Return to Classroom

Generate TV Scripts

REVIEW

CODE REVIEW

HISTORY

▼ problem_unittests.py

```
1  from unittest.mock import MagicMock, patch
2  import numpy as np
3  import torch
4
5
6  class _TestNN(torch.nn.Module):
7      def __init__(self, input_size, output_size):
8          super(_TestNN, self).__init__()
9          self.decoder = torch.nn.Linear(input_size, output_size)
10         self.forward_called = False
11
12     def forward(self, nn_input, hidden):
13         self.forward_called = True
14         output = self.decoder(nn_input)
15
16         return output, hidden
17
18
19 def _print_success_message():
20     print('Tests Passed')
21
22
23 class AssertTest(object):
24     def __init__(self, params):
25         self.assert_param_message = '\n'.join([str(k) + ': ' + str(v) + '' for k, v i
26
       def test(self, assert_condition, assert_message):
```

```
        assert assert_condition, assert_message + '\n\nUnit Test Function Parameters\


def test_create_lookup_tables(create_lookup_tables):
    test_text = '''
        Moe_Szyslak Moe's Tavern Where the elite meet to drink
        Bart_Simpson Eh yeah hello is Mike there Last name Rotch
        Moe_Szyslak Hold on I'll check Mike Rotch Mike Rotch Hey has anybody seen Mike
        Moe_Szyslak Listen you little puke One of these days I'm gonna catch you and I
        Moe_Szyslak Whats the matter Homer You're not your normal effervescent self
        Homer_Simpson I got my problems Moe Give me another one
        Moe_Szyslak Homer hey you should not drink to forget your problems
        Barney_Gumble Yeah you should only drink to enhance your social skills'''

    test_text = test_text.lower()
    test_text = test_text.split()

    vocab_to_int, int_to_vocab = create_lookup_tables(test_text)

    # Check types
    assert isinstance(vocab_to_int, dict),\
        'vocab_to_int is not a dictionary.'
    assert isinstance(int_to_vocab, dict),\
        'int_to_vocab is not a dictionary.'

    # Compare lengths of dicts
    assert len(vocab_to_int) == len(int_to_vocab),\
        'Length of vocab_to_int and int_to_vocab don\'t match. ' \
        'vocab_to_int is length {}. int_to_vocab is length {}'.format(len(vocab_to_int

    # Make sure the dicts have the same words
    vocab_to_int_word_set = set(vocab_to_int.keys())
    int_to_vocab_word_set = set(int_to_vocab.values())

    assert not (vocab_to_int_word_set - int_to_vocab_word_set),\
    'vocab_to_int and int_to_vocab don\'t have the same words.' \
        '{} found in vocab_to_int, but not in int_to_vocab'.format(vocab_to_int_word_
    assert not (int_to_vocab_word_set - vocab_to_int_word_set),\
        'vocab_to_int and int_to_vocab don\'t have the same words.' \
        '{} found in int_to_vocab, but not in vocab_to_int'.format(int_to_vocab_word_

    # Make sure the dicts have the same word ids
    vocab_to_int_word_id_set = set(vocab_to_int.values())
    int_to_vocab_word_id_set = set(int_to_vocab.keys())

    assert not (vocab_to_int_word_id_set - int_to_vocab_word_id_set),\
        'vocab_to_int and int_to_vocab don\'t contain the same word ids.' \
        '{} found in vocab_to_int, but not in int_to_vocab'.format(vocab_to_int_word_
    assert not (int_to_vocab_word_id_set - vocab_to_int_word_id_set),\
        'vocab_to_int and int_to_vocab don\'t contain the same word ids.' \
        '{} found in int_to_vocab, but not in vocab_to_int'.format(int_to_vocab_word_

    # Make sure the dicts make the same lookup
    missmatches = [(word, id, id, int_to_vocab[id]) for word, id in vocab_to_int.items

    assert not missmatches,\
        'Found {} missmatche(s). First missmatch: vocab_to_int[{}] = {} and int_to_voc


    assert len(vocab_to_int) > len(set(test_text))/2,\
        'The length of vocab seems too small.  Found a length of {}'.format(len(vocab_
```

```
    _print_success_message()


def test_tokenize(token_lookup):
    symbols = set(['.', ',', '"', ';', '!', '?', '(', ')', '-', '\n'])
    token_dict = token_lookup()

    # Check type
    assert isinstance(token_dict, dict), \
        'Returned type is {}.'.format(type(token_dict))

    # Check symbols
    missing_symbols = symbols - set(token_dict.keys())
    unknown_symbols = set(token_dict.keys()) - symbols

    assert not missing_symbols, \
    'Missing symbols: {}'.format(missing_symbols)
    assert not unknown_symbols, \
        'Unknown symbols: {}'.format(unknown_symbols)

    # Check values type
    bad_value_type = [type(val) for val in token_dict.values() if not isinstance(val,

    assert not bad_value_type,\
        'Found token as {} type.'.format(bad_value_type[0])

    # Check for spaces
    key_has_spaces = [k for k in token_dict.keys() if ' ' in k]
    val_has_spaces = [val for val in token_dict.values() if ' ' in val]

    assert not key_has_spaces,\
        'The key "{}" includes spaces. Remove spaces from keys and values'.format(key_
    assert not val_has_spaces,\
    'The value "{}" includes spaces. Remove spaces from keys and values'.format(val_ha

    # Check for symbols in values
    symbol_val = ()
    for symbol in symbols:
        for val in token_dict.values():
            if symbol in val:
                symbol_val = (symbol, val)

    assert not symbol_val,\
    'Don\'t use a symbol that will be replaced in your tokens. Found the symbol {} in

    _print_success_message()


def test_rnn(RNN, train_on_gpu):
    batch_size = 50
    sequence_length = 3
    vocab_size = 20
    output_size=20
    embedding_dim=15
    hidden_dim = 10
    n_layers = 2

    # create test RNN
    # params: (vocab_size, output_size, embedding_dim, hidden_dim, n_layers)
    rnn = RNN(vocab_size, output_size, embedding_dim, hidden_dim, n_layers)
```

```
150
151     # create test input
152     a = np.random.randint(vocab_size, size=(batch_size, sequence_length))
153     #b = torch.LongTensor(a)
154     b = torch.from_numpy(a)
155     hidden = rnn.init_hidden(batch_size)
156
157
158     if(train_on_gpu):
159         rnn.cuda()
160         b = b.cuda()
161
162     output, hidden_out = rnn(b, hidden)
163
164     assert_test = AssertTest({
165                               'Input Size': vocab_size,
166                               'Output Size': output_size,
167                               'Hidden Dim': hidden_dim,
168                               'N Layers': n_layers,
169                               'Batch Size': batch_size,
170                               'Sequence Length': sequence_length,
171                               'Input': b})
172
173     # initialization
174     correct_hidden_size = (n_layers, batch_size, hidden_dim)
175     assert_condition = hidden[0].size() == correct_hidden_size
176     assert_message = 'Wrong hidden state size. Expected type {}. Got type {}'.format(
177     assert_test.test(assert_condition, assert_message)
178
179     # output of rnn
180     correct_hidden_size = (n_layers, batch_size, hidden_dim)
181     assert_condition = hidden_out[0].size() == correct_hidden_size
182     assert_message = 'Wrong hidden state size. Expected type {}. Got type {}'.format(
183     assert_test.test(assert_condition, assert_message)
184
185     correct_output_size = (batch_size, output_size)
186     assert_condition = output.size() == correct_output_size
187     assert_message = 'Wrong output size. Expected type {}. Got type {}'.format(correc
188     assert_test.test(assert_condition, assert_message)
189
190     _print_success_message()
191
192
193 def test_forward_back_prop(RNN, forward_back_prop, train_on_gpu):
194     batch_size = 200
195     input_size = 20
196     output_size = 10
197     sequence_length = 3
198     embedding_dim=15
199     hidden_dim = 10
200     n_layers = 2
201     learning_rate = 0.01
202
203     # create test RNN
204     rnn = RNN(input_size, output_size, embedding_dim, hidden_dim, n_layers)
205
206     mock_decoder = MagicMock(wraps=_TestNN(input_size, output_size))
207     if train_on_gpu:
208         mock_decoder.cuda()
209
210     mock_decoder_optimizer = MagicMock(wraps=torch.optim.Adam(mock_decoder.parameters
```

```
    mock_criterion = MagicMock(wraps=torch.nn.CrossEntropyLoss())

    with patch.object(torch.autograd, 'backward', wraps=torch.autograd.backward) as m
        inp = torch.FloatTensor(np.random.rand(batch_size, input_size))
        target = torch.LongTensor(np.random.randint(output_size, size=batch_size))

        hidden = rnn.init_hidden(batch_size)

        loss, hidden_out = forward_back_prop(mock_decoder, mock_decoder_optimizer, mo

    assert (hidden_out[0][0]==hidden[0][0]).sum()==batch_size*hidden_dim, 'Returned h

    assert mock_decoder.zero_grad.called or mock_decoder_optimizer.zero_grad.called,
    assert mock_decoder.forward_called, 'Forward propagation not called.'
    assert mock_autograd_backward.called, 'Backward propagation not called'
    assert mock_decoder_optimizer.step.called, 'Optimization step not performed'
    assert type(loss) == float, 'Wrong return type. Expected {}, got {}'.format(float

    _print_success_message()
```

▸ helper.py

RETURN TO PATH