FancierSleepingBears

A Deep Learning Final Project

Progress Report

Dataset

We have obtained the Kaggle headline data for classification and ran the following preprocessing steps;

1. Blew up contractions into component words.
2. Lowered all cases.
3. Replaced all numbers with Number Tokens.
4. Blew up Possessives into original word and a target noun token
5. Preserved full vocabulary size (no rare word replacement or extraction)

Encoding

First we attempted to do an encoding of word to vector, implementing it within the pytorch framework. However, upon running over out entire dataset we found that even on a reasonably modern GPU time to run was too long to be useful. Instead we train the embedding layer in our model. This performs much quicker with no significant loss in accuracy.

Test Networks

We built a network using the following layers:

1. Data embedding layer
2. RNN layers -Last hidden state->
3. Linear layers

This was able to achieve around 90% accuracy after 1 epoch (about 1 minute to run). We accidently had an interesting bug where we originally soft-maxing over the wrong dimension and still getting reasonable outputs. If we ran the test dataset through the network with the same batch size it would perform with 90% accuracy, but if batch size was modified accuracy would tank. We then fixed that and are getting ~92% on our model with variable batch size.

Future Work

In the future plan to explore how modifying the RNNs embedding size, number of hidden units, LSTMs vs. GRUs, and number of layers within the net will affect our accuracy on the headlines dataset. Generalizing to the twitter dataset will be a little more challenging (e.g. how do you encode emojis) because of the need for improved pre-processing and the dataset being so much smaller. Finally, we plan to introduce convolutional based networks for comparison. For preprocessing we plan to test if eliminating single-used words will help to improve accuracy (the group is split its potential effectiveness).

Code

```
import collections
import os
import pandas
from six.moves import urllib
import zipfile
import re
import numpy as np
import pickle
```

```
import argparse
import datetime
import six
import math

import torch
import torch.nn as nn
import torch.utils.data
from torch.autograd import Variable
import numpy as np
import torch.functional as F
import torch.nn.functional as F
import torch.optim as optim

from torch.nn.utils.rnn import pad_packed_sequence, pack_padded_sequence

from tqdm import tqdm
from tensorboardX import SummaryWriter
import callbacks
import random
from random import shuffle


parser = argparse.ArgumentParser(description='Deep Learning JHU Assignment 1 - Fashion-MNIST')
parser.add_argument('--batch-size', type=int, default=256, metavar='B',
                    help='input batch size for training (default: 256)')
parser.add_argument('--test-batch-size', type=int, default=256, metavar='TB',
                    help='input batch size for testing (default: 1000)')

parser.add_argument('--embed-size', type=int, default=64, metavar='ES',
                    help='input batch size for testing (default: 64)')
parser.add_argument('--hidden-size', type=int, default=64, metavar='HS',
                    help='input batch size for testing (default: 64)')

parser.add_argument('--epochs', type=int, default=10, metavar='E',
                    help='number of epochs to train (default: 10)')
parser.add_argument('--lr', type=float, default=0.01, metavar='LR',
                    help='learning rate (default: 0.01)')
parser.add_argument('--optimizer', type=str, default='adam', metavar='O',
                    help='Optimizer options are sgd, adam, rms_prop')
# parser.add_argument('--momentum', type=float, default=0.5, metavar='MO',
                    # help='SGD momentum (default: 0.5)')
# parser.add_argument('--dropout', type=float, default=0.5, metavar='DO',
                    # help='Dropout probability (default: 0.5)')
parser.add_argument('--no-cuda', action='store_true', default=False,
                    help='disables CUDA training')
# parser.add_argument('--save_model', action='store_true', default=False,
                    # help='Save model')
parser.add_argument('--seed', type=int, default=1, metavar='S',
                    help='random seed (default: 1)')
parser.add_argument('--log_interval', type=int, default=0, metavar='I',
                    help="""how many batches to wait before logging detailed
                            training status, 0 means never log """)
# parser.add_argument('--dataset', type=str, default='mnist', metavar='D',
                    # help='Options are mnist and fashion_mnist')
# parser.add_argument('--data_dir', type=str, default='../data/', metavar='F',
                    # help='Where to put data')
# parser.add_argument('--load_model', type=str, metavar='LM',
                    # help='Where to load model from')
parser.add_argument('--log_dir', type=str, default='./data/', metavar='F',
                    help='Where to put logging')
# parser.add_argument('--name', type=str, default='', metavar='N',
                    # help="""A name for this training run, this
                            # affects the directory so use underscores and not spaces.""")
parser.add_argument('--model', type=str, default='default', metavar='M',
                    help="""Options are default, simple_rnn.""")
# parser.add_argument('--print_log', action='store_true', default=False,
                    # help='prints the csv log when training is complete')


args = parser.parse_args()

use_cuda = not args.no_cuda and torch.cuda.is_available()
```

```
batch_size = args.batch_size
test_batch_size = args.test_batch_size

embed_size = args.embed_size
hidden_size = args.hidden_size
log_interval = args.log_interval
epochs = args.epochs
learning_rate = args.lr
model_name = args.model
optimizer_name = args.optimizer

random.seed(args.seed)
torch.manual_seed(args.seed)
if use_cuda:
    torch.cuda.manual_seed(args.seed)

upper_limit_vocab_size = 1000000

if os.path.exists('news_data.p'):
    print('Load processed data')
    processed_data, vocabulary = pickle.load(open('news_data.p', 'rb'))
else:
    print('Create processed data')
    news_data = pandas.read_csv('uci-news-aggregator.csv')

    # skip_list=['plosser','noyer','bunds','cooperman','urgest','4b',"didn't",'chipotle','djia','5th','direxion']
    skip_list={}
    contractions_dict = {
    "ain't":'is not',
    "it'll":"it will",
    "there'll":"there will",
    "aren't": "are not",
    "can't": "cannot",
    "can't've": "cannot have",
    "'cause": "because",
    "could've": "could have",
    "couldn't": "could not",
    "couldn't've": "could not have",
    "didn't": "did not",
    "doesn't": "does not",
    "don't": "do not",
    "hadn't": "had not",
    "hadn't've": "had not have",
    "hasn't": "has not",
    "haven't": "have not",
    "he'd've": "he would have",
    "how'd": "how did",
    "how'd'y": "how do you",
    "how'll": "how will",
    "i'd've": "I would have",
    "i'd":'i would',
    "i'm": "I am",
    "i'll": "I will",
    "she'll": "she will",
    "he'll": "he will",
    "i've": "I have",
    "isn't": "is not",
    "he'd":"he would",
    "it'd've": "it would have",
    "let's": "let us",
    "ma'am": "madam",
    "mayn't": "may not",
    "might've": "might have",
    "mightn't": "might not",
    "mightn't've": "might not have",
    "must've": "must have",
    "mustn't": "must not",
    "mustn't've": "must not have",
    "needn't": "need not",
    "needn't've": "need not have",
    "o'clock": "of the clock",
    "oughtn't": "ought not",
    "oughtn't've": "ought not have",
    "shan't": "shall not",
```

```
"sha'n't": "shall not",
"shan't've": "shall not have",
"she'd've": "she would have",
"should've": "should have",
"shouldn't": "should not",
"shouldn't've": "should not have",
"so've": "so have",
"that'd've": "that would have",
"there'd've": "there would have",
"they'd've": "they would have",
"they're": "they are",
"they've": "they have",
"to've": "to have",
"wasn't": "was not",
"we'd've": "we would have",
"we'll": "we will",
"we'll've": "we will have",
"we're": "we are",
"we've": "we have",
"weren't": "were not",
"what're": "what are",
"what've": "what have",
"when've": "when have",
"where'd": "where did",
"they'd": "they did",
"where've": "where have",
"it'd":"it would",
"may've":"may have",
"who've": "who have",
"why've": "why have",
"will've": "will have",
"won't": "will not",
"won't've": "will not have",
"would've": "would have",
"wouldn't": "would not",
"wouldn't've": "would not have",
"y'all": "you all",
"y'all'd": "you all would",
"y'all'd've": "you all would have",
"y'all're": "you all are",
"y'all've": "you all have",
"you'd've": "you would have",
"who'd":"who would",
"you'd":"you would",
"why'd":"why would",
"you'll": "you will",
"who'll": "who will",
"what'll": "what will",
"you're": "you are",
"you've": "you have"
}

contractions_re = re.compile('(%s)' % '|'.join(contractions_dict.keys()))

def expand_contractions(s, contractions_dict=contractions_dict):
    def replace(match):
        return contractions_dict[match.group(0)]
    return contractions_re.sub(replace, s)

processed_data = []
vocabulary = []

for i, title in enumerate(news_data.TITLE):

    # Log progress
    if i % 10000 == 0:
        print(i, len(news_data.TITLE))

    # Replace contractions and split
    title = expand_contractions(title.lower())
    title = re.findall(r"[\w']+", title.lower())

    # Too long of a sentence
    if len(title) >= 20:
```

```
            continue

        sent = []
        for w in title:
            w = w.strip(', .:%\'')

            try:
                val = float(w)
                sent.append("<NUM>")
                continue
            except ValueError:
                pass

            if w[-2:] == "'s":
                w = w[-2:]

                sent.append(w)
                sent.append("'s")
            else:
                sent.append(w)

        processed_data.append((sent, news_data.CATEGORY[i]))
        vocabulary += sent

    pickle.dump((processed_data, vocabulary), open('news_data.p', 'wb'))



def build_dataset(words, n_words):
    """Process raw inputs into a dataset."""
    print('Build dataset')
    count = [['UNK', -1]]
    count.extend(collections.Counter(words).most_common(n_words - 1))
    dictionary = dict()
    for word, _ in count:
        dictionary[word] = len(dictionary)
    data = list()
    unk_count = 0
    for word in words:
        if word in dictionary:
            index = dictionary[word]
        else:
            index = 0  # dictionary['UNK']
            unk_count += 1
        data.append(index)
    count[0][1] = unk_count
    reversed_dictionary = dict(zip(dictionary.values(), dictionary.keys()))
    return data, count, dictionary, reversed_dictionary

data, count, dictionary, reversed_dictionary = build_dataset(vocabulary, upper_limit_vocab_size)



word2idx = dictionary
idx2word = reversed_dictionary
vocabulary = reversed_dictionary.values()
vocabulary_size = len(vocabulary)
tokenized_corpus = processed_data


corpus_dataset = []
# for each sentence
# i = 0
print('Create corpus dataset')
for sentence,label in tokenized_corpus:
    # i += 1
    # if i % 10000 == 0:
        # print(i, len(tokenized_corpus))
    indices = [word2idx[word] for word in sentence]
    corpus_dataset.append((indices, len(indices),label))


print('Create training and testing splits')
shuffle(corpus_dataset)
```

```
training_dataset = corpus_dataset[:int(0.8*len(corpus_dataset))]
test_dataset = corpus_dataset[int(0.8*len(corpus_dataset)):]

training_dataset = sorted(training_dataset, key=lambda x:x[1])
test_dataset = sorted(test_dataset, key=lambda x:x[1])

max_sent_len = max(training_dataset[-1][1], test_dataset[-1][1])

training = []
for sent, sent_len, label in training_dataset:
    for i in range(len(sent), max_sent_len):
        sent.append(0)
    sent = torch.Tensor([sent]).long()
    training.append((sent, sent_len,label))

test = []
for sent, sent_len, label in test_dataset:
    for i in range(len(sent), max_sent_len):
        sent.append(0)
    sent = torch.Tensor([sent]).long()
    test.append((sent, sent_len,label))

batches = list(range(len(training)))
shuffle(batches)



# Hard-coded the labels in
labels = ['m', 'b', 'e', 't']

class RNNModel(nn.Module):
    def __init__(self):
        super(RNNModel, self).__init__()
        self.embed = nn.Embedding(vocabulary_size,embed_size)
        self.rnn = nn.RNN(embed_size,hidden_size,1)
        self.w = nn.Linear(hidden_size,4)

    def forward(self,x,lengths):
        x = self.embed(x)
        y = pack_padded_sequence(x, lengths, batch_first=True)
        _,h = self.rnn(y)
        h = h.view(-1,hidden_size)
        # h = self.w(h)
        return F.log_softmax(self.w(h), dim=1)


if model_name == 'default' or model_name == 'simple_rnn':
    model = RNNModel()
else:
    raise ValueError('Unknown model type: ' + model_name)

print(model)
if use_cuda:
    model.cuda()

if optimizer_name == 'sgd':
    optimizer = optim.SGD(model.parameters(), lr=args.lr)
elif optimizer_name == 'adam':
    optimizer = optim.Adam(model.parameters(), lr=args.lr)
elif optimizer_name == 'rmsprop':
    optimizer = optim.RMSprop(model.parameters(), lr=args.lr)
else:
    raise ValueError('Unsupported optimizer: ' + optimizer_name)


# return optimizer
# optimizer = optim.Adam(model.parameters(), lr=learning_rate)


def train(tensorboard_writer, callbacklist, total_minibatch_count):

    correct_count = np.array(0)
    for batch_idx, i in enumerate(range(0, len(training), batch_size)):
```

```
        callbacklist.on_batch_begin(batch_idx)

        training_batch_idx = sorted(batches[i:min(i+batch_size, len(training))])
        training_batch = torch.Tensor(len(training_batch_idx), max_sent_len).long()
        y_true = torch.Tensor(len(training_batch_idx)).long()
        optimizer.zero_grad()
        sent_len = 0
        lengths = []
        for j, idx in enumerate(reversed(training_batch_idx)):
            sent_len = training[idx][1]
            lengths.append(sent_len)
            training_batch[j] = training[idx][0]
            y_true[j] = labels.index(training[idx][2])

        training_batch = training_batch[:,:lengths[0]]

        training_batch = Variable(training_batch)
        y_true = Variable(y_true)
        if use_cuda:
            test_batch, y_true = test_batch.cuda(), y_true.cuda()

        y_pred = model(training_batch,lengths)
        loss = F.nll_loss(y_pred, y_true)
        loss.backward()
        optimizer.step()

        _, argmax = torch.max(y_pred, 1)
        compare = (y_true == argmax).float()
        accuracy = compare.mean()
        correct_count += compare.int().sum().data.numpy()[0]

        batch_logs = {
            'loss': np.array(loss.data[0]),
            'acc': np.array(accuracy.data[0]),
            'size': np.array(len(y_true)),
            'batch': np.array(batch_idx)
        }

        callbacklist.on_batch_end(batch_idx, batch_logs)

        if log_interval != 0 and total_minibatch_count % log_interval == 0:
            # put all the logs in tensorboard
            for name, value in six.iteritems(batch_logs):
                tensorboard_writer.add_scalar(name, value, global_step=total_minibatch_count)

            # put all the parameters in tensorboard histograms
            # for name, param in model.named_parameters():
                # name = name.replace('.', '/')
                # tensorboard_writer.add_histogram(name, param.data.cpu().numpy(), global_step=total_minibatch_count)
                # tensorboard_writer.add_histogram(name + '/gradient', param.grad.data.cpu().numpy(), global_step=total_mi

        total_minibatch_count += 1
        # break

    return total_minibatch_count

def test_fn(tensorboard_writer, callbacklist, total_minibatch_count, epoch):

    test_size = np.array(len(test), np.float32)
    correct = 0
    test_loss = 0
    progress_bar = tqdm(range(0, len(test), test_batch_size), desc='Validation')
    # for i in range(0, len(test), test_batch_size):
    for i in progress_bar:

        test_batch_idx = range(i, min(i+test_batch_size, len(test)))
        # print(i, test_batch_idx)
        test_batch = torch.Tensor(len(test_batch_idx), max_sent_len).long()
        y_true = torch.Tensor(len(test_batch_idx)).long()

        sent_len = 0
        lengths = []
        for j, i in enumerate(reversed(test_batch_idx)):
```

```
            sent_len = test[i][1]
            lengths.append(sent_len)
            test_batch[j] = test[i][0]
            y_true[j] = labels.index(test[i][2])

        test_batch = test_batch[:,:lengths[0]]

        test_batch = Variable(test_batch,volatile=True)
        y_true = Variable(y_true,volatile=True)
        if use_cuda:
            test_batch, y_true = test_batch.cuda(), y_true.cuda()

        y_pred = model(test_batch, lengths)

        test_loss += F.nll_loss(y_pred, y_true, size_average=False).data[0]
        _, y_pred = torch.max(y_pred, dim=1)
        # print(y_pred, y_true)

        correct += (y_pred == y_true).sum().data.numpy()[0]
        # print(correct)
        total_minibatch_count += 1

    # print(correct, test_size)
    test_loss /= test_size
    acc = np.array(correct, np.float32) / test_size

    epoch_logs = {'val_loss': np.array(test_loss),
                  'val_acc': np.array(acc)}

    for name, value in six.iteritems(epoch_logs):
        tensorboard_writer.add_scalar(name, value, global_step=total_minibatch_count)
    if callbacklist is not None:
        callbacklist.on_epoch_end(epoch, epoch_logs)

    progress_bar.write(
        'Epoch: {} - validation test results - Average val_loss: {:.4f}, val_acc: {}/{} ({:.2f}%)'.format(
            epoch, test_loss, correct, test_size,
            100. * correct / test_size))



# Set up visualization and progress status update code
callback_params = {'epochs': epochs,
                   'samples': len(training),
                   'steps': len(training) / batch_size,
                   'metrics': {'acc': np.array([]),
                               'loss': np.array([]),
                               'val_acc': np.array([]),
                               'val_loss': np.array([])}}

output_on_train_end = os.sys.stdout

callbacklist = callbacks.CallbackList(
    [callbacks.BaseLogger(),
     callbacks.TQDMCallback(),
     callbacks.CSVLogger(filename="data/run1.csv",
                         output_on_train_end=output_on_train_end)])
callbacklist.set_params(callback_params)

tensorboard_writer = SummaryWriter(log_dir="data/", comment='simple_rnn_training')

total_minibatch_count = 0

callbacklist.on_train_begin()
for epoch in range(epochs):
    callbacklist.on_epoch_begin(epoch)
    total_minibatch_count = train(tensorboard_writer, callbacklist,total_minibatch_count)
    test_fn(tensorboard_writer, callbacklist, total_minibatch_count,epoch)
callbacklist.on_train_end()
tensorboard_writer.close()
```