

```
%matplotlib inline
```

Double-click (or enter) to edit

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

☞ Go to this URL in a browser: https://accounts.google.com/o/oauth2/auth?client_id=

Enter your authorization code:

.....

Mounted at /content/drive

```
import os
os.chdir('/content/drive/My Drive/cs505/char_rnn_tutorial') #achange 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 prediction is 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
```

Now we have `category_lines`, a dictionary mapping each category (language) to a list of lines (names) for 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', 'niujiaoxu', 'shuipo', 'daohugou']
```

Now that we have all the names organized, we need to turn them into Tensors to make any use of the

To represent a single letter, we use a "one-hot vector" of size $\langle 1 \times n_letters \rangle$. A one-hot vector is 1 for the current letter, e.g. "b" = $\langle 0 \ 1 \ 0 \ 0 \ 0 \ \dots \rangle$.

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

```

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., 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., 0.]])
torch.Size([5, 1, 57])

```

▼ Creating the Network

Before autograd, creating a recurrent neural network in Torch involved cloning the parameters of a lay hidden state and gradients which are now entirely handled by the graph itself. This means you can im regular feed-forward layers.

This RNN module (mostly copied from the PyTorch for Torch users tutorial <http://pytorch.org/tutorials/beginner/former_torchies/nn_tutorial.html#example-2-which-operate-on-an-input-and-hidden-state,with-a-LogSoftmax-layer-after-the-output> which operate on an input and hidden state, with a LogSoftmax layer after the output.

.. figure:: <https://i.imgur.com/Z2xbySQ.png> :alt:

```

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 hidden state (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([[ -2.2620, -2.1309, -2.1623, -2.1152, -2.1508, -2.2480, -2.2619, -2.2789,
           -2.1812]], 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([[ -2.2620, -2.1309, -2.1623, -2.1152, -2.1508, -2.2480, -2.2619, -2.2789,
           -2.1812]], 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 model, which is a list of probabilities for each category. We can use `max()` 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))
```

```
↳ ('fr', 3)
```

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)
    # 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 ===
('dankerode', 'khorkal', 'schaphausen', 'atin', 'leobalde')
('de', 'pk', 'de', 'in', 'de')
=== Validation ===
('galehye now abraj', 'kofime', 'xaffevillers', 'lujiawobao', 'kottenheide')
('ir', 'fi', 'fr', 'cn', '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 = 0.001 # If you set this too high, it might explode. If too low, it mi
```

```
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
plot_every = 1000 # 5000
```

```
# Keep track of losses for plotting
current_loss = 0
val_losses = 0.
train_acc_thru_time_aggregate, val_acc_thru_time_aggregate = 0., 0.
train_losses_thru_time = []
val_losses_thru_time = []
train_acc_thru_time = []
val_acc_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)
```

```
x_train, y_train, x_train_tensor, y_train_tensor = TrainingData()
x_val, y_val, x_val_tensor, y_val_tensor = ValidationData()
```

```
x_train_len = len(x_train)
x_val_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:
    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
val_correct_guess_count = 0
train_correct_guess_count = 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):
    # Train accuracy calc
    train_category, _, train_category_tensor, train_line_tensor = randomTrainingE
    train_output, train_loss = evaluate(train_line_tensor, train_category_tensor)
    train_guess, _ = categoryFromOutput(train_output)
    train_correct_guess_count += int(train_guess == train_category)

    # Validation accuracy calc
    val_category, _, val_category_tensor, val_line_tensor = randomValidationExamp
    val_output, val_loss = evaluate(val_line_tensor, val_category_tensor)
    val_guess, _ = categoryFromOutput(val_output)
    val_correct_guess_count += int(val_guess == val_category)

    val_loss_per_train_data += val_loss

# Aggregate accuracy
train_acc_per_train_data = train_correct_guess_count / x_val_len
train_acc_thru_time_aggregate += train_acc_per_train_data
val_acc_per_train_data = val_correct_guess_count / x_val_len
val_acc_thru_time_aggregate += val_acc_per_train_data

# Aggregate validation loss
val_loss_per_train_data_ave = val_loss_per_train_data / x_val_len
val_losses += val_loss_per_train_data_ave

# Print iter number, loss, name and guess
if i % print_every == 0:
    print("iter = {}({:d}%) | time taken = {} | train_loss={:.4f}, val_loss(ave)=
    debug_x, debug_y, debug_x_tensor, debug_y_tensor = [], [], [], []

# 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

    print("iter = {}({:d}%) | time taken = {} | train_acc_thru_time_ave={}, val_a

    train_acc_thru_time.append(train_acc_thru_time_aggregate / plot_every)
    val_acc_thru_time.append(val_acc_thru_time_aggregate / plot_every)
    train_acc_thru_time_aggregate = 0
    val_acc_thru_time_aggregate = 0

```

```

↳ learning rate = 0.001
x_train_len: 27000 , x_val_len: 10
iter = 0(0%) | time taken = 0m 3s | train_loss=2.2668, val_loss(ave)=2.1916 | tra
iter = 0(0%) | time taken = 0m 3s | train_acc_thru_time_ave=0.0002, val_acc_thru
iter = 1000(3%) | time taken = 0m 23s | train_loss=2.2082, val_loss(ave)=2.1939
iter = 1000(3%) | time taken = 0m 23s | train_acc_thru_time_ave=0.133199999999999
iter = 2000(7%) | time taken = 0m 44s | train_loss=2.1503, val_loss(ave)=2.1937
iter = 2000(7%) | time taken = 0m 44s | train_acc_thru_time_ave=0.139899999999999
iter = 3000(11%) | time taken = 1m 4s | train_loss=2.1270, val_loss(ave)=2.1756
iter = 3000(11%) | time taken = 1m 4s | train_acc_thru_time_ave=0.152099999999999
iter = 4000(14%) | time taken = 1m 25s | train_loss=2.2000, val_loss(ave)=2.2164
iter = 4000(14%) | time taken = 1m 25s | train_acc_thru_time_ave=0.164699999999999
iter = 5000(18%) | time taken = 1m 46s | train_loss=2.1128, val_loss(ave)=2.1962
iter = 5000(18%) | time taken = 1m 46s | train_acc_thru_time_ave=0.184699999999999
iter = 6000(22%) | time taken = 2m 7s | train_loss=2.2378, val_loss(ave)=2.2306
iter = 6000(22%) | time taken = 2m 7s | train_acc_thru_time_ave=0.206799999999999
iter = 7000(25%) | time taken = 2m 27s | train_loss=2.2748, val_loss(ave)=2.1729
iter = 7000(25%) | time taken = 2m 27s | train_acc_thru_time_ave=0.235099999999999
iter = 8000(29%) | time taken = 2m 48s | train_loss=2.0878, val_loss(ave)=2.1948
iter = 8000(29%) | time taken = 2m 48s | train_acc_thru_time_ave=0.250499999999999
iter = 9000(33%) | time taken = 3m 9s | train_loss=2.1558, val_loss(ave)=2.2312
iter = 9000(33%) | time taken = 3m 9s | train_acc_thru_time_ave=0.243699999999999
iter = 10000(37%) | time taken = 3m 30s | train_loss=2.0200, val_loss(ave)=2.2160
iter = 10000(37%) | time taken = 3m 30s | train_acc_thru_time_ave=0.250899999999999
iter = 11000(40%) | time taken = 3m 51s | train_loss=2.2084, val_loss(ave)=2.2437
iter = 11000(40%) | time taken = 3m 51s | train_acc_thru_time_ave=0.246599999999999
iter = 12000(44%) | time taken = 4m 11s | train_loss=2.2243, val_loss(ave)=2.2268
iter = 12000(44%) | time taken = 4m 11s | train_acc_thru_time_ave=0.244099999999999
iter = 13000(48%) | time taken = 4m 32s | train_loss=2.1562, val_loss(ave)=2.2060
iter = 13000(48%) | time taken = 4m 32s | train_acc_thru_time_ave=0.259699999999999
iter = 14000(51%) | time taken = 4m 52s | train_loss=2.0937, val_loss(ave)=2.2144
iter = 14000(51%) | time taken = 4m 52s | train_acc_thru_time_ave=0.270600000000000
iter = 15000(55%) | time taken = 5m 13s | train_loss=1.9818, val_loss(ave)=2.2228
iter = 15000(55%) | time taken = 5m 13s | train_acc_thru_time_ave=0.281400000000000
iter = 16000(59%) | time taken = 5m 34s | train_loss=2.3245, val_loss(ave)=2.2134
iter = 16000(59%) | time taken = 5m 34s | train_acc_thru_time_ave=0.2867, val_acc
iter = 17000(62%) | time taken = 5m 55s | train_loss=2.1697, val_loss(ave)=2.2132
iter = 17000(62%) | time taken = 5m 55s | train_acc_thru_time_ave=0.282100000000000
iter = 18000(66%) | time taken = 6m 16s | train_loss=2.0872, val_loss(ave)=2.2474
iter = 18000(66%) | time taken = 6m 16s | train_acc_thru_time_ave=0.285400000000000
iter = 19000(70%) | time taken = 6m 37s | train_loss=2.4316, val_loss(ave)=2.2461
iter = 19000(70%) | time taken = 6m 37s | train_acc_thru_time_ave=0.2919, val_acc
iter = 20000(74%) | time taken = 6m 58s | train_loss=2.2014, val_loss(ave)=2.1625
iter = 20000(74%) | time taken = 6m 58s | train_acc_thru_time_ave=0.299000000000000
iter = 21000(77%) | time taken = 7m 19s | train_loss=1.8570, val_loss(ave)=2.1631
iter = 21000(77%) | time taken = 7m 19s | train_acc_thru_time_ave=0.305500000000000
iter = 22000(81%) | time taken = 7m 40s | train_loss=2.0498, val_loss(ave)=2.2614
iter = 22000(81%) | time taken = 7m 40s | train_acc_thru_time_ave=0.305300000000000
iter = 23000(85%) | time taken = 8m 1s | train_loss=1.9411, val_loss(ave)=2.2629
iter = 23000(85%) | time taken = 8m 1s | train_acc_thru_time_ave=0.311499999999999
iter = 24000(88%) | time taken = 8m 22s | train_loss=1.9037, val_loss(ave)=2.2534
iter = 24000(88%) | time taken = 8m 22s | train_acc_thru_time_ave=0.311600000000000
iter = 25000(92%) | time taken = 8m 43s | train_loss=2.1473, val_loss(ave)=2.2960
iter = 25000(92%) | time taken = 8m 43s | train_acc_thru_time_ave=0.319299999999999

```

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

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

```
plt.figure()
train_loss_plot = plt.plot(train_losses_thru_time[1:], label='Train Loss')
val_loss_plot = plt.plot(val_losses_thru_time[1:], label="Val Loss")
plt.legend()
```

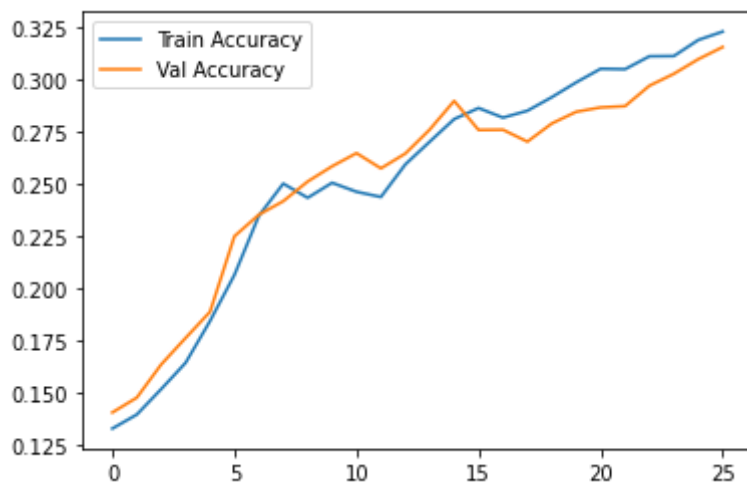
```
print("train_losses_thru_time")
print(train_losses_thru_time[1:])
print("val_losses_thru_time")
print(val_losses_thru_time[1:])
```

```
☐ train_losses_thru_time
[2.1931439759731295, 2.191539811849594, 2.1858927879333496, 2.1824070241451263, 2.1808927879333496, 2.17937879333496, 2.1778593787933496, 2.1763398787933496, 2.1748203787933496, 2.1733008787933496, 2.1717813787933496, 2.1702618787933496, 2.1687423787933496, 2.1672228787933496, 2.1657033787933496, 2.1641838787933496, 2.1626643787933496, 2.1611448787933496, 2.1596253787933496, 2.1581058787933496, 2.1565863787933496, 2.1550668787933496, 2.1535473787933496, 2.1520278787933496, 2.1505083787933496, 2.1489888787933496, 2.1474693787933496, 2.1459498787933496, 2.1444303787933496, 2.1429108787933496, 2.1413913787933496, 2.1398718787933496, 2.1383523787933496, 2.1368328787933496, 2.1353133787933496, 2.1337938787933496, 2.1322743787933496, 2.1307548787933496, 2.1292353787933496, 2.1277158787933496, 2.1261963787933496, 2.1246768787933496, 2.1231573787933496, 2.1216378787933496, 2.1201183787933496, 2.1185988787933496, 2.1170793787933496, 2.1155598787933496, 2.1140403787933496, 2.1125208787933496, 2.1109998787933496, 2.1094793787933496, 2.1079598787933496, 2.1064393787933496, 2.1049198787933496, 2.1033993787933496, 2.1018798787933496, 2.1003593787933496, 2.0988398787933496, 2.0973193787933496, 2.0957998787933496, 2.0942793787933496, 2.0927598787933496, 2.0912393787933496, 2.0897198787933496, 2.0881993787933496, 2.0866798787933496, 2.0851593787933496, 2.0836398787933496, 2.0821193787933496, 2.0805998787933496, 2.0790793787933496, 2.0775598787933496, 2.0760393787933496, 2.0745198787933496, 2.0729993787933496, 2.0714798787933496, 2.0699593787933496, 2.0684398787933496, 2.0669193787933496, 2.0653998787933496, 2.0638793787933496, 2.0623598787933496, 2.0608393787933496, 2.0593198787933496, 2.0577993787933496, 2.0562798787933496, 2.0547593787933496, 2.0532398787933496, 2.0517193787933496, 2.0501998787933496, 2.0486793787933496, 2.0471598787933496, 2.0456393787933496, 2.0441198787933496, 2.0425993787933496, 2.0410798787933496, 2.0395593787933496, 2.0380398787933496, 2.0365193787933496, 2.0349998787933496, 2.0334793787933496, 2.0319598787933496, 2.0304393787933496, 2.0289198787933496, 2.0273993787933496, 2.0258798787933496, 2.0243593787933496, 2.0228398787933496, 2.0213193787933496, 2.0197998787933496, 2.0182793787933496, 2.0167598787933496, 2.0152393787933496, 2.0137198787933496, 2.0121993787933496, 2.0106798787933496, 2.0091593787933496, 2.0076398787933496, 2.0061193787933496, 2.0045998787933496, 2.0030793787933496, 2.0015598787933496, 2.0000393787933496, 1.9985198787933496, 1.9969993787933496, 1.9954798787933496, 1.9939593787933496, 1.9924398787933496, 1.9909193787933496, 1.9893998787933496, 1.9878793787933496, 1.9863598787933496, 1.9848393787933496, 1.9833198787933496, 1.9817993787933496, 1.9802798787933496, 1.9787593787933496, 1.9772398787933496, 1.9757193787933496, 1.9741998787933496, 1.9726793787933496, 1.9711598787933496, 1.9696393787933496, 1.9681198787933496, 1.9665993787933496, 1.9650798787933496, 1.9635593787933496, 1.9620398787933496, 1.9605193787933496, 1.9589998787933496, 1.9574793787933496, 1.9559598787933496, 1.9544393787933496, 1.9529198787933496, 1.9513993787933496, 1.9498798787933496, 1.9483593787933496, 1.9468398787933496, 1.9453193787933496, 1.9437998787933496, 1.9422793787933496, 1.9407598787933496, 1.9392393787933496, 1.9377198787933496, 1.9361993787933496, 1.9346798787933496, 1.9331593787933496, 1.9316398787933496, 1.9301193787933496, 1.9285998787933496, 1.9270793787933496, 1.9255598787933496, 1.9240393787933496, 1.9225198787933496, 1.9209993787933496, 1.9194798787933496, 1.9179593787933496, 1.9164398787933496, 1.9149193787933496, 1.9133998787933496, 1.9118793787933496, 1.9103598787933496, 1.9088393787933496, 1.9073198787933496, 1.9057993787933496, 1.9042798787933496, 1.9027593787933496, 1.9012398787933496, 1.8997193787933496, 1.8981998787933496, 1.8966793787933496, 1.8951598787933496, 1.8936393787933496, 1.8921198787933496, 1.8905993787933496, 1.8890798787933496, 1.8875593787933496, 1.8860398787933496, 1.8845193787933496, 1.8829998787933496, 1.8814793787933496, 1.8799598787933496, 1.8784393787933496, 1.8769198787933496, 1.8753993787933496, 1.8738798787933496, 1.8723593787933496, 1.8708398787933496, 1.8693193787933496, 1.8677998787933496, 1.8662793787933496, 1.8647598787933496, 1.8632393787933496, 1.8617198787933496, 1.8601993787933496, 1.8586798787933496, 1.8571593787933496, 1.8556398787933496, 1.8541193787933496, 1.8525998787933496, 1.8510793787933496, 1.8495598787933496, 1.8480393787933496, 1.8465198787933496, 1.8449993787933496, 1.8434798787933496, 1.8419593787933496, 1.8404398787933496, 1.8389193787933496, 1.8373998787933496, 1.8358793787933496, 1.8343598787933496, 1.8328393787933496, 1.8313198787933496, 1.8297993787933496, 1.8282798787933496, 1.8267593787933496, 1.8252398787933496, 1.8237193787933496, 1.8221998787933496, 1.8206793787933496, 1.8191598787933496, 1.8176393787933496, 1.8161198787933496, 1.8145993787933496, 1.8130798787933496, 1.8115593787933496, 1.8100398787933496, 1.8085193787933496, 1.8069998787933496, 1.8054793787933496, 1.8039598787933496, 1.8024393787933496, 1.8009198787933496, 1.7993993787933496, 1.7978798787933496, 1.7963593787933496, 1.7948398787933496, 1.7933193787933496, 1.7917998787933496, 1.7902793787933496, 1.7887598787933496, 1.7872393787933496, 1.7857198787933496, 1.7841993787933496, 1.7826798787933496, 1.7811593787933496, 1.7796398787933496, 1.7781193787933496, 1.7765998787933496, 1.7750793787933496, 1.7735598787933496, 1.7720393787933496, 1.7705198787933496, 1.7689993787933496, 1.7674798787933496, 1.7659593787933496, 1.7644398787933496, 1.7629193787933496, 1.7613998787933496, 1.7598793787933496, 1.7583598787933496, 1.7568393787933496, 1.7553198787933496, 1.7537993787933496, 1.7522798787933496, 1.7507593787933496, 1.7492398787933496, 1.7477193787933496, 1.7461998787933496, 1.7446793787933496, 1.7431598787933496, 1.7416393787933496, 1.7401198787933496, 1.7385993787933496, 1.7370798787933496, 1.7355593787933496, 1.7340398787933496, 1.7325193787933496, 1.7309998787933496, 1.7294793787933496, 1.7279598787933496, 1.7264393787933496, 1.7249198787933496, 1.7233993787933496, 1.7218798787933496, 1.7203593787933496, 1.7188398787933496, 1.7173193787933496, 1.7157998787933496, 1.7142793787933496, 1.7127598787933496, 1.7112393787933496, 1.7097198787933496, 1.7081993787933496, 1.7066798787933496, 1.7051593787933496, 1.7036398787933496, 1.7021193787933496, 1.7005998787933496, 1.6990793787933496, 1.6975598787933496, 1.6960393787933496, 1.6945198787933496, 1.6929993787933496, 1.6914798787933496, 1.6899593787933496, 1.6884398787933496, 1.6869193787933496, 1.6853998787933496, 1.6838793787933496, 1.6823598787933496, 1.6808393787933496, 1.6793198787933496, 1.6777993787933496, 1.6762798787933496, 1.6747593787933496, 1.6732398787933496, 1.6717193787933496, 1.6701998787933496, 1.6686793787933496, 1.6671598787933496, 1.6656393787933496, 1.6641198787933496, 1.6625993787933496, 1.6610798787933496, 1.6595593787933496, 1.6580398787933496, 1.6565193787933496, 1.6549998787933496, 1.6534793787933496, 1.6519598787933496, 1.6504393787933496, 1.6489198787933496, 1.6473993787933496, 1.6458798787933496, 1.6443593787933496, 1.6428398787933496, 1.6413193787933496, 1.6397998787933496, 1.6382793787933496, 1.6367598787933496, 1.6352393787933496, 1.6337198787933496, 1.6321993787933496, 1.6306798787933496, 1.6291593787933496, 1.6276398787933496, 1.6261193787933496, 1.6245998787933496, 1.6230793787933496, 1.6215598787933496, 1.6200393787933496, 1.6185198787933496, 1.6169993787933496, 1.6154798787933496, 1.6139593787933496, 1.6124398787933496, 1.6109193787933496, 1.6093998787933496, 1.6078793787933496, 1.6063598787933496, 1.6048393787933496, 1.6033198787933496, 1.6017993787933496, 1.6002798787933496, 1.5987593787933496, 1.5972398787933496, 1.5957193787933496, 1.5941998787933496, 1.5926793787933496, 1.5911598787933496, 1.5896393787933496, 1.5881198787933496, 1.5865993787933496, 1.5850798787933496, 1.5835593787933496, 1.5820398787933496, 1.5805193787933496, 1.5789998787933496, 1.5774793787933496, 1.5759598787933496, 1.5744393787933496, 1.5729198787933496, 1.5713993787933496, 1.5698798787933496, 1.5683593787933496, 1.5668398787933496, 1.5653193787933496, 1.5637998787933496, 1.5622793787933496, 1.5607598787933496, 1.5592393787933496, 1.5577198787933496, 1.5561993787933496, 1.5546798787933496, 1.5531593787933496, 1.5516398787933496, 1.5501193787933496, 1.5485998787933496, 1.5470793787933496, 1.5455598787933496, 1.5440393787933496, 1.5425198787933496, 1.5409993787933496, 1.5394798787933496, 1.5379593787933496, 1.5364398787933496, 1.5349193787933496, 1.5333998787933496, 1.5318793787933496, 1.5303598787933496, 1.5288393787933496, 1.5273198787933496, 1.5257993787933496, 1.5242798787933496, 1.5227593787933496, 1.5212398787933496, 1.5197193787933496, 1.5181998787933496, 1.5166793787933496, 1.5151598787933496, 1.5136393787933496, 1.5121198787933496, 1.5105993787933496, 1.5090798787933496, 1.5075593787933496, 1.5060398787933496, 1.5045193787933496, 1.5029998787933496, 1.5014793787933496, 1.5000393787933496, 1.4985198787933496, 1.4969993787933496, 1.4954798787933496, 1.4939593787933496, 1.4924398787933496, 1.4909193787933496, 1.4893998787933496, 1.4878793787933496, 1.4863598787933496, 1.4848393787933496, 1.4833198787933496, 1.4817993787933496, 1.4802798787933496, 1.4787593787933496, 1.4772398787933496, 1.4757193787933496, 1.4741998787933496, 1.4726793787933496, 1.4711598787933496, 1.4696393787933496, 1.4681198787933496, 1.4665993787933496, 1.4650798787933496, 1.4635593787933496, 1.4620398787933496, 1.4605193787933496, 1.4589998787933496, 1.4574793787933496, 1.4559598787933496, 1.4544393787933496, 1.4529198787933496, 1.4513993787933496, 1.4498798787933496, 1.4483593787933496, 1.4468398787933496, 1.4453193787933496, 1.4437998787933496, 1.4422793787933496, 1.4407598787933496, 1.4392393787933496, 1.4377198787933496, 1.4361993787933496, 1.4346798787933496, 1.4331593787933496, 1.4316398787933496, 1.4301193787933496, 1.4285998787933496, 1.4270793787933496, 1.4255598787933496, 1.4240393787933496, 1.4225198787933496, 1.4209993787933496, 1.4194798787933496, 1.4179593787933496, 1.4164398787933496, 1.4149193787933496, 1.4133998787933496, 1.4118793787933496, 1.4103598787933496, 1.4088393787933496, 1.4073198787933496, 1.4057993787933496, 1.4042798787933496, 1.4027593787933496, 1.4012398787933496, 1.4000393787933496, 1.3985198787933496, 1.3969993787933496, 1.3954798787933496, 1.3939593787933496, 1.3924398787933496, 1.3909193787933496, 1.3893998787933496, 1.3878793787933496, 1.3863598787933496, 1.3848393787933496, 1.3833198787933496, 1.3817993787933496, 1.3802798787933496, 1.3787593787933496, 1.3772398787933496, 1.3757193787933496, 1.3741998787933496, 1.3726793787933496, 1.3711598787933496, 1.3696393787933496, 1.3681198787933496, 1.3665993787933496, 1.3650798787933496, 1.3635593787933496, 1.3620398787933496, 1.3605193787933496, 1.3589998787933496, 1.3574793787933496, 1.3559598787933496, 1.3544393787933496, 1.3529198787933496, 1.3513993787933496, 1.3498798787933496, 1.3483593787933496, 1.3468398787933496, 1.3453193787933496, 1.3437998787933496, 1.3422793787933496, 1.3407598787933496, 1.3392393787933496, 1.3377198787933496, 1.3361993787933496, 1.3346798787933496, 1.3331593787933496, 1.3316398787933496, 1.3301193787933496, 1.3285998787933496, 1.3270793787933496, 1.3255598787933496, 1.3240393787933496, 1.3225198787933496, 1.3209993787933496, 1.3194798787933496, 1.3179593787933496, 1.3164398787933496, 1.3149193787933496, 1.3133998787933496, 1.3118793787933496, 1.3103598787933496, 1.3088393787933496, 1.3073198787933496, 1.3057993787933496, 1.3042798787933496, 1.3027593787933496, 1.3012398787933496, 1.3000393787933496, 1.2985198787933496, 1.2969993787933496, 1.2954798787933496, 1.2939593787933496, 1.2924398787933496, 1.2909193787933496, 1.2893998787933496, 1.2878793787933496, 1.2863598787933496, 1.2848393787933496, 1.2833198787933496, 1.2817993787933496, 1.2802798787933496, 1.2787593787933496, 1.2772398787933496, 1.2757193787933496, 1.2741998787933496, 1.2726793787933496, 1.2711598787933
```

```

train_acc_thru_time
[0.13319999999999993, 0.13989999999999927, 0.1520999999999992, 0.16469999999999988,
val_acc_thru_time
[0.14089999999999992, 0.14799999999999944, 0.16389999999999902, 0.17659999999999999,

```



```

import time
import math

print_every = 1000 # total = 27000
plot_every = 1000 # 5000

# Keep track of losses for plotting
current_loss = 0
val_losses = 0.
train_acc_thru_time_aggregate, val_acc_thru_time_aggregate = 0., 0.
train_losses_thru_time = []
val_losses_thru_time = []
train_acc_thru_time = []
val_acc_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)

x_train, y_train, x_train_tensor, y_train_tensor = TrainingData()
x_val, y_val, x_val_tensor, y_val_tensor = ValidationData()

x_train_len = len(x_train)
x_val_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:
    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
    val_correct_guess_count = 0
    train_correct_guess_count = 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):
        # Train accuracy calc
        train_category, _, train_category_tensor, train_line_tensor = randomTrainingE
        train_output, train_loss = evaluate(train_line_tensor, train_category_tensor)
        train_guess, _ = categoryFromOutput(train_output)
        train_correct_guess_count += int(train_guess == train_category)

        # Validation accuracy calc
        val_category, _, val_category_tensor, val_line_tensor = randomValidationExamp
        val_output, val_loss = evaluate(val_line_tensor, val_category_tensor)
        val_guess, _ = categoryFromOutput(val_output)
        val_correct_guess_count += int(val_guess == val_category)

    val_loss_per_train_data += val_loss

# Aggregate accuracy
train_acc_per_train_data = train_correct_guess_count / x_val_len
train_acc_thru_time_aggregate += train_acc_per_train_data
val_acc_per_train_data = val_correct_guess_count / x_val_len
val_acc_thru_time_aggregate += val_acc_per_train_data

# Aggregate validation loss
val_loss_per_train_data_ave = val_loss_per_train_data / x_val_len
val_losses += val_loss_per_train_data_ave

# Print iter number, loss, name and guess
if i % print_every == 0:
    print("iter = {}({:d}%) | time taken = {} | train_loss={:.4f}, val_loss(ave)=
    debug_x, debug_y, debug_x_tensor, debug_y_tensor = [], [], [], []

# 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

print("iter = {}({:d}%) | train_acc_thru_time_ave={}, val_acc_thru_time_ave={

train_acc_thru_time.append(train_acc_thru_time_aggregate / plot_every)
val_acc_thru_time.append(val_acc_thru_time_aggregate / plot_every)
train_acc_thru_time_aggregate = 0
val_acc_thru_time_aggregate = 0
```



```

learning rate = 0.001
x_train_len: 27000 , x_val_len: 10
iter = 0(0%) | time taken = 0m 3s | train_loss=2.2222, val_loss(ave)=2.3266 | tra
iter = 0(0%) | train_acc_thru_time_ave=0m 3s, val_acc_thru_time_ave=0.0004
iter = 1000(3%) | time taken = 0m 24s | train_loss=1.5628, val_loss(ave)=2.3251
iter = 1000(3%) | train_acc_thru_time_ave=0m 24s, val_acc_thru_time_ave=0.319699
iter = 2000(7%) | time taken = 0m 45s | train_loss=1.8648, val_loss(ave)=2.3004
iter = 2000(7%) | train_acc_thru_time_ave=0m 45s, val_acc_thru_time_ave=0.318900
iter = 3000(11%) | time taken = 1m 6s | train_loss=1.5915, val_loss(ave)=2.7069
iter = 3000(11%) | train_acc_thru_time_ave=1m 6s, val_acc_thru_time_ave=0.329999
iter = 4000(14%) | time taken = 1m 26s | train_loss=1.7824, val_loss(ave)=2.2152
iter = 4000(14%) | train_acc_thru_time_ave=1m 26s, val_acc_thru_time_ave=0.337500
iter = 5000(18%) | time taken = 1m 47s | train_loss=1.9984, val_loss(ave)=2.4863
iter = 5000(18%) | train_acc_thru_time_ave=1m 47s, val_acc_thru_time_ave=0.350100
iter = 6000(22%) | time taken = 2m 7s | train_loss=2.3606, val_loss(ave)=2.5456
iter = 6000(22%) | train_acc_thru_time_ave=2m 7s, val_acc_thru_time_ave=0.346100
iter = 7000(25%) | time taken = 2m 28s | train_loss=1.1203, val_loss(ave)=2.5735
iter = 7000(25%) | train_acc_thru_time_ave=2m 28s, val_acc_thru_time_ave=0.355700
iter = 8000(29%) | time taken = 2m 48s | train_loss=1.5213, val_loss(ave)=2.5407
iter = 8000(29%) | train_acc_thru_time_ave=2m 48s, val_acc_thru_time_ave=0.367100
iter = 9000(33%) | time taken = 3m 9s | train_loss=3.1933, val_loss(ave)=2.7906
iter = 9000(33%) | train_acc_thru_time_ave=3m 9s, val_acc_thru_time_ave=0.361900
iter = 10000(37%) | time taken = 3m 29s | train_loss=1.2495, val_loss(ave)=2.6609
iter = 10000(37%) | train_acc_thru_time_ave=3m 29s, val_acc_thru_time_ave=0.3673
iter = 11000(40%) | time taken = 3m 49s | train_loss=1.8335, val_loss(ave)=2.6510
iter = 11000(40%) | train_acc_thru_time_ave=3m 49s, val_acc_thru_time_ave=0.373200
iter = 12000(44%) | time taken = 4m 9s | train_loss=1.4305, val_loss(ave)=2.3426
iter = 12000(44%) | train_acc_thru_time_ave=4m 9s, val_acc_thru_time_ave=0.375300
iter = 13000(48%) | time taken = 4m 29s | train_loss=1.6745, val_loss(ave)=2.4404
iter = 13000(48%) | train_acc_thru_time_ave=4m 29s, val_acc_thru_time_ave=0.3767
iter = 14000(51%) | time taken = 4m 49s | train_loss=1.8929, val_loss(ave)=2.6249
iter = 14000(51%) | train_acc_thru_time_ave=4m 49s, val_acc_thru_time_ave=0.39529
iter = 15000(55%) | time taken = 5m 9s | train_loss=1.5942, val_loss(ave)=2.5534
iter = 15000(55%) | train_acc_thru_time_ave=5m 9s, val_acc_thru_time_ave=0.3896
iter = 16000(59%) | time taken = 5m 29s | train_loss=1.8463, val_loss(ave)=2.6989
iter = 16000(59%) | train_acc_thru_time_ave=5m 29s, val_acc_thru_time_ave=0.38939
iter = 17000(62%) | time taken = 5m 48s | train_loss=0.6244, val_loss(ave)=2.6837
iter = 17000(62%) | train_acc_thru_time_ave=5m 48s, val_acc_thru_time_ave=0.40269
iter = 18000(66%) | time taken = 6m 8s | train_loss=1.7411, val_loss(ave)=2.9596
iter = 18000(66%) | train_acc_thru_time_ave=6m 8s, val_acc_thru_time_ave=0.353300
iter = 19000(70%) | time taken = 6m 27s | train_loss=1.7421, val_loss(ave)=3.2679
iter = 19000(70%) | train_acc_thru_time_ave=6m 27s, val_acc_thru_time_ave=0.35409
iter = 20000(74%) | time taken = 6m 47s | train_loss=1.8878, val_loss(ave)=2.9697
iter = 20000(74%) | train_acc_thru_time_ave=6m 47s, val_acc_thru_time_ave=0.37489
iter = 21000(77%) | time taken = 7m 6s | train_loss=0.8556, val_loss(ave)=2.2618
iter = 21000(77%) | train_acc_thru_time_ave=7m 6s, val_acc_thru_time_ave=0.39149
iter = 22000(81%) | time taken = 7m 25s | train_loss=1.6963, val_loss(ave)=2.4448
iter = 22000(81%) | train_acc_thru_time_ave=7m 25s, val_acc_thru_time_ave=0.38409
iter = 23000(85%) | time taken = 7m 43s | train_loss=1.9805, val_loss(ave)=1.9584
iter = 23000(85%) | train_acc_thru_time_ave=7m 43s, val_acc_thru_time_ave=0.39040
iter = 24000(88%) | time taken = 8m 2s | train_loss=2.0390, val_loss(ave)=3.9155
iter = 24000(88%) | train_acc_thru_time_ave=8m 2s, val_acc_thru_time_ave=0.406600
iter = 25000(92%) | time taken = 8m 21s | train_loss=1.5916, val_loss(ave)=2.8011
iter = 25000(92%) | train_acc_thru_time_ave=8m 21s, val_acc_thru_time_ave=0.42970

```


▼ Plotting the Results

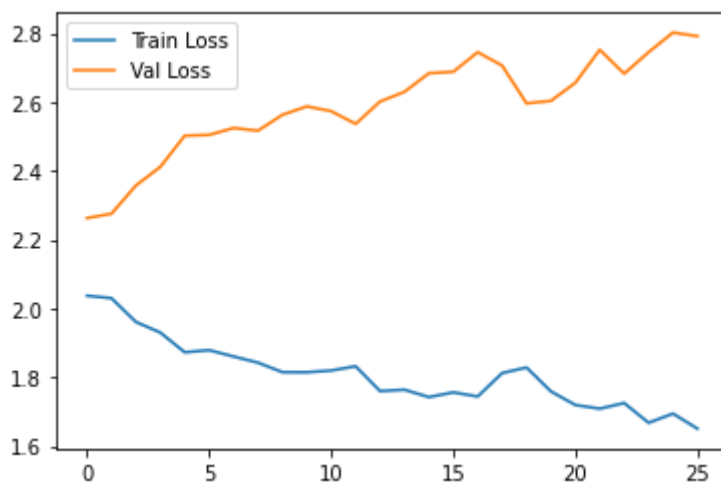
Plotting the historical loss from `all_losses` shows the network learning:

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

```
plt.figure()
train_loss_plot = plt.plot(train_losses_thru_time[1:], label='Train Loss')
val_loss_plot = plt.plot(val_losses_thru_time[1:], label="Val Loss")
plt.legend()
```

```
print("train_losses_thru_time")
print(train_losses_thru_time[1:])
print("val_losses_thru_time")
print(val_losses_thru_time[1:])
```

```
☞ train_losses_thru_time
[2.0370060900449753, 2.029802362084389, 1.9607643482089043, 1.9299266896247864, :
val_losses_thru_time
[2.2626027950167646, 2.2753594054341266, 2.3573232454776756, 2.41132256001234, 2.
```



```
train_acc_thru_time_aggregate
```

```
☞ 427.19999999999993
```

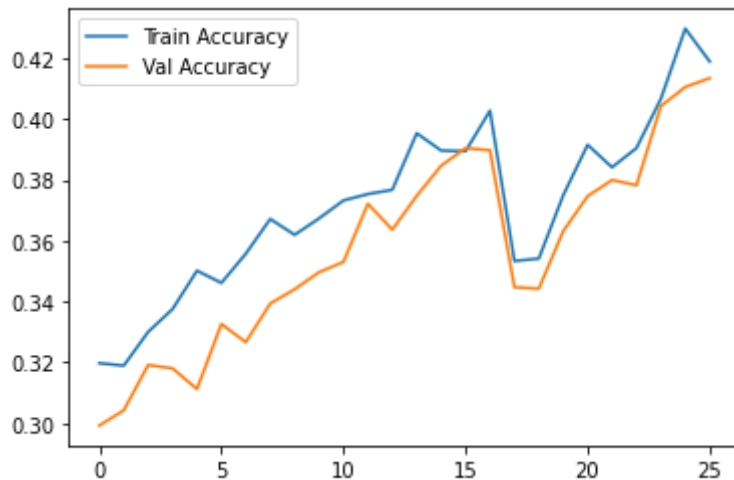
```
import matplotlib.pyplot as plt
import matplotlib.ticker as ticker
```

```
plt.figure()
train_acc_plot = plt.plot(train_acc_thru_time[1:], label='Train Accuracy')
val_acc_plot = plt.plot(val_acc_thru_time[1:], label="Val Accuracy")
plt.legend()
```

```
print("train_acc_thru_time")
```

```
print(train_acc_thru_time[1:])
print("val_acc_thru_time")
print(val_acc_thru_time[1:])
```

```
↳ train_acc_thru_time
[0.31969999999999993, 0.31890000000000007, 0.32999999999999998, 0.33750000000000006,
val_acc_thru_time
[0.29920000000000005, 0.30419999999999998, 0.31909999999999999, 0.31800000000000017,
```



▼ 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 evaluated using `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()
```

```

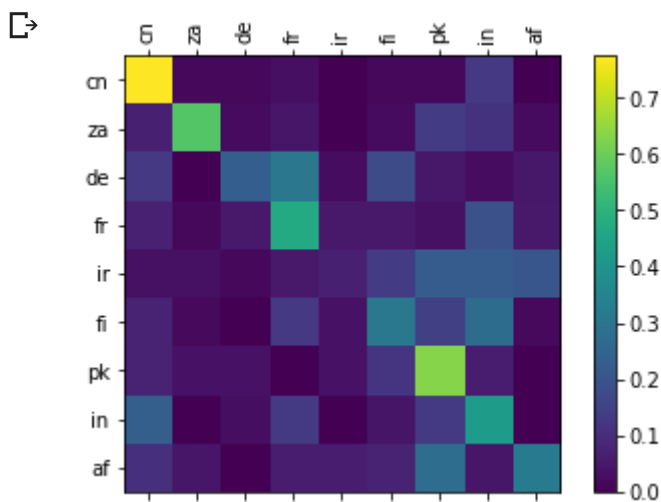
fig = plt.figure()
ax = fig.add_subplot(111)
cax = ax.matshow(confusion.numpy())
fig.colorbar(cax)

# 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 = torch.sum(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)
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 = 0.4259999990463257

```

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

