Character RNN Classification Demo

Adapted from PyTorch tutorial

https://pytorch.org/tutorials/intermediate/char_rnn_classification_tutorial.html

Originally accessed on 03-28-2023

David I. Inouye

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

Original Author: Sean Robertson

- We will be building and training a basic character-level RNN to classify words.
- 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
```

Download and extract name data

```python
import urllib.request
import zipfile
import os

# Check if data directory already exists
if os.path.exists('data/names') and os.listdir('data/names'):
    print("Data already exists, skipping download.")
else:
    # Download the data
    url = "https://download.pytorch.org/tutorial/data.zip"
    filename = "data.zip"

    print("Downloading and extracting data...")
    urllib.request.urlretrieve(url, filename)
    with zipfile.ZipFile(filename, 'r') as zip_ref:
        zip_ref.extractall('.')
    os.remove(filename)
    print("Data downloaded and extracted successfully!")
```

```
Data already exists, skipping download.
```

- 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.

Check the data and setup possible characters

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

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

print(findFiles('data/names/*.txt'))

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

# Turn a Unicode string to plain ASCII, thanks to https://stackoverflow.com/a/518232/2809427
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/Greek.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
```

Build dictionary of names

```
# 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)
print(category_lines['Italian'][:5])
```

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

- 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.

Turning Names into Tensors

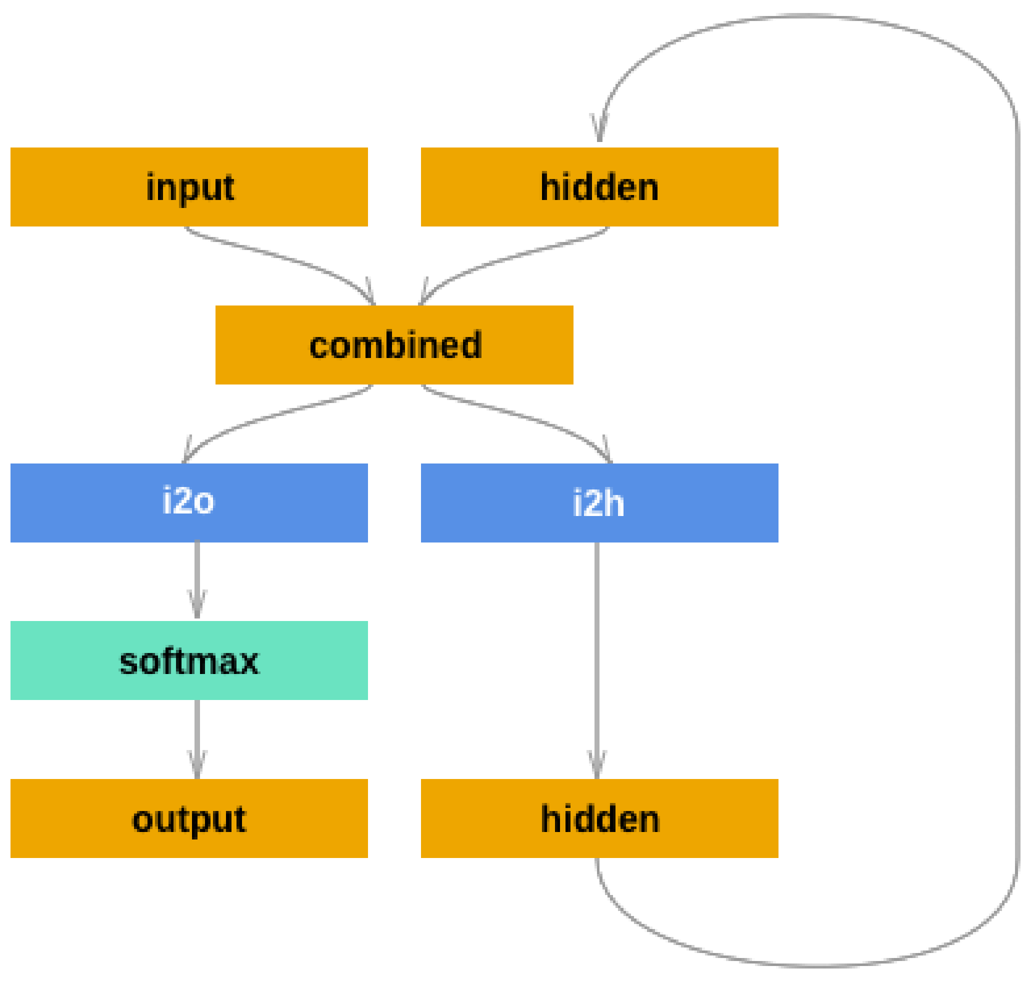
- To represent a single letter, we use a “one-hot vector” of size `<1 x n_letters>`.
  - A one-hot vector is filled with 0s except for a 1 at index of the current letter, e.g. `"b" = <0 1 0 0 0 ...>`.
- To make a word we join a bunch of those into a 2D matrix `<line_length x 1 x n_letters>`.
  - That extra 1 dimension is because PyTorch assumes everything is in batches - we’re just using a batch size of 1 here.

Turning Names into Tensors

```python
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., 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])
```

Creating the Network




```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
        self.i2o = nn.Linear(input_size + hidden_size, output
        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)
```

Running this RNN

- 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 = lineToTensor('Albert')
hidden = torch.zeros(1, n_hidden)
output, next_hidden = rnn(input[0], hidden)
print(output)
```

```
tensor([[-2.8752, -2.8544, -2.8862, -2.9186, -2.9392, -2.9018, -2.9728, -2.8836,
         -2.9022, -2.9242, -2.8132, -2.7843, -2.8920, -2.9446, -2.8656, -2.8909,
         -2.9023, -2.8926]], grad_fn=<LogSoftmaxBackward0>)
```

- 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).

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):
    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)
```

Preparing for Training

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

```python
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)], dtype=torch.long)
    line_tensor = lineToTensor(line)
    return category, line, category_tensor, line_tensor

for i in range(5):
    category, line, category_tensor, line_tensor = randomTrainingExample()
    print('category =', category, '/ line =', line)
```

```
category = Czech / line = Kirchma
category = Scottish / line = Young
category = Polish / line = Zawisza
category = Spanish / line = De leon
category = Japanese / line = Jippensha
```

Training the Network

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

```
criterion = nn.NLLLoss()
learning_rate = 0.005 # If you set this too high, it might ex

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

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 b
    for p in rnn.parameters():
        p.data.add_(p.grad.data, alpha=-learning_rate)

    return output, loss.item()
```

Training: Looping through names

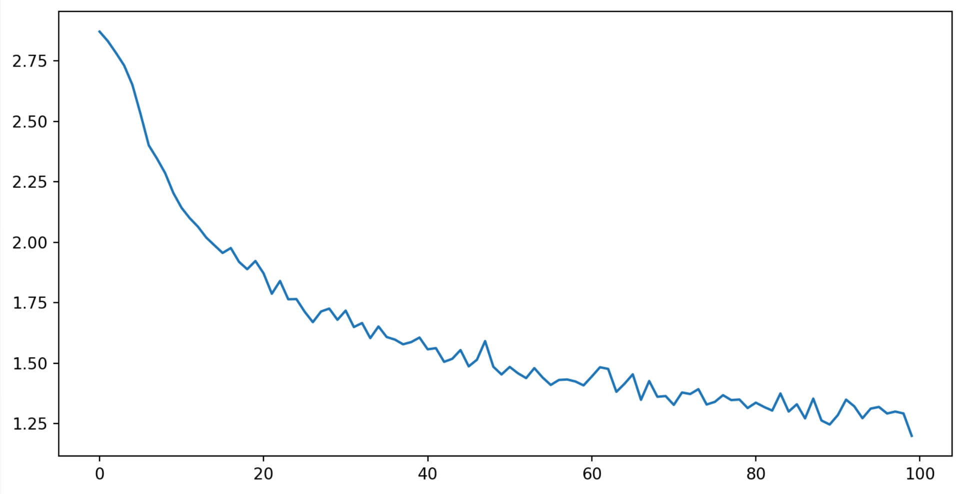
```python
import time
import math

n_iters = 100000
print_every = 5000
plot_every = 1000
current_loss = 0
all_losses = []
start = time.time()
for iter in range(1, n_iters + 1):
    category, line, category_tensor, line_tensor = randomTrai
    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)' % ca
        print('%d %d%% (%s) %.4f %s / %s %s' % (iter, iter /

    # 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 2s) 3.0481 Mikhail / Greek ✗ (Arabic)
10000 10% (0m 5s) 3.3924 Hachirobei / Italian ✗ (Japanese)
15000 15% (0m 7s) 1.1472 Dao / Chinese ✗ (Vietnamese)
20000 20% (0m 10s) 1.0616 Nahas / Arabic ✓
25000 25% (0m 13s) 2.4398 Attia / Spanish ✗ (Arabic)
30000 30% (0m 15s) 1.8406 Buhr / German ✓
35000 35% (0m 18s) 1.2269 Santos / Arabic ✗ (Portuguese)
40000 40% (0m 20s) 1.6114 Eoin / Irish ✓
45000 45% (0m 23s) 1.8291 Armando / Italian ✗ (Spanish)
50000 50% (0m 26s) 2.0408 Tyler / German ✗ (English)
55000 55% (0m 28s) 2.1278 Mordovin / French ✗ (Russian)
60000 60% (0m 31s) 2.0012 Rinn / Chinese ✗ (Irish)
65000 65% (0m 33s) 2.5728 Pinter / German ✗ (Czech)
70000 70% (0m 36s) 1.1657 Hong / Chinese ✗ (Korean)
75000 75% (0m 39s) 0.6283 Filipek / Polish ✓
80000 80% (0m 41s) 1.7897 Dertilis / Portuguese ✗ (Greek)
85000 85% (0m 44s) 2.4244 Kumiega / Japanese ✗ (Polish)
90000 90% (0m 46s) 1.3739 Atherstone / French ✗ (English)
95000 95% (0m 49s) 1.9437 Adlam / Arabic ✗ (English)
100000 100% (0m 51s) 1.2604 Kenward / English ✓
```

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)
```



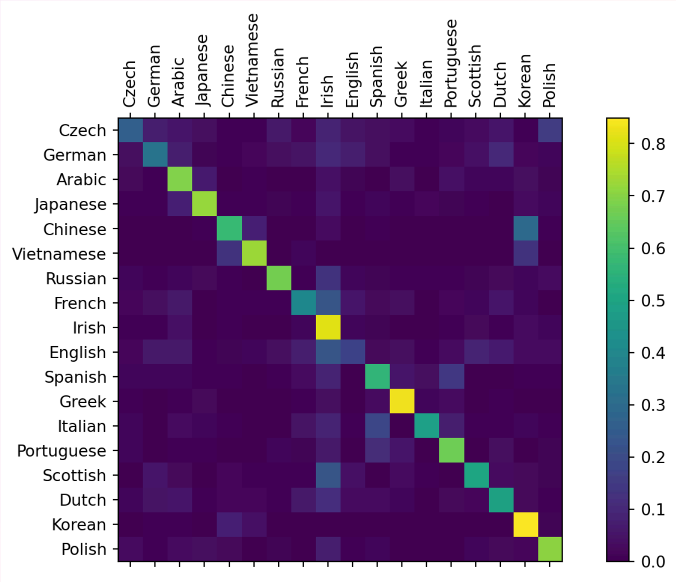
Evaluating the Results via Confusion Matrix

- To see how well the network performs on different categories, we will create a confusion matrix.
  - A confusion matrix indicates 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.

```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 correct
for i in range(n_confusion):
    category, line, category_tensor, line_tensor = randomTrai
    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()
```

Evaluating the Results via Confusion Matrix

```
# 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()
```



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).

Deployment: Running on User Input

```python
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[catego
            predictions.append([value, all_categories[categor

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

```
> Dovesky
(-0.48) Russian
(-1.31) Czech
(-2.95) English

> Jackson
(-0.16) Scottish
(-2.44) English
(-3.90) Dutch

> Satoshi
(-1.13) Polish
(-1.61) Japanese
(-1.91) Italian
```