

```
%matplotlib inline
```

Double-click (or enter) to edit

```
from google.colab import drive
drive.mount('/content/drive')
```

☞ Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.mount(

```
import os
os.chdir('/content/drive/My Drive/cs505/char_rnn_tutorial') #change dir
!pwd
```

☞ /content/drive/My Drive/cs505/char_rnn_tutorial

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. A character-level RNN outputs a prediction and "hidden state" at each step, feeding its previous hidden state into each new step. 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 the word belongs to.

::

```
$ 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:

- <http://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 <<http://karpathy.github.io/2015/05/21/rnn-effectiveness/>> __ shows a bunch of real life examples
- Understanding LSTM Networks <<http://colah.github.io/posts/2015-08-Understanding-LSTMs/>> but also informative about RNNs in general

▼ Preparing the Data

.. Note:: Download the data from here <<https://download.pytorch.org/tutorial/data.zip>> _{
Included in the `data/names` directory are 18 text files named as "[Language].txt". Each file contains a 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 general language and name in our case) are used for later extensibility.

```
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/cities_train/*.txt'))

import unicodedata
import string

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

# Turn a Unicode string to plain ASCII, thanks to http://stackoverflow.com/a/518232/2
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 = {}
val_category_lines = {}
all_categories = []
val_categories = []
```

```

all_categories = []

# Read a file and split into lines
def readLines(filename):
    lines = open(filename, encoding="ISO-8859-1").read().split('\n')
    return [unicodeToAscii(line) for line in lines]

for filename in findFiles('data/cities_train/*.txt'):
    category = os.path.splitext(os.path.basename(filename))[0]
    all_categories.append(category)
    lines = readLines(filename)[-1]
    category_lines[category] = lines

n_categories = len(all_categories)

for filename in findFiles('data/cities_val/*.txt'):
    category = os.path.splitext(os.path.basename(filename))[0]
    val_categories.append(category)
    lines = readLines(filename)[-1]
    val_category_lines[category] = lines

☞ ['data/cities_train/cn.txt', 'data/cities_train/za.txt', 'data/cities_train/de.txt',
   'data/cities_train/sl.txt', 'data/cities_train/slusarski.txt']

```

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

```

print(category_lines['cn'][-5:])
print(val_category_lines['cn'][-5:])

☞ ['cuizongzhuang', 'hetou', 'hulstai', 'shuanglazi', 'tebongori']
   ['xueguangzhang', 'ian', 'niujiang', 'shuipo', 'daohugou']

```

▼ 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 $\langle 1 \times n_{\text{letters}} \rangle$. A one-hot vector is 1 at the index of the current letter, e.g. "b" = $\langle 0 \ 1 \ 0 \ 0 \ 0 \ \dots \rangle$.

To make a word we join a bunch of those into a 2D matrix $\langle \text{line_length} \times 1 \times n_{\text{letters}} \rangle$.

That extra 1 dimension is because PyTorch assumes everything is in batches - we're just using a batch of size 1.

```

import torch

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

```



```

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 initialize as zeros at first). We'll get back the output (probability of each language) and a next hidden state.

```

input = letterToTensor('A')
hidden = torch.zeros(1, n_hidden)

output, next_hidden = rnn(input, hidden)
print(output)

```

```

↳ tensor([[ -2.1730, -2.1819, -2.1385, -2.2335, -2.2882, -2.2238, -2.1204, -2.1685,
           -2.2598]], grad_fn=<LogSoftmaxBackward>)

```

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

```

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

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

```

```

↳ tensor([[ -2.1730, -2.1819, -2.1385, -2.2335, -2.2882, -2.2238, -2.1204, -2.1685,
           -2.2598]], grad_fn=<LogSoftmaxBackward>)

```

As you can see the output is a `<1 x n_categories>` Tensor, where every item is the likelihood of the

▼ 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 as the 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))
```

```
↳ ('pk', 6)
```

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(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
```

```
def randomValidationExample():
    category = randomChoice(val_categories)
    line = randomChoice(val_category_lines[category])
    val_category_tensor = torch.tensor([val_categories.index(category)], dtype=torch.long)
    val_line_tensor = lineToTensor(line)
    return category, line, val_category_tensor, val_line_tensor
```

```
def shuffle_arrs(a,b,c,d):
    combined = list(zip(a, b, c, d))
    random.shuffle(combined)
    a, b, c, d = zip(*combined)
    return a,b,c,d
```

```
def genData(category_line_hash, categories_arr):
    x, y, x_tensor, y_tensor = [], [], [], []
    for y_category in category_line_hash.keys():
        for x_line in category_line_hash[y_category]:
            y.append(y_category)
            x.append(x_line)
            y_tensor.append(torch.tensor([categories_arr.index(y_category)], dtype=torch.long))
            x_tensor.append(lineToTensor(x_line))
    x, y, x_tensor, y_tensor = shuffle_arrs(x, y, x_tensor, y_tensor)
    return x, y, x_tensor, y_tensor
```

```
def TrainingData():
    return genData(category_lines, all_categories)

# v = 1
```

```

π y = []
# x = []
# for y_category in category_lines.keys():
#     for x_line in category_lines[y_category]:
#         y.append(y_category)
#         x.append(x_line)
#         y_tensor.append(torch.tensor([val_categories.index(category)], dtype=to
#         x_tensor.append(lineToTensor(x_line))
# x, y, x_tensor, y_tensor = shuffle_arrs(x, y, x_tensor, y_tensor)
# return x, y

def ValidationData():
    return genData(val_category_lines, val_categories)
# y = []
# x = []
# y_tensor = []
# x_tensor = []
# for y_category in val_category_lines.keys():
#     for x_line in val_category_lines[y_category]:
#         y.append(y_category)
#         x.append(x_line)
#         y_tensor.append(torch.tensor([val_categories.index(category)], dtype=to
#         x_tensor.append(lineToTensor(x_line))
# x, y, x_tensor, y_tensor = shuffle_arrs(x, y, x_tensor, y_tensor)
# return x, y

print("=== Train ===")
x,y,x_tensor,y_tensor= TrainingData()
print(x[:5])
print(y[:5])
# print(x_tensor[:1])
# print(y_tensor[:1])

print("=== Validation ===")
x,y,x_tensor,y_tensor = ValidationData()
print(x[:5])
print(y[:5])
# print(x_tensor[:1])
# print(y_tensor[:1])

☐> === Train ===
('iskatel', 'kleinneundorf', 'kolkhozabag', 'jhamrah', 'bodah')
('af', 'in', 'za', 'pk', 'af')
=== Validation ===
('counenguious', 'zasun', 'bagzai', 'asolmah', 'klein muckrow')
('fr', 'za', 'pk', 'af', 'de')

```

▼ 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
 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 = 0.000005 # If you set this too high, it might explode. If too low, it
```

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

    rnn.zero_grad()

    # print("category_tensor={}, line_tensor.size()[0]={}".format(category_tensor, li
    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 rate
    for p in rnn.parameters():
        p.data.add_(-learning_rate, p.grad.data)

    return output, loss.item()

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

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

    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 the loss, we also keep track of loss for plotting. Since there are 1000s of examples we print only every `print_every` loss.

```

import time
import math

print_every = 1000 # total = 27000
print_every = 10 # 5000

# keep track of losses for plotting
current_loss = 0
all_losses = []
timeSince = []
all_losses_thru_time = []

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()

print("learning rate = ", learning_rate)

train_loader, y_train, x_train_tensor, y_train_tensor = TrainingDataLoader(
    data_loader, y_val, x_val_tensor, y_val_tensor = ValidationDataLoader()

train_loader_len = len(x_train)
val_loader_len = 10 # len(x_val)
print("x_train_len:", x_train_len, ", x_val_len:", x_val_len)

for i in range(x_train_len):
    # category, line, category_tensor, line_tensor = randomTrainingExample() # TODO: re
    category = y_train[i]
    line = x_train[i]
    category_tensor = y_train_tensor[i]
    line_tensor = x_train_tensor[i]

    output, loss = train(category_tensor, line_tensor)
    current_loss += loss

    val_loss_per_train_data = 0
    # for j in range(x_val_len):
    #     val_output, val_loss = evaluate(x_val_tensor[j], y_val_tensor[j])
    #     val_loss_per_train_data += val_loss
    for j in range(x_val_len):

```

```
_, _, val_category_tensor, val_line_tensor = randomValidationExample() # TODO:
val_output, val_loss = evaluate(val_line_tensor, val_category_tensor)
val_loss_per_train_data += val_loss
```

```
val_loss_per_train_data_ave = val_loss_per_train_data / x_val_len
val_losses += val_loss_per_train_data
```

```
# Print iter number, loss, name and guess
```

```
if i % print_every == 0:
```

```
    print("iter = {}({:d}%) | time taken = {} | train_loss={:.4f}, val_loss(ave)={:.4f}"
```

```
# Add current loss avg to list of losses
```

```
if i % plot_every == 0:
```

```
    train_losses_thru_time.append(current_loss / plot_every)
```

```
    val_losses_thru_time.append(val_losses / plot_every)
```

```
    current_loss = 0
```

```
    val_losses = 0
```

```
↳ learning rate = 5e-06
```

```
x_train_len: 27000 , x_val_len: 10
```

```
iter = 0(0%) | time taken = 0m 3s | train_loss=1.1295, val_loss(ave)=3.6670
```

```
iter = 1000(3%) | time taken = 0m 13s | train_loss=0.3921, val_loss(ave)=4.0183
```

```
iter = 2000(7%) | time taken = 0m 23s | train_loss=1.7498, val_loss(ave)=3.1904
```

```
iter = 3000(11%) | time taken = 0m 34s | train_loss=2.3342, val_loss(ave)=3.9280
```

```
iter = 4000(14%) | time taken = 0m 44s | train_loss=0.1390, val_loss(ave)=3.0630
```

```
iter = 5000(18%) | time taken = 0m 54s | train_loss=0.2785, val_loss(ave)=4.5074
```

```
iter = 6000(22%) | time taken = 1m 4s | train_loss=2.0313, val_loss(ave)=3.9842
```

```
iter = 7000(25%) | time taken = 1m 15s | train_loss=1.2922, val_loss(ave)=3.7304
```

```
iter = 8000(29%) | time taken = 1m 26s | train_loss=1.7926, val_loss(ave)=3.5959
```

```
iter = 9000(33%) | time taken = 1m 36s | train_loss=1.2283, val_loss(ave)=3.8076
```

```
iter = 10000(37%) | time taken = 1m 46s | train_loss=1.0459, val_loss(ave)=3.1557
```

```
iter = 11000(40%) | time taken = 1m 57s | train_loss=0.3086, val_loss(ave)=4.0040
```

```
iter = 12000(44%) | time taken = 2m 7s | train_loss=3.2619, val_loss(ave)=3.8911
```

```
iter = 13000(48%) | time taken = 2m 18s | train_loss=0.2181, val_loss(ave)=3.4830
```

```
iter = 14000(51%) | time taken = 2m 28s | train_loss=0.7031, val_loss(ave)=3.1017
```

```
iter = 15000(55%) | time taken = 2m 39s | train_loss=3.6834, val_loss(ave)=3.6361
```

```
iter = 16000(59%) | time taken = 2m 49s | train_loss=1.0188, val_loss(ave)=4.5630
```

```
iter = 17000(62%) | time taken = 2m 59s | train_loss=2.6995, val_loss(ave)=2.8297
```

```
iter = 18000(66%) | time taken = 3m 9s | train_loss=1.0234, val_loss(ave)=3.5749
```

```
iter = 19000(70%) | time taken = 3m 20s | train_loss=1.2653, val_loss(ave)=4.1659
```

```
iter = 20000(74%) | time taken = 3m 30s | train_loss=0.5556, val_loss(ave)=3.0668
```

```
iter = 21000(77%) | time taken = 3m 40s | train_loss=1.9955, val_loss(ave)=3.6242
```

```
iter = 22000(81%) | time taken = 3m 50s | train_loss=3.7275, val_loss(ave)=3.0829
```

```
iter = 23000(85%) | time taken = 4m 1s | train_loss=1.6796, val_loss(ave)=3.0614
```

```
iter = 24000(88%) | time taken = 4m 11s | train_loss=1.0166, val_loss(ave)=3.1981
```

```
iter = 25000(92%) | time taken = 4m 21s | train_loss=2.4082, val_loss(ave)=4.1479
```

▼ Plotting the Results

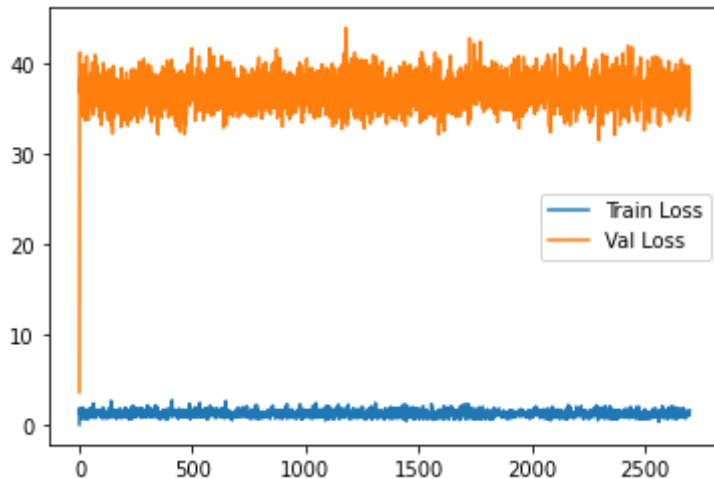
Plotting the historical loss from `all_losses` shows the network learning:

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

```
import matplotlib.ticker as ticker
```

```
plt.figure()
train_loss_plot = plt.plot(train_losses_thru_time, label='Train Loss')
val_loss_plot = plt.plot(val_losses_thru_time, label="Val Loss")
plt.legend()
```

☐ <matplotlib.legend.Legend at 0x7ff669db7320>



▼ Evaluating the Results

To see how well the network performs on different categories, we will create a confusion matrix, indicating which language the network guesses (columns). To calculate the confusion matrix a bunch of samples are passed through `evaluate()`, which is the same as `train()` minus the backprop.

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

n_confusion = 500 # 10000

# 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, category_tensor)
    guess, guess_i = categoryFromOutput(output)
    category_i = all_categories.index(category)
    confusion[category_i][guess_i] += 1
    confusion_no_norm[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()
```

```

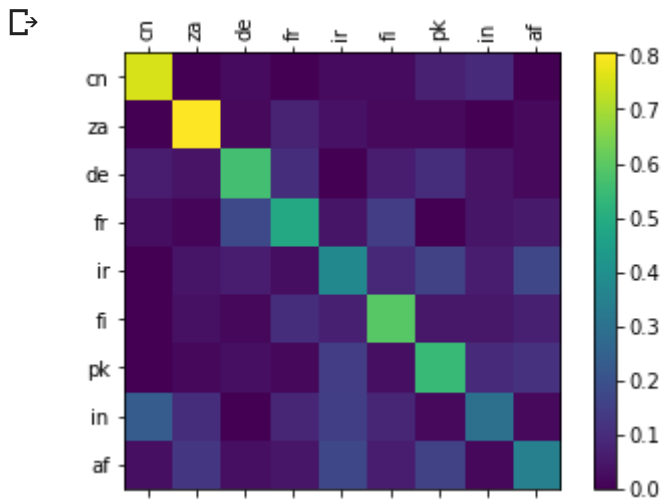
ax = fig.add_subplot(111)
cax = ax.matshow(confusion.numpy())
fig.colorbar(cax)

# 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()

```



```

print("confusion matrix (no normalization)")
print(confusion_no_norm)

def get_multi_class_accuracy(confusion):
    total = len(confusion)
    correct = 0
    for guess_i in range(len(confusion)):
        correct += confusion[guess_i][guess_i]
    return correct / total

print("accuracy, multi-class = {}".format(get_multi_class_accuracy(confusion_no_norm)))

confusion matrix (no normalization)
tensor([[31., 0., 1., 0., 1., 1., 3., 4., 0.],
        [ 0., 41., 1., 4., 2., 1., 1., 0., 1.],
        [ 3., 2., 27., 5., 0., 3., 5., 2., 1.],
        [ 2., 1., 12., 33., 3., 10., 0., 3., 4.],
        [ 0., 3., 4., 2., 24., 6., 10., 4., 11.],
        [ 0., 2., 1., 6., 4., 34., 3., 3., 4.],
        [ 0., 1., 2., 1., 9., 2., 33., 6., 7.],
        [11., 5., 0., 4., 7., 4., 1., 14., 1.],
        [ 2., 8., 2., 3., 11., 4., 10., 1., 22.]])
accuracy, multi-class = 28.77777862548828

```

```

def get_pos_tp(target_i, confusion):
    pos = torch.sum(confusion[:, target_i])
    tp = confusion[target_i][target_i]
    return pos, tp

def get_multi_class_precision(confusion):
    '''
    multi-class-precision = sum(all tp's across class) / sum(all pos' across class)
    '''
    pos = 0
    tp = 0
    for i in range(len(confusion)):
        target_pos, target_tp = get_pos_tp(i, confusion)
        pos += target_pos
        tp += target_tp

    precision = tp / pos
    return precision

print("precision, multi-class = {}".format(get_multi_class_precision(confusion_no_nor

↳ precision, multi-class = 0.5180000066757202

```

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

▼ Running on User Input

```

# 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')

```

The final versions of the scripts in the Practical PyTorch repo <<https://github.com/spro/p-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
```

Run `server.py` and visit <http://localhost:5533/Yourname> to get JSON output of predictions.

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

