1.1. (15 points) Effect of hidden state length - run the script for hidden state sizes of 2, 8 and 32 by modifying the value of variable n_hidden.

Submitted by:

Aradhya Mathur

Lakshmi Nikhil Goduguluri

What is the Accuracy yielded for different hidden state sizes? Also, include a graph of the loss function and the confusion matrix for each case.
Note that you will have to modify the existing evaluation function, as it measures accuracy from a randomly sampled population, which could lead to biased results:
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
You will want to measure performance on every sample in the dataset by comparing the label of the sample to the output of your trained network, and report the average accuracy.

In [1]:

```python
#!/usr/bin/env python
# coding: utf-8

# In[2]:
from __future__ import unicode_literals, print_function, division
from io import open
import glob
import os

print('hello')

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

#print(findFiles('C:/Users/*.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.com/a/5182
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'))

# %%
# 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]

# PLEASE UPDATE THE FILE PATH BELOW FOR YOUR SYSTEM
for filename in findFiles(r"C:\Users\aradh\Desktop\Fall 22\TSA\Project 3.1\data\da
    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 ''

findName(names,'Bernal')
findName(names,'Johnson')

# Now we have ``names``, a dictionary mapping each category
# (language) to a list of lines (names). We also kept track of
# ``languages`` (just a list of languages) and ``n_categories`` for
# later reference.
#

print(names['Italian'][:5])

# %%
# Turning Names into Tensors
# --------------------------
#
# Now that we have all the names organized, we need to turn them into
# Tensors to make any use of them.
#
# To represent a single letter, we use a "one-hot vector" of size
# ``<1 x n_letters>``. A one-hot vector is filled with 0s except for a 1
# at index of the current letter, e.g. ``"b" = <0 1 0 0 0 ...>``.
#
# To make a word we join a bunch of those into a 2D matrix
# ``<line_length x 1 x n_letters>``.
#
# That extra 1 dimension is because PyTorch assumes everything is in
# batches - we're just using a batch size of 1 here.

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

print(letterToTensor('J'))

print(nameToTensor('Jones').size())


# %% Creating the Network
# ====================
#
# Before autograd, creating a recurrent neural network in Torch involved
# cloning the parameters of a layer over several timesteps. The layers
# held hidden state and gradients which are now entirely handled by the
# graph itself. This means you can implement a RNN in a very "pure" way,
# as regular feed-forward layers.
#
# This RNN module (mostly copied from `the PyTorch for Torch users
# tutorial <https://pytorch.org/tutorials/beginner/former_torchies/
# nn_tutorial.html#example-2-recurrent-net>`__)
# is just 2 linear layers which operate on an input and hidden state, with
# a LogSoftmax layer after the output.
#
# .. figure:: https://i.imgur.com/Z2xbySO.png
#    :alt:
```

```
hello
Slusarski
['Abandonato', 'Abatangelo', 'Abatantuono', 'Abate', 'Abategiovanni']
tensor([[0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
         0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 1.,
         0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
         0., 0., 0.]])
torch.Size([5, 1, 57])
```

Hidden Layer = 2

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
```

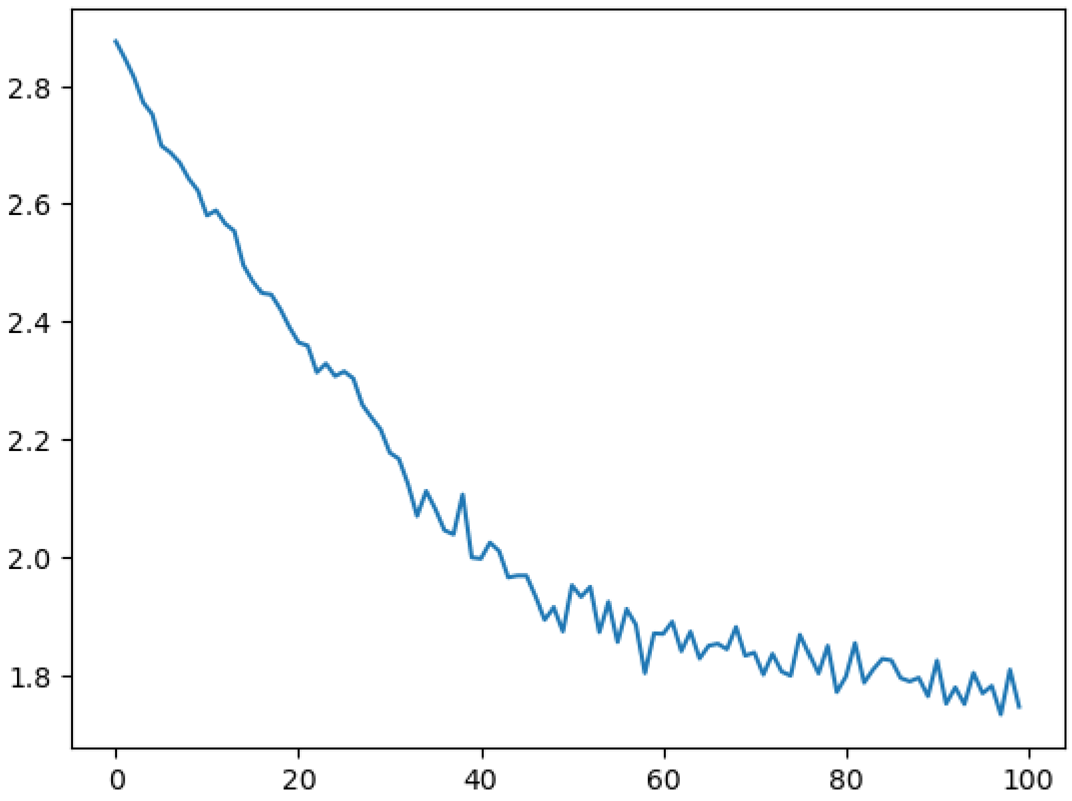
```python
    def initHidden(self):
        return torch.zeros(1, self.hidden_size)

#n_hidden = 128
n_hidden = 2
rnn = RNN(n_letters, n_hidden, n_categories)

# %% To run a step of this network we need to pass an input (in our case, the
# Tensor for the current letter) and a previous hidden state (which we
# initialize as zeros at first). We'll get back the output (probability of
# each language) and a next hidden state (which we keep for the next
# step).

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)

# As you can see the output is a ``<1 x n_categories>`` Tensor, where
# every item is the likelihood of that category (higher is more likely).
#

# %% Training
# ========
# Preparing for Training
# ----------------------
#
# Before going into training we should make a few helper functions. The
# first is to interpret the output of the network, which we know to be a
# likelihood of each category. We can use ``Tensor.topk`` to get the index
# of the greatest value:
#

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))

# We will also want a quick way to get a training example (a name and its
# language):
#
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=torch.long)
    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)

# %% Training the Network
# --------------------
#
# Now all it takes to train the network is show it a bunch of examples,
# have it make guesses, and compare its output against labels.
#
# For the loss function ``nn.NLLLoss`` is appropriate, since the last
# layer of the RNN is ``nn.LogSoftmax``.
#
criterion = nn.NLLLoss()

# Each loop of training will:
#
# -  Create input and target tensors
# -  Create a zeroed initial hidden state
# -  Read each letter in and
#
#    -  Keep hidden state for next letter
#
# -  Compare final output to target
# -  Back-propagate
# -  Return the output and loss
#

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 sequence
    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()

# Now we just have to run that with a bunch of examples. Since the
# ``train`` function returns both the output and loss we can print its
# guesses and also keep track of loss for plotting. Since there are 1000s
# of examples we print only every ``print_every`` examples, and take an
# average of the loss.
#
import time
import math

n_iters = 100000
print_every = 5000
plot_every = 1000
```

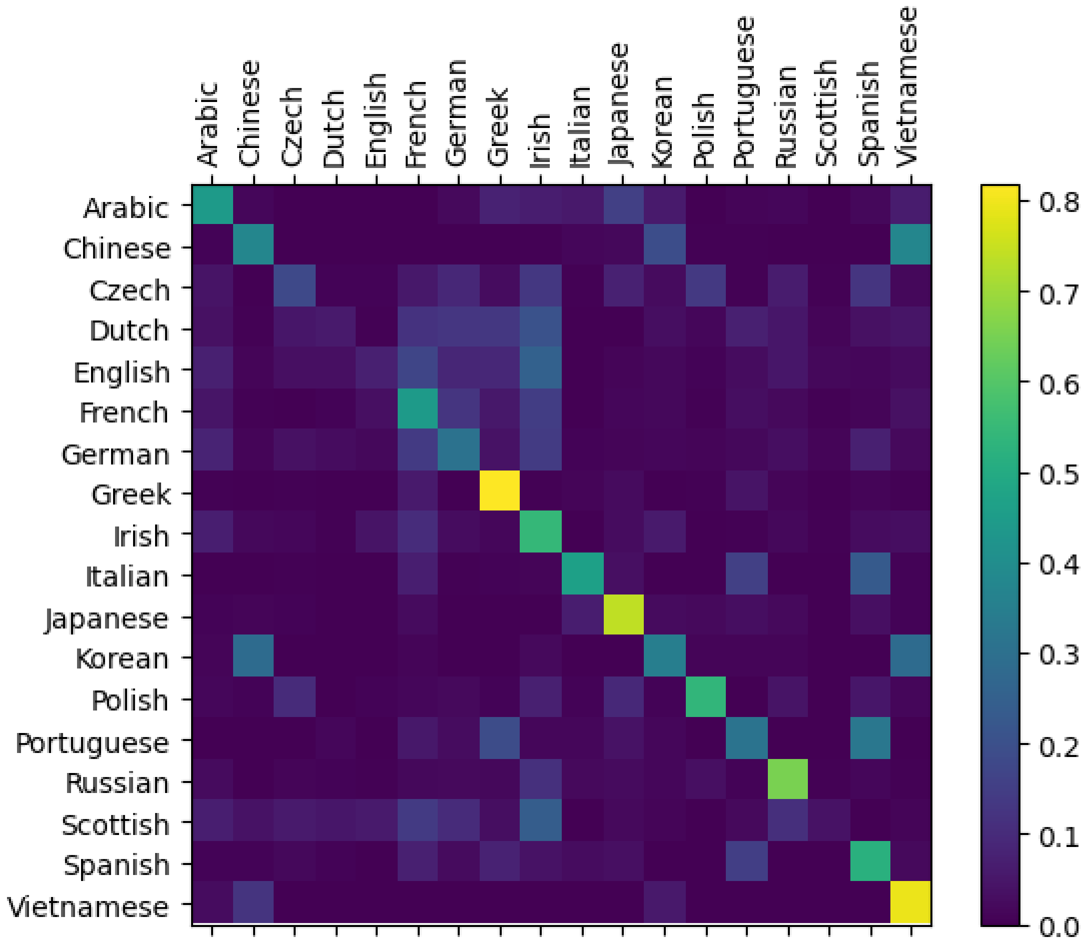
```
# Keep track of loss for plotting
current_loss = 0
all_losses = []

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, timeSi

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


# Plotting the Results
# --------------------
#
# Plotting the historical loss from ``all_losses`` shows the network
# learning:
#

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

plt.figure()
plt.plot(all_losses)


# Evaluating the Results
# ======================
#
# To see how well the network performs on different categories, we will
# create a confusion matrix, indicating for every actual language (rows)
# which language the network guesses (columns). To calculate the confusion
# matrix a bunch of samples are run through the network with
# ``evaluate()``, which is the same as ``train()`` minus the backprop.
#

# In[14]:

# Keep track of correct guesses in a confusion matrix
confusion = torch.zeros(n_categories, n_categories)
n_confusion = 20000

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

```python
    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 languages:
    for j in names[i]:
        category_tensor = torch.tensor([languages.index(i)], dtype=torch.long) # C
        name_tensor = nameToTensor(j) # Line 103 first chunk

        #As it is.
        output = evaluate(name_tensor)
        guess, guess_i = categoryFromOutput(output)
        category_i = languages.index(i)
        confusion[category_i][guess_i] += 1

#     category = randomTrainingExample()
#     name = randomTrainingExample()
#     category_tensor = randomTrainingExample()
#     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())
# Normalize by dividing every row by its sum
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([[-2.8017, -2.8688, -3.0663, -3.0088, -2.7096, -2.9778, -2.9813, -2.9558,
         -2.7024, -2.7537, -2.8271, -3.1401, -2.9319, -2.8975, -2.8582, -2.9462,
         -2.8012, -2.9181]], grad_fn=<LogSoftmaxBackward0>)
('Irish', 8)
category = Scottish / name = Gibson
category = Irish / name = O'Donnell
category = Vietnamese / name = Diep
category = Irish / name = Donoghue
category = Polish / name = Kijek
category = Dutch / name = Robert
category = Arabic / name = Tahan
category = Scottish / name = Clark
category = Chinese / name = Lam
category = Vietnamese / name = Luong
5000 5% (0m 4s) 2.5563 Chu / Vietnamese ✓
10000 10% (0m 9s) 2.8272 Jimbo / Portuguese ✗ (Japanese)
15000 15% (0m 14s) 2.6842 Kappel / Czech ✗ (Dutch)
20000 20% (0m 18s) 1.9290 Rooijakker / German ✗ (Dutch)
25000 25% (0m 23s) 2.9622 Oriol / Czech ✗ (Spanish)
30000 30% (0m 27s) 2.4541 Neisser / German ✗ (Czech)
35000 35% (0m 32s) 0.3110 Yuzhakov / Russian ✓
40000 40% (0m 36s) 2.1142 Haik / Korean ✗ (Arabic)
45000 45% (0m 41s) 0.5599 Bekyros / Greek ✓
50000 50% (0m 45s) 2.5146 Ri / Italian ✗ (Korean)
55000 55% (0m 50s) 1.5757 Zheromsky / Polish ✗ (Russian)
60000 60% (0m 54s) 0.6076 Albani / Italian ✓
65000 65% (0m 58s) 2.7184 Lucas / Portuguese ✗ (Dutch)
70000 70% (1m 3s) 0.1068 Makhutov / Russian ✓
75000 75% (1m 7s) 3.1829 Falkenrath / Russian ✗ (German)
80000 80% (1m 12s) 1.9087 Ghannam / Korean ✗ (Arabic)
85000 85% (1m 16s) 0.5993 Kanavos / Greek ✓
90000 90% (1m 21s) 1.9834 Ritchie / French ✗ (Scottish)
95000 95% (1m 25s) 0.1121 Mulatov / Russian ✓
100000 100% (1m 30s) 1.5265 Sung / Chinese ✗ (Korean)
Accuracy is 0.475790
```

```
C:\Users\aradh\AppData\Local\Temp\ipykernel_63236\3417164399.py:252: UserWarning:
FixedFormatter should only be used together with FixedLocator
  ax.set_xticklabels([''] + languages, rotation=90)
C:\Users\aradh\AppData\Local\Temp\ipykernel_63236\3417164399.py:253: UserWarning:
FixedFormatter should only be used together with FixedLocator
  ax.set_yticklabels([''] + languages)
```





In [7]:
```
languages
```

Out[7]:
```
['Arabic',
 'Chinese',
 'Czech',
 'Dutch',
 'English',
 'French',
 'German',
 'Greek',
 'Irish',
 'Italian',
 'Japanese',
 'Korean',
 'Polish',
 'Portuguese',
 'Russian',
 'Scottish',
 'Spanish',
 'Vietnamese']
```

Hidden Layer = 8

In [8]:
```python
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 = 8
rnn = RNN(n_letters, n_hidden, n_categories)

# %% To run a step of this network we need to pass an input (in our case, the
# Tensor for the current letter) and a previous hidden state (which we
# initialize as zeros at first). We'll get back the output (probability of
# each language) and a next hidden state (which we keep for the next
# step).

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.
#
```

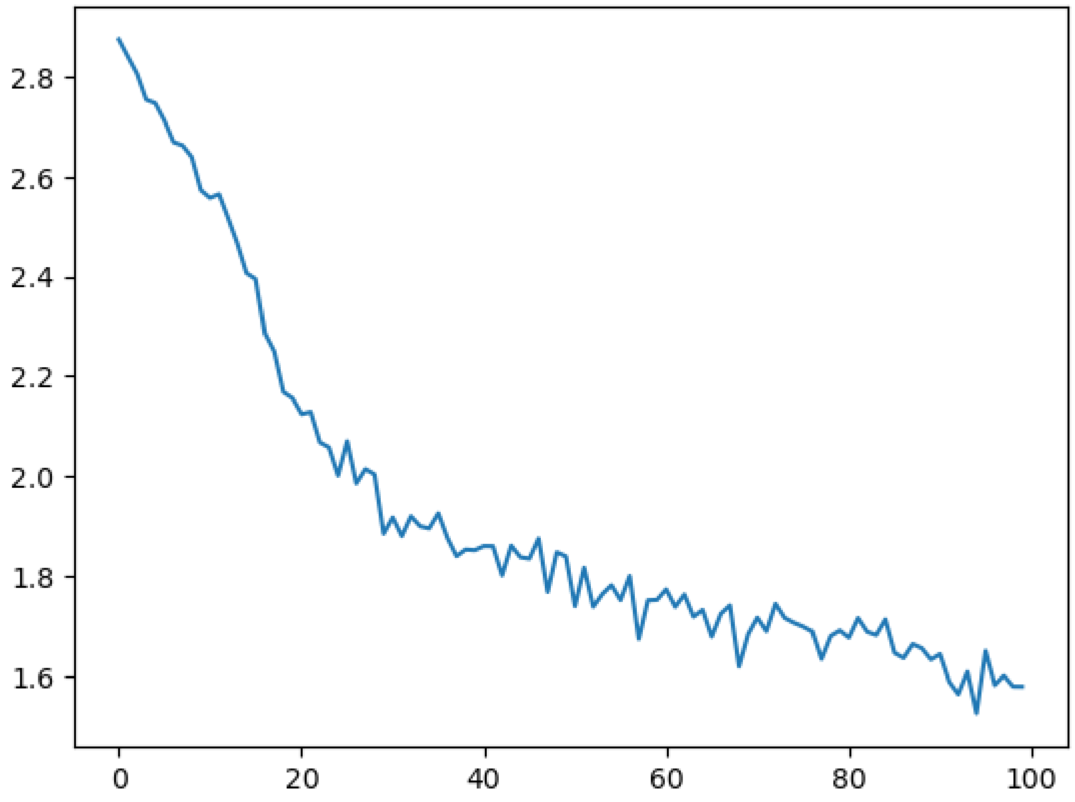
```
input = nameToTensor('Albert')
hidden = torch.zeros(1, n_hidden)

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

# As you can see the output is a ``<1 x n_categories>`` Tensor, where
# every item is the likelihood of that category (higher is more likely).
#

# %% Training
# ========
# Preparing for Training
# ----------------------
#
# Before going into training we should make a few helper functions. The
# first is to interpret the output of the network, which we know to be a
# likelihood of each category. We can use ``Tensor.topk`` to get the index
# of the greatest value:
#

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))

# We will also want a quick way to get a training example (a name and its
# language):
#
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=torch.long)
    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)

# %% Training the Network
# --------------------
#
# Now all it takes to train the network is show it a bunch of examples,
# have it make guesses, and compare its output against labels.
#
# For the loss function ``nn.NLLLoss`` is appropriate, since the last
# layer of the RNN is ``nn.LogSoftmax``.
#
criterion = nn.NLLLoss()

# Each loop of training will:
#
# -  Create input and target tensors
# -  Create a zeroed initial hidden state
# -  Read each letter in and
#
#    -  Keep hidden state for next letter
#
# -  Compare final output to target
# -  Back-propagate
# -  Return the output and loss
#

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 sequence
    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()

# Now we just have to run that with a bunch of examples. Since the
# ``train`` function returns both the output and loss we can print its
# guesses and also keep track of loss for plotting. Since there are 1000s
# of examples we print only every ``print_every`` examples, and take an
# average of the loss.
#
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
```

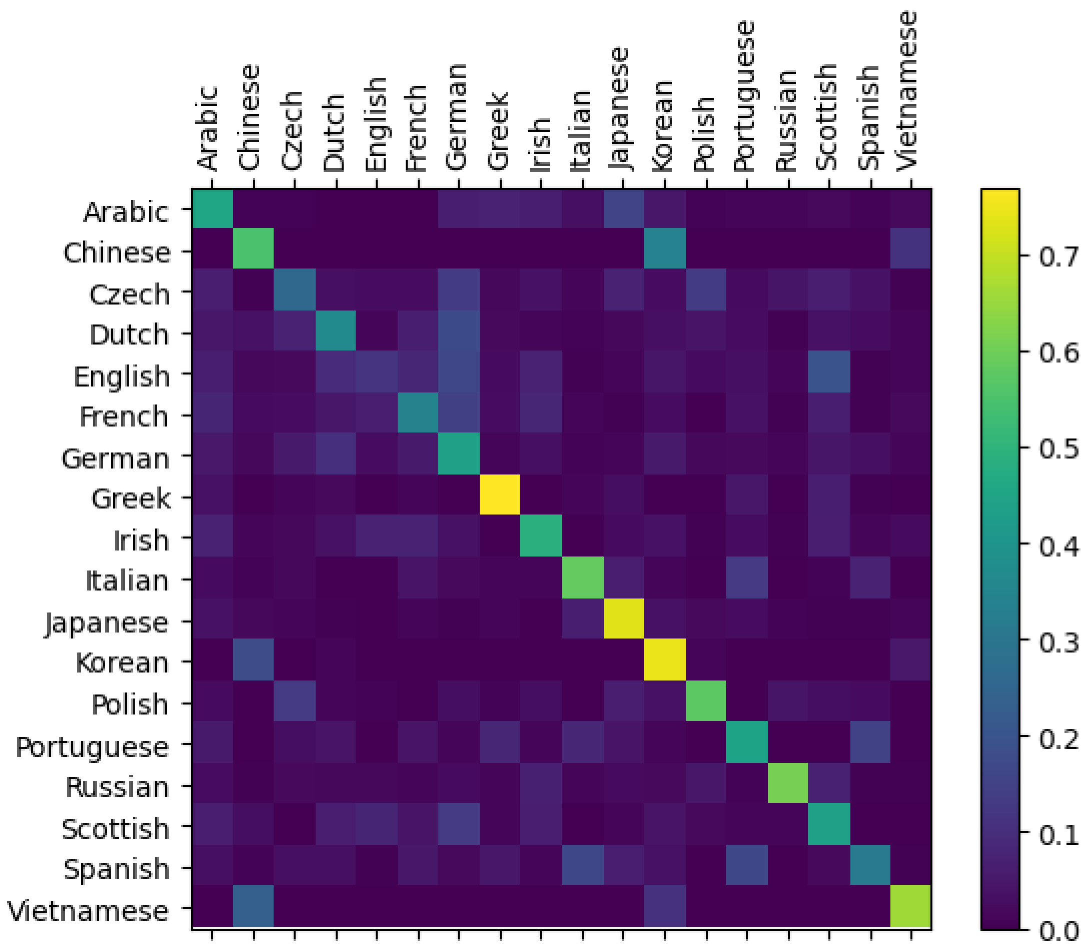
```
            print('%d %d%% (%s) %.4f %s / %s %s' % (iter, iter / n_iters * 100, timeSi

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


# Plotting the Results
# --------------------
#
# Plotting the historical loss from ``all_losses`` shows the network
# learning:
#

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

plt.figure()
plt.plot(all_losses)


# Evaluating the Results
# ======================
#
# To see how well the network performs on different categories, we will
# create a confusion matrix, indicating for every actual language (rows)
# which language the network guesses (columns). To calculate the confusion
# matrix a bunch of samples are run through the network with
# ``evaluate()``, which is the same as ``train()`` minus the backprop.
#

# In[14]:

# Keep track of correct guesses in a confusion matrix
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 languages:
    for j in names[i]:
        category_tensor = torch.tensor([languages.index(i)], dtype=torch.long) # C
        name_tensor = nameToTensor(j) # Line 103 first chunk


        #As it is.
        output = evaluate(name_tensor)
        guess, guess_i = categoryFromOutput(output)
        category_i = languages.index(i)
        confusion[category_i][guess_i] += 1

#     category = randomTrainingExample()
#     name = randomTrainingExample()
#     category_tensor = randomTrainingExample()
```

```python
#     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())
# Normalize by dividing every row by its sum
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([[-2.9269, -3.0566, -2.8221, -2.9978, -2.8771, -3.0599, -2.9346, -2.7241,
         -2.9725, -3.0181, -2.8442, -2.8744, -2.7432, -2.9502, -2.7832, -2.9155,
         -2.8378, -2.7790]], grad_fn=<LogSoftmaxBackward0>)
('Greek', 7)
category = Dutch / name = Haanraads
category = Vietnamese / name = Ngo
category = Japanese / name = Ozu
category = Czech / name = Wood
category = Vietnamese / name = Doan
category = French / name = Desrochers
category = German / name = Kneib
category = German / name = Lewerenz
category = Spanish / name = Jimenez
category = Italian / name = Vico
5000 5% (0m 4s) 2.7614 Silveira / Japanese ✗ (Portuguese)
10000 10% (0m 10s) 2.4135 Ferguson / Irish ✗ (Scottish)
15000 15% (0m 15s) 2.9844 Ha / Japanese ✗ (Vietnamese)
20000 20% (0m 19s) 0.8260 Malinowski / Polish ✓
25000 25% (0m 24s) 1.7088 Pasternack / Czech ✗ (Polish)
30000 30% (0m 28s) 3.0003 Bell / German ✗ (Scottish)
35000 35% (0m 33s) 2.3070 Elizondo / Italian ✗ (Spanish)
40000 40% (0m 37s) 1.0913 Kanavos / Greek ✓
45000 45% (0m 42s) 2.4830 Truong / German ✗ (Vietnamese)
50000 50% (0m 47s) 1.1374 Aswad / Arabic ✓
55000 55% (0m 51s) 4.3070 Sereda / Spanish ✗ (Russian)
60000 60% (0m 56s) 1.6078 Hutmacher / French ✗ (German)
65000 65% (1m 0s) 0.9246 Ursler / German ✓
70000 70% (1m 5s) 2.4583 Sze  / Korean ✗ (Chinese)
75000 75% (1m 9s) 2.6338 Breda / Japanese ✗ (Italian)
80000 80% (1m 14s) 2.3985 Buffone / French ✗ (Italian)
85000 85% (1m 18s) 3.2857 De fiore / French ✗ (Italian)
90000 90% (1m 23s) 2.4594 Campbell / Dutch ✗ (Scottish)
95000 95% (1m 27s) 0.9451 Blazek / Czech ✓
100000 100% (1m 32s) 1.3569 Grosz / German ✓
Accuracy is 0.481817
```

```
C:\Users\aradh\AppData\Local\Temp\ipykernel_63236\4147703795.py:253: UserWarning:
FixedFormatter should only be used together with FixedLocator
  ax.set_xticklabels([''] + languages, rotation=90)
C:\Users\aradh\AppData\Local\Temp\ipykernel_63236\4147703795.py:254: UserWarning:
FixedFormatter should only be used together with FixedLocator
  ax.set_yticklabels([''] + languages)
```





Hidden Layer = 32

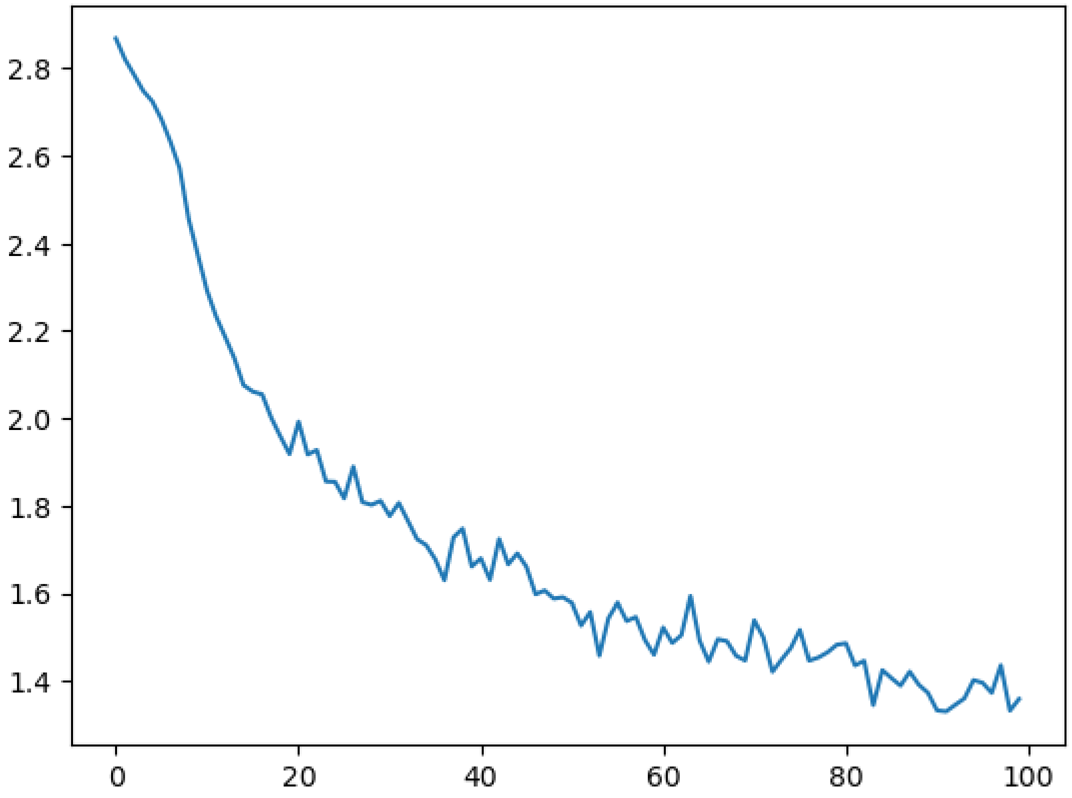
In [9]:
```python
import torch.nn as nn

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 = 32
rnn = RNN(n_letters, n_hidden, n_categories)

# %% To run a step of this network we need to pass an input (in our case, the
# Tensor for the current letter) and a previous hidden state (which we
# initialize as zeros at first). We'll get back the output (probability of
# each language) and a next hidden state (which we keep for the next
# step).

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)

# As you can see the output is a ``<1 x n_categories>`` Tensor, where
# every item is the likelihood of that category (higher is more likely).
#

# %% Training
# ========
# Preparing for Training
# ----------------------
#
# Before going into training we should make a few helper functions. The
# first is to interpret the output of the network, which we know to be a
# likelihood of each category. We can use ``Tensor.topk`` to get the index
# of the greatest value:
#

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))

# We will also want a quick way to get a training example (a name and its
# language):
#
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=torch.long)
    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)

# %% Training the Network
# --------------------
#
# Now all it takes to train the network is show it a bunch of examples,
# have it make guesses, and compare its output against labels.
#
# For the loss function ``nn.NLLLoss`` is appropriate, since the last
# layer of the RNN is ``nn.LogSoftmax``.
#
criterion = nn.NLLLoss()

# Each loop of training will:
#
# -  Create input and target tensors
# -  Create a zeroed initial hidden state
# -  Read each letter in and
#
#    -  Keep hidden state for next letter
#
# -  Compare final output to target
# -  Back-propagate
# -  Return the output and loss
#

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 sequence
    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()

# Now we just have to run that with a bunch of examples. Since the
# ``train`` function returns both the output and loss we can print its
# guesses and also keep track of loss for plotting. Since there are 1000s
# of examples we print only every ``print_every`` examples, and take an
# average of the loss.
#
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, timeSi

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


# Plotting the Results
# --------------------
#
# Plotting the historical loss from ``all_losses`` shows the network
# learning:
#

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

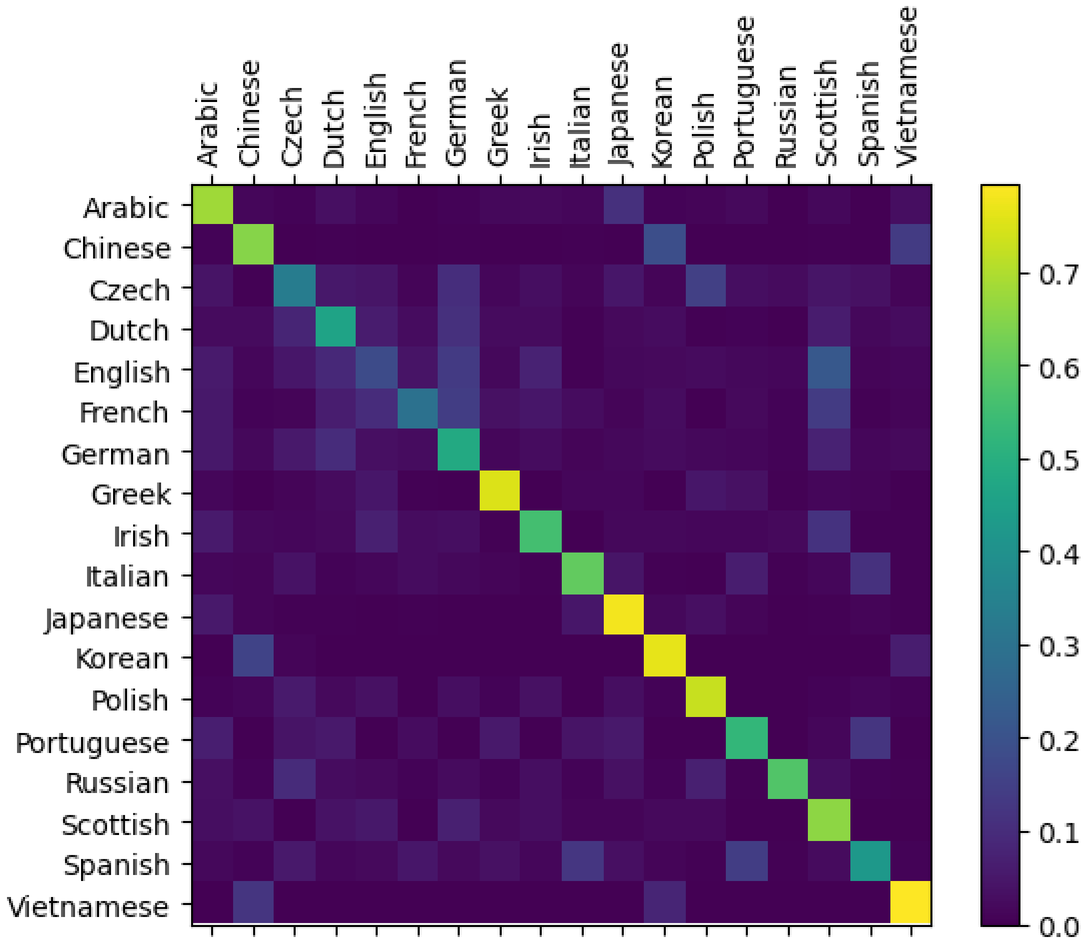
```
plt.figure()
plt.plot(all_losses)


# Evaluating the Results
# ======================
#
# To see how well the network performs on different categories, we will
# create a confusion matrix, indicating for every actual language (rows)
# which language the network guesses (columns). To calculate the confusion
# matrix a bunch of samples are run through the network with
# ``evaluate()``, which is the same as ``train()`` minus the backprop.
#

# In[14]:

# Keep track of correct guesses in a confusion matrix
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 languages:
    for j in names[i]:
        category_tensor = torch.tensor([languages.index(i)], dtype=torch.long) # Co
        name_tensor = nameToTensor(j) # Line 103 first chunk


        #As it is.
        output = evaluate(name_tensor)
        guess, guess_i = categoryFromOutput(output)
        category_i = languages.index(i)
        confusion[category_i][guess_i] += 1

#     category = randomTrainingExample()
#     name = randomTrainingExample()
#     category_tensor = randomTrainingExample()
#     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())
# Normalize by dividing every row by its sum
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)
```

```python
# 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([[-2.9212, -2.9554, -2.8929, -2.8022, -2.9413, -2.9033, -2.9413, -2.8562,
         -2.8592, -2.8305, -2.7164, -2.9603, -2.9362, -2.9310, -2.9286, -2.7897,
         -3.0263, -2.8820]], grad_fn=<LogSoftmaxBackward0>)
('Japanese', 10)
category = Scottish / name = Munro
category = Irish / name = O'Grady
category = Dutch / name = Kann
category = Czech / name = Driml
category = Dutch / name = Peusen
category = Japanese / name = Kakutama
category = Dutch / name = Simonis
category = Dutch / name = Daalen
category = Vietnamese / name = Cao
category = Arabic / name = Hakimi
5000 5% (0m 4s) 2.7046 Ganem / Arabic ✓
10000 10% (0m 9s) 2.3604 Shiskikura / Spanish ✗ (Japanese)
15000 15% (0m 13s) 1.4427 Than / Chinese ✗ (Vietnamese)
20000 20% (0m 18s) 1.2977 Villanueva / Spanish ✓
25000 25% (0m 23s) 1.5226 Gaber / Arabic ✓
30000 30% (0m 27s) 2.1537 Snaijer / Arabic ✗ (Dutch)
35000 35% (0m 32s) 0.9631 Rang / Chinese ✓
40000 40% (0m 36s) 1.9624 Pokorny / Russian ✗ (Polish)
45000 45% (0m 41s) 0.6439 Wan / Chinese ✓
50000 50% (0m 45s) 0.9516 Lao / Vietnamese ✗ (Chinese)
55000 55% (0m 50s) 2.4968 Kanne / English ✗ (Dutch)
60000 60% (0m 55s) 0.5525 Paloumbas / Greek ✓
65000 65% (0m 59s) 1.1811 Orellana / Italian ✗ (Spanish)
70000 70% (1m 4s) 1.1594 Welter / German ✓
75000 75% (1m 8s) 1.7559 Kozlow / English ✗ (Polish)
80000 80% (1m 13s) 0.3110 Choi / Korean ✓
85000 85% (1m 18s) 2.0909 Glaisyer / German ✗ (French)
90000 90% (1m 22s) 0.1869 Pavlyuchenko / Russian ✓
95000 95% (1m 27s) 0.6712 Jang / Korean ✓
100000 100% (1m 31s) 1.4395 Chi / Korean ✗ (Chinese)
Accuracy is 0.514347
```

```
C:\Users\aradh\AppData\Local\Temp\ipykernel_63236\656936235.py:253: UserWarning: F
ixedFormatter should only be used together with FixedLocator
  ax.set_xticklabels([''] + languages, rotation=90)
C:\Users\aradh\AppData\Local\Temp\ipykernel_63236\656936235.py:254: UserWarning: F
ixedFormatter should only be used together with FixedLocator
  ax.set_yticklabels([''] + languages)
```


2.8
2.6
2.4
2.2
2.0
1.8
1.6
1.4
0
20
40
60
80
100


Arabic
Chinese
Czech
Dutch
English
French
German
Greek
Irish
Italian
Japanese
Korean
Polish
Portuguese
Russian
Scottish
Spanish
Vietnamese
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0

```
In [ ]:
```

```
In [ ]:
```

```
In [ ]:
```

```
In [ ]:
```