Problem Set 6

Basil R. Yap
50.021 Artificial Intelligence - Term 8

June 27, 2018

1 Theory Component

[Q1] The following is the AdaGrad algorithm for weight update.

$$cache_i = cache_i + (\nabla_{w_i} L)^2$$
$$w_i = w_i - \frac{\eta}{\sqrt{cache_i} + \epsilon} \nabla_{w_i} L$$

where w_i is the weight to be updated, $\nabla_{w_i} L$ is the gradient of the loss w.r.t w_i, ϵ is a hyper-paremeter between 10^{-8} and 10^{-4} and η is a hyperparameter similar to step size in SGD. List one difference between AdaGrad and SGD in terms of step size and **explain** what effects you expect from this difference.

Solution Stochastic Gradient Descent uses a constant step size, this results in variable learning rates depending on the gradient of the minibatch. The learning rate is prone to extreme changes in learning rate when the gradient changes drastically.

AdaGrad tries to overcome this problem by taking the root mean square of the current gradient and previous gradients, the number of gradients considered is determined by the size of the cache. With a larger cache size, the slower the descent and the more tolerant the descent is to extreme changes in gradients.

[Q2] The following are the defining equations for a LSTM cell,

$$\begin{aligned}
i_t &= \sigma(W^i x_t + U^i h_{t-1}) \\
f_t &= \sigma(W^f x_t + U^f h_{t-1}) \\
o_t &= \sigma(W^o x_t + U^o h_{t-1}) \\
\tilde{c}_t &= \tanh(W^c x_t + U^c h_{t-1}) \\
c_t &= f_t \circ c_{t-1} + i_t \circ \tilde{c}_t \\
h_t &= o_t \circ \tanh(c_t)
\end{aligned}$$

The symbol $\circ$ denotes element-wise multiplication and $\sigma(x) = \frac{1}{1+e^{-x}}$ is the sigmoid function. Answer True/False to the following questions and give not more than 2 sentences explanation.

1. If $x_t = 0$ vector then $h_t = h_{t-1}$.
2. If f_t is very small or zero, then the error will not be back-propagated to earlier time steps.
3. The entries of f_t, i_t, o_t are non-negative.
4. f_t, i_t, o_t can be seen as probability distributions, which means that their entries are non-negative and their entries sum to 1.

Solution

1. False, the forward pass is affected by f_t, i_t and o_t.
2. False, gradient update is affected by i_t and o_t.
3. True, all three values are the output of sigmoid functions, which range $[0, 1)$.
4. False, the sigmoid function maps the input space of $[0, \infty)$ to $[0, 1)$. So, the entries do not sum to 1 and once passed through the sigmoid function never reaches 1.

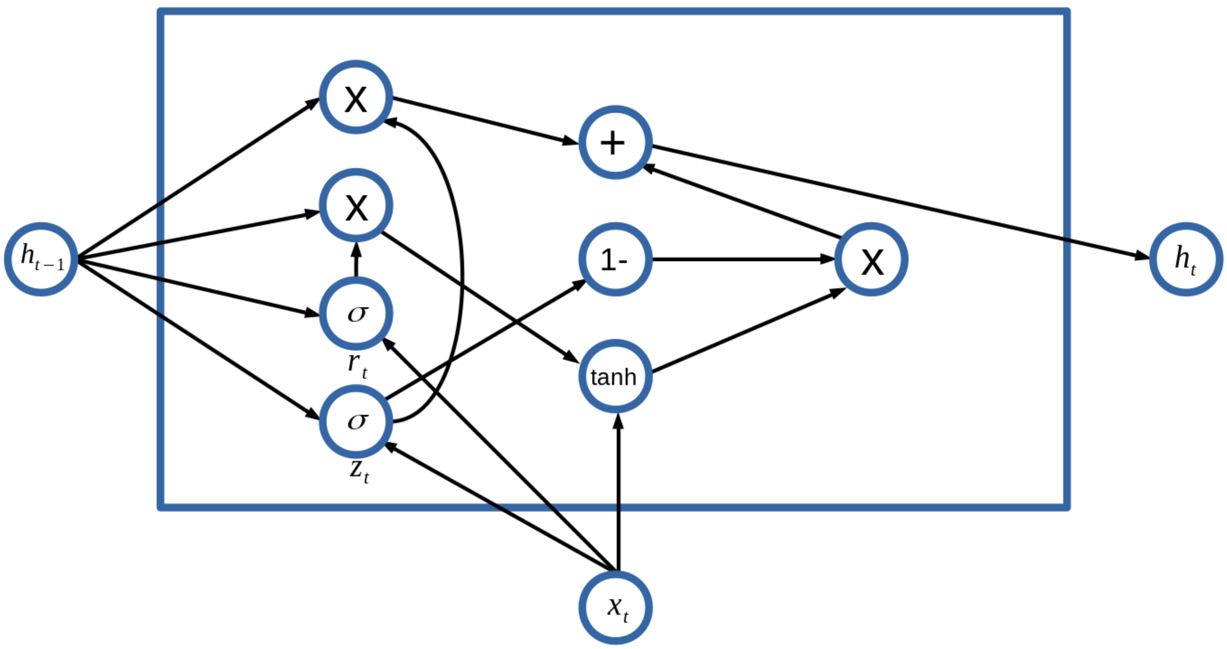
[Q3] The defining equations for a GRU cell are,

$$\begin{aligned}
z_t &= \sigma(W^z x_t + U^z h_{t-1}) \\
r_t &= \sigma(W^r x_t + U^r h_{t-1}) \\
\tilde{h}_t &= \tanh(W x_t + r_t \circ U h_{t-1}) \\
h_t &= z_t \circ h_{t-1} + (1 - z_t) \circ \tilde{h}_t
\end{aligned}$$

1. Draw a diagram of this GRU cell.
2. Assume h_t and x_t are column vectors, with dimensions d_h and d_x respectively. What are the dimensions (rows $\times$ columns) of the weight matrices W^z, W^r, W, U^z, U^r, and U?
3. Like LSTM cells, GRU cells can tackle vanishing or exploding gradient problem too. By taking a look at the formula for LSTM in Q2, what is the main advantage of using GRU cells over LSTMs for some problems? Give an answer it at most 5 sentences.
   Hint: We expect a qualitative answer (deep math proofs are not required) that comes with an explanation of the answer.

Solution

1.



2.

$$
\begin{aligned}
d_{W^z} &= d_h \times d_x \\
d_{U^z} &= d_h \times d_h \\
d_{W^r} &= d_h \times d_x \\
d_{U^r} &= d_h \times d_h \\
d_U &= d_h \times d_h
\end{aligned}
$$

3. GRU utilises two gates instead of three. By removing the memory unit present in the LSTM, GRU requires less parameters in order to train. Given a problem where the difference in performance of a single GRU and LSTM cell are negligible, GRU would be more computationally efficient than LSTM. This should be the case for models where long-term memory does not have a significant impact on future values.

2 Coding Component

```
'''
Support code for Homework 6
LSTM - Star Trek Script
'''
```

```
import os.path
import time
import unicodedata
import string
import random

import torch
import numpy.random as rand
import pickle

'''
----------- Helper Function ---------
'''
charspace = string.ascii_letters + string.digits + " ?!.,:;'-\n"  # use \n as

def letter_index(letter):
    return charspace.find(letter)

# strip unacceptable symbols


def strip_symbols(s, to_space='/+()[]'):
    for t in to_space:
        s = s.replace(t, ' ')
    return s

# turn unicode string to ascii


def convert_to_ascii(unicode_strings):
    return ''.join(c for c in unicodedata.normalize('NFD', unicode_strings.st
                   if unicodedata.category(c) != 'Mn' and c in charspace)

# convert string to a one-hot tensor


def string_to_tensor(inputstring):
    tensor = torch.zeros(len(inputstring), len(charspace))
    for i, letter in enumerate(inputstring):
        tensor[i][letter_index(letter)] = 1
    return tensor

# split csv according to ',' but preserving any ', ' which is usually in a li
```

```
def split_csv(s, ignore=',␣'):
    # uses '$' to replace ‘, ‘, and ^ to replace ','
    # and then split by '^' and convert '$' back to ', '
    s = s.replace(ignore, '$')
    s = s.replace(',', '^')
    s = s.replace('$', ignore)
    return s.split('^')


'''
―――――――― Dataset ――――――――
'''
from torch.utils.data import Dataset, DataLoader
# make it a dataset class


class MovieScriptDataset(Dataset):
    def __init__(self, root_path, ext='.csv', filterwords=[], shuffle=True):
        self.meta = {"root_path": root_path,
                     "ext": ext,
                     "filterwords": filterwords,
                     "shuffle": shuffle}
        self.line_list = []
        if root_path is not None:
            self.read_script()
        if shuffle:
            self.shuffle()

    def shuffle(self):
        random.shuffle(self.line_list)

    def read_script(self):
        '''
        Read the script given in root_path.
        '''
        filename = self.meta['root_path']
        with open(filename, 'r') as infile:
            filecontent = strip_symbols(infile.read())
            if self.meta['ext'] == '.csv':
                filecontent = split_csv(filecontent)
            else:
                filecontent = filecontent.split('\n')
            lines = [convert_to_ascii(content.strip()) for content in filecon
                     if len(content.strip()) > 1 and content.strip() not in s
```

```
        self.line_list += lines

    def split_train_test(self, train_fraction=0.7):
        '''
        Creates 2 new MovieScriptDataset and fill them
        with 70/30 of own data.
        '''
        midline = int(len(self.line_list) * train_fraction)
        train = MovieScriptDataset(None)
        train.meta = self.meta
        train.line_list = self.line_list[:midline]

        test = MovieScriptDataset(None)
        test.meta = self.meta
        test.line_list = self.line_list[midline:]
        return train, test

    def __len__(self):
        return len(self.line_list)

    def __getitem__(self, index):
        '''
        Take the name-language pair and convert them to tensors
        '''
        line = self.line_list[index]
        line_tensor = string_to_tensor(line)
        # since the goal is to predict the next letter,
        # the label should be the line shifted, plus EOL (\n)
        label = line[1:] + charspace[-1]
        label_tensor = torch.Tensor([charspace.index(l) for l in label])
        return {'input': line_tensor,
                'label': label_tensor,
                'seq_len': len(line)}

# must be executed at top-level, otherwise cant pickle


def MovieScriptCollator(tensorlist):
    # tensorlist: a list of tensors from __getitem__
    tensorlist = sorted(tensorlist, key=lambda x: x['seq_len'], reverse=True)
    data_tensor = rnn_utils.pack_sequence([val['input'] for val in tensorlist
    label_tensor = rnn_utils.pack_sequence(
        [val['label'] for val in tensorlist])
    return {'input': data_tensor,
            'label': label_tensor,
```

```python
            'batch_size': len(tensorlist)}


'''
------------ RNN/LSTM ------------
'''
import torch.nn as nn
import torch.nn.utils.rnn as rnn_utils


class CoveredLSTM(nn.Module):
    '''
    A stack of LSTM that is covered by a fully-connected layer
    as the last layer. The fully-connected layer is fed the hidden
    state of the last LSTM stack (if any). CrossEntropyLoss should be
    used as the output is a tensor of length num_class.
    '''

    def __init__(self, input_size, hidden_size, num_layers, num_classes):
        super(CoveredLSTM, self).__init__()
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.num_layers = num_layers

        self.lstm = nn.LSTM(input_size, hidden_size, num_layers)
        self.stack_fc = nn.Linear(hidden_size, hidden_size)
        self.fc_dropout = nn.Dropout(0.1)
        self.cover_fc = nn.Linear(hidden_size, num_classes)
        self.softmax = nn.Softmax(dim=0)

    def forward(self, inputs, cache):
        '''
        Forward through the lstm, then check if PackedSequence
        '''
        output, (hn, cn) = self.lstm(inputs, cache)

        if isinstance(output, rnn_utils.PackedSequence):
            stack_output = self.stack_fc(output.data)
            dropped_stack_output = self.fc_dropout(stack_output)
            covered_output = self.cover_fc(dropped_stack_output)
            return covered_output, (hn, cn)
        else:
            stack_output = self.stack_fc(output)
            dropped_stack_output = self.fc_dropout(stack_output)
            covered_output = self.cover_fc(dropped_stack_output)
```

```python
        return covered_output, (hn, cn)

    def init_cache(self, batch=1, use_gpu=True):
        '''
        "batch" parameter is added in case a
        stacked multiple hidden matrix is needed
        (e.g. for multibatch forward pass)
        '''
        h0 = torch.zeros(self.num_layers, batch, self.hidden_size)
        c0 = torch.zeros(self.num_layers, batch, self.hidden_size)
        if use_gpu:
            h0, c0 = h0.cuda(), c0.cuda()
        return (h0, c0)

    def sample(self, start_letter=None, max_length=100,
               use_gpu=True, temperature=0.5):
        '''
        With the current model, get a sample line.
        category should already be parsed (an integer/tensor)
        '''
        if start_letter == None:
            start_letter = random.choice("ABCDEFGHIJKLMNOPRSTUVWZ")

        with torch.no_grad():
            inputs = string_to_tensor(start_letter)
            cache = self.init_cache()
            output_line = start_letter

            for i in range(max_length):
                if use_gpu:
                    inputs = inputs.cuda()
                inputs = rnn_utils.pack_sequence([inputs])
                output, cache = self.forward(inputs, cache)
                output = self.softmax(output.view(-1) / temperature)
                multinom = torch.multinomial(output, 1)
                gen = multinom.item()
                # check EOL
                if gen >= len(charspace) - 1:  # meaning \n
                    break  # EOL reached - stop
                else:
                    new_letter = charspace[gen]
                    output_line += new_letter
                    inputs = string_to_tensor(new_letter)
        return output_line
```

```python
    def save_model(self, filename):
        '''
        Save the model parameters as a pickled object.
        '''
        with open(filename, 'wb') as outfile:
            pickle.dump(self.state_dict(), outfile)

    def load_model(self, filename):
        '''
        Load model parameters from specified file.
        '''
        with open(filename, 'rb') as infile:
            loaded_dict = pickle.load(infile)
            self.load_state_dict(loaded_dict)


'''
Training
'''
from torch.autograd import Variable
import torch.optim as optim


def train(train_dataset, test_dataset, model,
          batch_size=8, use_gpu=True, learnrate=5e-4, epoch=5, lr_gamma=0.95,
          print_every=1, sample_every=800, resume_from=0, save_model_every=5)
    '''
    Loop through epoch and execute train_single
    '''
    optimizer = optim.SGD(model.parameters(), lr=learnrate, momentum=0.9)
    lr_scheduler = torch.optim.lr_scheduler.StepLR(
        optimizer, lr_step, lr_gamma)
    criterion = nn.CrossEntropyLoss()

    if resume_from > 0:
        model.load_model('model/trekmodel_e{}.clstm'.format(resume_from - 1))

    train_loss_acc = []
    test_loss_acc = []

    for e in range(resume_from, epoch):
        # training
        print('EPOCH', e)
        model, loss, acc = train_single(train_dataset, model, optimizer, crit
                                        batch_size=batch_size, use_gpu=use_gp
```

```
                                        print_every=print_every, sample_every
        train_loss_acc.append((loss, acc))
        # testing
        model, loss, acc = train_single(test_dataset, model, optimizer, criter
                                        batch_size=batch_size, use_gpu=use_gp
                                        print_every=print_every, sample_every
        test_loss_acc.append((loss, acc))

        # line testing
        random_idx = random.randint(0, len(test_dataset))
        random_line = test_dataset.line_list[random_idx]
        print('      sample line:', random_line)
        random_input = test_dataset[random_idx]['input']
        if use_gpu:
            random_input = random_input.cuda()
        cache = model.init_cache(batch=1, use_gpu=use_gpu)
        output, _ = model(random_input.view(-1, 1, len(charspace)), cache)
        _, pred = output.topk(1)

        random_output = random_line[0] + \
            ''.join([charspace[idx] for idx in pred.view(-1)])
        print('      sample output:', random_output[:-1])

        lr_scheduler.step()

        if (e + 1) % save_model_every == 0:
            model.save_model("models/trekmodel_e{}.clstm".format(e))

        sample_filename = "samples/treksample_e{}.txt".format(e)
        with open(sample_filename, 'w') as samplefile:
            for i in 'ABCDEFGHIJKLMNOPRSTUVWZ':
                samplefile.write(model.sample() + '\n')

        statistic_filename = "trekstats.stt"
        with open(statistic_filename, 'wb') as statfile:
            data = {'train': train_loss_acc,
                    'test': test_loss_acc}
            pickle.dump(data, statfile)

    model.save_model("models/trekmodel_latest.clstm")
    return model, train_loss_acc, test_loss_acc


def train_single(dataset, model, optimizer, criterion, batch_size=8, use_gpu=
                 mode='train', print_every=1, sample_every=800):
```

```python
loader = DataLoader(dataset, batch_size=batch_size,
                    collate_fn=MovieScriptCollator, num_workers=4)
model.train(mode == 'train')
model.zero_grad()
total_iter_count = (len(dataset) // batch_size) + 1

running_loss = 0.0
running_corrects = 0
total_letters = 0
epoch_start = time.clock()

iterr = 0
for data in loader:
    iterr += 1
    optimizer.zero_grad()
    inputs, labels = data['input'], data['label']
    if use_gpu:
        inputs, labels = inputs.cuda(), labels.cuda()

    cache = model.init_cache(batch=data['batch_size'], use_gpu=use_gpu)
    # from MovieScriptCollator, each batch has a 'batch size' to handle e
    # e.g. if the last batch is less than batch_size

    model.zero_grad()
    output, _ = model(inputs, cache)
    _, pred = output.topk(1)

    loss = criterion(output, labels.data.long())
    running_loss += loss.item() / output.size()[0]
    if mode == 'train':
        loss.backward()
        optimizer.step()

    running_corrects += (pred.view(-1) == labels.data.long()).sum().item(
    total_letters += labels.data.size()[0]

    if (iterr % print_every) == 0:
        print('␣␣␣␣␣␣...iteration␣{}/{}'.format(iterr,
                                              total_iter_count), end='\
    if (iterr % sample_every) == 0 and mode == 'train':
        epoch_time = time.clock() - epoch_start
        print('␣␣␣␣␣␣generated_sample:', model.sample(),
              '[loss:{:.5f}␣||␣acc:{:.3f}␣||␣in␣{:.4f}s]'
              .format(running_loss, running_corrects / total_letters, epo
```

```
    epoch_time = time.clock() - epoch_start
    print("      >> Epoch loss {:.5f} accuracy {:.3f}        \
            in {:.4f}s".format(running_loss, running_corrects / total_letters

    return model, running_loss, running_corrects / total_letters


'''
------ plotting ------
'''
import matplotlib.pyplot as plt


def plot_over_epoch(train_loss_acc, test_loss_acc):
    # plot loss
    train_loss, train_acc = zip(*train_loss_acc)
    test_loss, test_acc = zip(*test_loss_acc)

    plt.figure()
    plt.subplot(121)
    plt.plot(train_loss, color='purple')
    plt.plot(test_loss, color='red')
    plt.title('Losses')

    plt.subplot(122)
    plt.plot(train_acc, color='blue')
    plt.plot(test_acc, color='cyan')
    plt.title('Accuracy')

    plt.savefig('plot_over_epoch.png')


def main():
    # split_csv test
    cssv = "I am groot.,Groot, I am."

    star_filter = ['NEXTEPISODE']
    dataset = MovieScriptDataset('../dataset/startrek/star_trek_transcripts_a
                                 filterwords=star_filter)
    # dataset, _ = dataset.split_train_test(train_fraction=0.001) # getting s
    train_data, test_data = dataset.split_train_test()

    lstm_mod = CoveredLSTM(len(charspace), 200, 3, len(charspace)).cuda()

    trained_model, train_loss_acc, test_loss_acc = train(train_data, test_dat
```

```
                                                          learnrate=1e-1, batc
    plot_over_epoch(train_loss_acc, test_loss_acc)


if __name__ == '__main__':
    main()
```