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

```
    Drive already mounted at /content/gdrive; to attempt to forcibly remount, call
```

```
! 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
```

```
    Requirement already satisfied: unidecode in /usr/local/lib/python3.6/dist-pack
```

```
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 characteristics of the individual books rather than the language).

```
from unidecode import unidecode as unicodeToAscii

all_characters = string.printable
```

```
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
['german', 'czech', 'norwegian', 'finnish', 'turkish', 'vietnamese', 'swedish'
```

# Data processing

```
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()
```

```
    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=torc
        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_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.

```
class RNN(nn.Module):
    def __init__(self, input_size, hidden_size, output_size, model_type="rnn", n_la
        super(RNN, self).__init__()
        """
        Initialize the RNN model.

        You should create:
        - An Embedding object which will learn a mapping from tensors
        of dimension input_size to embedding of dimension hidden_size.
        - Your RNN network which takes the embedding as input (use models
        in torch.nn). This network should have input size hidden_size and
        output size hidden_size.
        - A linear layer of dimension hidden_size x output_size which
        will predict output scores.

        Inputs:
        - input_size: Dimension of individual element in input sequence to model
        - hidden_size: Hidden layer dimension of RNN model
        - output_size: Dimension of individual element in output sequence from mode
```

```
        - model_type: RNN network type can be "rnn" (for basic rnn), "gru", or "lst
        - n_layers: number of layers in your RNN network
        """

        self.model_type = model_type
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.output_size = output_size
        self.n_layers = n_layers

        ####################################
        #          YOUR CODE HERE          #
        ####################################

        self.input2hidden  = nn.Embedding(input_size, hidden_size)
        self.hidden2output = nn.Linear(hidden_size, output_size)

        if model_type == "rnn":
            self.rnn = nn.RNN(hidden_size, hidden_size, n_layers)
        elif model_type == "lstm":
            self.rnn = nn.LSTM(hidden_size, hidden_size, n_layers)
        elif model_type == "gru":
            self.rnn = nn.GRU(hidden_size, hidden_size, n_layers)

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



    def forward(self, input, hidden):
        """
        Forward pass through RNN model. Use your Embedding object to create
        an embedded input to your RNN network. You should then use the
        linear layer to get an output of self.output_size.

        Inputs:
        - input: the input data tensor to your model of dimension (batch_size)
        - hidden: the hidden state tensor of dimension (n_layers x batch_size x hid

        Returns:
        - output: the output of your linear layer
        - hidden: the output of the RNN network before your linear layer (hidden st
        """

        output = None

        ####################################
        #          YOUR CODE HERE          #
        ####################################
        batch_size = input.shape[0]
        embedding = self.input2hidden(input).view(1, batch_size, -1)
        # required input size for nn.RNN: seq_len, batch, input_size
        output, hidden = self.rnn(embedding, hidden)
        output = self.hidden2output(output.view(1, batch_size, -1))
        output = output.view(batch_size, -1)
```

```
        ##########       END      ##########
        return output, hidden

    def init_hidden(self, batch_size, device=None):
        """
        Initialize hidden states to all 0s during training.

        Hidden states should be initilized to dimension (n_layers x batch_size x hi

        Inputs:
        - batch_size: batch size

        Returns:
        - hidden: initialized hidden values for input to forward function
        """

        hidden = None

        ####################################
        #          YOUR CODE HERE          #
        ####################################

        if self.model_type == "lstm":
            hidden = (torch.zeros(self.n_layers, batch_size, self.hidden_size).to(d
                      torch.zeros(self.n_layers, batch_size, self.hidden_size).to(dev
        else:
            hidden = torch.zeros(self.n_layers, batch_size, self.hidden_size).to(de

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

        return hidden
```

## Train RNN

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

双击（或按回车键）即可修改

```
chunk_len = 50

BATCH_SIZE = 100
n_epochs = 2000
hidden_size = 100
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_la
```

**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_S
    """
    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, 0

    ####################################
    #          YOUR CODE HERE          #
    ####################################
    batch_size = data_tensor.shape[0]
    optimizer.zero_grad()
    hidden = rnn.init_hidden(batch_size, device = device)

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

    loss += criterion(output, target_tensor.squeeze())

    loss.backward()
    optimizer.step()


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

    return output, loss



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()


n_iters = 10000
print_every = 100
plot_every = 100


# 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=1e-5)


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

```
Train accuracy: 0.9907
7100 71% (14m 32s) 0.0018 0.0018 gloron kaj potencon. Tributu al la Eternulo l
Train accuracy: 0.9912
7200 72% (14m 44s) 0.0444 0.0375 ai mot can va rong mot can. Giua cac phong, c
Train accuracy: 0.9901
7300 73% (14m 55s) 0.0357 0.0317 n fajron. Mi ja vin baptas per akvo por pentc
Train accuracy: 0.9922
7400 74% (15m 8s) 0.0487 0.0445 a pononga, a ma matou e whakaatu tona tikanga.
Train accuracy: 0.9914
7500 75% (15m 20s) 0.0053 0.0051  suoi nuoc. Nham ngay muoi lam thang hai, sau
Train accuracy: 0.9904
7600 76% (15m 32s) 0.0130 0.0125 halakat, es halakat advan, megtore, es ada az
Train accuracy: 0.9898
7700 77% (15m 44s) 0.0233 0.0126  diru al gxi:Vi estas lando ne purigata, ne p
Train accuracy: 0.9928
7800 78% (15m 56s) 0.0418 0.0298 i i mohio ki taua mea huna, mauria ana tetahi
Train accuracy: 0.9932
7900 79% (16m 8s) 0.0091 0.0097 i elinesibini ka-Ahazi ukumkani wakwaYuda,  uH
Train accuracy: 0.9919
8000 80% (16m 21s) 0.0082 0.0061 fruto, e a terra dara a sua novidade, e estar
Train accuracy: 0.992
8100 81% (16m 33s) 0.0434 0.0223 gent, disant chaque jour: Ou est ton Dieu? Pc
Train accuracy: 0.9919
8200 82% (16m 46s) 0.0227 0.0110 Dieve. Viespatie, mano Dieve, teisk mane, vac
Train accuracy: 0.9924
8300 83% (16m 58s) 0.0782 0.0285 don vi venis? cxu ne por kapti kaptajxon vi k
Train accuracy: 0.9911
8400 84% (17m 10s) 0.0144 0.0113 ne time mbi shtepine e Perendise tim, pervec
Train accuracy: 0.9914
8500 85% (17m 22s) 0.0290 0.0180 stele au venit peste tara Egiptului, si s'au
Train accuracy: 0.9928
8600 86% (17m 35s) 0.0115 0.0079 nd with me. Shall evil be recompensed for goc
Train accuracy: 0.9908
8700 87% (17m 47s) 0.0643 0.0426 en birini<br />Ya da ortusu olmayan bir yoksu
Train accuracy: 0.9934
8800 88% (17m 59s) 0.0325 0.0176 tet me veglat e tyre muzikore te kushtuara Zc
Train accuracy: 0.9898
8900 89% (18m 11s) 0.0296 0.0266 : "Ala pelkaa. Mina haluan osoittaa sinulle y
Train accuracy: 0.9913
9000 90% (18m 23s) 0.0045 0.0046  adini verdi. agirligi ve degeri bilinmeyen k
Train accuracy: 0.9926

9100 91% (18m 35s) 0.0592 0.0505 inen Hauch vergehen und ihre Jahre durch plot
Train accuracy: 0.9934
9200 92% (18m 47s) 0.0386 0.0275 vost jak polni kviti. Trava usycha, kvet vadr
Train accuracy: 0.9922
9300 93% (18m 59s) 0.0028 0.0027 g till Absalom -- flydde han till HERRENS tal
Train accuracy: 0.9928
9400 94% (19m 11s) 0.0094 0.0073 u'ils mesurent le plan de cette maison. Quanc
Train accuracy: 0.9929
9500 95% (19m 23s) 0.0019 0.0017 mezi Judove od vychodni strany po zapadni buc
Train accuracy: 0.9917
9600 96% (19m 35s) 0.0070 0.0072 amajai aukai; ozi aukai uz nuodeme; padekos a
Train accuracy: 0.9918
9700 97% (19m 48s) 0.0119 0.0087 chlachtopfer und Brandopfer in unsere Hande g
Train accuracy: 0.9929
9800 98% (20m 0s) 0.0148 0.0104 intea Lui: asteapta -L! Dar, pentru ca minia I
Train accuracy: 0.9919
9900 99% (20m 12s) 0.0049 0.0037 tini. A, ko nga hoiho o Horomona, he mea mau
Train accuracy: 0.9927
```

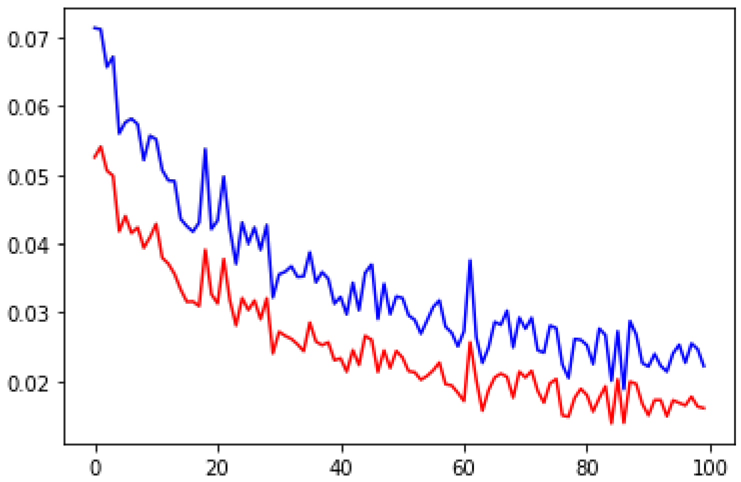
```
10000 100% (30m 35s) 0.0336 0.0289
```

## Save the model

## Plot loss functions

```
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')
```

```
[<matplotlib.lines.Line2D at 0x7f60456d35c0>]
```




## 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.

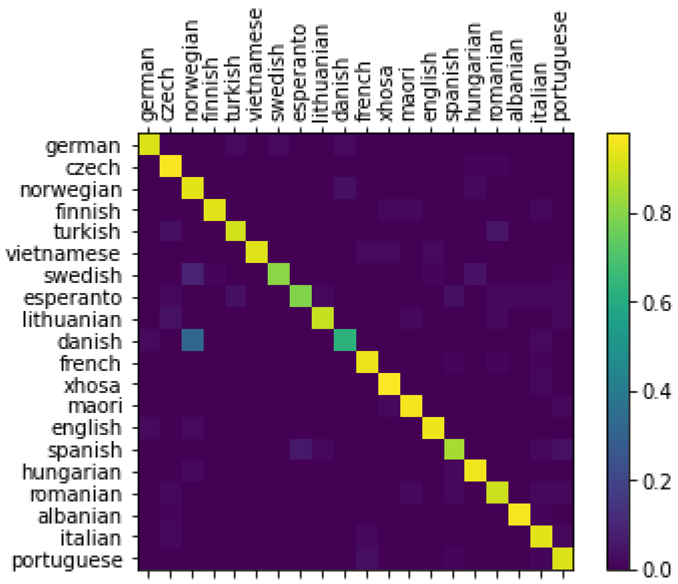
In this evaluation we consider sequences of variable sizes rather than the fixed length sequences we used for training.

```
eval_batch_size = 1  # needs to be set to 1 for evaluating different sequence lengt

# 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 sequenc
    input_data, target_category, text_data = load_random_batch(test_category_data,
```

```
    output = evaluate(rnn, input_data, seq_len=eval_chunk_len, batch_size=eval_batc

    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:  0.907
```




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

```
> This is a phrase to test the model on user input
(14.91) english
(2.18) french
(1.30) swedish
(1.17) spanish
(0.84) german
```

# 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
    writer = csv.writer(f)
    writer.writerows(output_rows)
```