to someone. Messages can be written to the terminal using print() or saved to file, for example using the Logger module. Sometimes it's worthwhile to catch and log exceptions during a long-running operation so that the operation itself is not aborted.

Reproducibility:

- Reproducibility is perhaps the biggest issue in machine learning right now. With so many moving parts present in the code (data, hyperparameters, etc) it is imperative that the instructions and code make it easy for anyone to get exactly the same results (just imagine debugging an ML pipeline where the data changes every time and so you cannot get the same result twice).
- Also consider using random seeds to make your data more reproducible.

Debugging:

- Check out this guide on debugging in python

Optimization and Profiling:

- Monitoring progress and debugging with Tensorboard: This tool can log detailed information about the model, data, hyperparameters, and more. Tensorboard can be used with Pytorch as well.
- Profiling with Pytorch: Pytorch's profiler can be used to break down profiling information by operations (convolution, pooling, batch norm) and identify performance bottlenecks. The performance traces can be viewed in the browser itself. The profiler is a great tool for quickly comparing GPU vs CPU speedups for example.

All Required Files and Tests

✓

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

Jupyter notebook is included in the submission zip. ✅

Suggestion:
You can export your conda environment into `environment.yaml` file so that you can recreate your conda environment later while practicing on your own system. Use the following command -

```
conda env export -f environment.yaml
```

✓

All the unit tests in project have passed.

The implementation passes all the unit tests laid out throughout the project notebook ✅

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).

`create_lookup_tables` generates a vocabulary from the text input, creates the `vocab_to_int` and reverse `int_to_vocab` dictionary and returns them as a tuple.

Suggestion:

While Python's `Counter` module is a convenient here, it has some extra overhead we don't need.
So you can use a `set` instead: `set(text)`
The sorting operation is also unnecessary and you only need to enumerate once, if you create both dicts inside a single for loop. All of these things will save some compute power/time.

```
vocab = set(text)

int_to_vocab = {}
vocab_to_int = {}

for idx, word in enumerate(vocab):
    int_to_vocab[idx] = word
    vocab_to_int[word] = idx

return (vocab_to_int, int_to_vocab)
```

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

The function `token_lookup` creates a dictionary that maps symbols/punctuations into unique tokens and returns this dictionary.

Why do we need to preprocess the input data before passing it into a Neural network?

Text data is represented on computers using an encoding scheme such as ASCII or UNICODE, that maps every character to a number. Computers store and transmit these values as binary. So a string such as "UDACITY" is internally stored just as an array of binary values. The neural network won't be able to extract any meaningful information either from the binary values or from the encoding scheme values.

This is why pre-processing is extremely important. During the pre-processing phase we might remove source specific markers (such as HTML tags from website data), punctuations, stopwords, etc.

While some preprocessing steps are language agnostic, others are heavily dependent on the language we are working with.

e.g. Languages like French have punctuations as part of the words. As such we need to carefully evaluate our data before we perform pre-processing.

Batching Data

✓

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

`batch_data` breaks up word id's into the appropriate sequence lengths.

You could use `append` operations on lists to generate the target and feature tensors and then convert the final arrays to numpy arrays. Append on numpy arrays is really slow and therefore anytime we want to build up our data, it's faster to build the entire list first and then convert to numpy.

✓

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

Implementation loads the sequenced data into Tensors and then uses PyTorch's `TensorDataset` utility to generate the dataset.

```
data = TensorDataset(feature_tensors, target_tensors)
```

✓

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

Function returns a Dataloader as expected.
DataLoader is very a useful PyTorch module that makes loading data a breeze. It also supports unique features like automatic batching.

You can check out more features of [Dataloader here]

Build the RNN

✓

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

The RNN class has been defined appropriately and `__init__`, `forward`, and `init_hidden` functions from the base class `nn.Module` have been overriden in the RNN class description.

The advantage of using a Deep Learning library such as PyTorch is that you only need to define the `forward` function and the `backward` function (i.e. Backpropagation step) is defined automatically by the built-in autograd module. Things would be quite hard if we had to worry about all those gradients and calculus and implement chain rule manually!

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

```
nn.LSTM(embedding_dim, hidden_dim, n_layers, dropout=dropout, batch_first=True)
```

RNN implements an `LSTM` Layer, and initializes it appropriately.

Suggested Reading: Visual guide on LSTMs by Chris Olah: https://colah.github.io/posts/2015-08-Understanding-LSTMs/

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.

```
sequence_length = 10
batch_size = 128
```

```
num_epochs = 50
# Learning Rate
learning_rate = 0.001

vocab_size = len(vocab_to_int)+len(token_dict)+1
output_size = vocab_size
embedding_dim = 900
hidden_dim = 512
n_layers = 3
```

Sensible hyperparameters have been selected for the RNN model.

When it comes to hyperparameters, there is no universal answer to what works well. Therefore, it's best to experiment with a range of different values to check which hyperparameters result in the best model.

Check out this wonderful guide on HyperParameter Optimization for Deep Neural Networks

✓

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

Model loss drops throughout the training phase and finally reaches `1.88` which is below the required threshold for passing this requirement.

Ever wanted to peek behind the complex and confusing mathematical equation of cross-entropy-loss?

These two guides explain the fundamentals of cross-entropy-loss beautifully:

- Visual Explanation of Binary Cross-Entropy Loss
- Introduction to Cross-Entropy Loss

✓

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

You've provided succinct and lucid explanation on your choice of various hyperparameters. Good job! 👍

In **generation** tasks such as these, there's no concrete way to perform cross validation unlike other standard learning tasks. Therefore, we have to rely on our intuition to make sure the network's output makes sense and it is not merely generating nonsense. And since we cannot perform cross validation, the task of choosing appropriate hyperparameters is tougher and here too we need to decide based on our own judgement and intuition.

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.

Output script looks structurally similar to the Dataset Script.
However, you can clearly see that the Neural Networks still cannot make sense of grammar or semantics like humans can. They don't really have any intuition about what words really mean.

DOWNLOAD PROJECT

RETURN TO PATH