```python
import torch
import torch.nn as nn
from torch.autograd import Variable
from torch.nn.functional import one_hot
import random
import time
import math
import unicodedata
import string
```

Data Prep

```python
import requests
url = "https://api.github.com/repos/DrUzair/NLP/contents/textclassification/surnames/names_data/names"
response = requests.get(url)
category_lines = {}
all_categories = []
if response.status_code == 200:
    # Parse the JSON response
    files_info = response.json()
    for file_info in files_info:
      file_name = file_info['name']
      category = file_name.split('/')[-1].split('.')[0]
      all_categories.append(category)
      download_url = file_info['download_url']
      file_response = requests.get(download_url)
      if file_response.status_code == 200:
        file_content = file_response.content.decode('utf-8')
        names = [name for name in file_content.split('\n') if len(name.strip())>1]
        category_lines[category] = names

      else:
        print("Error occurred:", response.status_code)
else:
    print("Error occurred:", response.status_code)

n_categories = len(all_categories)

```

```python
n_categories = len(all_categories)
n_categories

category_lines
```

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

# Turn a Unicode string to plain ASCII, thanks to http://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
    )

# 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]
```

```python

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

# Turn a name into a <name_length x 1 x n_letters>,
# or an array of one-hot letter vectors
def nameToTensor(name):
    tensor = torch.zeros(len(name), 1, n_letters)
    for li, letter in enumerate(name):
        tensor[li][0][letterToIndex(letter)] = 1
    return tensor

def categoryFromOutput(output):
    top_n, top_i = output.data.topk(1)  # Tensor out of Variable with .data
    category_i = top_i[0][0]
    return all_categories[category_i], category_i

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

def randomTrainingPair():
    category = randomChoice(all_categories)
    name = randomChoice(category_lines[category])
    category_tensor = Variable(torch.LongTensor([all_categories.index(category)]))
    name_tensor = Variable(nameToTensor(name))
    return category, name, category_tensor, name_tensor

```

```python
category, name, category_tensor, name_tensor = randomTrainingPair()
print(f"category: {category}")
print(f"category_tensor: {category_tensor}")
print(f"category_tensor shape: {category_tensor.shape}")
print(f"name: {name}")
print(f"name_tensor shape: {name_tensor.shape}")
print(f"name_tensor: {name_tensor}")
```

```
category: Spanish
category_tensor: tensor([16])
category_tensor shape: torch.Size([1])
name: Estévez
name_tensor shape: torch.Size([7, 1, 58])
name_tensor: 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., 1., 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., 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., 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., 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., 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., 1., 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., 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., 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., 0., 0.,
          0., 0., 0., 0., 0., 0., 0.]]])
```

RNN Model

Textbook RNN

```
class RNN_Textbook(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(RNN_Textbook, self).__init__()
        self.hidden_size = hidden_size
        self.W = nn.Linear(input_size, hidden_size)
        self.U = nn.Linear(hidden_size, hidden_size)
        self.V = nn.Linear(hidden_size, output_size)
        self.softmax = nn.LogSoftmax(dim=1)

    def forward(self, input):
        self.hidden = torch.zeros(1, self.hidden_size)
        for i in range(input.size(0)):  # Iterate through the time steps
            self.hidden = torch.tanh(self.W(input[i]) + self.U(self.hidden))
        output = self.V(self.hidden)
        output = self.softmax(output)
        return output

# Example usage:
input_size = 3 # sequence length
hidden_size = 20
output_size = 18
batch_size = 1

rnn = RNN_Textbook(input_size, hidden_size, output_size)
input = torch.randn(input_size, batch_size, input_size)  # Sequence length x batch size x input size
output = rnn(input)
print(output)  # This will be the output for the last time step

```

```
tensor([[-2.3270, -3.0113, -3.1159, -2.9885, -2.5024, -3.3357, -2.6729, -3.5345,
         -2.5955, -2.4811, -3.0357, -3.1022, -3.0319, -2.9861, -2.9602, -2.9689,
         -2.9868, -3.2392]], grad_fn=<LogSoftmaxBackward0>)
```

Pytorch RNN

```
import torch.nn as nn
import torch.optim as optim

# Define your RNN model
class RNN_Pytorch(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
```

```
        super(RNN_Pytorch, self).__init__()
        self.hidden_size = hidden_size
        self.rnn = nn.RNN(input_size, hidden_size)
        # output projection layer
        self.fc = nn.Linear(hidden_size, output_size)
        # softmax
        self.softmax = nn.LogSoftmax(dim=1)

    def forward(self, input):
        self.hidden = torch.zeros(1, input.size(1), self.hidden_size)
        output, self.hidden = self.rnn(input,  self.hidden)
        output_last = output[-1]  # Selecting the output of the last time step
        output = self.fc(output_last)
        output = self.softmax(output)
        return output

# Example usage:
input_size = 3 # sequence length
hidden_size = 20
output_size = 18
batch_size = 1

rnn = RNN_Pytorch(input_size, hidden_size, output_size)
input = torch.randn(input_size, batch_size, input_size)  # Sequence length x batch size x input size
output = rnn(input)
print(output)  # This will be the output for the last time step
```

```
tensor([[-2.9847, -2.9793, -2.7283, -2.7699, -2.7077, -2.9586, -2.8452, -2.8118,
         -2.8397, -2.9154, -2.8262, -3.0134, -3.0826, -3.2816, -2.7720, -3.0925,
         -2.6094, -3.0386]], grad_fn=<LogSoftmaxBackward0>)
```

Model training

train function

```
n_hidden = 128
n_epochs = 100000
print_every = 1000
plot_every = 1000
learning_rate = 0.0001
batch_size = 1

#rnn = RNN_Textbook(input_size=n_letters, hidden_size=n_hidden, output_size=n_categories)
rnn = RNN_Pytorch(input_size=n_letters, hidden_size=n_hidden, output_size=n_categories)

#optimizer = torch.optim.SGD(rnn.parameters(), lr=learning_rate)
optimizer = torch.optim.Adam(rnn.parameters(), lr=learning_rate)
criterion = nn.NLLLoss()
# criterion = nn.CrossEntropyLoss()
def train(category_tensor, name_tensor):
    optimizer.zero_grad()  # set gradients to zero
    output = rnn(name_tensor)
    loss = criterion(output, category_tensor)
    loss.backward()
    nn.utils.clip_grad_norm_(rnn.parameters(), 1) # gradient clipping : max_norm=1
    optimizer.step()
    return output, loss.item()
```

```
category, name, category_tensor, name_tensor = randomTrainingPair()
```

```
output, loss = train(category_tensor, name_tensor)
print(output)
print(loss)
```

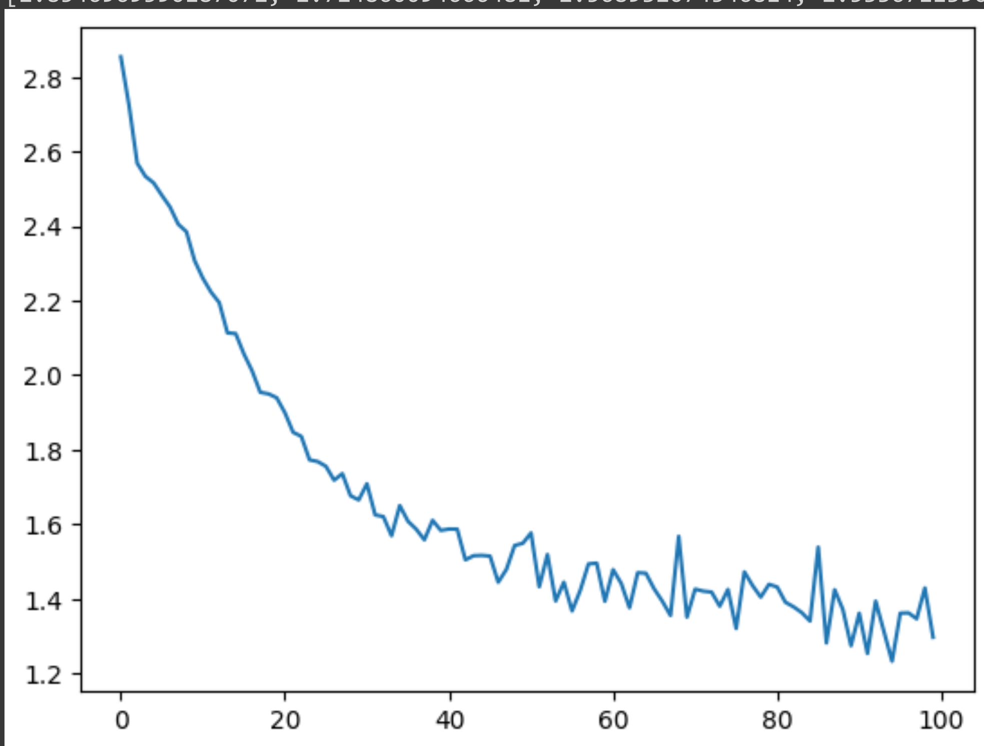
```
tensor([[-2.9041, -2.8077, -3.0249, -2.8745, -2.8807, -2.9602, -2.8875, -2.8095,
         -2.9399, -2.8979, -2.9369, -2.8887, -2.8028, -2.7793, -2.9290, -2.9360,
         -2.8327, -2.9720]], grad_fn=<LogSoftmaxBackward0>)
2.9719746112823486
```

```
# 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 epoch in range(1, n_epochs + 1):
    category, name, category_tensor, name_tensor = randomTrainingPair()
    output, loss = train(category_tensor, name_tensor)
    current_loss += loss

    # Print epoch number, loss, name and guess
    if epoch % print_every == 0:
        guess, guess_i = categoryFromOutput(output)
        correct = '✓' if guess == category else '✗ (%s)' % category
        print('%d %d%% (%s) %.4f %s / %s %s' % (
        epoch, epoch / n_epochs * 100, timeSince(start), loss, name, guess, correct))

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

print(all_losses)
import matplotlib.pyplot as plt
import matplotlib.ticker as ticker

plt.figure()
plt.plot(all_losses)

torch.save(rnn, 'char-rnn-classification.pt')
```

```
1000 1% (0m 3s) 2.5599 Estéves / Portuguese ✓
2000 2% (0m 6s) 3.1171 Blanc / Arabic ✗ (French)
3000 3% (0m 9s) 3.0677 Driffield / Russian ✗ (English)
4000 4% (0m 12s) 2.9712 Johanson / Greek ✗ (English)
5000 5% (0m 14s) 2.5062 Solomon / Scottish ✗ (French)
6000 6% (0m 17s) 3.0029 Jordan / Scottish ✗ (Polish)
7000 7% (0m 19s) 1.9398 Zogby / Arabic ✓
8000 8% (0m 22s) 2.6028 Poingdestre / Greek ✗ (French)
9000 9% (0m 25s) 2.4254 Furtsch / French ✗ (Czech)
10000 10% (0m 27s) 1.8077 Bachmeier / German ✓
11000 11% (0m 30s) 2.4414 Reid / Arabic ✗ (Scottish)
12000 12% (0m 32s) 2.2399 Ferrara / Spanish ✗ (Italian)
13000 13% (0m 35s) 1.5480 Yamaguchi / Japanese ✓
14000 14% (0m 38s) 2.8735 Vaca / Japanese ✗ (Czech)
15000 15% (0m 41s) 2.9698 Hassel / Arabic ✗ (Dutch)
```

```
15000 15% (0m 41s) 2.9698 Hassel / Arabic ✗ (Dutch)
16000 16% (0m 44s) 3.3294 O'Shea / Vietnamese ✗ (Irish)
17000 17% (0m 47s) 0.6767 Mei / Chinese ✓
18000 18% (0m 50s) 1.8815 Jang / Chinese ✗ (Korean)
19000 19% (0m 53s) 0.3236 Paschalis / Greek ✓
20000 20% (0m 55s) 2.3132 Lachance / English ✗ (French)
21000 21% (0m 58s) 3.5679 Cham / Korean ✗ (Arabic)
22000 22% (1m 1s) 1.9511 Serafim / Portuguese ✓
23000 23% (1m 4s) 1.0964 Ryoo / Korean ✓
24000 24% (1m 7s) 1.4432 Mach / Vietnamese ✓
25000 25% (1m 10s) 0.5233 Shim / Korean ✓
26000 26% (1m 14s) 0.6155 Ricchetti / Italian ✓
27000 27% (1m 17s) 2.2074 Bezubyak / Polish ✗ (Russian)
28000 28% (1m 19s) 0.3697 Shimanouchi / Japanese ✓
29000 28% (1m 22s) 0.5895 Parisi / Italian ✓
30000 30% (1m 25s) 4.3715 Fergus / Portuguese ✗ (Irish)
31000 31% (1m 28s) 1.0753 Colbert / French ✓
32000 32% (1m 30s) 4.2674 Tunison / Irish ✗ (Dutch)
33000 33% (1m 33s) 3.6062 Tsai  / Arabic ✗ (Korean)
34000 34% (1m 35s) 0.1777 Thach / Vietnamese ✓
35000 35% (1m 38s) 1.1755 Armbrüster / German ✓
36000 36% (1m 41s) 0.5626 Xie / Chinese ✓
37000 37% (1m 43s) 1.2461 Ryu / Chinese ✗ (Korean)
38000 38% (1m 46s) 0.0245 Rutkowski / Polish ✓
39000 39% (1m 50s) 0.4158 Brown / Scottish ✓
40000 40% (1m 52s) 1.0670 Coelho / Italian ✗ (Portuguese)
41000 41% (1m 55s) 1.5658 Nunes / Portuguese ✓
42000 42% (1m 57s) 3.5859 Voakes / Greek ✗ (English)
43000 43% (2m 0s) 0.9420 Gwang  / Korean ✓
44000 44% (2m 3s) 2.8157 Breda / Spanish ✗ (Dutch)
45000 45% (2m 6s) 1.9343 Janda / Czech ✗ (Polish)
46000 46% (2m 8s) 2.9485 Hrula / Scottish ✗ (Czech)
47000 47% (2m 12s) 1.1284 Parent / French ✓
48000 48% (2m 15s) 1.1297 Achthoven / Dutch ✓
49000 49% (2m 17s) 6.6543 Budny / Scottish ✗ (Polish)
50000 50% (2m 20s) 1.9576 Tso / Vietnamese ✗ (Chinese)
51000 51% (2m 22s) 2.3607 Nose / Chinese ✗ (Japanese)
52000 52% (2m 25s) 5.0440 Nifterik / Czech ✗ (Dutch)
53000 53% (2m 29s) 1.0574 Gaber / German ✗ (Arabic)
54000 54% (2m 31s) 0.0361 Kefalas / Greek ✓
55000 55% (2m 34s) 0.0815 Ri / Korean ✓
56000 56% (2m 36s) 0.0095 O'Loughlin / Irish ✓
57000 56% (2m 39s) 0.1525 Hyun  / Korean ✓
58000 57% (2m 42s) 0.0037 Szczepanski / Polish ✓
59000 59% (2m 44s) 4.0565 Abadi / Italian ✗ (Arabic)
60000 60% (2m 47s) 3.3001 Jacques / Portuguese ✗ (French)
61000 61% (2m 49s) 2.2287 Wotherspoon / Russian ✗ (English)
62000 62% (2m 53s) 1.8636 Kouri / Japanese ✗ (Arabic)
63000 63% (2m 55s) 1.4969 Alves / Portuguese ✗ (Spanish)
64000 64% (2m 58s) 2.6644 Torres / Portuguese ✗ (Spanish)
65000 65% (3m 0s) 0.0008 Kozlowski / Polish ✓
66000 66% (3m 3s) 0.8767 Chi / Korean ✓
67000 67% (3m 6s) 0.0039 Ieyasu / Japanese ✓
68000 68% (3m 9s) 0.0981 Demetrious / Greek ✓
69000 69% (3m 12s) 3.8374 Ferraro / Portuguese ✗ (Italian)
70000 70% (3m 14s) 0.0204 Mikolajczak / Polish ✓
71000 71% (3m 17s) 1.2271 Maroun / Arabic ✓
72000 72% (3m 20s) 0.2042 Said / Arabic ✓
73000 73% (3m 23s) 0.0686 Safar / Arabic ✓
74000 74% (3m 27s) 2.5649 Vargas / Portuguese ✗ (Spanish)
75000 75% (3m 31s) 2.5453 Spiller / Dutch ✗ (English)
76000 76% (3m 34s) 0.0028 Altoviti / Italian ✓
77000 77% (3m 38s) 1.0132 Sato / Japanese ✓
78000 78% (3m 40s) 0.1125 Ilyahin / Russian ✓
```

```
79000 79% (3m 43s) 0.8231 Shaughnessy / English ✓
80000 80% (3m 46s) 0.2092 Quraishi / Arabic ✓
81000 81% (3m 48s) 0.0653 Whyte / Scottish ✓
82000 82% (3m 51s) 0.4245 Khouri / Arabic ✓
83000 83% (3m 55s) 0.2920 Wright / Scottish ✓
84000 84% (3m 57s) 0.5137 Kijek / Polish ✓
85000 85% (4m 0s) 3.8501 Braden / Dutch ✗ (Irish)
86000 86% (4m 3s) 0.0084 Kouretas / Greek ✓
87000 87% (4m 7s) 2.3374 Garland / French ✗ (English)
88000 88% (4m 9s) 0.0207 Kwang  / Korean ✓
89000 89% (4m 12s) 0.1893 Escamilla / Spanish ✓
90000 90% (4m 14s) 5.7808 Chmiel / English ✗ (Polish)
91000 91% (4m 17s) 0.0024 Górski / Polish ✓
92000 92% (4m 21s) 0.0010 Yagubov / Russian ✓
93000 93% (4m 24s) 1.6385 Bishop / English ✓
94000 94% (4m 26s) 0.0837 Paredes / Portuguese ✓
95000 95% (4m 29s) 0.0000 Matsakov / Russian ✓
96000 96% (4m 32s) 3.7199 Otsu / Arabic ✗ (Japanese)
97000 97% (4m 35s) 0.4769 Vega / Spanish ✓
98000 98% (4m 37s) 0.0023 Ferreiro / Portuguese ✓
99000 99% (4m 40s) 4.3900 Albuquerque / French ✗ (Portuguese)
100000 100% (4m 43s) 6.7209 Adam / Arabic ✗ (French)
[2.8546969590187072, 2.724860094666481, 2.568932074546814, 2.533671139061451, 2.5160853575468063, 2.48
```



Prediction

```
import sys
rnn = torch.load('char-rnn-classification.pt')
```

```
# Just return an output given a name
def evaluate(name_tensor):
  output = rnn(name_tensor)
  return output


```

```python
def predict(line, n_predictions=3):
    output = evaluate(Variable(nameToTensor(line)))

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

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

    return predictions



predict('ahmad')

```

```
(-0.01) Arabic
(-6.22) English
(-6.38) French
[[tensor(-0.0068), 'Arabic'],
 [tensor(-6.2206), 'English'],
 [tensor(-6.3834), 'French']]
```