Data downloading

```
!wget --no-check-certificate --content-disposition https://raw.githubusercontent.com/Ju

!unzip language_data.zip
!rm language_data.zip
```

import packges

```
!pip install unidecode
```

```
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
```

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

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 "你好" or "γεια σας"), 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

✓ 47s completed at 10:51 AM

```
from unidecode import unidecode as unicodeToAscii

all_characters = string.printable
n_letters = len(all_characters)

print(unicodeToAscii('těší mě'))
```

```
tesi me
```

```
# 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.

```
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
['lithuanian', 'xhosa', 'english', 'romanian', 'portuguese', 'norwegian', 'esperanto'
```

Data processing

```
def categoryFromOutput(output):
    top_n, top_i = output.topk(1, dim=1)
```

```
# 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.
        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
```

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_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.

Train RNN

```
from rnn.model import RNN
```

```
chunk_len = 100

BATCH_SIZE = 100
n_epochs = 2000
hidden_size = 120
n_layers = 4
learning_rate = 0.003
```

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.

```
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          #
    ####################################
    # with torch.no_grad():
    data_tensor = data_tensor.to(device)
    hidden = rnn.init_hidden(batch_size, device=device)
    rnn.zero_grad()
    for c in range(chunk_len):
      # output, hidden = rnn.forward(data_tensor, hidden)
      output, hidden = rnn(data_tensor[:,c], hidden)

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

    return output, loss.item()

```

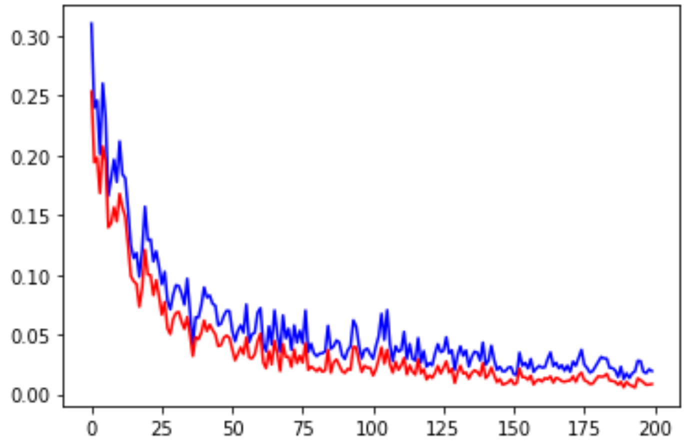
```
def evaluate(rnn, data_tensor, seq_len=chunk_len, batch_size=BATCH_SIZE):
    with torch.no_grad():
        data_tensor = data_tensor.to(device)
        hidden = rnn.init_hidden(batch_size, device=device)
        for i in range(seq_len):
            output, hidden = rnn(data_tensor[:,i], hidden)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

```
    return output, loss.item()
```

```
n_iters = 10000 #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)


number_correct = 0
for iter in range(1, n_iters + 1):
    input_data, target_category, text_data = load_random_batch(train_category_data, c
    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
        print('Train accuracy: {}'.format(float(number_correct)/float(print_every*BAT
        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)
```

```
250 2% (0m 36s) 0.0707 0.0710 i cieli. Quando fa udire la sua voce, v'e un rumor d'ac
Train accuracy: 0.9134
300 3% (0m 43s) 0.1365 0.1173 lkeskare ham. Og se, to blinde sade ved Vejen, og da de
Train accuracy: 0.9244
350 3% (0m 50s) 0.1764 0.1601 a no coracao para fazer por Jerusalem. Nao havia comigo
Train accuracy: 0.938
400 4% (0m 57s) 0.0550 0.0424 t op rad som blir til skam for ditt hus, lagt op rad om
Train accuracy: 0.938
450 4% (1m 5s) 0.1527 0.1085 ju edhe per pak kohe; ecni gjersa keni drite, qe te mos
Train accuracy: 0.9342
500 5% (1m 12s) 0.1983 0.2706 luyordu. Umma, Afek ve Rehov; koyleriyle birlikte yirmi
Train accuracy: 0.9432
550 5% (1m 19s) 0.0934 0.0697 udesta. Missa elattekin, ette saa syoda lintujen etteka
Train accuracy: 0.9304
600 6% (1m 26s) 0.1016 0.1012 igliuoli d'Aaronne, sara unto per succedergli, fara anc
Train accuracy: 0.9436
650 6% (1m 33s) 0.1681 0.1083  forte dig ud af AEgypten, af Traellehuset. Du ma ikke
Train accuracy: 0.9362
700 7% (1m 41s) 0.1030 0.0707 c Gie-ho-va cam dong Gia-ha-xi-en, con trai Xa-cha-ri,
Train accuracy: 0.9498
750 7% (1m 48s) 0.0889 0.0838  murte igjen for mig, sa jeg ikke kan komme ut; han gjo
Train accuracy: 0.958
800 8% (1m 55s) 0.0741 0.0665 ''. Dhe ai veproi keshtu dhe dora e tij iu shendosh si
Train accuracy: 0.9624
850 8% (2m 2s) 0.0855 0.0613 s vel klause: "Ka jai padaryti?" Gehazis atsiliepe: "Ji
Train accuracy: 0.9602
900 9% (2m 9s) 0.0778 0.0592 adu tautai ar karalystei ja statyti ir itvirtinti, bet j
Train accuracy: 0.9718
950 9% (2m 17s) 0.2178 0.1646 dan, viidenkymmenen ja kymmenen paallikoiksi. He ratkoi
Train accuracy: 0.9658
1000 10% (2m 24s) 0.0971 0.0919  porem, aquele, o Espirito da verdade, ele vos guiara
Train accuracy: 0.9528
1050 10% (2m 31s) 0.0731 0.0565 ezzel borita be. Es a rudakat betola az oltar oldalai
Train accuracy: 0.9564
1100 11% (2m 38s) 0.2225 0.1830 es brought Aaron's sons, and clothed them with coats,
Train accuracy: 0.9606
1150 11% (2m 45s) 0.1095 0.0788 vim, ze me uz neuvidi nikdo z vas, k nimz jsem na svy
Train accuracy: 0.9678
1200 12% (2m 52s) 0.1361 0.1039 apenbaret dem det. For hans usynlige vesen, bade hans
Train accuracy: 0.959
1250 12% (3m 0s) 0.0771 0.0758 a unjengaye. Akabuyanga noko uYehova ekuvutheni komsin
Train accuracy: 0.963
1300 13% (3m 7s) 0.0911 0.0694 o, beza kwaYuda naseYerusalem; ngokuba uYarobheham noo
Train accuracy: 0.9714
1350 13% (3m 14s) 0.0945 0.0791 sidererete come impuri. Pero, una fonte o una cistern
Train accuracy: 0.9668
1400 14% (3m 21s) 0.1444 0.1039 apatat astfel fiinta`, -zice Domnul. -,Iata spre cine
Train accuracy: 0.9722
1450 14% (3m 28s) 0.0105 0.0097 uizo. Porque para todo proposito ha tempo e juizo; po
Train accuracy: 0.9784
```

```
[<matplotlib.lines.Line2D at 0x7f8fb35cde50>]
```



Evaluate results

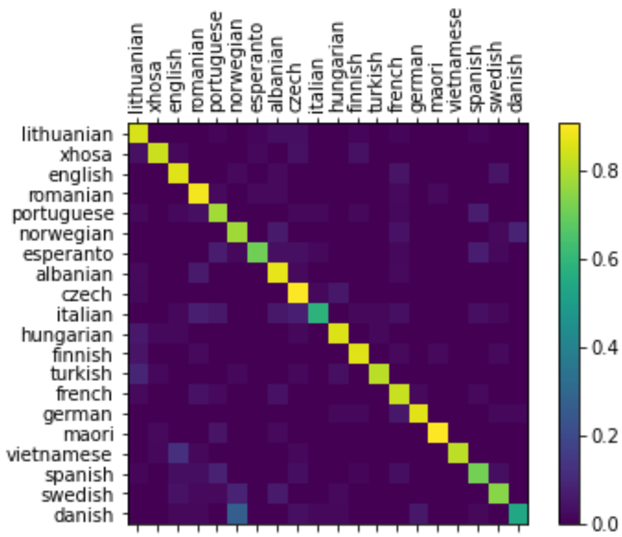
We now vizualize the performance of our model by creating a confusion matrix. The ground truth languages of samples are represented by rows in the matrix while the predicted languages are represented by columns.

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

```
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 o
    input_data, target_category, text_data = load_random_batch(test_category_data, chun
    output = evaluate(rnn, input_data, seq_len=eval_chunk_len, batch_size=eval_batch_si

    guess_i = categoryFromOutput(output)
    category_i = [int(target_category[idx]) for idx in range(len(target_category))]
    for j in range(eval_batch_size):
```

```
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:  0.797
```



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

Run on User Input

Now you can test your model on your own input.

```
def predict(input_line, n_predictions=5):
```

```
        predictions.append([value, all_categories[category_index]])

predict('This is a phrase to test the model on user input')

```

```
> This is a phrase to test the model on user input
(8.24) albanian
(3.16) english
(2.05) french
(-0.90) romanian
(-0.93) hungarian
```

Output Kaggle submission file

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

```
### 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
```