In [1]:

```
from __future__ import unicode_literals, print_function, division
from io import open
import glob
import os

def findFiles(path): return glob.glob(path)
```

In [2]:

```
print(findFiles('data/names/*.txt'))
# %%

import unicodedata
import string

all_letters = string.ascii_letters + " .,;'"
n_letters = len(all_letters)

# Turn a Unicode string to plain ASCII, thanks to https://stackoverflow.co
def unicodeToAscii(s):
    return ''.join(
        c for c in unicodedata.normalize('NFD', s)
        if unicodedata.category(c) != 'Mn'
        and c in all_letters
    )

print(unicodeToAscii('Ślusàrski'))
```

```
['data/names/Czech.txt', 'data/names/German.txt', 'data/names/Arabic.txt',
'data/names/Japanese.txt', 'data/names/Chinese.txt', 'data/names/Vietnamese
.txt', 'data/names/Russian.txt', 'data/names/French.txt', 'data/names/Irish
.txt', 'data/names/English.txt', 'data/names/Spanish.txt', 'data/names/Gree
k.txt', 'data/names/Italian.txt', 'data/names/Portuguese.txt', 'data/names/
Scottish.txt', 'data/names/Dutch.txt', 'data/names/Korean.txt', 'data/names
/Polish.txt']
Slusarski
```

In [3]:

```
# %%
# Build the names dictionary, a list of names per language
# di tionary keys are languages, values are names
names = {}
languages = []

# Read a file and split into lines
def readLines(filename):
    lines = open(filename, encoding='utf-8').read().strip().split('\n')
    return [unicodeToAscii(line) for line in lines]
```

In [4]:

```
# PLEASE UPDATE THE FILE PATH BELOW FOR YOUR SYSTEM
for filename in findFiles('data/names/*.txt'):
    category = os.path.splitext(os.path.basename(filename))[0]
    languages.append(category)
    lines = readLines(filename)
    names[category] = lines

n_categories = len(languages)

def findName(dict, name):
    keys = dict.keys()
    for key in keys:
        if name in dict[key]:
            return key
    return ''
```

In [5]:

```
import torch

# Find letter index from all_letters, e.g. "a" = 0
def letterToIndex(letter):
    return all_letters.find(letter)

# Just for demonstration, turn a letter into a <1 x n_letters> Tensor
def letterToTensor(letter):
    tensor = torch.zeros(1, n_letters)
    tensor[0][letterToIndex(letter)] = 1
    return tensor

# Turn a line into a <line_length x 1 x n_letters>,
# or an array of one-hot letter vectors
def nameToTensor(name):
    tensor = torch.zeros(len(name), 1, n_letters)
    for li, letter in enumerate(name):
        tensor[li][0][letterToIndex(letter)] = 1
    return tensor
```

In [6]:

```
import torch.nn as nn

class RNN(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(RNN, self).__init__()

        self.hidden_size = hidden_size
        self.i2h = nn.Linear(input_size + hidden_size, hidden_size)
        self.i2o = nn.Linear(input_size + hidden_size, output_size)
        self.softmax = nn.LogSoftmax(dim=1)

    def forward(self, input, hidden):
        combined = torch.cat((input, hidden), 1)
        hidden = self.i2h(combined)
        output = self.i2o(combined)
        output = self.softmax(output)
        return output, hidden
```

```
    def initHidden(self):
        return torch.zeros(1, self.hidden_size)

#n_hidden = 128
n_hidden = 2
rnn = RNN(n_letters, n_hidden, n_categories)
input = letterToTensor('A')
hidden = torch.zeros(1, n_hidden)
output, next_hidden = rnn(input, hidden)

# For the sake of efficiency we don't want to be creating a new Tensor for
# every step, so we will use ``nameToTensor`` instead of
# ``letterToTensor`` and use slices. This could be further optimized by
# pre-computing batches of Tensors.
#

input = nameToTensor('Albert')
hidden = torch.zeros(1, n_hidden)

output, next_hidden = rnn(input[0], hidden)
print(output)
def categoryFromOutput(output):
    # compute max
    top_n, top_i = output.topk(1)
    # output index of max
    category_i = top_i.item()
    return languages[category_i], category_i


import random

def randomChoice(l):
    return l[random.randint(0, len(l) - 1)]

def randomTrainingExample():
    category = randomChoice(languages)
    name = randomChoice(names[category])
    category_tensor = torch.tensor([languages.index(category)], dtype=torcl
    name_tensor = nameToTensor(name)
    return category, name, category_tensor, name_tensor

for i in range(10):
    category, name, category_tensor, name_tensor = randomTrainingExample()
```

```
tensor([[-2.7200, -2.8972, -2.7771, -2.9742, -2.8518, -3.0428, -2.8471, -2.
8535,
         -2.7805, -2.9748, -3.0088, -2.8572, -2.8798, -2.8693, -2.8837, -3.
0511,
         -2.7972, -3.0451]], grad_fn=<LogSoftmaxBackward0>)
```

In [7]:

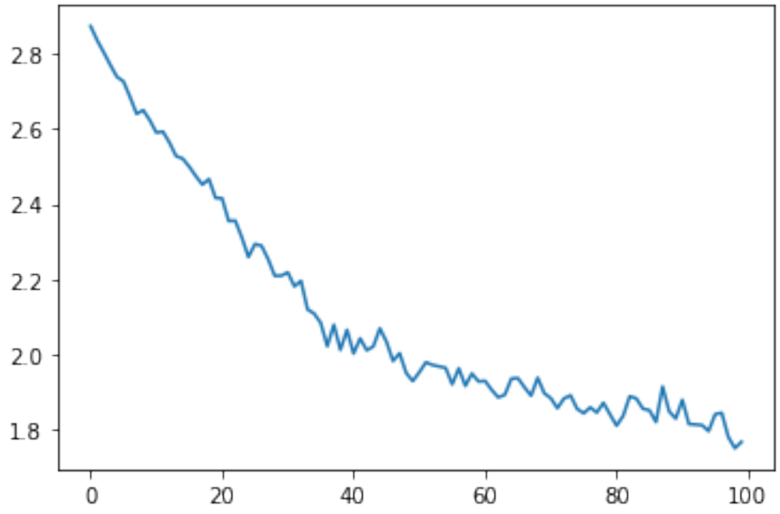
```
criterion = nn.NLLLoss()
learning_rate = 0.005 # For this example, we keep the learning rate fixed
def train(category_tensor, name_tensor):
    # initialize hidden state - do this every time before passing an input
    hidden = rnn.initHidden()
    # reset grad counters - do this every time after backprop
    rnn.zero_grad()
    # manually go through each element in input sequence
    for i in range(name_tensor.size()[0]):
        output, hidden = rnn(name_tensor[i], hidden)
    # backpropagate based on loss at last element only
    loss = criterion(output, category_tensor)
    loss.backward()

    # Update network parameters
    for p in rnn.parameters():
        p.data.add_(-learning_rate, p.grad.data)

    return output, loss.item()
import time
import math

n_iters = 100000
print_every = 5000
plot_every = 1000

# Keep track of loss for plotting
current_loss = 0
all_losses = []
def timeSince(since):
    now = time.time()
    s = now - since
    m = math.floor(s / 60)
    s -= m * 60
    return '%dm %ds' % (m, s)

start = time.time()

for iter in range(1, n_iters + 1):
    category, name, category_tensor, name_tensor = randomTrainingExample()
    output, loss = train(category_tensor, name_tensor)
    current_loss += loss

    # Print iter number, loss, name and guess
    if iter % print_every == 0:
        guess, guess_i = categoryFromOutput(output)
        correct = '✓' if guess == category else '✗ (%s)' % category
        print('%d %d%% (%s) %.4f %s / %s %s' % (iter, iter / n_iters * 100

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

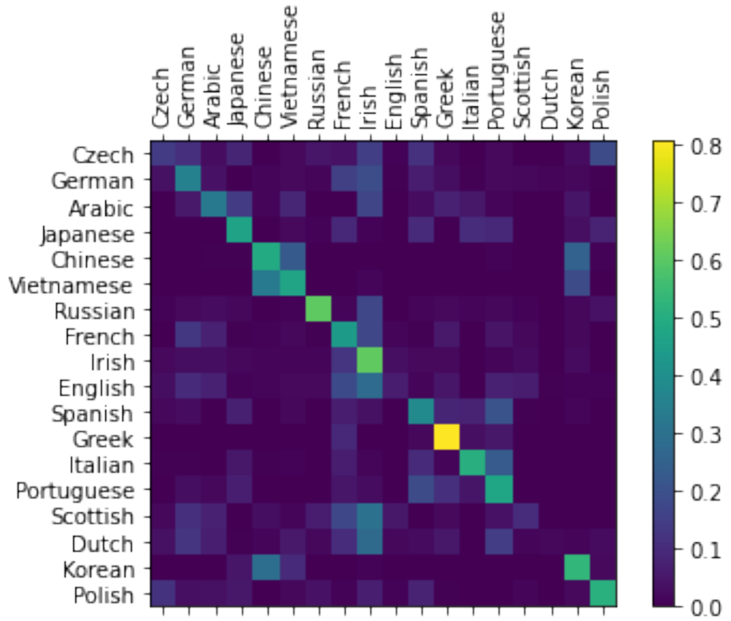
```
<ipython-input-7-90e0b1012043>:17: UserWarning: This overload of add_ is de
precated:
        add_(Number alpha, Tensor other)
Consider using one of the following signatures instead:
        add_(Tensor other, *, Number alpha) (Triggered internally at  ../to
rch/csrc/utils/python_arg_parser.cpp:1050.)
  p.data.add_(-learning_rate, p.grad.data)
5000 5% (0m 8s) 2.5462 Waxweiler / German ✓
10000 10% (0m 14s) 2.7613 Pho / Portuguese ✗ (Vietnamese)
15000 15% (0m 17s) 2.7325 Klein / Irish ✗ (Dutch)
20000 20% (0m 23s) 1.1500 Bakirov / Russian ✓
25000 25% (0m 26s) 2.4674 Stumpf / English ✗ (German)
30000 30% (0m 33s) 3.4384 Piazza / Japanese ✗ (Italian)
35000 35% (0m 36s) 1.6667 Lemaire / French ✓
40000 40% (0m 39s) 1.5559 Nisi / Italian ✓
45000 45% (0m 43s) 1.9593 Tailler / German ✗ (French)
50000 50% (0m 46s) 1.6715 Mclean / Scottish ✓
55000 55% (0m 49s) 2.1529 Russell / German ✗ (Scottish)
60000 60% (0m 52s) 0.5148 Kyritsis / Greek ✓
65000 65% (0m 56s) 1.3683 Ballaltick / Polish ✗ (Czech)
70000 70% (0m 59s) 1.8015 Ha / Japanese ✗ (Korean)
75000 75% (1m 2s) 1.1274 Werner / German ✓
80000 80% (1m 5s) 3.0836 Uerling / Scottish ✗ (Czech)
85000 85% (1m 9s) 0.8249 Shui / Chinese ✓
90000 90% (1m 12s) 1.9042 Murray / Irish ✗ (Scottish)
95000 95% (1m 15s) 0.3380 Poplawski / Polish ✓
100000 100% (1m 18s) 2.0589 Nguyen / Irish ✗ (Vietnamese)
```

In [8]:

```
import matplotlib.pyplot as plt
import matplotlib.ticker as ticker

plt.figure()
plt.plot(all_losses)
confusion = torch.zeros(n_categories, n_categories)
n_confusion = 20000

# return an output given an input name
def evaluate(name_tensor):
    hidden = rnn.initHidden()

    for i in range(name_tensor.size()[0]):
        output, hidden = rnn(name_tensor[i], hidden)

    return output

# Go through a bunch of examples and record which are correctly guessed
for i in range(n_confusion):
    category, name, category_tensor, name_tensor = randomTrainingExample()
    output = evaluate(name_tensor)
    guess, guess_i = categoryFromOutput(output)
    category_i = languages.index(category)
    confusion[category_i][guess_i] += 1

accuracy = sum(confusion.diag())/sum(sum(confusion))
print('Accuracy is %f' % accuracy.item())

for i in range(n_categories):
    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([''] + languages, rotation=90)
ax.set_yticklabels([''] + languages)

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

# sphinx_gallery_thumbnail_number = 2
plt.show()
```

```
Accuracy is 0.404650
<ipython-input-8-e82daa84c18a>:39: UserWarning: FixedFormatter should only
be used together with FixedLocator
  ax.set_xticklabels([''] + languages, rotation=90)
<ipython-input-8-e82daa84c18a>:40: UserWarning: FixedFormatter should only
be used together with FixedLocator
  ax.set_yticklabels([''] + languages)
```





In [9]:

```
class RNN(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(RNN, self).__init__()

        self.hidden_size = hidden_size
        self.i2h = nn.Linear(input_size + hidden_size, hidden_size)
        self.i2o = nn.Linear(input_size + hidden_size, output_size)
        self.softmax = nn.LogSoftmax(dim=1)

    def forward(self, input, hidden):
        combined = torch.cat((input, hidden), 1)
        hidden = self.i2h(combined)
        output = self.i2o(combined)
        output = self.softmax(output)
```

```
        return output, hidden

    def initHidden(self):
        return torch.zeros(1, self.hidden_size)

#n_hidden = 128
n_hidden = 8
rnn = RNN(n_letters, n_hidden, n_categories)
input = letterToTensor('A')
hidden = torch.zeros(1, n_hidden)
output, next_hidden = rnn(input, hidden)

# For the sake of efficiency we don't want to be creating a new Tensor for
# every step, so we will use ``nameToTensor`` instead of
# ``letterToTensor`` and use slices. This could be further optimized by
# pre-computing batches of Tensors.
#

input = nameToTensor('Albert')
hidden = torch.zeros(1, n_hidden)

output, next_hidden = rnn(input[0], hidden)
print(output)
def categoryFromOutput(output):
    # compute max
    top_n, top_i = output.topk(1)
    # output index of max
    category_i = top_i.item()
    return languages[category_i], category_i

print(categoryFromOutput(output))

import random

def randomChoice(l):
    return l[random.randint(0, len(l) - 1)]

def randomTrainingExample():
    category = randomChoice(languages)
    name = randomChoice(names[category])
    category_tensor = torch.tensor([languages.index(category)], dtype=torcl
    name_tensor = nameToTensor(name)
    return category, name, category_tensor, name_tensor

for i in range(10):
    category, name, category_tensor, name_tensor = randomTrainingExample()
    print('category =', category, '/ name =', name)

criterion = nn.NLLLoss()
learning_rate = 0.005 # For this example, we keep the learning rate fixed
def train(category_tensor, name_tensor):
    # initialize hidden state - do this every time before passing an input
    hidden = rnn.initHidden()
    # reset grad counters - do this every time after backprop
    rnn.zero_grad()
    # manually go through each element in input sequence
    for i in range(name_tensor.size()[0]):
```

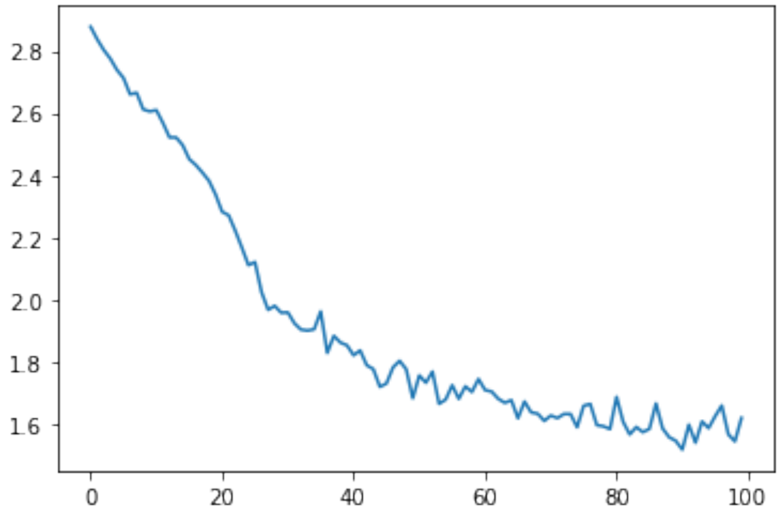
```
        output, hidden = rnn(name_tensor[i], hidden)
    # backpropagate based on loss at last element only
    loss = criterion(output, category_tensor)
    loss.backward()

    # Update network parameters
    for p in rnn.parameters():
        p.data.add_(-learning_rate, p.grad.data)

    return output, loss.item()
import time
import math

n_iters = 100000
print_every = 5000
plot_every = 1000

# Keep track of loss for plotting
current_loss = 0
all_losses = []
def timeSince(since):
    now = time.time()
    s = now - since
    m = math.floor(s / 60)
    s -= m * 60
    return '%dm %ds' % (m, s)

start = time.time()

for iter in range(1, n_iters + 1):
    category, name, category_tensor, name_tensor = randomTrainingExample()
    output, loss = train(category_tensor, name_tensor)
    current_loss += loss

    # Print iter number, loss, name and guess
    if iter % print_every == 0:
        guess, guess_i = categoryFromOutput(output)
        correct = '✓' if guess == category else '✗ (%s)' % category
        print('%d %d%% (%s) %.4f %s / %s %s' % (iter, iter / n_iters * 100

    # Add current loss avg to list of losses
    if iter % plot_every == 0:
        all_losses.append(current_loss / plot_every)
        current_loss = 0
plt.figure()
plt.plot(all_losses)
confusion = torch.zeros(n_categories, n_categories)
n_confusion = 20000

# return an output given an input name
def evaluate(name_tensor):
    hidden = rnn.initHidden()

    for i in range(name_tensor.size()[0]):
        output, hidden = rnn(name_tensor[i], hidden)

    return output
```

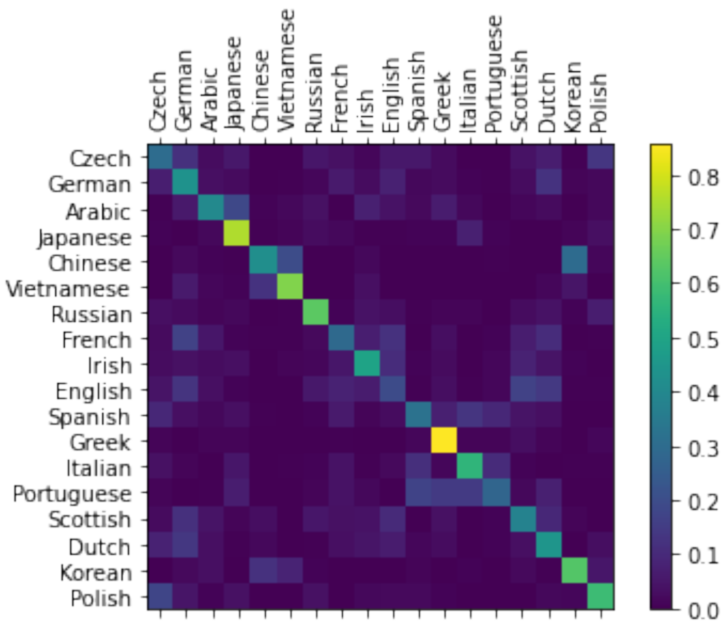
```
# Go through a bunch of examples and record which are correctly guessed
for i in range(n_confusion):
    category, name, category_tensor, name_tensor = randomTrainingExample()
    output = evaluate(name_tensor)
    guess, guess_i = categoryFromOutput(output)
    category_i = languages.index(category)
    confusion[category_i][guess_i] += 1

accuracy = sum(confusion.diag())/sum(sum(confusion))
print('Accuracy is %f' % accuracy.item())

for i in range(n_categories):
    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([''] + languages, rotation=90)
ax.set_yticklabels([''] + languages)

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

# sphinx_gallery_thumbnail_number = 2
plt.show()
```

```
tensor([[-2.9420, -2.8126, -2.7924, -2.7428, -2.9492, -2.8277, -2.9743, -3.
0314,
         -2.9307, -2.9263, -2.8868, -2.8546, -2.7532, -2.9134, -3.0626, -2.
9595,
         -2.8408, -2.8935]], grad_fn=<LogSoftmaxBackward0>)
('Japanese', 3)
category = Chinese / name = Wan
category = Italian / name = Bellomi
category = Arabic / name = Botros
category = Chinese / name = Nie
category = Irish / name = Raghailligh
category = Japanese / name = Shioya
category = Dutch / name = Peter
category = Portuguese / name = Mata
category = Greek / name = Nikolaou
category = Korean / name = Chang
5000 5% (0m 3s) 2.7188 Bakhorin / Irish ✗ (Russian)
10000 10% (0m 6s) 3.1222 Avis / Greek ✗ (English)
15000 15% (0m 9s) 2.8496 An / Irish ✗ (Vietnamese)
20000 20% (0m 13s) 2.1280 Roche / French ✓
25000 25% (0m 16s) 0.7645 Manos / Greek ✓
30000 30% (0m 19s) 0.4376 Georgeakopoulos / Greek ✓
35000 35% (0m 23s) 2.1268 Martel / French ✓
40000 40% (0m 26s) 2.9956 Franke / Polish ✗ (German)
45000 45% (0m 29s) 1.1889 Landi / Italian ✓
50000 50% (0m 32s) 2.8799 Okanao / Portuguese ✗ (Japanese)
```

```
55000 55% (0m 36s) 1.0919 Mooren / Dutch ✓
60000 60% (0m 39s) 1.1959 Do / Vietnamese ✓
65000 65% (0m 42s) 1.8284 Durante / French ✗ (Italian)
70000 70% (0m 45s) 2.1992 Plastow / Scottish ✗ (English)
75000 75% (0m 49s) 1.0511 Mai / Chinese ✗ (Vietnamese)
80000 80% (0m 52s) 1.2557 Aswad / Arabic ✓
85000 85% (0m 55s) 2.2190 Watt / German ✗ (Scottish)
90000 90% (0m 59s) 3.6380 Marqueringh / Irish ✗ (Dutch)
95000 95% (1m 2s) 0.6968 Hofmeister / German ✓
100000 100% (1m 5s) 2.8142 Oshin / Irish ✗ (Japanese)
Accuracy is 0.485300
```

```
<ipython-input-9-306d6f3898fa>:151: UserWarning: FixedFormatter should only
be used together with FixedLocator
  ax.set_xticklabels([''] + languages, rotation=90)
<ipython-input-9-306d6f3898fa>:152: UserWarning: FixedFormatter should only
be used together with FixedLocator
  ax.set_yticklabels([''] + languages)
```





In [10]:

```
class RNN(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(RNN, self).__init__()
```

```
        self.hidden_size = hidden_size
        self.i2h = nn.Linear(input_size + hidden_size, hidden_size)
        self.i2o = nn.Linear(input_size + hidden_size, output_size)
        self.softmax = nn.LogSoftmax(dim=1)

    def forward(self, input, hidden):
        combined = torch.cat((input, hidden), 1)
        hidden = self.i2h(combined)
        output = self.i2o(combined)
        output = self.softmax(output)
        return output, hidden

    def initHidden(self):
        return torch.zeros(1, self.hidden_size)

#n_hidden = 128
n_hidden = 32
rnn = RNN(n_letters, n_hidden, n_categories)
input = letterToTensor('A')
hidden = torch.zeros(1, n_hidden)
output, next_hidden = rnn(input, hidden)

# For the sake of efficiency we don't want to be creating a new Tensor for
# every step, so we will use ``nameToTensor`` instead of
# ``letterToTensor`` and use slices. This could be further optimized by
# pre-computing batches of Tensors.
#

input = nameToTensor('Albert')
hidden = torch.zeros(1, n_hidden)

output, next_hidden = rnn(input[0], hidden)
print(output)
def categoryFromOutput(output):
    # compute max
    top_n, top_i = output.topk(1)
    # output index of max
    category_i = top_i.item()
    return languages[category_i], category_i

print(categoryFromOutput(output))

import random

def randomChoice(l):
    return l[random.randint(0, len(l) - 1)]

def randomTrainingExample():
    category = randomChoice(languages)
    name = randomChoice(names[category])
    category_tensor = torch.tensor([languages.index(category)], dtype=torcl
    name_tensor = nameToTensor(name)
    return category, name, category_tensor, name_tensor

for i in range(10):
    category, name, category_tensor, name_tensor = randomTrainingExample()
    print('category =', category, '/ name =', name)
```

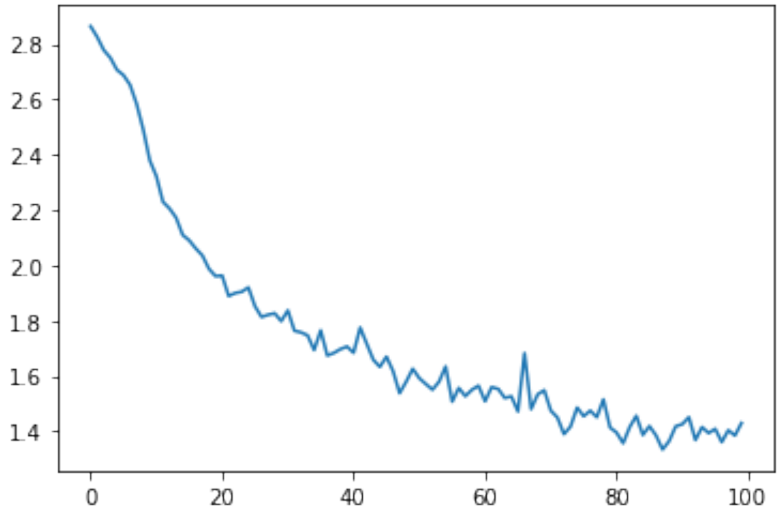
```
criterion = nn.NLLLoss()
learning_rate = 0.005 # For this example, we keep the learning rate fixed
def train(category_tensor, name_tensor):
    # initialize hidden state - do this every time before passing an input
    hidden = rnn.initHidden()
    # reset grad counters - do this every time after backprop
    rnn.zero_grad()
    # manually go through each element in input sequence
    for i in range(name_tensor.size()[0]):
        output, hidden = rnn(name_tensor[i], hidden)
    # backpropagate based on loss at last element only
    loss = criterion(output, category_tensor)
    loss.backward()

    # Update network parameters
    for p in rnn.parameters():
        p.data.add_(-learning_rate, p.grad.data)

    return output, loss.item()
import time
import math

n_iters = 100000
print_every = 5000
plot_every = 1000

# Keep track of loss for plotting
current_loss = 0
all_losses = []
def timeSince(since):
    now = time.time()
    s = now - since
    m = math.floor(s / 60)
    s -= m * 60
    return '%dm %ds' % (m, s)

start = time.time()

for iter in range(1, n_iters + 1):
    category, name, category_tensor, name_tensor = randomTrainingExample()
    output, loss = train(category_tensor, name_tensor)
    current_loss += loss

    # Print iter number, loss, name and guess
    if iter % print_every == 0:
        guess, guess_i = categoryFromOutput(output)
        correct = '✓' if guess == category else '✗ (%s)' % category
        print('%d %d%% (%s) %.4f %s / %s %s' % (iter, iter / n_iters * 100

    # Add current loss avg to list of losses
    if iter % plot_every == 0:
        all_losses.append(current_loss / plot_every)
        current_loss = 0
plt.figure()
plt.plot(all_losses)
confusion = torch.zeros(n_categories, n_categories)
```

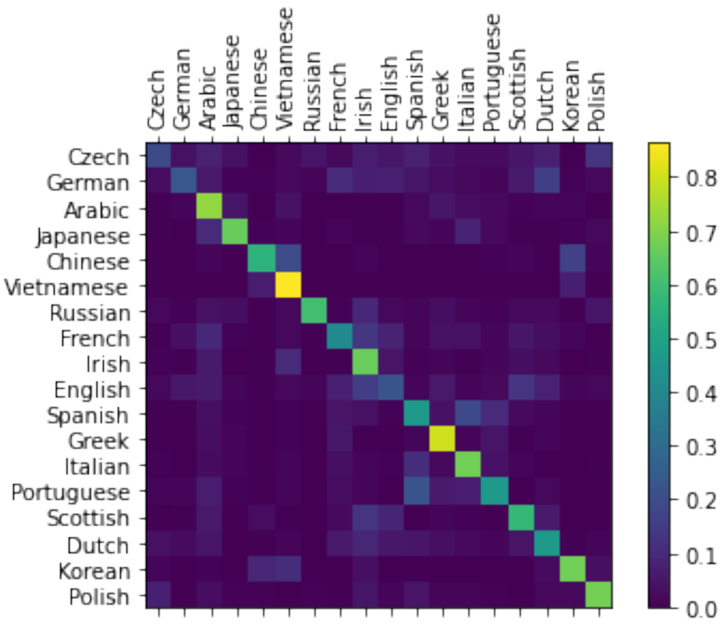
```
n_confusion = 20000

# return an output given an input name
def evaluate(name_tensor):
    hidden = rnn.initHidden()

    for i in range(name_tensor.size()[0]):
        output, hidden = rnn(name_tensor[i], hidden)

    return output

# Go through a bunch of examples and record which are correctly guessed
for i in range(n_confusion):
    category, name, category_tensor, name_tensor = randomTrainingExample()
    output = evaluate(name_tensor)
    guess, guess_i = categoryFromOutput(output)
    category_i = languages.index(category)
    confusion[category_i][guess_i] += 1

accuracy = sum(confusion.diag())/sum(sum(confusion))
print('Accuracy is %f' % accuracy.item())

for i in range(n_categories):
    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([''] + languages, rotation=90)
ax.set_yticklabels([''] + languages)

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

# sphinx_gallery_thumbnail_number = 2
plt.show()
```

```
tensor([[-2.8541, -2.8732, -2.7899, -2.7970, -2.9860, -2.9214, -2.9843, -2.
7860,
         -2.9346, -3.0470, -2.8497, -2.8526, -2.8048, -2.9888, -2.7874, -2.
9631,
         -2.9571, -2.9078]], grad_fn=<LogSoftmaxBackward0>)
('French', 7)
category = French / name = Delacroix
category = French / name = Roux
category = Portuguese / name = Gouveia
category = French / name = Gardinier
category = Chinese / name = Mai
category = German / name = Weiman
category = Irish / name = Sioda
category = Irish / name = Sluaghadhan
category = Polish / name = Andrysiak
category = English / name = Lynes
```

```
5000 5% (0m 3s) 3.0100 Weiss / Greek ✗ (Czech)
10000 10% (0m 6s) 2.6013 Rosales / Greek ✗ (Spanish)
15000 15% (0m 9s) 2.4169 Brodeur / Dutch ✗ (French)
20000 20% (0m 13s) 2.4064 Echevarria / Greek ✗ (Spanish)
25000 25% (0m 17s) 0.7115 Zhang / Chinese ✓
30000 30% (0m 21s) 0.7864 Gui / Chinese ✓
35000 35% (0m 24s) 0.3620 Takeshita / Japanese ✓
40000 40% (0m 27s) 2.6238 Russell / German ✗ (Scottish)
45000 45% (0m 31s) 1.4158 Parent / French ✓
50000 50% (0m 34s) 1.7263 Bellerose / German ✗ (French)
55000 55% (0m 37s) 4.7118 Botros / Greek ✗ (Arabic)
60000 60% (0m 41s) 0.3451 Ferreiro / Portuguese ✓
65000 65% (0m 44s) 1.4327 Antwerp / Dutch ✓
70000 70% (0m 47s) 2.9333 Mayer / Arabic ✗ (Czech)
75000 75% (0m 51s) 1.1464 Simpson / Scottish ✓
80000 80% (0m 54s) 2.5086 Mulder / German ✗ (Dutch)
85000 85% (0m 57s) 1.2301 Pyhtin / Russian ✓
90000 90% (1m 1s) 0.6724 Jo / Korean ✓
95000 95% (1m 4s) 3.1572 Stegon / English ✗ (Czech)
100000 100% (1m 7s) 0.1630 Egonidis / Greek ✓
Accuracy is 0.554000
```

```
<ipython-input-10-09b0b499240b>:151: UserWarning: FixedFormatter should onl
y be used together with FixedLocator
  ax.set_xticklabels([''] + languages, rotation=90)
<ipython-input-10-09b0b499240b>:152: UserWarning: FixedFormatter should onl
y be used together with FixedLocator
  ax.set_yticklabels([''] + languages)
```





In [11]:

```
class RNN(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(RNN, self).__init__()

        self.hidden_size = hidden_size
        self.i2h = nn.Linear(input_size + hidden_size, hidden_size)
        self.i2o = nn.Linear(input_size + hidden_size, output_size)
        self.softmax = nn.LogSoftmax(dim=1)

    def forward(self, input, hidden):
        combined = torch.cat((input, hidden), 1)
        hidden = self.i2h(combined)
        output = self.i2o(combined)
        output = self.softmax(output)
        return output, hidden

    def initHidden(self):
        return torch.zeros(1, self.hidden_size)

#n_hidden = 128
n_hidden = 2
rnn = RNN(n_letters, n_hidden, n_categories)
input = letterToTensor('A')
hidden = torch.zeros(1, n_hidden)
output, next_hidden = rnn(input, hidden)

# For the sake of efficiency we don't want to be creating a new Tensor for
# every step, so we will use ``nameToTensor`` instead of
# ``letterToTensor`` and use slices. This could be further optimized by
# pre-computing batches of Tensors.
#

input = nameToTensor('Albert')
hidden = torch.zeros(1, n_hidden)

output, next_hidden = rnn(input[0], hidden)
```

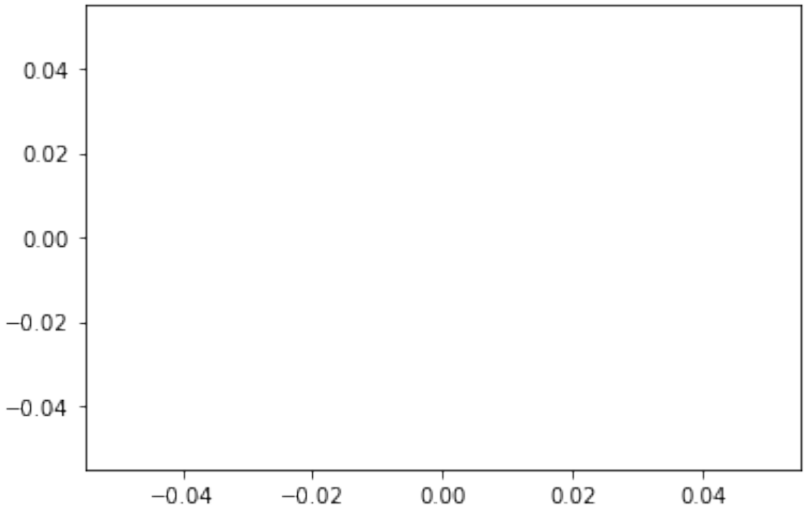
```
print(output)
def categoryFromOutput(output):
    # compute max
    top_n, top_i = output.topk(1)
    # output index of max
    category_i = top_i.item()
    return languages[category_i], category_i

print(categoryFromOutput(output))

import random


def randomChoice(l):
    return l[random.randint(0, len(l) - 1)]

def randomTrainingExample():
    category = randomChoice(languages)
    name = randomChoice(names[category])
    category_tensor = torch.tensor([languages.index(category)], dtype=torcl
    name_tensor = nameToTensor(name)
    return category, name, category_tensor, name_tensor

for i in range(10):
    category, name, category_tensor, name_tensor = randomTrainingExample()
    print('category =', category, '/ name =', name)

criterion = nn.NLLLoss()

learning_rate = 0.005 # For this example, we keep the learning rate fixed
def train(category_tensor, name_tensor):
    # initialize hidden state - do this every time before passing an input
    hidden = rnn.initHidden()
    # reset grad counters - do this every time after backprop
    rnn.zero_grad()
    # manually go through each element in input sequence
    for i in range(name_tensor.size()[0]):
        output, hidden = rnn(name_tensor[i], hidden)
    # backpropagate based on loss at last element only
    loss = criterion(output, category_tensor)
    loss.backward()

    # Update network parameters
    for p in rnn.parameters():
        p.data.add_(-learning_rate, p.grad.data)

    return output, loss.item()


import time
import math

n_iters = 100000
print_every = 5000
plot_every = 1000

# Keep track of loss for plotting
```

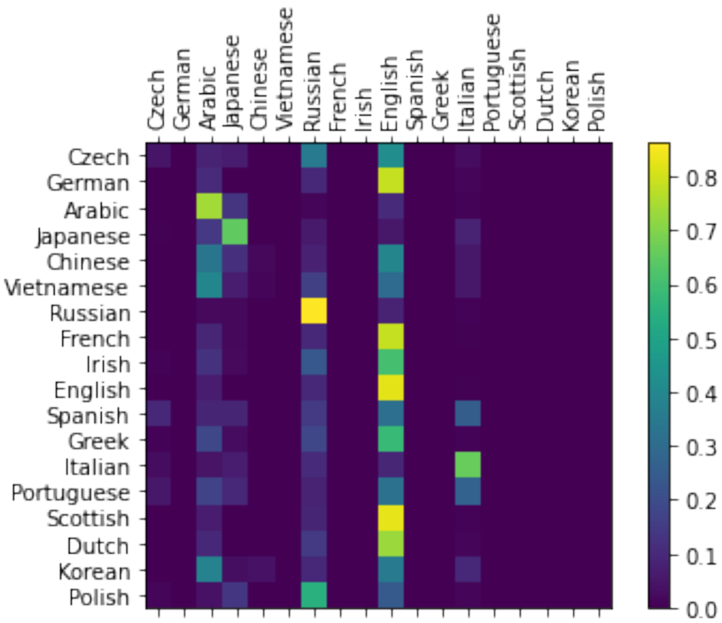
```
current_loss = 0
all_losses = []

def timeSince(since):
    now = time.time()
    s = now - since
    m = math.floor(s / 60)
    s -= m * 60
    return '%dm %ds' % (m, s)

start = time.time()

list_data = []
for category in languages:
    for name in names[category]:
        list_data.append((name,category))
iter = 0

for _ in range(5):
    random.shuffle(list_data)
    for name, category in list_data:
        iter+= 1
        category_tensor = torch.tensor([languages.index(category)], dtype =
        name_tensor = nameToTensor(name)
        output, loss = train(category_tensor,name_tensor)
        current_loss += loss

    # Print iter number, loss, name and guess
    if iter % print_every == 0:
        guess, guess_i = categoryFromOutput(output)
        correct = '✓' if guess == category else '✗ (%s)' % category
        print('%d %d%% (%s) %.4f %s / %s %s' % (iter, iter / n_iters * 100

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

import matplotlib.pyplot as plt
import matplotlib.ticker as ticker

plt.figure()
plt.plot(all_losses)
confusion = torch.zeros(n_categories, n_categories)
n_confusion = 20000

# return an output given an input name
def evaluate(name_tensor):
    hidden = rnn.initHidden()

    for i in range(name_tensor.size()[0]):
        output, hidden = rnn(name_tensor[i], hidden)

    return output

# Go through a bunch of examples and record which are correctly guessed
'''for i in range(n_confusion):
```

```
    category, name, category_tensor, name_tensor = randomTrainingExample()
    output = evaluate(name_tensor)
    guess, guess_i = categoryFromOutput(output)
    category_i = languages.index(category)
    confusion[category_i][guess_i] += 1'''

for category in languages:
    for name in names[category]:
        category_tensor = torch.tensor([languages.index(category)], dtype =
        name_tensor = nameToTensor(name)
        output= evaluate(name_tensor)
        guess, guess_i = categoryFromOutput(output)
        category_i = languages.index(category)
        confusion[category_i][guess_i] += 1


accuracy = sum(confusion.diag())/sum(sum(confusion))
print('Accuracy is %f' % accuracy.item())

for i in range(n_categories):
    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([''] + languages, rotation=90)
ax.set_yticklabels([''] + languages)

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

# sphinx_gallery_thumbnail_number = 2
plt.show()
```

```
tensor([[-2.8866, -2.7654, -2.9102, -2.9833, -2.7995, -2.9017, -2.9638, -3.
0253,
         -3.0443, -2.8880, -2.9585, -2.9171, -2.7914, -2.8397, -2.6966, -2.
8338,
         -2.9658, -2.9293]], grad_fn=<LogSoftmaxBackward0>)
('Scottish', 14)
category = Vietnamese / name = Dang
category = Scottish / name = Millar
category = Chinese / name = Ang
category = Japanese / name = Muso
category = Portuguese / name = Araujo
category = Greek / name = Grammatakakis
category = Polish / name = Stawski
category = Japanese / name = Kaza
category = Portuguese / name = Gomes
category = Korean / name = Suk
Accuracy is 0.686161
```

```
<ipython-input-11-c6296066d37f>:180: UserWarning: FixedFormatter should onl
y be used together with FixedLocator
  ax.set_xticklabels([''] + languages, rotation=90)
<ipython-input-11-c6296066d37f>:181: UserWarning: FixedFormatter should onl
y be used together with FixedLocator
  ax.set_yticklabels([''] + languages)
```





In [14]:

```
class RNN(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(RNN, self).__init__()

        self.hidden_size = hidden_size
        self.i2h = nn.Linear(input_size + hidden_size, hidden_size)
        self.i2o = nn.Linear(input_size + hidden_size, output_size)
        self.softmax = nn.LogSoftmax(dim=1)

    def forward(self, input, hidden):
        combined = torch.cat((input, hidden), 1)
        hidden = self.i2h(combined)
        output = self.i2o(combined)
        output = self.softmax(output)
```

```
        return output, hidden

    def initHidden(self):
        return torch.zeros(1, self.hidden_size)

#n_hidden = 128
n_hidden = 8
rnn = RNN(n_letters, n_hidden, n_categories)
input = letterToTensor('A')
hidden = torch.zeros(1, n_hidden)
output, next_hidden = rnn(input, hidden)

# For the sake of efficiency we don't want to be creating a new Tensor for
# every step, so we will use ``nameToTensor`` instead of
# ``letterToTensor`` and use slices. This could be further optimized by
# pre-computing batches of Tensors.
#

input = nameToTensor('Albert')
hidden = torch.zeros(1, n_hidden)

output, next_hidden = rnn(input[0], hidden)
print(output)
def categoryFromOutput(output):
    # compute max
    top_n, top_i = output.topk(1)
    # output index of max
    category_i = top_i.item()
    return languages[category_i], category_i

print(categoryFromOutput(output))

import random


def randomChoice(l):
    return l[random.randint(0, len(l) - 1)]

def randomTrainingExample():
    category = randomChoice(languages)
    name = randomChoice(names[category])
    category_tensor = torch.tensor([languages.index(category)], dtype=torcl
    name_tensor = nameToTensor(name)
    return category, name, category_tensor, name_tensor

for i in range(10):
    category, name, category_tensor, name_tensor = randomTrainingExample()
    print('category =', category, '/ name =', name)

criterion = nn.NLLLoss()

learning_rate = 0.005 # For this example, we keep the learning rate fixed
def train(category_tensor, name_tensor):
    # initialize hidden state - do this every time before passing an input
    hidden = rnn.initHidden()
    # reset grad counters - do this every time after backprop
    rnn.zero_grad()
```

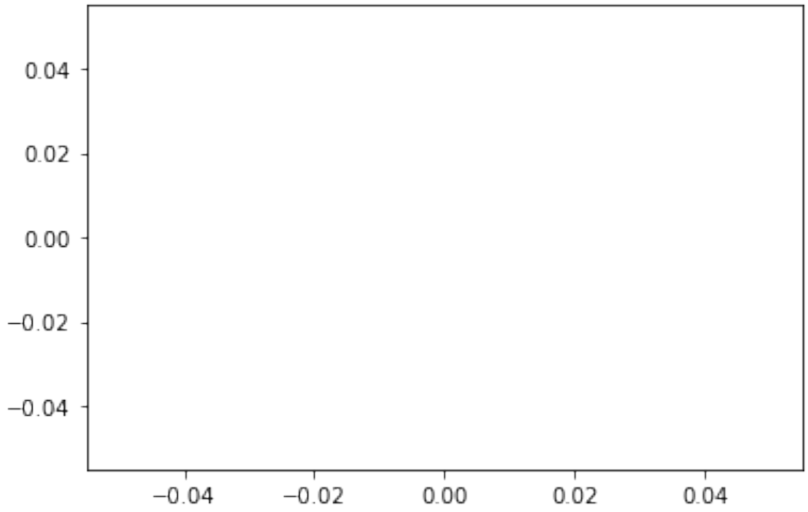
```
    # manually go through each element in input sequence
    for i in range(name_tensor.size()[0]):
        output, hidden = rnn(name_tensor[i], hidden)
    # backpropagate based on loss at last element only
    loss = criterion(output, category_tensor)
    loss.backward()

    # Update network parameters
    for p in rnn.parameters():
        p.data.add_(-learning_rate, p.grad.data)

    return output, loss.item()


import time
import math

n_iters = 100000
print_every = 5000
plot_every = 1000

# Keep track of loss for plotting
current_loss = 0
all_losses = []

def timeSince(since):
    now = time.time()
    s = now - since
    m = math.floor(s / 60)
    s -= m * 60
    return '%dm %ds' % (m, s)

start = time.time()

list_data = []
for category in languages:
    for name in names[category]:
        list_data.append((name,category))
iter = 0

for _ in range(5):
    random.shuffle(list_data)
    for name, category in list_data:
        iter+= 1
        category_tensor = torch.tensor([languages.index(category)], dtype =
        name_tensor = nameToTensor(name)
        output, loss = train(category_tensor,name_tensor)
        current_loss += loss

    # Print iter number, loss, name and guess
    if iter % print_every == 0:
        guess, guess_i = categoryFromOutput(output)
        correct = '✓' if guess == category else '✗ (%s)' % category
        print('%d %d%% (%s) %.4f %s / %s %s' % (iter, iter / n_iters * 100

    # Add current loss avg to list of losses
    if iter % plot_every == 0:
```

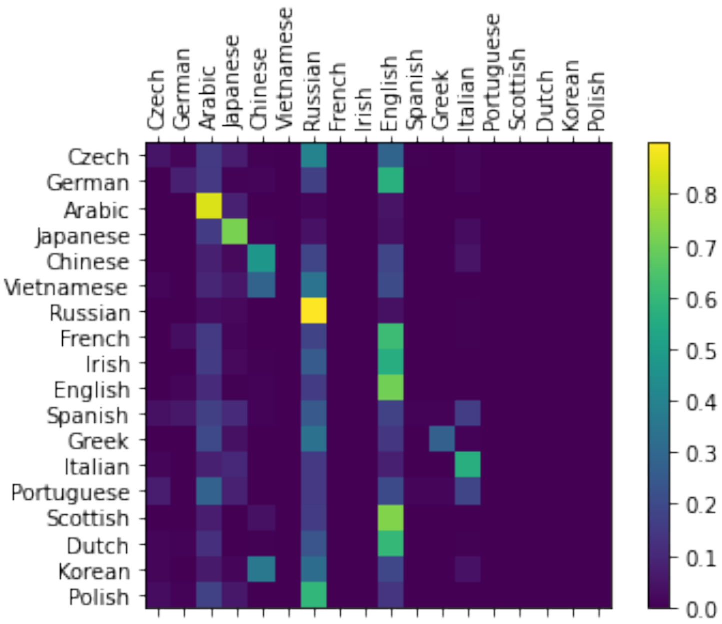
```
        all_losses.append(current_loss / plot_every)
        current_loss = 0

import matplotlib.pyplot as plt
import matplotlib.ticker as ticker

plt.figure()
plt.plot(all_losses)
confusion = torch.zeros(n_categories, n_categories)
n_confusion = 20000

# return an output given an input name
def evaluate(name_tensor):
    hidden = rnn.initHidden()

    for i in range(name_tensor.size()[0]):
        output, hidden = rnn(name_tensor[i], hidden)

    return output

# Go through a bunch of examples and record which are correctly guessed
'''for i in range(n_confusion):
    category, name, category_tensor, name_tensor = randomTrainingExample()
    output = evaluate(name_tensor)
    guess, guess_i = categoryFromOutput(output)
    category_i = languages.index(category)
    confusion[category_i][guess_i] += 1'''

for category in languages:
    for name in names[category]:
        category_tensor = torch.tensor([languages.index(category)], dtype =
        name_tensor = nameToTensor(name)
        output= evaluate(name_tensor)
        guess, guess_i = categoryFromOutput(output)
        category_i = languages.index(category)
        confusion[category_i][guess_i] += 1


accuracy = sum(confusion.diag())/sum(sum(confusion))
print('Accuracy is %f' % accuracy.item())

for i in range(n_categories):
    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([''] + languages, rotation=90)
ax.set_yticklabels([''] + languages)

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

```
# sphinx_gallery_thumbnail_number = 2
plt.show()
```

```
tensor([[-2.9639, -3.0885, -3.0036, -2.9959, -2.8934, -2.9379, -2.9988, -2.
9712,
         -2.8457, -2.8058, -2.7336, -2.7161, -2.8682, -2.8551, -2.8532, -2.
8319,
         -2.8238, -2.9225]], grad_fn=<LogSoftmaxBackward0>)
('Greek', 11)
category = German / name = Metz
category = Scottish / name = Wallace
category = Korean / name = Han
category = Dutch / name = Paulissen
category = German / name = Gaertner
category = Arabic / name = Baba
category = English / name = Grenard
category = Chinese / name = Mah
category = Vietnamese / name = Nguyen
category = Japanese / name = Modegi
Accuracy is 0.704792
```

```
<ipython-input-14-9f66052e8e55>:180: UserWarning: FixedFormatter should onl
y be used together with FixedLocator
  ax.set_xticklabels([''] + languages, rotation=90)
<ipython-input-14-9f66052e8e55>:181: UserWarning: FixedFormatter should onl
y be used together with FixedLocator
  ax.set_yticklabels([''] + languages)
```





In [15]:

```
class RNN(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(RNN, self).__init__()

        self.hidden_size = hidden_size
        self.i2h = nn.Linear(input_size + hidden_size, hidden_size)
        self.i2o = nn.Linear(input_size + hidden_size, output_size)
        self.softmax = nn.LogSoftmax(dim=1)

    def forward(self, input, hidden):
        combined = torch.cat((input, hidden), 1)
        hidden = self.i2h(combined)
        output = self.i2o(combined)
        output = self.softmax(output)
        return output, hidden

    def initHidden(self):
        return torch.zeros(1, self.hidden_size)

#n_hidden = 128
n_hidden = 32
rnn = RNN(n_letters, n_hidden, n_categories)
input = letterToTensor('A')
hidden = torch.zeros(1, n_hidden)
output, next_hidden = rnn(input, hidden)

# For the sake of efficiency we don't want to be creating a new Tensor for
# every step, so we will use ``nameToTensor`` instead of
# ``letterToTensor`` and use slices. This could be further optimized by
# pre-computing batches of Tensors.
#

input = nameToTensor('Albert')
hidden = torch.zeros(1, n_hidden)

output, next_hidden = rnn(input[0], hidden)
```

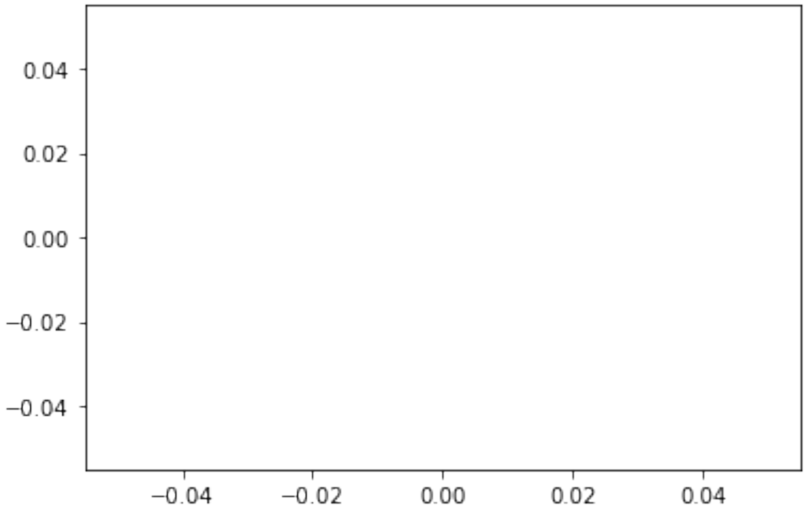
```
print(output)
def categoryFromOutput(output):
    # compute max
    top_n, top_i = output.topk(1)
    # output index of max
    category_i = top_i.item()
    return languages[category_i], category_i

print(categoryFromOutput(output))

import random


def randomChoice(l):
    return l[random.randint(0, len(l) - 1)]

def randomTrainingExample():
    category = randomChoice(languages)
    name = randomChoice(names[category])
    category_tensor = torch.tensor([languages.index(category)], dtype=torcl
    name_tensor = nameToTensor(name)
    return category, name, category_tensor, name_tensor

for i in range(10):
    category, name, category_tensor, name_tensor = randomTrainingExample()
    print('category =', category, '/ name =', name)

criterion = nn.NLLLoss()

learning_rate = 0.005 # For this example, we keep the learning rate fixed
def train(category_tensor, name_tensor):
    # initialize hidden state - do this every time before passing an input
    hidden = rnn.initHidden()
    # reset grad counters - do this every time after backprop
    rnn.zero_grad()
    # manually go through each element in input sequence
    for i in range(name_tensor.size()[0]):
        output, hidden = rnn(name_tensor[i], hidden)
    # backpropagate based on loss at last element only
    loss = criterion(output, category_tensor)
    loss.backward()

    # Update network parameters
    for p in rnn.parameters():
        p.data.add_(-learning_rate, p.grad.data)

    return output, loss.item()


import time
import math

n_iters = 100000
print_every = 5000
plot_every = 1000

# Keep track of loss for plotting
```

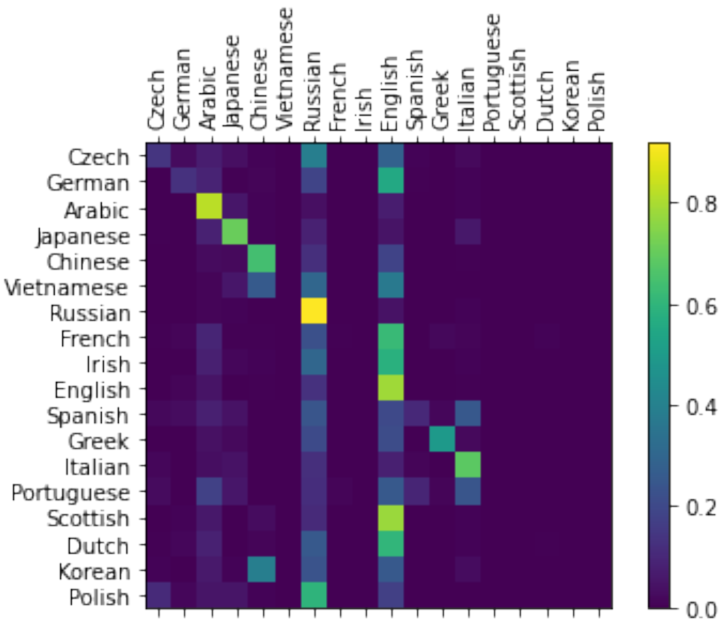
```
current_loss = 0
all_losses = []

def timeSince(since):
    now = time.time()
    s = now - since
    m = math.floor(s / 60)
    s -= m * 60
    return '%dm %ds' % (m, s)

start = time.time()

list_data = []
for category in languages:
    for name in names[category]:
        list_data.append((name,category))
iter = 0

for _ in range(5):
    random.shuffle(list_data)
    for name, category in list_data:
        iter+= 1
        category_tensor = torch.tensor([languages.index(category)], dtype =
        name_tensor = nameToTensor(name)
        output, loss = train(category_tensor,name_tensor)
        current_loss += loss

    # Print iter number, loss, name and guess
    if iter % print_every == 0:
        guess, guess_i = categoryFromOutput(output)
        correct = '✓' if guess == category else '✗ (%s)' % category
        print('%d %d%% (%s) %.4f %s / %s %s' % (iter, iter / n_iters * 100

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

import matplotlib.pyplot as plt
import matplotlib.ticker as ticker

plt.figure()
plt.plot(all_losses)
confusion = torch.zeros(n_categories, n_categories)
n_confusion = 20000

# return an output given an input name
def evaluate(name_tensor):
    hidden = rnn.initHidden()

    for i in range(name_tensor.size()[0]):
        output, hidden = rnn(name_tensor[i], hidden)

    return output

# Go through a bunch of examples and record which are correctly guessed
'''for i in range(n_confusion):
```

```
    category, name, category_tensor, name_tensor = randomTrainingExample()
    output = evaluate(name_tensor)
    guess, guess_i = categoryFromOutput(output)
    category_i = languages.index(category)
    confusion[category_i][guess_i] += 1'''

for category in languages:
    for name in names[category]:
        category_tensor = torch.tensor([languages.index(category)], dtype =
        name_tensor = nameToTensor(name)
        output= evaluate(name_tensor)
        guess, guess_i = categoryFromOutput(output)
        category_i = languages.index(category)
        confusion[category_i][guess_i] += 1


accuracy = sum(confusion.diag())/sum(sum(confusion))
print('Accuracy is %f' % accuracy.item())

for i in range(n_categories):
    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([''] + languages, rotation=90)
ax.set_yticklabels([''] + languages)

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

# sphinx_gallery_thumbnail_number = 2
plt.show()
```

```
tensor([[-3.0487, -2.7359, -2.8197, -2.8126, -2.9079, -2.9532, -2.9347, -2.
7833,
         -2.8305, -2.8627, -2.8034, -2.9956, -2.9444, -2.9290, -2.9177, -2.
8873,
         -2.9548, -2.9632]], grad_fn=<LogSoftmaxBackward0>)
('German', 1)
category = Korean / name = Gil
category = Greek / name = Demas
category = German / name = Hartmann
category = Greek / name = Karahalios
category = Chinese / name = Lai
category = Vietnamese / name = Bach
category = German / name = Welter
category = Japanese / name = Kurmochi
category = Czech / name = Bruckner
category = English / name = Holloway
Accuracy is 0.738069
```

```
<ipython-input-15-93967a0c31df>:180: UserWarning: FixedFormatter should onl
y be used together with FixedLocator
  ax.set_xticklabels([''] + languages, rotation=90)
<ipython-input-15-93967a0c31df>:181: UserWarning: FixedFormatter should onl
y be used together with FixedLocator
  ax.set_yticklabels([''] + languages)
```





In [12]:

```python
from __future__ import unicode_literals, print_function, division
from io import open
import glob
import os
import numpy as np
import pandas as pd
import unicodedata
import string
import torch
import torch.nn as nn
import random
import matplotlib.pyplot as plt
import matplotlib.ticker as ticker
```

In [13]:
```
def findFiles(path):
    return glob.glob(path)

all_letters = string.ascii_letters + " .,;'"
n_letters = len(all_letters)

def unicodeToAscii(s):
    return ''.join(
        c for c in unicodedata.normalize('NFD', s)
        if unicodedata.category(c) != 'Mn'
        and c in all_letters
    )
names = {}
languages = []


def readLines(filename):
    lines = open(filename, encoding='utf-8').read().strip().split('\n')
    return [unicodeToAscii(line) for line in lines]

# (TO DO:) CHANGE FILE PATH AS NECESSARY
for filename in findFiles('data/names/*.txt'):
    category = os.path.splitext(os.path.basename(filename))[0]
    languages.append(category)
    lines = readLines(filename)
    names[category] = lines
    n_categories = len(languages)

def letterToIndex(letter):
    return all_letters.find(letter)


def nameToTensor(name):
    tensor = torch.zeros(len(name), 1, n_letters)
    for li, letter in enumerate(name):
        tensor[li][0][letterToIndex(letter)] = 1
    return tensor

class RNN(nn.Module):
    def __init__(self, INPUT_SIZE, HIDDEN_SIZE, N_LAYERS,OUTPUT_SIZE):
        super(RNN, self).__init__()
        self.rnn = nn.RNN(
            input_size = INPUT_SIZE,
            hidden_size = HIDDEN_SIZE, # number of hidden units
            num_layers = N_LAYERS, # number of layers
            batch_first = True)
        self.out = nn.Linear(HIDDEN_SIZE, OUTPUT_SIZE)

    def forward(self, x):
        r_out, h = self.rnn(x, None) # None represents zero initial hidden
        out = self.out(r_out[:, -1, :])
        return out

n_hidden = 32
#rnn = RNN(n_letters, n_hidden, n_categories)
allnames = [] # Create list of all names and corresponding output language
for language in list(names.keys()):
```

```
    for name in names[language]:
        allnames.append([name, language])

## (TO DO:) Determine Padding length (this is the length of the longest st
maxlen = max(len(name[0]) for name in allnames) # Add code here to compute

n_letters = len(all_letters)
n_categories = len(languages)

def categoryFromOutput(output):
    top_n, top_i = output.topk(1)
    category_i = top_i.item()
    return languages[category_i], category_i

learning_rate = 0.005
rnn = RNN(n_letters, 128, 1, n_categories)
optimizer = torch.optim.Adam(rnn.parameters(), lr=learning_rate)   # optim
loss_func = nn.CrossEntropyLoss()

for epoch in range(5):
    batch_size = len(allnames)
    random.shuffle(allnames)

    # if "b_in" and "b_out" are the variable names for input and output te

    b_in = torch.zeros(batch_size, maxlen, n_letters)  # (TO DO:) Initiali
    b_out =torch.zeros(batch_size, n_categories, dtype= torch.long)  # (TO

    def split(word):
        return[char for char in word]
    # (TO DO:) Populate "b_in" tensor
    for name in allnames:
        i= allnames.index(name)
        list1= split(name[0])
        for m in range(len(name[0])):
            b_in[i][m][letterToIndex(list1[m])]=1

    # (TO DO:) Populate "b_out" tensor
    for name in allnames:
        i= allnames.index(name)
        lan= name[1]
        l = languages.index(lan)
        b_out[i][l]=1


    labels= torch.max(b_out,1)[1]
    output = rnn(b_in)
    #(TO DO:)
    loss = loss_func(output, labels)
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

    # Print accuracy
    test_output = rnn(b_in)                   #
    pred_y = torch.max(test_output, 1)[1].data.numpy().squeeze()
    test_y = torch.max(b_out, 1)[1].data.numpy().squeeze()
```

```
    accuracy = sum(pred_y == test_y)/batch_size
    print("Epoch: ", epoch, "| train loss: %.4f" % loss.item(), '| accuracy
```

```
Epoch:  0 | train loss: 2.8884 | accuracy: 0.47
Epoch:  1 | train loss: 2.6692 | accuracy: 0.47
Epoch:  2 | train loss: 2.1616 | accuracy: 0.47
Epoch:  3 | train loss: 1.9342 | accuracy: 0.47
Epoch:  4 | train loss: 1.9238 | accuracy: 0.47
```

Case 1: Batch Size = 1000

In [24]:

```
learning_rate = 0.005
rnn = RNN(n_letters, 128, 1, n_categories)
optimizer = torch.optim.Adam(rnn.parameters(), lr=learning_rate)   # optim
loss_func = nn.CrossEntropyLoss()
avg_accuracy = []
for epoch in range(5):
    batch_size = len(allnames)
    random.shuffle(allnames)

    length =0
    accuracy_list=[]
    while(length<=20074):
        #
        #print("length: ",length)
        try:
            max_sub= 0
            for i in range(length,length+1000):
                if(len(allnames[i][0]) > max_sub):
                    max_sub = len(allnames[i][0])
                    ans = allnames[i][0]
                    idx = i

            b_in = torch.zeros(batch_size, max_sub, n_letters)   # (TO DO:
            b_out = torch.zeros(batch_size, n_categories)  # (TO DO:) Init

            for i in range(length,length+1000):
                if(i>=20074): break
                else:
                    b_in[i] = nameToTensor3(allnames[i][0],max_sub)
                    category_index = torch.tensor([languages.index(allnames
                    b_out[i][category_index] = 1

            length=i+1
        except:
            length = length+1000
            continue


        output = rnn(b_in)                                  # rnn output
        #(TO DO:)
        loss = loss_func(output, b_out)   # (TO DO:) Fill "...." to calcul
        optimizer.zero_grad()                              # clear gradients
        loss.backward()                                    # backpropagation,
        optimizer.step()                                   # apply gradients

    # Print accuracy
        test_output = rnn(b_in)                        #
        pred_y = torch.max(test_output, 1)[1].data.numpy().squeeze()
        test_y = torch.max(b_out, 1)[1].data.numpy().squeeze()
        accuracy = sum(pred_y == test_y)/1000   #since batch size is 1000
        print("Epoch: ", epoch, "|Batch No.",length//1000,"| train loss: %
        accuracy_list.append(accuracy)
    avg_accuracy.append(sum(accuracy_list)/20)
```

```
Epoch:  0 |Batch No. 1 | train loss: 0.1430 | accuracy: 0.46
Epoch:  0 |Batch No. 2 | train loss: 0.1325 | accuracy: 0.47
```

```
Epoch:  0 |Batch No. 3 | train loss: 0.1079 | accuracy: 0.48
Epoch:  0 |Batch No. 4 | train loss: 0.0961 | accuracy: 0.47
Epoch:  0 |Batch No. 5 | train loss: 0.0976 | accuracy: 0.48
Epoch:  0 |Batch No. 6 | train loss: 0.0913 | accuracy: 0.48
Epoch:  0 |Batch No. 7 | train loss: 0.0984 | accuracy: 0.45
Epoch:  0 |Batch No. 8 | train loss: 0.0940 | accuracy: 0.47
Epoch:  0 |Batch No. 9 | train loss: 0.0918 | accuracy: 0.47
Epoch:  0 |Batch No. 10 | train loss: 0.0937 | accuracy: 0.46
Epoch:  0 |Batch No. 11 | train loss: 0.0939 | accuracy: 0.45
Epoch:  0 |Batch No. 12 | train loss: 0.0965 | accuracy: 0.45
Epoch:  0 |Batch No. 13 | train loss: 0.0930 | accuracy: 0.47
Epoch:  0 |Batch No. 14 | train loss: 0.0907 | accuracy: 0.48
Epoch:  0 |Batch No. 15 | train loss: 0.0960 | accuracy: 0.45
Epoch:  0 |Batch No. 16 | train loss: 0.0929 | accuracy: 0.47
Epoch:  0 |Batch No. 17 | train loss: 0.0947 | accuracy: 0.46
Epoch:  0 |Batch No. 18 | train loss: 0.0909 | accuracy: 0.51
Epoch:  0 |Batch No. 19 | train loss: 0.0924 | accuracy: 0.47
Epoch:  0 |Batch No. 20 | train loss: 0.0942 | accuracy: 0.47
Epoch:  1 |Batch No. 1 | train loss: 0.0953 | accuracy: 0.45
Epoch:  1 |Batch No. 2 | train loss: 0.0931 | accuracy: 0.46
Epoch:  1 |Batch No. 3 | train loss: 0.0912 | accuracy: 0.47
Epoch:  1 |Batch No. 4 | train loss: 0.0940 | accuracy: 0.44
Epoch:  1 |Batch No. 5 | train loss: 0.0929 | accuracy: 0.47
Epoch:  1 |Batch No. 6 | train loss: 0.0944 | accuracy: 0.45
Epoch:  1 |Batch No. 7 | train loss: 0.0944 | accuracy: 0.45
Epoch:  1 |Batch No. 8 | train loss: 0.0913 | accuracy: 0.47
Epoch:  1 |Batch No. 9 | train loss: 0.0909 | accuracy: 0.48
Epoch:  1 |Batch No. 10 | train loss: 0.0934 | accuracy: 0.47
Epoch:  1 |Batch No. 11 | train loss: 0.0935 | accuracy: 0.48
Epoch:  1 |Batch No. 12 | train loss: 0.0918 | accuracy: 0.48
Epoch:  1 |Batch No. 13 | train loss: 0.0915 | accuracy: 0.49
Epoch:  1 |Batch No. 14 | train loss: 0.0911 | accuracy: 0.47
Epoch:  1 |Batch No. 15 | train loss: 0.0927 | accuracy: 0.47
Epoch:  1 |Batch No. 16 | train loss: 0.0918 | accuracy: 0.47
Epoch:  1 |Batch No. 17 | train loss: 0.0919 | accuracy: 0.47
Epoch:  1 |Batch No. 18 | train loss: 0.0927 | accuracy: 0.47
Epoch:  1 |Batch No. 19 | train loss: 0.0893 | accuracy: 0.49
Epoch:  1 |Batch No. 20 | train loss: 0.0873 | accuracy: 0.48
Epoch:  2 |Batch No. 1 | train loss: 0.0950 | accuracy: 0.46
Epoch:  2 |Batch No. 2 | train loss: 0.0920 | accuracy: 0.47
Epoch:  2 |Batch No. 3 | train loss: 0.0927 | accuracy: 0.48
Epoch:  2 |Batch No. 4 | train loss: 0.0916 | accuracy: 0.46
Epoch:  2 |Batch No. 5 | train loss: 0.0941 | accuracy: 0.44
Epoch:  2 |Batch No. 6 | train loss: 0.0924 | accuracy: 0.48
Epoch:  2 |Batch No. 7 | train loss: 0.0913 | accuracy: 0.49
Epoch:  2 |Batch No. 8 | train loss: 0.0898 | accuracy: 0.49
Epoch:  2 |Batch No. 9 | train loss: 0.0926 | accuracy: 0.46
Epoch:  2 |Batch No. 10 | train loss: 0.0911 | accuracy: 0.48
Epoch:  2 |Batch No. 11 | train loss: 0.0919 | accuracy: 0.48
Epoch:  2 |Batch No. 12 | train loss: 0.0905 | accuracy: 0.46
Epoch:  2 |Batch No. 13 | train loss: 0.0951 | accuracy: 0.45
Epoch:  2 |Batch No. 14 | train loss: 0.0907 | accuracy: 0.47
Epoch:  2 |Batch No. 15 | train loss: 0.0930 | accuracy: 0.45
Epoch:  2 |Batch No. 16 | train loss: 0.0912 | accuracy: 0.46
Epoch:  2 |Batch No. 17 | train loss: 0.0924 | accuracy: 0.47
Epoch:  2 |Batch No. 18 | train loss: 0.0916 | accuracy: 0.47
Epoch:  2 |Batch No. 19 | train loss: 0.0935 | accuracy: 0.46
Epoch:  2 |Batch No. 20 | train loss: 0.0924 | accuracy: 0.49
Epoch:  3 |Batch No. 1 | train loss: 0.0908 | accuracy: 0.46
Epoch:  3 |Batch No. 2 | train loss: 0.0910 | accuracy: 0.45
Epoch:  3 |Batch No. 3 | train loss: 0.0939 | accuracy: 0.46
```

```
Epoch:  3 |Batch No. 4 | train loss: 0.0922 | accuracy: 0.46
Epoch:  3 |Batch No. 5 | train loss: 0.0934 | accuracy: 0.47
Epoch:  3 |Batch No. 6 | train loss: 0.0946 | accuracy: 0.47
Epoch:  3 |Batch No. 7 | train loss: 0.0944 | accuracy: 0.45
Epoch:  3 |Batch No. 8 | train loss: 0.0913 | accuracy: 0.47
Epoch:  3 |Batch No. 9 | train loss: 0.0907 | accuracy: 0.49
Epoch:  3 |Batch No. 10 | train loss: 0.0929 | accuracy: 0.47
Epoch:  3 |Batch No. 11 | train loss: 0.0938 | accuracy: 0.47
Epoch:  3 |Batch No. 12 | train loss: 0.0955 | accuracy: 0.45
Epoch:  3 |Batch No. 13 | train loss: 0.0946 | accuracy: 0.46
Epoch:  3 |Batch No. 14 | train loss: 0.0901 | accuracy: 0.48
Epoch:  3 |Batch No. 15 | train loss: 0.0884 | accuracy: 0.49
Epoch:  3 |Batch No. 16 | train loss: 0.0915 | accuracy: 0.48
Epoch:  3 |Batch No. 17 | train loss: 0.0922 | accuracy: 0.47
Epoch:  3 |Batch No. 18 | train loss: 0.0903 | accuracy: 0.48
Epoch:  3 |Batch No. 19 | train loss: 0.0901 | accuracy: 0.48
Epoch:  3 |Batch No. 20 | train loss: 0.0905 | accuracy: 0.48
Epoch:  4 |Batch No. 1 | train loss: 0.0933 | accuracy: 0.45
Epoch:  4 |Batch No. 2 | train loss: 0.0941 | accuracy: 0.45
Epoch:  4 |Batch No. 3 | train loss: 0.0907 | accuracy: 0.47
Epoch:  4 |Batch No. 4 | train loss: 0.0922 | accuracy: 0.47
Epoch:  4 |Batch No. 5 | train loss: 0.0903 | accuracy: 0.47
Epoch:  4 |Batch No. 6 | train loss: 0.0899 | accuracy: 0.50
Epoch:  4 |Batch No. 7 | train loss: 0.0957 | accuracy: 0.46
Epoch:  4 |Batch No. 8 | train loss: 0.0896 | accuracy: 0.48
Epoch:  4 |Batch No. 9 | train loss: 0.0962 | accuracy: 0.44
Epoch:  4 |Batch No. 10 | train loss: 0.0905 | accuracy: 0.48
Epoch:  4 |Batch No. 11 | train loss: 0.0901 | accuracy: 0.49
Epoch:  4 |Batch No. 12 | train loss: 0.0905 | accuracy: 0.48
Epoch:  4 |Batch No. 13 | train loss: 0.0913 | accuracy: 0.49
Epoch:  4 |Batch No. 14 | train loss: 0.0966 | accuracy: 0.46
Epoch:  4 |Batch No. 15 | train loss: 0.0942 | accuracy: 0.47
Epoch:  4 |Batch No. 16 | train loss: 0.0905 | accuracy: 0.48
Epoch:  4 |Batch No. 17 | train loss: 0.0929 | accuracy: 0.45
Epoch:  4 |Batch No. 18 | train loss: 0.0914 | accuracy: 0.45
Epoch:  4 |Batch No. 19 | train loss: 0.0918 | accuracy: 0.46
Epoch:  4 |Batch No. 20 | train loss: 0.0908 | accuracy: 0.47
```

Batch Size: 2000

In [25]:

```
learning_rate = 0.005
rnn = RNN(n_letters, 128, 1, n_categories)
optimizer = torch.optim.Adam(rnn.parameters(), lr=learning_rate)   # optim
loss_func = nn.CrossEntropyLoss()
avg_accuracy = []
for epoch in range(5):
    batch_size = len(allnames)
    random.shuffle(allnames)
    #padded_length = maxlen

    # if "b_in" and "b_out" are the variable names for input and output te

    #b_in = torch.zeros(batch_size, padded_length, n_letters)   # (TO DO:)
    #b_out = torch.zeros(batch_size, n_categories)  # (TO DO:) Initialize


    # (TO DO:) Populate "b_in" tensor

    # (TO DO:) Populate "b_out" tensor
```

```
    length =0
    accuracy_list=[]
    while(length<=20074):

        try:
            max_sub= 0

            for i in range(length,length+2000):
                if(len(allnames[i][0]) > max_sub):
                    max_sub = len(allnames[i][0])
                    ans = allnames[i][0]
                    idx = i

            b_in = torch.zeros(batch_size, max_sub, n_letters)   # (TO DO:
            b_out = torch.zeros(batch_size, n_categories)  # (TO DO:) Init

            for i in range(length,length+2000):
                if(i>=20074): break
                else:
                    b_in[i] = nameToTensor3(allnames[i][0],max_sub)
                    category_index = torch.tensor([languages.index(allname
                    b_out[i][category_index] = 1

            length=i+1
        except:
            length = length+2000
            continue


        output = rnn(b_in)                                        # rnn output
        #(TO DO:)
        loss = loss_func(output, b_out)   # (TO DO:) Fill "...." to calcul
        optimizer.zero_grad()                                    # clear gradients
        loss.backward()                                          # backpropagation,
        optimizer.step()                                         # apply gradients

    # Print accuracy
        test_output = rnn(b_in)                         #
        pred_y = torch.max(test_output, 1)[1].data.numpy().squeeze()
        test_y = torch.max(b_out, 1)[1].data.numpy().squeeze()
        accuracy = sum(pred_y == test_y)/2000   #since batch size is 2000
        print("Epoch: ", epoch, "|Batch No.",length//2000,"| train loss: %

        accuracy_list.append(round(accuracy,2))
```

```
Epoch:  0 |Batch No. 1 | train loss: 0.2884 | accuracy: 0.48
Epoch:  0 |Batch No. 2 | train loss: 0.2704 | accuracy: 0.47
Epoch:  0 |Batch No. 3 | train loss: 0.2369 | accuracy: 0.46
Epoch:  0 |Batch No. 4 | train loss: 0.1939 | accuracy: 0.47
Epoch:  0 |Batch No. 5 | train loss: 0.1912 | accuracy: 0.46
Epoch:  0 |Batch No. 6 | train loss: 0.1866 | accuracy: 0.46
Epoch:  0 |Batch No. 7 | train loss: 0.1910 | accuracy: 0.48
Epoch:  0 |Batch No. 8 | train loss: 0.1886 | accuracy: 0.47
Epoch:  0 |Batch No. 9 | train loss: 0.1856 | accuracy: 0.47
Epoch:  0 |Batch No. 10 | train loss: 0.1937 | accuracy: 0.46
Epoch:  1 |Batch No. 1 | train loss: 0.1887 | accuracy: 0.48
Epoch:  1 |Batch No. 2 | train loss: 0.1887 | accuracy: 0.46
Epoch:  1 |Batch No. 3 | train loss: 0.1896 | accuracy: 0.46
Epoch:  1 |Batch No. 4 | train loss: 0.1827 | accuracy: 0.48
Epoch:  1 |Batch No. 5 | train loss: 0.1887 | accuracy: 0.47
Epoch:  1 |Batch No. 6 | train loss: 0.1868 | accuracy: 0.46
Epoch:  1 |Batch No. 7 | train loss: 0.1844 | accuracy: 0.47
Epoch:  1 |Batch No. 8 | train loss: 0.1849 | accuracy: 0.46
Epoch:  1 |Batch No. 9 | train loss: 0.1839 | accuracy: 0.48
Epoch:  1 |Batch No. 10 | train loss: 0.1814 | accuracy: 0.47
Epoch:  2 |Batch No. 1 | train loss: 0.1891 | accuracy: 0.46
Epoch:  2 |Batch No. 2 | train loss: 0.1886 | accuracy: 0.45
Epoch:  2 |Batch No. 3 | train loss: 0.1872 | accuracy: 0.48
Epoch:  2 |Batch No. 4 | train loss: 0.1850 | accuracy: 0.47
Epoch:  2 |Batch No. 5 | train loss: 0.1818 | accuracy: 0.47
Epoch:  2 |Batch No. 6 | train loss: 0.1871 | accuracy: 0.46
Epoch:  2 |Batch No. 7 | train loss: 0.1796 | accuracy: 0.48
Epoch:  2 |Batch No. 8 | train loss: 0.1802 | accuracy: 0.48
Epoch:  2 |Batch No. 9 | train loss: 0.1877 | accuracy: 0.46
Epoch:  2 |Batch No. 10 | train loss: 0.1824 | accuracy: 0.47
Epoch:  3 |Batch No. 1 | train loss: 0.1867 | accuracy: 0.45
Epoch:  3 |Batch No. 2 | train loss: 0.1821 | accuracy: 0.48
Epoch:  3 |Batch No. 3 | train loss: 0.1849 | accuracy: 0.48
Epoch:  3 |Batch No. 4 | train loss: 0.1828 | accuracy: 0.48
Epoch:  3 |Batch No. 5 | train loss: 0.1844 | accuracy: 0.47
Epoch:  3 |Batch No. 6 | train loss: 0.1883 | accuracy: 0.45
Epoch:  3 |Batch No. 7 | train loss: 0.1845 | accuracy: 0.48
Epoch:  3 |Batch No. 8 | train loss: 0.1878 | accuracy: 0.47
Epoch:  3 |Batch No. 9 | train loss: 0.1807 | accuracy: 0.47
Epoch:  3 |Batch No. 10 | train loss: 0.1819 | accuracy: 0.46
Epoch:  4 |Batch No. 1 | train loss: 0.1839 | accuracy: 0.45
Epoch:  4 |Batch No. 2 | train loss: 0.1867 | accuracy: 0.47
Epoch:  4 |Batch No. 3 | train loss: 0.1855 | accuracy: 0.47
Epoch:  4 |Batch No. 4 | train loss: 0.1866 | accuracy: 0.47
Epoch:  4 |Batch No. 5 | train loss: 0.1766 | accuracy: 0.50
Epoch:  4 |Batch No. 6 | train loss: 0.1885 | accuracy: 0.45
Epoch:  4 |Batch No. 7 | train loss: 0.1784 | accuracy: 0.49
Epoch:  4 |Batch No. 8 | train loss: 0.1878 | accuracy: 0.46
Epoch:  4 |Batch No. 9 | train loss: 0.1838 | accuracy: 0.47
Epoch:  4 |Batch No. 10 | train loss: 0.1846 | accuracy: 0.46
```

Batch Size: 3000

In [26]:

```
learning_rate = 0.005
rnn = RNN(n_letters, 128, 1, n_categories)
optimizer = torch.optim.Adam(rnn.parameters(), lr=learning_rate)   # optim
loss_func = nn.CrossEntropyLoss()
avg_accuracy = []
for epoch in range(5):
    batch_size = len(allnames)
```

```
    random.shuffle(allnames)
    #padded_length = maxlen

    # if "b_in" and "b_out" are the variable names for input and output te

    #b_in = torch.zeros(batch_size, padded_length, n_letters)   # (TO DO:)
    #b_out = torch.zeros(batch_size, n_categories)  # (TO DO:) Initialize


    # (TO DO:) Populate "b_in" tensor

    # (TO DO:) Populate "b_out" tensor
    length =0
    accuracy_list=[]
    while(length<=20074):

        try:
            max_sub= 0

            for i in range(length,length+3000):
                if(len(allnames[i][0]) > max_sub):
                    max_sub = len(allnames[i][0])
                    ans = allnames[i][0]
                    idx = i

            b_in = torch.zeros(batch_size, max_sub, n_letters)   # (TO DO:
            b_out = torch.zeros(batch_size, n_categories)  # (TO DO:) Init

            for i in range(length,length+3000):
                if(i>=20074): break
                else:
                    b_in[i] = nameToTensor3(allnames[i][0],max_sub)
                    category_index = torch.tensor([languages.index(allname
                    b_out[i][category_index] = 1

            length=i+1
        except:
            length = length+3000
            continue


        output = rnn(b_in)                                    # rnn output
        #(TO DO:)
        loss = loss_func(output, b_out)   # (TO DO:) Fill "...." to calcul
        optimizer.zero_grad()                                # clear gradients
        loss.backward()                                      # backpropagation,
        optimizer.step()                                     # apply gradients

    # Print accuracy
        test_output = rnn(b_in)                        #
        pred_y = torch.max(test_output, 1)[1].data.numpy().squeeze()
        test_y = torch.max(b_out, 1)[1].data.numpy().squeeze()
        accuracy = sum(pred_y == test_y)/3000   #since batch size is 2000
        print("Epoch: ", epoch, "|Batch No.",length//3000,"| train loss: %
```

```
Epoch:  0 |Batch No. 1 | train loss: 0.4308 | accuracy: 0.47
Epoch:  0 |Batch No. 2 | train loss: 0.4003 | accuracy: 0.46
Epoch:  0 |Batch No. 3 | train loss: 0.3157 | accuracy: 0.49
Epoch:  0 |Batch No. 4 | train loss: 0.3127 | accuracy: 0.45
Epoch:  0 |Batch No. 5 | train loss: 0.2899 | accuracy: 0.48
Epoch:  0 |Batch No. 6 | train loss: 0.2882 | accuracy: 0.46
Epoch:  1 |Batch No. 1 | train loss: 0.2922 | accuracy: 0.46
Epoch:  1 |Batch No. 2 | train loss: 0.2842 | accuracy: 0.47
Epoch:  1 |Batch No. 3 | train loss: 0.2777 | accuracy: 0.48
Epoch:  1 |Batch No. 4 | train loss: 0.2797 | accuracy: 0.47
Epoch:  1 |Batch No. 5 | train loss: 0.2799 | accuracy: 0.47
Epoch:  1 |Batch No. 6 | train loss: 0.2841 | accuracy: 0.46
Epoch:  2 |Batch No. 1 | train loss: 0.2828 | accuracy: 0.47
Epoch:  2 |Batch No. 2 | train loss: 0.2729 | accuracy: 0.48
Epoch:  2 |Batch No. 3 | train loss: 0.2792 | accuracy: 0.46
Epoch:  2 |Batch No. 4 | train loss: 0.2745 | accuracy: 0.47
Epoch:  2 |Batch No. 5 | train loss: 0.2815 | accuracy: 0.47
Epoch:  2 |Batch No. 6 | train loss: 0.2790 | accuracy: 0.47
Epoch:  3 |Batch No. 1 | train loss: 0.2762 | accuracy: 0.47
Epoch:  3 |Batch No. 2 | train loss: 0.2811 | accuracy: 0.46
Epoch:  3 |Batch No. 3 | train loss: 0.2769 | accuracy: 0.47
Epoch:  3 |Batch No. 4 | train loss: 0.2763 | accuracy: 0.47
Epoch:  3 |Batch No. 5 | train loss: 0.2763 | accuracy: 0.47
Epoch:  3 |Batch No. 6 | train loss: 0.2750 | accuracy: 0.48
Epoch:  4 |Batch No. 1 | train loss: 0.2756 | accuracy: 0.47
Epoch:  4 |Batch No. 2 | train loss: 0.2788 | accuracy: 0.46
Epoch:  4 |Batch No. 3 | train loss: 0.2790 | accuracy: 0.46
Epoch:  4 |Batch No. 4 | train loss: 0.2693 | accuracy: 0.48
Epoch:  4 |Batch No. 5 | train loss: 0.2809 | accuracy: 0.47
Epoch:  4 |Batch No. 6 | train loss: 0.2748 | accuracy: 0.47
```

In [ ]: