In [1]:
```
from google.colab import drive
drive.mount("/content/gdrive")
import os
os.chdir("/content/gdrive/My Drive/cs747/CS747_Assignment4/")
%ls
import torch
!pip install Unidecode
```

```
Mounted at /content/gdrive
download_language.sh           language_data/
kaggle_rnn_submission.txt      MP4_classification.ipynb
kaggle_rnn_submission_v2.txt   MP4_generation.ipynb
kaggle_rnn_submission_v3.txt   MP4_generation-Part2.ipynb
kaggle_rnn_submission_v4.txt   rnn/
kaggle_rnn_submission_v5.txt   rnn_generator.pth
kaggle_rnn_submission_v6.txt   Spring_2023_CS_747_Deep_Learning_Assignment-4.
pdf
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab
-wheels/public/simple/
Collecting Unidecode
  Downloading Unidecode-1.3.6-py3-none-any.whl (235 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 235.9/235.9 KB 19.3 MB/s eta 0:
00:00
Installing collected packages: Unidecode
Successfully installed Unidecode-1.3.6
```

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

import torch
import torch.nn as nn
print(torch.__version__)
from rnn.helpers import time_since

from torch.optim.lr_scheduler import ReduceLROnPlateau
from torch.optim.lr_scheduler import CosineAnnealingLR

%matplotlib inline
```

```
2.0.0+cu118
```

In [ ]:
```
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).

In [ ]:
```
from unidecode import unidecode as unicodeToAscii

all_characters = string.printable
n_letters = len(all_characters)

print(unicodeToAscii('těší mě'))
```

```
tesi me
```

In [ ]:
```
# 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('_'
        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.

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

Data processing

In [ ]:
```
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)], dty
        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

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

In [ ]:
```
chunk_len = 50

BATCH_SIZE = 250
n_epochs = 2000
hidden_size = 250 #250 #200 #100
n_layers = 1
learning_rate = 0.001
model_type = 'lstm'

criterion = nn.CrossEntropyLoss()
rnn = RNN(n_letters, hidden_size, n_languages, model_type=model_type, n_laye
```

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.

In [ ]:
```
def train(rnn, target_tensor, data_tensor, optimizer, criterion, 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_l
    - 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          #
    ####################################

    hidden = rnn.init_hidden(batch_size, device=device)
    rnn.zero_grad()
    for char in range(chunk_len):
        output, hidden = rnn(data_tensor[:, char], hidden)
        loss += criterion(output.view(batch_size, -1), target_tensor.squeeze

    loss.data = loss.data/chunk_len
    loss.backward()
    torch.nn.utils.clip_grad_norm_(rnn.parameters(), max_norm=100)
    optimizer.step()


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

    return output, loss.item()
```

In [ ]:
```
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()
```

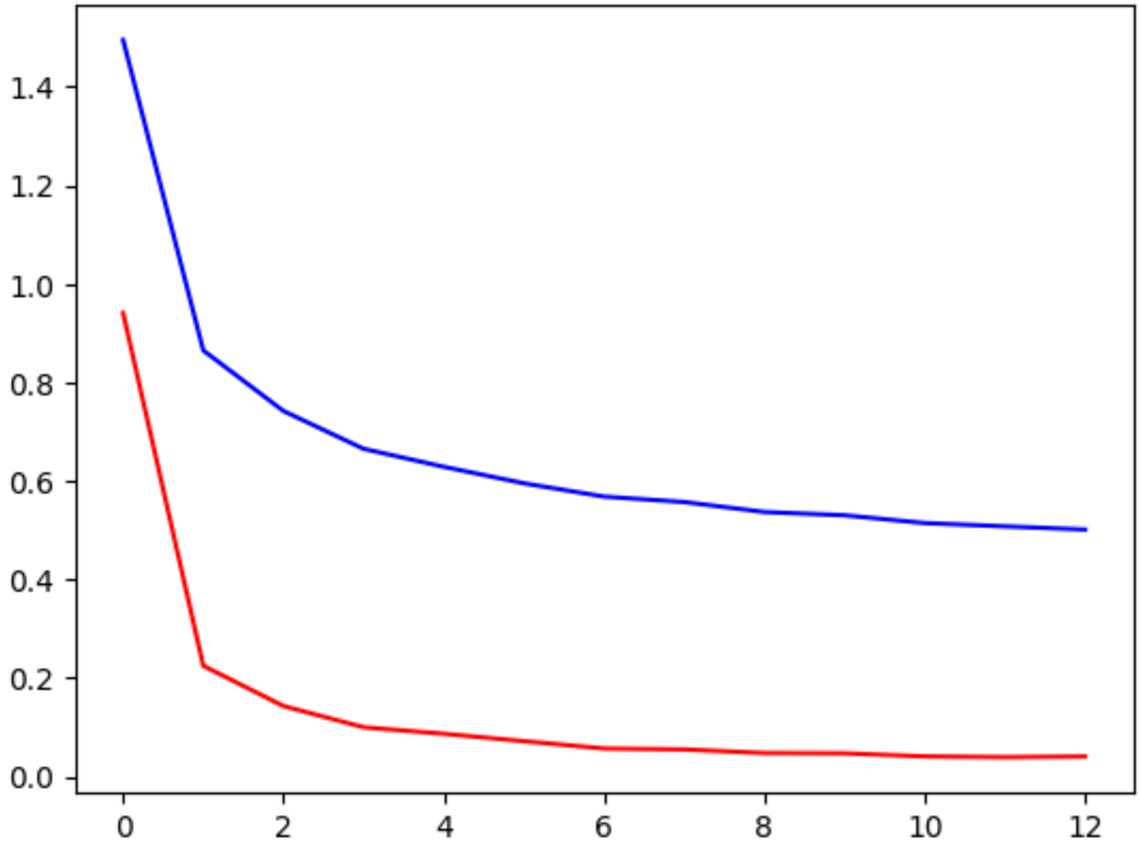
In [ ]:
```
n_iters = 2000 #5000 #2000 #100000
print_every = 150
plot_every = 150


# 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)
#scheduler = ReduceLROnPlateau(optimizer, mode='min', factor=0.5, patience=1
scheduler = CosineAnnealingLR(optimizer, T_max=2000, eta_min=1e-5)

number_correct = 0
for iter in range(1, n_iters + 1):
    input_data, target_category, text_data = load_random_batch(train_categor
    output, loss = train(rnn, target_category, input_data, optimizer, criter
    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().

    # 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 *
        print('Train accuracy: {}'.format(float(number_correct)/float(print_
        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
        scheduler.step(all_test_losses[-1])
```

```
150 7% (0m 24s) 0.9305 0.2809 n hallitsemaan viitta kaupunkia.' "Mutta kun s
eura / finnish ✓
Train accuracy: 0.692
300 15% (0m 49s) 0.7237 0.1473 i uomini. Questo e il sogno che io, il re Neb
ucadn / italian ✓
Train accuracy: 0.9172266666666666
450 22% (1m 13s) 0.6599 0.0983 edig a hir o feloluk a jeruzsalemi gyulekezet
fule / hungarian ✓
Train accuracy: 0.94592
600 30% (1m 38s) 0.7154 0.1299 r, kaj kia estas via peto? se tio estas ecx d
uono  / esperanto ✓
Train accuracy: 0.9632
750 37% (2m 2s) 0.6659 0.1079 ins in the cities of Judah. Princes were hange
d up / english ✓
Train accuracy: 0.96592
900 45% (2m 27s) 0.6037 0.1045 sok, Emoreusok, Perizeusok, Jebuzeusok, Girga
zeuso / hungarian ✓
Train accuracy: 0.9722933333333333
1050 52% (2m 51s) 0.5174 0.0379 or vosotros, oh casa de Israel, sino por cau
sa de  / spanish ✓
Train accuracy: 0.978
1200 60% (3m 16s) 0.5563 0.0955  un ied, si a trimes lui Saul aceste lucruri
, prin / romanian ✓
Train accuracy: 0.9783733333333333
1350 67% (3m 40s) 0.5110 0.0319 r negali paskui mane sekti, bet veliau nusek
si man / lithuanian ✓
Train accuracy: 0.9808
1500 75% (4m 5s) 0.5056 0.0278  isalleen: "Ala vihastu, herrani, vaikka en v
oikaa / finnish ✓
Train accuracy: 0.9814666666666667
1650 82% (4m 29s) 0.5159 0.0354  da Mara e giunsero ad Elim; ad Elim c'erano
dodic / italian ✓
Train accuracy: 0.98424
1800 90% (4m 54s) 0.5344 0.0496  hoki he tangata hei tinei. P Ko te kupu i k
itea e / maori ✓
Train accuracy: 0.9844533333333333
1950 97% (5m 18s) 0.5177 0.0254 wiri, tenei tiakanga i te ritenga mai o tene
i tiak / maori ✓
Train accuracy: 0.98416
```

Plot loss functions

In [ ]:
```python
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')
```

Out[ ]:
```
[<matplotlib.lines.Line2D at 0x7f3d4938c0d0>]
```



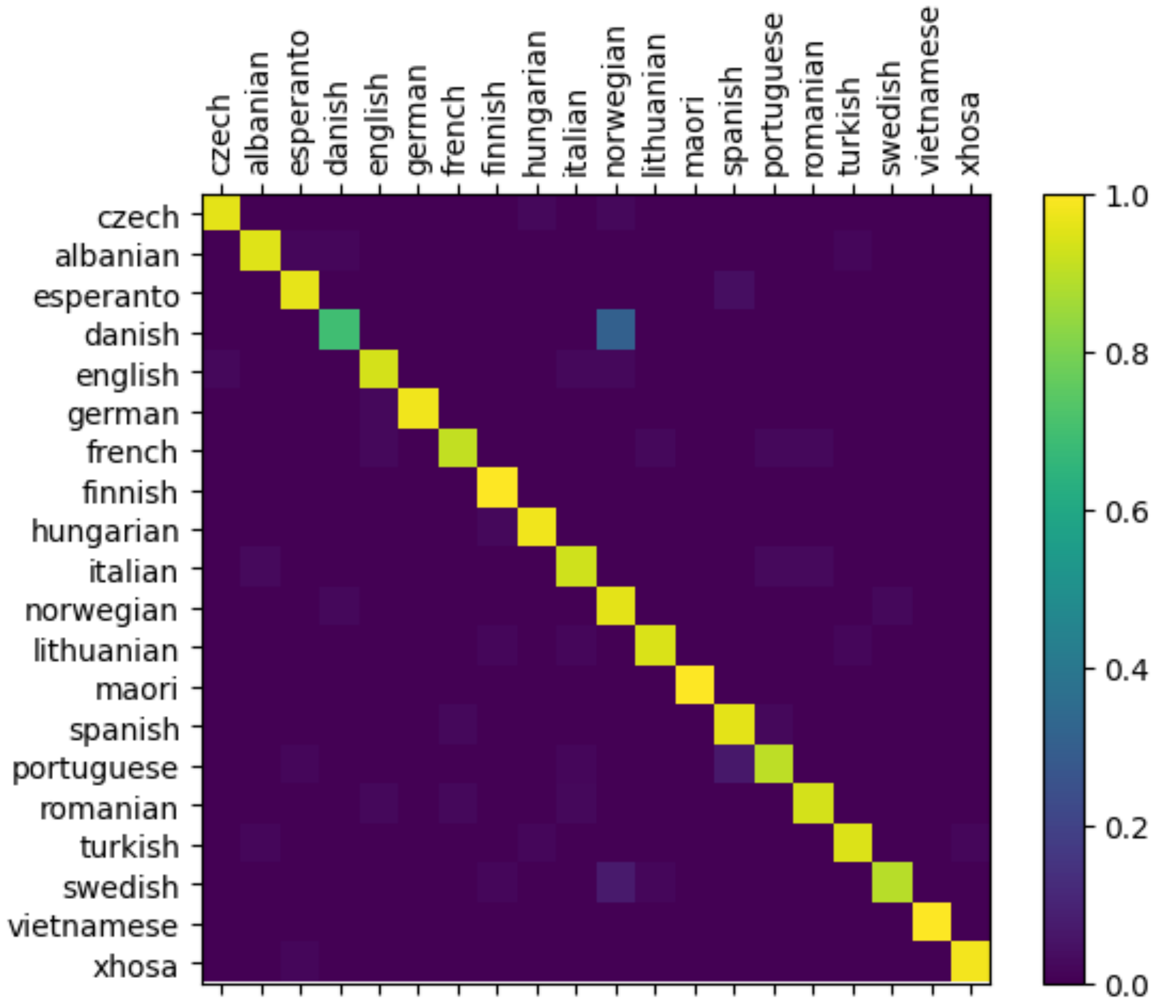
Evaluate results

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.

In [ ]:
```
eval_batch_size = 1  # needs to be set to 1 for evaluating different sequenc

# 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
    input_data, target_category, text_data = load_random_batch(test_category
    output = evaluate(rnn, input_data, seq_len=eval_chunk_len, batch_size=ev

    guess_i = categoryFromOutput(output)
    category_i = [int(target_category[idx]) for idx in range(len(target_cate
    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_siz

# 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:  0.939
<ipython-input-33-4480673938c5>:35: UserWarning: FixedFormatter should only
be used together with FixedLocator
  ax.set_xticklabels([''] + all_categories, rotation=90)
<ipython-input-33-4480673938c5>:36: UserWarning: FixedFormatter should only
be used together with FixedLocator
  ax.set_yticklabels([''] + all_categories)
```



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.

In [ ]:
```
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(devic
        output = evaluate(rnn, input_data, seq_len=len(input_line), batch_si

    # 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
(8.98) english
(0.93) french
(0.61) czech
(0.57) german
(0.46) swedish
```

Output Kaggle submission file

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

In [ ]:

```
### DO NOT CHANGE KAGGLE SUBMISSION CODE ####
import csv

kaggle_test_file_path = 'language_data/kaggle_rnn_language_classification_te
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=
    guess_i = categoryFromOutput(output)
    output_rows.append((str(i+1), all_categories[guess_i]))

submission_file_path = 'kaggle_rnn_submission_v6.txt'
with open(submission_file_path, 'w') as f:
    output_rows = [('id', 'category')] + output_rows
    writer = csv.writer(f)
    writer.writerows(output_rows)
```