RTML-Final-2021

May 7, 2021

1 RTML Final 2021

In this exam, we'll have some practical exercises using RNNs and some short answer questions regarding the Transformer/attention and reinforcement learning.

Consider the AGNews text classification dataset:

[60]:
```
from torchtext.datasets import AG_NEWS
from torchtext.data.utils import get_tokenizer
from collections import Counter
from torchtext.vocab import Vocab
import pandas as pd
import string
import re

#train_iter = pd.read_csv("./data/.data/train.csv")
#train_iter['summary'] = train_iter['Title'] + ' ' + train_iter['Description']
train_iter = AG_NEWS(root='.data',split='train')
tokenizer = get_tokenizer('basic_english')
counter = Counter()


def clean(line):
    line = line.replace('\\', ' ')
    line = re.sub('(\s+)(a|an|and|the)(\s+)', '\1\3', line)
    line = re.sub('[%s]' % re.escape(string.punctuation), '', line)

    return line

labels = {}
for (label, line) in train_iter:
    if label in labels:
        labels[label] += 1
    else:
        labels[label] = 1
    counter.update(tokenizer(clean(line)))

vocab = Vocab(counter, min_freq=0, max_size=1000)
```

```
print('Label frequencies:', labels)
print('A few token frequencies:', vocab.freqs.most_common(5))
print('Label meanings: 1: World news, 2: Sports news, 3: Business news, 4: Sci/
↪Tech news')
```

```
Label frequencies: {3: 30000, 4: 30000, 2: 30000, 1: 30000}
A few token frequencies: [('to', 106167), ('of', 71310), ('in', 64953), ('on',
47273), ('for', 36960)]
Label meanings: 1: World news, 2: Sports news, 3: Business news, 4: Sci/Tech
news
```

Here's how we can get a sequence of tokens for a sentence with the cleaner, tokenizer, and vocabulary:

```
[54]: [vocab[token] for token in tokenizer(clean('Bangkok, or The Big Mango, is one␣
      ↪of the great cities of Asia'))]
```

```
[54]: [3914, 96, 8, 291, 0, 11, 45, 7438, 1990, 3, 1057]
```

Let's make pipelines for processing a news story and a label:

```
[55]: text_pipeline = lambda x: [vocab[token] for token in tokenizer(clean(x))]
      label_pipeline = lambda x: int(x) - 1
```

Here's how to create dataloaders for the training and test datasets:

```
[56]: import torch
      from torch.utils.data import DataLoader
      from torch.nn.utils.rnn import pad_sequence
      device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

      def collate_batch(batch):
          label_list, text_list, length_list = [], [], []
          for (_label, _text) in batch:
              label_list.append(label_pipeline(_label))
              processed_text = torch.tensor(text_pipeline(_text), dtype=torch.int64)
              length_list.append(processed_text.shape[0])
              text_list.append(processed_text)
          label_list = torch.tensor(label_list, dtype=torch.int64)
          text_list = pad_sequence(text_list, padding_value=0)
          length_list = torch.tensor(length_list, dtype=torch.int64)
          return label_list.to(device), text_list.to(device), length_list.to(device)

      train_iter = AG_NEWS(split='train')
      train_dataset = list(train_iter)
      test_iter = AG_NEWS(split='test')
      test_dataset = list(test_iter)
```

```
train_dataloader = DataLoader(train_dataset, batch_size=8, shuffle=True,␣
 ↪collate_fn=collate_batch)
test_dataloader = DataLoader(test_dataset, batch_size=8, shuffle=False,␣
 ↪collate_fn=collate_batch)
```

Here's how to get a batch from one of these dataloaders. The first entry is a 1D tensor of labels for the batch (8 values between 0 and 3), then a 2D tensor representing the stories with dimension T x B (number of tokens x batch size).

[57]:
```
batch = next(enumerate(train_dataloader))
print(batch)
```

```
(0, (tensor([3, 1, 0, 0, 0, 2, 3, 1], device='cuda:0'), tensor([[   491,    136,
1145,  59356,   5454,   3052,   1388,   9830],
        [  3516,    895,    272,   1124,   1435,      2,   1352, 231994],
        [  4962,   2025,    424,   5895,   1637,    251,   3800,    242],
        [ 50357,   2308,     16,  54052,   1083,   9010,    220,      6],
        [   491,     23,  49994,  13942,    846,      6,    457,     22],
        [    16,   5530,     19,     39,     72,  18237,      6,   2812],
        [213605,  41516,     19,  12709,     72,     60,  96872,   9830],
        [    65,   7562,    744,  39289,   1486,    177,     19, 140479],
        [   201,  40739,    424,     25,     56,     17,     19, 227501],
        [  7628,    540,  72416,  29607,   2217,   3052, 165581,    634],
        [ 50357,  39511,  71440,    843,   1435,     84,    363,      4],
        [ 29398,  11110,   1145,     72,   8047,   1913,    162,   1191],
        [   129,  12040,    272,     72,  57174,     24,  18318,   9286],
        [ 93032,   2744,     13,   1528,    532,    251,   1388,   2512],
        [     8,    386,     42,      3,      6,   9010,     53,     54],
        [  1232,     38,  46270,  12709,    122,    222,     11,  86764],
        [    77,      9,    306, 174853,     15,     81,  64921,   1106],
        [    24,     11,    169,  25762,     56,      6,    317,  62271],
        [    29,   3682,   1748,   1399,  12892,  18237,  11578,   1417],
        [   841,    166,     23,  59356,  13762,     60,     77,   5311],
        [179826,     11,   6132,     73,  10927,      4,  15087,   1309],
        [    25,    539,  69627,   6815,   6858,    742,      4,      0],
        [ 14337,      2,     35,   7726,   8903,   1229,    131,      0],
        [ 46362,    207,   1015,  24279,     12,   2211,   7894,      0],
        [244690,    540,   1304,    378,     34,    149,      3,      0],
        [     0,     94,  32492, 192727,    840,  15387, 107799,      0],
        [     0,     92,      3,    106,      2,   4192,  11001,      0],
        [     0,    136,     23,     86,   4128,     78,      6,      0],
        [     0,    895,   1779,   3912,     23,  20429,     67,      0],
        [     0,   1020,      0,     13,   5288,  12909,  96871,      0],
        [     0,   7984,      0,    213,    153,    751,     60,      0],
        [     0,    866,      0, 135719,  88659, 128079,      4,      0],
        [     0, 130247,      0,     26,     21,   3159,  27837,      0],
        [     0,  41516,      0, 225089,    618,   2891,      0,      0],
        [     0,   6464,      0,      4,      0,    996,      0,      0],
        [     0,    801,      0,   1555,      0, 227468,      0,      0],
        [     0,  11110,      0,     18,      0,      0,      0,      0],
        [     0,     18,      0, 253998,      0,      0,      0,      0],
        [     0,  41516,      0,   6391,      0,      0,      0,      0],
        [     0,  14744,      0, 152416,      0,      0,      0,      0],
        [     0,    609,      0,   1077,      0,      0,      0,      0],
        [     0,      0,      0,  10437,      0,      0,      0,      0],
        [     0,      0,      0,    843,      0,      0,      0,      0]],
       device='cuda:0'), tensor([25, 41, 29, 43, 34, 36, 33, 21],
device='cuda:0')))
```

1.1 Question 1, 10 points

The vocabulary currently is too large for a simple one-hot embedding. Let's reduce the vocabulary size so that we can use one-hot. First, add a step that removes tokens from a list of "stop words" to the `text_pipeline` function. You probably want to remove punctuation ('.', ',', '-', etc.) and articles ("a", "the").

Once you've removed stop words, modify the vocabulary to include only the most frequent 1000 tokens (including 0 for an unknown/infrequent word).

Write your revised code in the cell below and output the 999 top words with their frequencies:

[6]:
```python
# Place code for Question 1 here
def clean(line):
    line = line.replace('\\', ' ')
    #Add for question one
    line = re.sub('(\s+)(a|an|and|the)(\s+)', '\1\3', line)
    line = re.sub('[%s]' % re.escape(string.punctuation), '', line)
    return line
```

1.2 Question 2, 30 points

Next, let's build a simple RNN for classification of the AGNews dataset. Use a one-hot embedding of the vocabulary entries and the basic RNN from Lab 10. Use the lengths tensor (the third element in the batch returned by the dataloaders) to determine which output to apply the loss to.

Place your training code below, and plot the training and test accuracy as a function of epoch. Finally, output a confusion matrix for the test set.

Do not spend a lot of time on the training! A few minutes is enough. The point is to show that the model is learning, not to get the best possible performance.

[90]:
```python
# Place code for Question 2 here
import torch
import torch.nn as nn
import torch.optim as optim
num_class = len(set([label for (label, text) in train_iter]))
```

```
vocab_size = len(vocab)
emsize = 64
n_hidden = 128
n_tag = 4

class RNN(nn.Module):
    #Create a constructor
    def __init__(self, vocab_size,input_size,hidden_size, output_size):
        super(RNN, self).__init__()
        self.embedding = nn.EmbeddingBag(vocab_size, input_size, sparse=True)
        self.hidden_size = hidden_size
        self.rnn_cell = nn.RNN(input_size, hidden_size)
        self.h20 = nn.Linear(hidden_size, output_size)
        self.softmax = nn.LogSoftmax(dim = 1)

    #create a forward pass function
    def forward(self, input_,offsets_,hidden = None, batch_size = 1):
        embedded = self.embedding(input_,offsets_)
        out, hidden = self.rnn_cell(embedded, hidden)
        output = self.h20(hidden.view(-1, self.hidden_size))
        output = self.softmax(output)
        return output, hidden

    def init_hidden(self, batch_size = 1):
        #function to init the hidden layers
        return torch.zeros(1, batch_size, self.hidden_size)
#train
net = RNN(vocab_size, emsize,n_hidden ,num_class).to(device)
criterion = nn.NLLLoss()
opt = optim.SGD(net.parameters(), lr=0.01, momentum=0.9)
```

[91]:
```
import time

def train(dataloader):
    net.train()
    total_acc, total_count = 0, 0
    log_interval = 500
    start_time = time.time()

    for idx, (label, text, offsets) in enumerate(dataloader):
        opt.zero_grad()
        predited_label = net(text, offsets)
        loss = criterion(predited_label, label)
        loss.backward()
        torch.nn.utils.clip_grad_norm_(model.parameters(), 0.1)
        opt.step()
        total_acc += (predited_label.argmax(1) == label).sum().item()
        total_count += label.size(0)
        if idx % log_interval == 0 and idx > 0:
            elapsed = time.time() - start_time
            print('| epoch {:3d} | {:5d}/{:5d} batches '
                  '| accuracy {:8.3f}'.format(epoch, idx, len(dataloader),
                                              total_acc/total_count))
            total_acc, total_count = 0, 0
            start_time = time.time()

def evaluate(dataloader):
    net.eval()
    total_acc, total_count = 0, 0

    with torch.no_grad():
        for idx, (label, text, offsets) in enumerate(dataloader):
            predited_label = net(text, offsets)
            loss = criterion(predited_label, label)
            total_acc += (predited_label.argmax(1) == label).sum().item()
            total_count += label.size(0)
    return total_acc/total_count
```

[92]:
```
EPOCHS = 10 # epoch
LR = 5  # learning rate
BATCH_SIZE = 64 # batch size for training

for epoch in range(1, EPOCHS + 1):
    epoch_start_time = time.time()
    train(train_dataloader)
    accu_val = evaluate(valid_dataloader)
    if total_accu is not None and total_accu > accu_val:
      scheduler.step()
    else:
       total_accu = accu_val
    print('-' * 59)
    print('| end of epoch {:3d} | time: {:5.2f}s | '
          'valid accuracy {:8.3f} '.format(epoch,
                                           time.time() - epoch_start_time,
                                           accu_val))
    print('-' * 59)
```

```
      ␣
↪-----------------------------------------------------------------------------
```

```
    ValueError                                Traceback (most recent call␣
↪last)

    <ipython-input-92-6cc7538ef1ad> in <module>
      5 for epoch in range(1, EPOCHS + 1):
      6     epoch_start_time = time.time()
----> 7     train(train_dataloader)
      8     accu_val = evaluate(valid_dataloader)
      9     if total_accu is not None and total_accu > accu_val:


    <ipython-input-91-9686dc2f9267> in train(dataloader)
      9     for idx, (label, text, offsets) in enumerate(dataloader):
     10         opt.zero_grad()
---> 11         predited_label = net(text, offsets)
     12         loss = criterion(predited_label, label)
     13         loss.backward()


    /usr/local/lib/python3.6/dist-packages/torch/nn/modules/module.py in␣
↪_call_impl(self, *input, **kwargs)
    887             result = self._slow_forward(*input, **kwargs)
    888         else:
--> 889             result = self.forward(*input, **kwargs)
    890         for hook in itertools.chain(
    891                 _global_forward_hooks.values(),


    <ipython-input-90-0e6b9c63588a> in forward(self, input_, offsets_,␣
↪hidden, batch_size)
     21     #create a forward pass function
     22     def forward(self, input_,offsets_,hidden = None, batch_size = 1):
---> 23         embedded = self.embedding(input_,offsets_)
     24         out, hidden = self.rnn_cell(embedded, hidden)
     25         output = self.h20(hidden.view(-1, self.hidden_size))


    /usr/local/lib/python3.6/dist-packages/torch/nn/modules/module.py in␣
↪_call_impl(self, *input, **kwargs)
    887             result = self._slow_forward(*input, **kwargs)
    888         else:
--> 889             result = self.forward(*input, **kwargs)
    890         for hook in itertools.chain(
    891                 _global_forward_hooks.values(),
    /usr/local/lib/python3.6/dist-packages/torch/nn/modules/sparse.py in␣
↪forward(self, input, offsets, per_sample_weights)
    340                                 self.max_norm, self.norm_type,
    341                                 self.scale_grad_by_freq, self.mode,␣
↪self.sparse,
--> 342                                 per_sample_weights, self.
↪include_last_offset)
    343
    344     def extra_repr(self) -> str:


    /usr/local/lib/python3.6/dist-packages/torch/nn/functional.py in␣
↪embedding_bag(input, weight, offsets, max_norm, norm_type, scale_grad_by_freq,␣
↪mode, sparse, per_sample_weights, include_last_offset)
   2045                 ", as input is treated is a mini-batch of"
   2046                 " fixed length sequences. However, found "
-> 2047                 "offsets of type {}".format(type_str)
   2048             )
   2049         offsets = torch.arange(0, input.numel(), input.size(1),␣
↪dtype=input.dtype, device=input.device)


    ValueError: if input is 2D, then offsets has to be None, as input is␣
↪treated is a mini-batch of fixed length sequences. However, found offsets of␣
↪type <class 'torch.Tensor'>
```

1.3 Question 3, 10 points

Next, replace the SRNN from Question 2 with a single-layer LSTM. Give the same output (training and testing accuracy as a function of epoch, as well as confusion matrix for the test set). Comment on the differences you observe between the two models.

[8]:
```
# Place code for Question 3 here
```

1.4 Question 4, 10 points

Explain how you could use the Transformer model to perform the same task you explored in Questions 2 and 3. How would attention be useful for this text classification task? Give a precise and detailed answer. Be sure to discuss what parts of the original Transformer you would use and what you would have to remove.

Answer: Transformer model can use a sequence of text as a input of model. This model can do as parallelization of sequence. For this task, there are many tag or many word in sentence. self-Attention machanism could be help for AgNews dataset which it try to looks at an input sequence and decides at each step which other parts of the sequence are important. To modify code for self-Attention, we need to separate part into docoder and encoder,and add more linear layer as a V, Q, K (multihead attention) after that we use same softmax function for output.

1.5 Question 5, 10 points

In Lab 13, you implemented a DQN model for tic-tac-toe. You method learned to play against a fairly dumb `expert_action` opponent, however. Also, DQN has proven to be less stable than other methods such as Double DQN, also discussed in Lab 13.

Explain below how you would apply double DQN and self-play to improve your tic-tac-toe agent. Provide pseudocode for the algorithm below.

Answer: Double DQN is part of RL model which contain 2 neural network model.First, model learn during the experience play as same as DQN. Second, copy last episode of the first model to compare for Q-value. If values of the second model are lower that the main model, we use second model to attain Q-value. Sometime DQN overestimate the reward so double DQN decoupling the actions selection from the action evaluation. To apply double DQN for tic-tac-toe, the input is as same as DQN but we need to create 2 neural network. First neural network will decides which one is the best next action and then second network evaluates this action to know Q-value.

[ ]:
```
#Pseudo code
def select_greedy_actions(states: torch.Tensor, q_network: nn.Module) -> torch.
↪Tensor:

    _, actions = q_network(states).max(dim=1, keepdim=True)
    return actions



def evaluate_selected_actions(states: torch.Tensor,
                              actions: torch.Tensor,
                              rewards: torch.Tensor,
                              dones: torch.Tensor,
                              gamma: float,
                              q_network: nn.Module) -> torch.Tensor:
    next_q_values = q_network(states).gather(dim=1, index=actions)
    q_values = rewards + (gamma * next_q_values * (1 - dones))
    return q_values
def double_q_learning_update(states: torch.Tensor,
                             rewards: torch.Tensor,
                             dones: torch.Tensor,
                             gamma: float,
                             q_network_1: nn.Module,
                             q_network_2: nn.Module) -> torch.Tensor:

    actions = select_greedy_actions(states, q_network_1)
    q_values = evaluate_selected_actions(states, actions, rewards, dones,␣
↪gamma, q_network_2)
    return q_values
```

2 Pseudo code

q__network1 is first neural network

q__network2 is secone neural network

Select__greedy__actions(state,q__network)

```
    Selection action for q_network
```

evaluate__action(state,action,reward,done,q__network)

```
    findding a next q value from q_network

    evaluate q value
```

DDQN__update(state,reward,done,q__network1,q__network2)

```
    selection from Select_greedy_actions by using state and q q_network1

    evaluate action from evaluate_action which using q_network2 which return q value from seco
```

2.1 Question 6, 30 points

Based on your existing DQN implementation, implement the double DQN and self-play training method you just described. After some training (don't spend too much time on training -- again, we just want to see that the model can learn), show the result you playing a game against your learned agent.

[9]:
```
# Code for training and playing goes here
```

[ ]:
```
```