MP4_P2_classification

November 23, 2020

[1]:
```
from google.colab import drive
drive.mount('/content/gdrive')
import os
os.chdir("gdrive/My Drive/assignment4")
```

```
Mounted at /content/gdrive
```

[2]:
```
!pip install unidecode
```

```
Collecting unidecode
  Downloading https://files.pythonhosted.org/packages/d0/42/d9edfed04228ba
cea2d824904cae367ee9efd05e6cce7ceaaedd0b0ad964/Unidecode-1.1.1-py2.py3-none-
any.whl (238kB)
     |                 | 245kB 12.2MB/s
Installing collected packages: unidecode
Successfully installed unidecode-1.1.1
```

[3]:
```
import os
import time
import math
import glob
import string
import random

import torch
import torch.nn as nn

from rnn.helpers import time_since

%matplotlib inline
```

[4]:
```
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
```

1 Language recognition with an RNN

If you've ever used an online translator you've probably seen a feature that automatically detects the input language. While this might be easy to do if you input unicode characters that are unique to one or a small group of languages (like “ ” or “ ”), this problem is more challenging if the input only uses the available ASCII characters. In this case, something like “těší mě” would beome “tesi me” in the ascii form. This is a more challenging problem in which the language must be recognized purely by the pattern of characters rather than unique unicode characters.

We will train an RNN to solve this problem for a small set of languages thta can be converted to romanized ASCII form. For training data it would be ideal to have a large and varied dataset in different language styles. However, it is easy to find copies of the Bible which is a large text translated to different languages but in the same easily parsable format, so we will use 20 different copies of the Bible as training data. Using the same book for all of the different languages will hopefully prevent minor overfitting that might arise if we used different books for each language (fitting to common characteristics of the individual books rather than the language).

[5]:
```python
from unidecode import unidecode as unicodeToAscii

all_characters = string.printable
n_letters = len(all_characters)

print(unicodeToAscii('těší mě'))
```

```
tesi me
```

[6]:
```python
# Read a file and split into lines
def readFile(filename):
    data = open(filename, encoding='utf-8').read().strip()
    return unicodeToAscii(data)

def get_category_data(data_path):
    # Build the category_data dictionary, a list of names per language
    category_data = {}
    all_categories = []
    for filename in glob.glob(data_path):
        category = os.path.splitext(os.path.basename(filename))[0].split('_')[0]
        all_categories.append(category)
        data = readFile(filename)
        category_data[category] = data

    return category_data, all_categories
```

The original text is split into two parts, train and test, so that we can make sure that the model is not simply memorizing the train data.

[7]:
```python
train_data_path = 'language_data/train/*_train.txt'
test_data_path = 'language_data/test/*_test.txt'

train_category_data, all_categories = get_category_data(train_data_path)
test_category_data, test_all_categories = get_category_data(test_data_path)
```

```
n_languages = len(all_categories)

print(len(all_categories))
print(all_categories)
```

```
20
['albanian', 'english', 'czech', 'danish', 'esperanto', 'finnish', 'french',
'german', 'hungarian', 'italian', 'lithuanian', 'maori', 'norwegian',
'portuguese', 'romanian', 'spanish', 'swedish', 'turkish', 'vietnamese',
'xhosa']
```

2 Data processing

[8]:
```
def categoryFromOutput(output):
    top_n, top_i = output.topk(1, dim=1)
    category_i = top_i[:, 0]
    return category_i

# Turn string into long tensor
def stringToTensor(string):
    tensor = torch.zeros(len(string), requires_grad=True).long()
    for c in range(len(string)):
        tensor[c] = all_characters.index(string[c])
    return tensor

def load_random_batch(text, chunk_len, batch_size):
    input_data = torch.zeros(batch_size, chunk_len).long().to(device)
    target = torch.zeros(batch_size, 1).long().to(device)
    input_text = []
    for i in range(batch_size):
        category = all_categories[random.randint(0, len(all_categories) - 1)]
        line_start = random.randint(0, len(text[category])-chunk_len)
        category_tensor = torch.tensor([all_categories.index(category)],␣
 ↪dtype=torch.long)
        line = text[category][line_start:line_start+chunk_len]
        input_text.append(line)
        input_data[i] = stringToTensor(line)
        target[i] = category_tensor
    return input_data, target, input_text
```

3 Implement Model

For this classification task, we can use the same model we implement for the generation task which is located in `rnn/model.py`. See the `MP4_P2_generation.ipynb` notebook for more instructions. In this case each output vector of our RNN will have the dimension of the number of possible languages (i.e. `n_languages`). We will use this vector to predict a distribution over the languages.

In the generation task, we used the output of the RNN at every time step to predict the next letter and our loss included the output from each of these predictions. However, in this task we use the output of the RNN at the end of the sequence to predict the language, so our loss function will use only the predicted output from the last time step.

4 Train RNN

[9]:
```
from rnn.model import RNN
```

[169]:
```
chunk_len = 10

BATCH_SIZE = 100
n_epochs = 5000
hidden_size = 350
n_layers = 2
learning_rate = 0.001
model_type = 'lstm'

criterion = nn.CrossEntropyLoss()
rnn = RNN(n_letters, hidden_size, n_languages, model_type=model_type,␣
 ↪n_layers=n_layers).to(device)
```

TODO: Fill in the train function. You should initialize a hidden layer representation using your RNN's `init_hidden` function, set the model gradients to zero, and loop over each time step (character) in the input tensor. For each time step compute the output of the of the RNN and the next hidden layer representation. The cross entropy loss should be computed over the last RNN output scores from the end of the sequence and the target classification tensor. Lastly, call backward on the loss and take an optimizer step.

[40]:
```
def train(rnn, target_tensor, data_tensor, optimizer, criterion,␣
 ↪batch_size=BATCH_SIZE):
    """
    Inputs:
    - rnn: model
    - target_target: target character data tensor of shape (batch_size, 1)
    - data_tensor: input character data tensor of shape (batch_size, chunk_len)
    - optimizer: rnn model optimizer
    - criterion: loss function
    - batch_size: data batch size

    Returns:
    - output: output from RNN from end of sequence
    - loss: computed loss value as python float

    """

    output, loss = None, None
    ####################################
    #          YOUR CODE HERE          #
    ####################################

    hidden = rnn.init_hidden(batch_size, device)
    optimizer.zero_grad()

    for i in range(chunk_len):
      output, hidden = rnn.forward(data_tensor[:, i], hidden)

    loss = criterion(output, target_tensor.squeeze())
    loss.backward()
    optimizer.step()

    ##########       END      ##########

    return output, loss
```

[153]:
```
def evaluate(rnn, data_tensor, seq_len=chunk_len, batch_size=BATCH_SIZE):
    with torch.no_grad():
        data_tensor = data_tensor.to(device)
        hidden = rnn.init_hidden(batch_size, device=device)
        for i in range(seq_len):
            output, hidden = rnn(data_tensor[:,i], hidden)

        return output

def eval_test(rnn, category_tensor, data_tensor):
    with torch.no_grad():
        output = evaluate(rnn, data_tensor)
        loss = criterion(output, category_tensor.squeeze())
        return output, loss.item()
```

[170]:
```
n_iters = 2000 #2000 #100000
print_every = 50
plot_every = 50


# Keep track of losses for plotting
current_loss = 0
current_test_loss = 0
all_losses = []
all_test_losses = []

start = time.time()
```

```
optimizer = torch.optim.Adam(rnn.parameters(), lr=learning_rate, weight_decay=0.
↪0001)


number_correct = 0
for iter in range(1, n_iters + 1):
    input_data, target_category, text_data =␣
↪load_random_batch(train_category_data, chunk_len, BATCH_SIZE)
    output, loss = train(rnn, target_category, input_data, optimizer, criterion)
    current_loss += loss

    _, test_loss = eval_test(rnn, target_category, input_data)
    current_test_loss += test_loss

    guess_i = categoryFromOutput(output)
    number_correct += (target_category.squeeze()==guess_i.squeeze()).long().
↪sum()

    # Print iter number, loss, name and guess
    if iter % print_every == 0:
        sample_idx = 0
        guess = all_categories[guess_i[sample_idx]]

        category = all_categories[int(target_category[sample_idx])]

        correct = ' ' if guess == category else '  (%s)' % category
        print('%d %d%% (%s) %.4f %.4f %s / %s %s' % (iter, iter / n_iters *␣
↪100, time_since(start), loss, test_loss, text_data[sample_idx], guess,␣
↪correct))
        print('Train accuracy: {}'.format(float(number_correct)/
↪float(print_every*BATCH_SIZE)))
        number_correct = 0

    # Add current loss avg to list of losses
    if iter % plot_every == 0:
        all_losses.append(current_loss / plot_every)
        current_loss = 0
        all_test_losses.append(current_test_loss / plot_every)
        current_test_loss = 0
```

```
50 2% (0m 1s) 2.1417 2.0639 is eram um / romanian   (portuguese)
Train accuracy: 0.2052
100 5% (0m 3s) 1.5823 1.4894 g. Khi mot / vietnamese
Train accuracy: 0.3694
150 7% (0m 4s) 1.4208 1.3352 erusalim.  / turkish   (romanian)
Train accuracy: 0.451
```

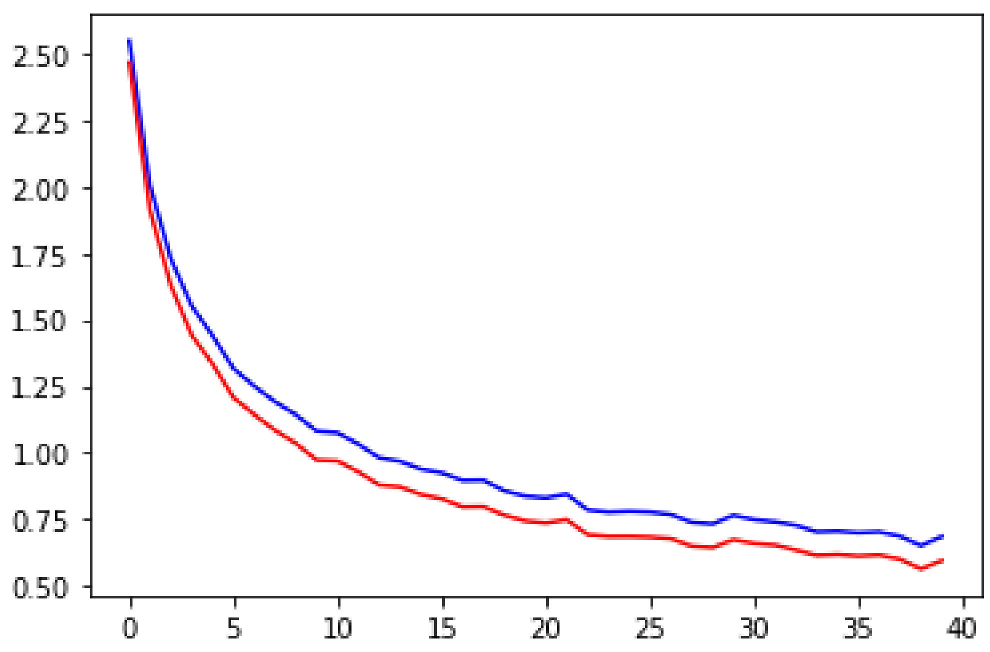
```
200 10% (0m 6s) 1.5657 1.4441 lle: 'Mina / romanian  (finnish)
Train accuracy: 0.504
250 12% (0m 7s) 1.5853 1.4667  haere ki  / norwegian  (maori)
Train accuracy: 0.5312
300 15% (0m 9s) 1.2209 1.0903  tye emehl / english  (xhosa)
Train accuracy: 0.5672
350 17% (0m 11s) 1.0095 0.9133 , Bot-ra,  / vietnamese
Train accuracy: 0.592
400 20% (0m 12s) 1.2434 1.1273 excito Jeh / spanish
Train accuracy: 0.6122
450 22% (0m 14s) 1.1080 0.9851 le, vlozil / czech
Train accuracy: 0.622
500 25% (0m 15s) 1.1990 1.0832 dejci, kte / czech
Train accuracy: 0.6406
550 27% (0m 17s) 0.8984 0.8130 he garden  / albanian  (english)
Train accuracy: 0.6464
600 30% (0m 19s) 1.0246 0.9366 to i te wh / maori
Train accuracy: 0.6556
650 32% (0m 20s) 0.9207 0.8300 tonu te ri / maori
Train accuracy: 0.6772
700 35% (0m 22s) 1.0030 0.9003 or sus nom / portuguese  (spanish)
Train accuracy: 0.6806
750 37% (0m 23s) 0.9115 0.8135 iaure nuo  / lithuanian
Train accuracy: 0.691
800 40% (0m 25s) 0.9090 0.8037 om da til  / norwegian
Train accuracy: 0.6962
850 42% (0m 27s) 0.9595 0.8449 mig." Thi  / english  (danish)
Train accuracy: 0.7044
900 45% (0m 28s) 0.9186 0.8400 ner fra Gu / danish  (norwegian)
Train accuracy: 0.7024
950 47% (0m 30s) 0.9946 0.9095 , co mu pa / vietnamese  (czech)
Train accuracy: 0.7208
1000 50% (0m 31s) 0.7254 0.6382 ner var Am / danish  (norwegian)
Train accuracy: 0.7178
1050 52% (0m 33s) 0.8275 0.7383 laktoffer  / swedish
Train accuracy: 0.7212
1100 55% (0m 35s) 0.7037 0.6308 e henkensa / swedish  (finnish)
Train accuracy: 0.7204
1150 57% (0m 36s) 0.8430 0.7442 ndepladen, / swedish  (danish)
Train accuracy: 0.7388
1200 60% (0m 38s) 0.8591 0.7784 Egiptujon, / esperanto
Train accuracy: 0.7402
1250 62% (0m 39s) 0.8894 0.7902 eg vende o / norwegian  (danish)
Train accuracy: 0.7408
1300 65% (0m 41s) 0.7120 0.6164 ose uzmoke / czech  (lithuanian)
Train accuracy: 0.742
1350 67% (0m 43s) 0.7178 0.6375 n animon k / esperanto
Train accuracy: 0.7398
```

```
1400 70% (0m 44s) 0.7849 0.6742 rit e bage / albanian
Train accuracy: 0.7524
1450 72% (0m 46s) 0.6043 0.5326 in ne krye / albanian
Train accuracy: 0.7548
1500 75% (0m 47s) 0.8754 0.7798 as de Saul / spanish  (portuguese)
Train accuracy: 0.7404
1550 77% (0m 49s) 0.7761 0.6785  consagrad / portuguese
Train accuracy: 0.7502
1600 80% (0m 50s) 0.6663 0.5972  nel suo p / italian
Train accuracy: 0.7512
1650 82% (0m 52s) 0.7647 0.6774 ir!> diyer / turkish
Train accuracy: 0.7514
1700 85% (0m 54s) 0.6937 0.6136 Ryktet har / norwegian  (swedish)
Train accuracy: 0.7622
1750 87% (0m 55s) 0.5960 0.5238 e Zeiten?  / german
Train accuracy: 0.7576
1800 90% (0m 57s) 0.8396 0.7365 lten sich  / german
Train accuracy: 0.7668
1850 92% (0m 58s) 0.7715 0.6573  Joosua ja / finnish
Train accuracy: 0.761
1900 95% (1m 0s) 0.6429 0.5575  arin, arg / turkish  (albanian)
Train accuracy: 0.7686
1950 97% (1m 1s) 0.7208 0.6311  Uro. Bedr / lithuanian  (danish)
Train accuracy: 0.7816
2000 100% (1m 3s) 0.7442 0.6175  nebi zabu / czech
Train accuracy: 0.7672
```

4.1 Plot loss functions

[171]:
```
import matplotlib.pyplot as plt
import matplotlib.ticker as ticker

plt.figure()
plt.plot(all_losses, color='b')
plt.plot(all_test_losses, color='r')
```

[171]:
```
[<matplotlib.lines.Line2D at 0x7faf357a95c0>]
```



4.2 Evaluate results

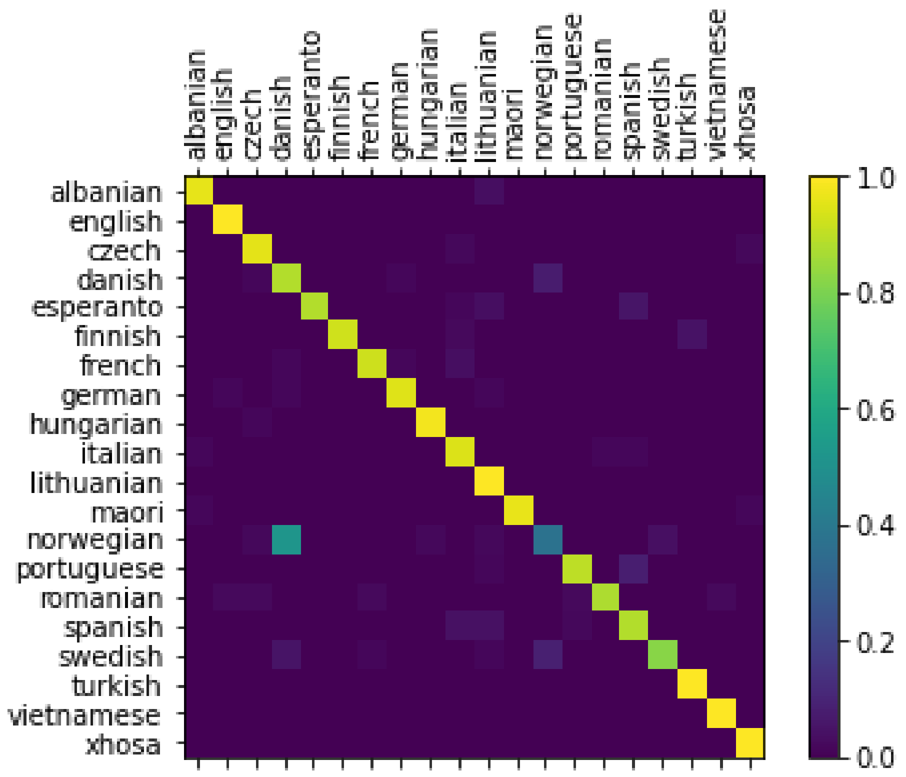
We now vizualize the performance of our model by creating a confusion matrix. The ground truth languages of samples are represented by rows in the matrix while the predicted languages are represented by columns.

In this evaluation we consider sequences of variable sizes rather than the fixed length sequences we used for training.

[172]:
```
eval_batch_size = 1  # needs to be set to 1 for evaluating different sequence␣
↪lengths

# Keep track of correct guesses in a confusion matrix
confusion = torch.zeros(n_languages, n_languages)
n_confusion = 1000
num_correct = 0
total = 0

for i in range(n_confusion):
    eval_chunk_len = random.randint(10, 50) # in evaluation we will look at␣
↪sequences of variable sizes
    input_data, target_category, text_data =␣
↪load_random_batch(test_category_data, chunk_len=eval_chunk_len,␣
↪batch_size=eval_batch_size)
    output = evaluate(rnn, input_data, seq_len=eval_chunk_len,␣
↪batch_size=eval_batch_size)
```

```
    guess_i = categoryFromOutput(output)
    category_i = [int(target_category[idx]) for idx in␣
 ↪range(len(target_category))]
    for j in range(eval_batch_size):
        category = all_categories[category_i[j]]
        confusion[category_i[j]][guess_i[j]] += 1
        num_correct += int(guess_i[j]==category_i[j])
        total += 1

print('Test accuracy: ', float(num_correct)/float(n_confusion*eval_batch_size))

# Normalize by dividing every row by its sum
for i in range(n_languages):
    confusion[i] = confusion[i] / confusion[i].sum()

# Set up plot
fig = plt.figure()
ax = fig.add_subplot(111)
cax = ax.matshow(confusion.numpy())
fig.colorbar(cax)

# Set up axes
ax.set_xticklabels([''] + all_categories, rotation=90)
ax.set_yticklabels([''] + all_categories)

# Force label at every tick
ax.xaxis.set_major_locator(ticker.MultipleLocator(1))
ax.yaxis.set_major_locator(ticker.MultipleLocator(1))

plt.show()
```

```
Test accuracy:  0.912
```



You can pick out bright spots off the main axis that show which languages it guesses incorrectly.

4.3 Run on User Input

Now you can test your model on your own input.

```
[175]: def predict(input_line, n_predictions=5):
           print('\n> %s' % input_line)
           with torch.no_grad():
               input_data = stringToTensor(input_line).long().unsqueeze(0).to(device)
               output = evaluate(rnn, input_data, seq_len=len(input_line),␣
        ↪batch_size=1)

           # Get top N categories
           topv, topi = output.topk(n_predictions, dim=1)
           predictions = []

           for i in range(n_predictions):
               topv.shape
               topi.shape
               value = topv[0][i].item()
               category_index = topi[0][i].item()
               print('(%.2f) %s' % (value, all_categories[category_index]))
               predictions.append([value, all_categories[category_index]])

       predict('Did you ever hear the tragedy of Darth Plagueis The Wise?')
```

```
> Did you ever hear the tragedy of Darth Plagueis The Wise?
(11.72) english
(4.40) german
(1.27) spanish
(0.92) danish
(0.54) hungarian
```

5 Output Kaggle submission file

Once you have found a good set of hyperparameters submit the output of your model on the Kaggle test file.

[176]:
```
### DO NOT CHANGE KAGGLE SUBMISSION CODE ####
import csv

kaggle_test_file_path = 'language_data/kaggle_rnn_language_classification_test.
 ↪txt'
with open(kaggle_test_file_path, 'r') as f:
    lines = f.readlines()

output_rows = []
for i, line in enumerate(lines):
    sample = line.rstrip()
    sample_chunk_len = len(sample)
    input_data = stringToTensor(sample).unsqueeze(0)
    output = evaluate(rnn, input_data, seq_len=sample_chunk_len, batch_size=1)
    guess_i = categoryFromOutput(output)
    output_rows.append((str(i+1), all_categories[guess_i]))

submission_file_path = 'kaggle_rnn_submission.txt'
with open(submission_file_path, 'w') as f:
    output_rows = [('id', 'category')] + output_rows
    writer = csv.writer(f)
    writer.writerows(output_rows)
```

[ ]:
```
```