Adapted from PyTorch tutorial https://pytorch.org/tutorials/intermediate/char_rnn_classification_tutori (https://pytorch.org/tutorials/intermediate/char_rnn_classification_tutor originally accessed on 03-28-2023

In [1]:
```
# For tips on running notebooks in Google Colab, see
# https://pytorch.org/tutorials/beginner/colab
%matplotlib inline
```

NLP From Scratch: Classifying Names with a Character-Level RNN

Author: Sean Robertson (https://github.com/spro/practical-pytorch)

We will be building and training a basic character-level RNN to classify words. This tutorial, along with the following two, show how to do preprocess data for NLP modeling "from scratch", in particular not using many of the convenience functions of `torchtext`, so you can see how preprocessing for NLP modeling works at a low level.

A character-level RNN reads words as a series of characters - outputting a prediction and "hidden state" at each step, feeding its previous hidden state into each next step. We take the final prediction to be the output, i.e. which class the word belongs to.

Specifically, we'll train on a few thousand surnames from 18 languages of origin, and predict which language a name is from based on the spelling:

::

```
$ python predict.py Hinton
(-0.47) Scottish
(-1.52) English
(-3.57) Irish

$ python predict.py Schmidhuber
(-0.19) German
(-2.48) Czech
(-2.68) Dutch
```

Recommended Reading:

I assume you have at least installed PyTorch, know Python, and understand Tensors:

- https://pytorch.org/ (https://pytorch.org/) For installation instructions
- :doc: `/beginner/deep_learning_60min_blitz` to get started with PyTorch in general
- :doc: `/beginner/pytorch_with_examples` for a wide and deep overview
- :doc: `/beginner/former_torchies_tutorial` if you are former Lua Torch user

It would also be useful to know about RNNs and how they work:

- The Unreasonable Effectiveness of Recurrent Neural Networks (https://karpathy.github.io/2015/05/21/rnn-effectiveness/)_ shows a bunch of real life examples
- Understanding LSTM Networks (https://colah.github.io/posts/2015-08-Understanding-LSTMs/)_ is about LSTMs specifically but also informative about RNNs in general

Preparing the Data

.. Note:: Download the data from here (https://download.pytorch.org/tutorial/data.zip) and extract it to the current directory.

Included in the `data/names` directory are 18 text files named as "[Language].txt". Each file contains a bunch of names, one name per line, mostly romanized (but we still need to convert from Unicode to ASCII).

We'll end up with a dictionary of lists of names per language, `{language: [names ...]}`. The generic variables "category" and "line" (for language and name in our case) are used for later extensibility.

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

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

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.c
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 category_lines dictionary, a list of names per language
category_lines = {}
all_categories = []

# 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]

for filename in findFiles('data/names/*.txt'):
    category = os.path.splitext(os.path.basename(filename))[0]
    all_categories.append(category)
    lines = readLines(filename)
    category_lines[category] = lines

n_categories = len(all_categories)
```

```
['data/names/Czech.txt', 'data/names/German.txt', 'data/names/Arabic.tx
t', 'data/names/Japanese.txt', 'data/names/Chinese.txt', 'data/names/Vi
etnamese.txt', 'data/names/Russian.txt', 'data/names/French.txt', 'dat
a/names/Irish.txt', 'data/names/English.txt', 'data/names/Spanish.txt',
'data/names/Greek.txt', 'data/names/Italian.txt', 'data/names/Portugues
e.txt', 'data/names/Scottish.txt', 'data/names/Dutch.txt', 'data/names/
Korean.txt', 'data/names/Polish.txt']
Slusarski
```

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

In [3]:
```
print(category_lines['Italian'][:5])
```

```
['Abandonato', 'Abatangelo', 'Abatantuono', 'Abate', 'Abategiovanni']
```

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.

In [4]:
```
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 lineToTensor(line):
    tensor = torch.zeros(len(line), 1, n_letters)
    for li, letter in enumerate(line):
        tensor[li][0][letterToIndex(letter)] = 1
    return tensor

print(letterToTensor('J'))

print(lineToTensor('Jones').size())
```

```
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.,
0., 0., 1.,
         0., 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])
```

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/ (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 (https://i.imgur.com/Z2xbySO.png) :alt:

In [5]:
```
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
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).

In [6]:
```
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 `lineToTensor` instead of `letterToTensor` and use slices. This could be further optimized by pre-computing batches of Tensors.

In [7]:
```
input = lineToTensor('Albert')
hidden = torch.zeros(1, n_hidden)

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

```
tensor([[-2.8889, -2.8474, -2.9432, -2.8653, -2.8664, -2.8983, -2.9383,
-2.8155,
         -3.0116, -3.0238, -2.8319, -2.7677, -2.9007, -2.9664, -2.8175,
-2.8376,
         -2.9150, -2.9317]], grad_fn=<LogSoftmaxBackward>)
```

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:

In [8]:
```
def categoryFromOutput(output):
    top_n, top_i = output.topk(1)
    category_i = top_i[0].item()
    return all_categories[category_i], category_i

print(categoryFromOutput(output))
```

```
('Greek', 11)
```

We will also want a quick way to get a training example (a name and its language):

In [9]:
```
import random

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

def randomTrainingExample():
    category = randomChoice(all_categories)
    line = randomChoice(category_lines[category])
    category_tensor = torch.tensor([all_categories.index(category)], dtyp
    line_tensor = lineToTensor(line)
    return category, line, category_tensor, line_tensor

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

```
category = Dutch / line = Mulder
category = Spanish / line = Ramos
category = English / line = Danby
category = Scottish / line = Mackenzie
category = German / line = Derrick
category = Dutch / line = Klein
category = Spanish / line = Rivera
category = Polish / line = Nowak
category = Russian / line = Kachur
category = Chinese / line = Tong
```

Training the Network

Now all it takes to train this network is show it a bunch of examples, have it make guesses, and tell it if it's wrong.

For the loss function `nn.NLLLoss` is appropriate, since the last layer of the RNN is `nn.LogSoftmax`.

In [10]:
```
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

In [11]:
```
learning_rate = 0.005 # If you set this too high, it might explode. If to

def train(category_tensor, line_tensor):
    hidden = rnn.initHidden()

    rnn.zero_grad()

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

    loss = criterion(output, category_tensor)
    loss.backward()

    # Add parameters' gradients to their values, multiplied by learning r
    for p in rnn.parameters():
        p.data.add_(p.grad.data, alpha=-learning_rate)

    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.

In [12]:

```
import time
import math

n_iters = 100000
print_every = 5000
plot_every = 1000



# Keep track of losses 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, line, category_tensor, line_tensor = randomTrainingExample(
    output, loss = train(category_tensor, line_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 * 10

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

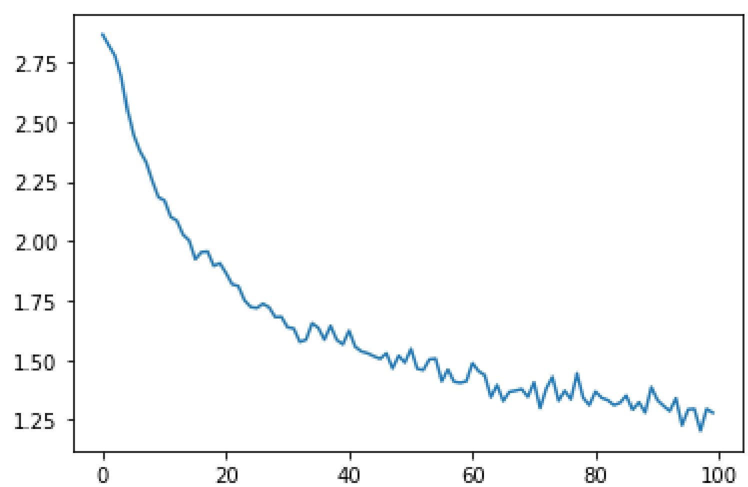
```
5000 5% (0m 6s) 1.7866 Zhong / Chinese ✓
10000 10% (0m 12s) 2.1799 Kang / Chinese ✗ (Korean)
15000 15% (0m 19s) 2.4574 Dunmore / French ✗ (English)
20000 20% (0m 25s) 2.3446 Arnall / Irish ✗ (English)
25000 25% (0m 32s) 0.9424 She / Chinese ✓
30000 30% (0m 38s) 2.1284 Severijns / Greek ✗ (Dutch)
35000 35% (0m 44s) 2.1564 Espinosa / Czech ✗ (Spanish)
40000 40% (0m 52s) 2.3306 Charnock / Scottish ✗ (English)
45000 45% (0m 59s) 0.8680 Banos / Greek ✓
50000 50% (1m 5s) 2.0736 Ubina / Italian ✗ (Spanish)
55000 55% (1m 12s) 1.5325 Bock / Czech ✓
60000 60% (1m 18s) 0.0819 Niemczyk / Polish ✓
65000 65% (1m 24s) 3.6960 Rey / Korean ✗ (Spanish)
70000 70% (1m 31s) 0.1907 Tzarevsky / Russian ✓
75000 75% (1m 37s) 2.6453 Essop / Scottish ✗ (English)
80000 80% (1m 43s) 0.4020 Yeon / Korean ✓
85000 85% (1m 48s) 0.5061 Hatov / Russian ✓
90000 90% (1m 54s) 2.0221 Koeman / Irish ✗ (Dutch)
95000 95% (2m 0s) 1.5055 Peatain / French ✗ (Irish)
100000 100% (2m 6s) 0.0251 Wronski / Polish ✓
```

Plotting the Results

Plotting the historical loss from `all_losses` shows the network learning:

In [13]:
```
import matplotlib.pyplot as plt
import matplotlib.ticker as ticker

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

Out[13]:
```
[<matplotlib.lines.Line2D at 0x7fb03083b430>]
```

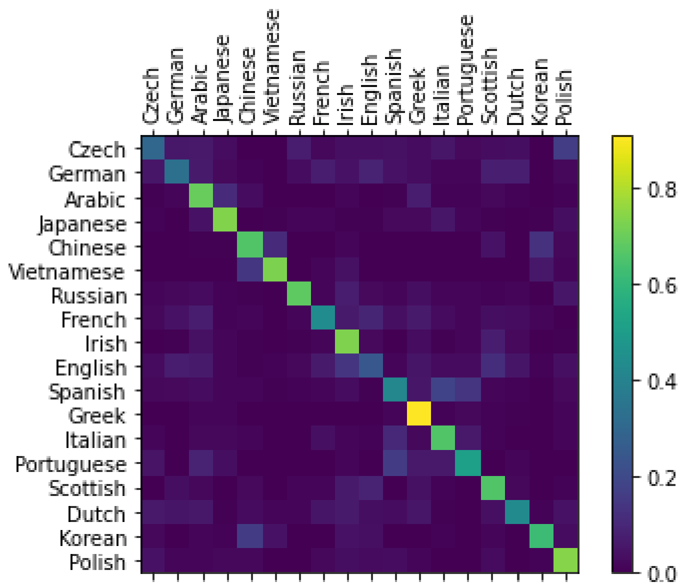


Evaluating the Results

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]:
```python
# Keep track of correct guesses in a confusion matrix
confusion = torch.zeros(n_categories, n_categories)
n_confusion = 10000

# Just return an output given a line
def evaluate(line_tensor):
    hidden = rnn.initHidden()

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

    return output

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

# 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([''] + all_categories, rotation=90)
ax.set_yticklabels([''] + all_categories)

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

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

```
<ipython-input-14-a5b341ffc3a3>:33: UserWarning: FixedFormatter should
only be used together with FixedLocator
  ax.set_xticklabels([''] + all_categories, rotation=90)
<ipython-input-14-a5b341ffc3a3>:34: UserWarning: FixedFormatter should
only be used together with FixedLocator
  ax.set_yticklabels([''] + all_categories)
```



You can pick out bright spots off the main axis that show which languages it guesses incorrectly, e.g. Chinese for Korean, and Spanish for Italian. It seems to do very well with Greek, and very poorly with English (perhaps because of overlap with other languages).

Running on User Input

In [15]:
```
def predict(input_line, n_predictions=3):
    print('\n> %s' % input_line)
    with torch.no_grad():
        output = evaluate(lineToTensor(input_line))

        # Get top N categories
        topv, topi = output.topk(n_predictions, 1, True)
        predictions = []

        for i in range(n_predictions):
            value = topv[0][i].item()
            category_index = topi[0][i].item()
            print('(%.2f) %s' % (value, all_categories[category_index]))
            predictions.append([value, all_categories[category_index]])

predict('Dovesky')
predict('Jackson')
predict('Satoshi')
```

```
> Dovesky
(-0.46) Russian
(-1.56) Czech
(-2.61) English

> Jackson
(-0.37) Scottish
(-1.73) English
(-2.83) Russian

> Satoshi
(-0.73) Japanese
(-1.58) Arabic
(-2.10) Italian
```

The final versions of the scripts [in the Practical PyTorch repo (https://github.com/spro/practical-pytorch/tree/master/char-rnn-classification)](https://github.com/spro/practical-pytorch/tree/master/char-rnn-classification)_ split the above code into a few files:

- `data.py` (loads files)
- `model.py` (defines the RNN)
- `train.py` (runs training)
- `predict.py` (runs `predict()` with command line arguments)
- `server.py` (serve prediction as a JSON API with bottle.py)

Run `train.py` to train and save the network.

Run `predict.py` with a name to view predictions:

::

```
$ python predict.py Hazaki
(-0.42) Japanese
(-1.39) Polish
(-3.51) Czech
```

Exercises

- Try with a different dataset of line -> category, for example:
  - Any word -> language
  - First name -> gender
  - Character name -> writer
  - Page title -> blog or subreddit
- Get better results with a bigger and/or better shaped network
  - Add more linear layers
  - Try the `nn.LSTM` and `nn.GRU` layers
  - Combine multiple of these RNNs as a higher level network