In [130…
```
import glob
import unicodedata
import string
import requests
import pandas as pd
import random
import sys
import torch
from torch.autograd import Variable
from torch import nn
from torch.utils.data import Dataset, DataLoader
import torch.optim as optim
from sklearn.model_selection import train_test_split

import random
import time
import math
from torch.autograd import Variable
import matplotlib.pyplot as plt
import matplotlib.ticker as ticker
import os
```

Data Loading

In [131…
```
data = []
for path in glob.glob("./names/*.txt"):
    nationality = os.path.basename(path).replace(".txt","")

    with open(path, encoding = "utf-8") as f:
        for line in f:
            name = line.strip()
            if name:
                data.append({"Name":line.strip(), "Nationality":nationali


df = pd.DataFrame(data)
```

In [132…
```
df.head()
```

Out[132…

	Name	Nationality
0	Abl	Czech
1	Adsit	Czech
2	Ajdrna	Czech
3	Alt	Czech
4	Antonowitsch	Czech

In [133…
```
# nationality to numbers
nationalities = df['Nationality'].unique()
nationality_to_index = {}
index_to_nationality = []
```

```
current_idx = 0
for nat in nationalities:
    nationality_to_index[nat] = current_idx
    index_to_nationality.append(nat)
    current_idx = current_idx + 1

labels = []
for nat in df['Nationality']:
    labels.append(nationality_to_index[nat])
target_tensor = torch.tensor(labels, dtype=torch.long)
```

In [134…
```
df.shape
```

Out[134… (20074, 2)

eda

In [135…
```
df["Nationality"].value_counts() # HUGE IMBALANCE
```

Out[135…
```
Nationality
Russian       9408
English       3668
Arabic        2000
Japanese       991
German         724
Italian        709
Czech          519
Spanish        298
Dutch          297
French         277
Chinese        268
Irish          232
Greek          203
Polish         139
Scottish       100
Korean          94
Portuguese      74
Vietnamese      73
Name: count, dtype: int64
```

In [136…
```
target_size = 500
balanced_list = []

for nationality in df['Nationality'].unique():
    subset = df[df['Nationality'] == nationality]
    if len(subset) > target_size:
        subset = subset.sample(n=target_size, random_state=1)
    balanced_list.append(subset)

df = pd.concat(balanced_list)
df = df.sample(frac=1, random_state=1).reset_index(drop=True)
```

In [137…
```
df["Nationality"].value_counts()
```

Out[137…
```
Nationality
German        500
Arabic        500
Russian       500
Czech         500
Italian       500
Japanese      500
English       500
Spanish       298
Dutch         297
French        277
Chinese       268
Irish         232
Greek         203
Polish        139
Scottish      100
Korean         94
Portuguese     74
Vietnamese     73
Name: count, dtype: int64
```

In [138…
```
df.head()
```

Out[138…

	Name	Nationality
0	Straub	German
1	Marquardt	German
2	Palladino	Italian
3	Xing	Chinese
4	O'Brien	Irish

In [139…
```
df.shape
```

Out[139…
```
(5555, 2)
```

Text Preprocessing

- Character-to-Index Conversion: Each character in a name is mapped to an index based on a predefined vocabulary.

In [140…
```
all_letters = string.ascii_letters

char_to_index = {}
char_to_index["<PAD>"] = 0
char_to_index["<SOS>"] = 1
char_to_index["<EOS>"] = 2

current_index = 3
for char in all_letters:
    char_to_index[char] = current_index
    current_index += 1
```

In [141…
```
char_to_index
```

Out[141…
```
{'<PAD>': 0,
 '<SOS>': 1,
 '<EOS>': 2,
 'a': 3,
 'b': 4,
 'c': 5,
 'd': 6,
 'e': 7,
 'f': 8,
 'g': 9,
 'h': 10,
 'i': 11,
 'j': 12,
 'k': 13,
 'l': 14,
 'm': 15,
 'n': 16,
 'o': 17,
 'p': 18,
 'q': 19,
 'r': 20,
 's': 21,
 't': 22,
 'u': 23,
 'v': 24,
 'w': 25,
 'x': 26,
 'y': 27,
 'z': 28,
 'A': 29,
 'B': 30,
 'C': 31,
 'D': 32,
 'E': 33,
 'F': 34,
 'G': 35,
 'H': 36,
 'I': 37,
 'J': 38,
 'K': 39,
 'L': 40,
 'M': 41,
 'N': 42,
 'O': 43,
 'P': 44,
 'Q': 45,
 'R': 46,
 'S': 47,
 'T': 48,
 'U': 49,
 'V': 50,
 'W': 51,
 'X': 52,
 'Y': 53,
 'Z': 54}
```

- Padding: Names are padded to a consistent length to ensure they have the same dimensions.

In [142…
```
def get_length_of_name(name):
    return len(name)
```

In [143…
```
max_length_of_name = df["Name"].apply(get_length_of_name).max()
```

In [144…
```
def name_to_indices(name, max_len):
    result = []

    for char in name:
        if char in char_to_index:
            index_value = char_to_index[char]
            result.append(index_value)

    # Padding
    while len(result) < max_len:
        result.append(0)

    return result
```

In [145…
```
encoded_names = []
for name in df["Name"]:
    encoded_list = name_to_indices(name, max_length_of_name)
    encoded_names.append(encoded_list)

labels_list = []
for nat in df['Nationality']:
    labels_list.append(nationality_to_index[nat])

input_tensor = torch.tensor(encoded_names, dtype=torch.long)
target_tensor = torch.tensor(labels_list, dtype=torch.long)
```

In [161…
```
X_train, X_test, y_train, y_test = train_test_split(
    input_tensor, target_tensor, test_size=0.2, random_state=42
)
```

In [146…
```
df['Encoded_Name'] = encoded_names
```

In [147…
```
df.head()
```

Out[147…

	Name	Nationality	Encoded_Name
0	Straub	German	[47, 22, 20, 3, 23, 4, 0, 0, 0, 0, 0, 0, 0, 0,...
1	Marquardt	German	[41, 3, 20, 19, 23, 3, 20, 6, 22, 0, 0, 0, 0, ...
2	Palladino	Italian	[44, 3, 14, 14, 3, 6, 11, 16, 17, 0, 0, 0, 0, ...
3	Xing	Chinese	[52, 11, 16, 9, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ...
4	O'Brien	Irish	[43, 30, 20, 11, 7, 16, 0, 0, 0, 0, 0, 0, 0, 0...

- Character Embeddings: The character indices are transformed into dense vectors using an embedding layer. This helps capture more nuanced information about the characters.

In [148…
```
vocab_size = len(char_to_index)
embedding_dim = 16
```

In [149…
```
embedding_layer = nn.Embedding(num_embeddings=vocab_size,
                               embedding_dim=embedding_dim,
                               padding_idx=0)
```

In [150…
```
all_rows = []
for encoded_name in df['Encoded_Name']:
    all_rows.append(encoded_name)
```

In [151…
```
input_tensor = torch.tensor(all_rows, dtype=torch.long)
embedded_output = embedding_layer(input_tensor)
```

In [152…
```
first_name = df['Name'][0]
first_letter_vector = embedded_output[0][0]
```

In [153…
```
first_name
```

Out[153…
```
'Straub'
```

In [154…
```
first_letter_vector
```

Out[154…
```
tensor([ 1.3659, -0.7645,  0.3487,  0.0278, -0.8061,  0.3338,  0.5660, -
1.7055,
         0.8877,  0.0146,  0.9174,  1.0222,  0.8110,  0.3444, -0.8231,
0.8194],
       grad_fn=<SelectBackward0>)
```

Prediction

- Multi-Layer Perceptron (MLP)
  -- Embedding Layer: Transforms character indices into dense vectors.
  -- Fully Connected Layer: Flattens the embeddings and passes them through a linear layer to generate predictions.
  -- Output Layer: Produces probability scores for each nationality.

The model is trained using cross-entropy loss and optimized with the Adam optimizer. After training, the model can predict the nationality of new names based on the learned patterns.

In [155…
```
class MLP(nn.Module):
    def __init__(self, vocab_size, embed_dim, sequence_length, num_classe
        super(MLP, self).__init__()

        self.embedding = nn.Embedding(vocab_size, embed_dim, padding_idx=
        self.fc1 = nn.Linear(embed_dim * sequence_length, 128)
```

```
        self.fc2 = nn.Linear(128, num_classes)
        self.relu = nn.ReLU()
    def forward(self, x):


        x = self.embedding(x) # shape: [batch_size, sequence_length, embe
        x = x.view(x.size(0), -1) # shape: [batch_size, sequence_length *

        x = self.fc1(x)
        x = self.relu(x)
        x = self.fc2(x)

        return x
```

In [156…
```
model = MLP(
    vocab_size=len(char_to_index),
    embed_dim=16,
    sequence_length=max_length_of_name,
    num_classes=len(nationalities)
)
```

In [157…
```
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)
```

In [158…
```
epochs = 100
for epoch in range(epochs):
    optimizer.zero_grad()
    outputs = model(input_tensor)
    loss = criterion(outputs, target_tensor)
    loss.backward()

    #Update weights
    optimizer.step()

    if (epoch + 1) % 5 == 0:
        print(f"Epoch [{epoch+1}/{epochs}], Loss: {loss.item():.4f}")
```

```
Epoch [5/100], Loss: 2.8159
Epoch [10/100], Loss: 2.7040
Epoch [15/100], Loss: 2.6007
Epoch [20/100], Loss: 2.5054
Epoch [25/100], Loss: 2.4169
Epoch [30/100], Loss: 2.3327
Epoch [35/100], Loss: 2.2512
Epoch [40/100], Loss: 2.1720
Epoch [45/100], Loss: 2.0955
Epoch [50/100], Loss: 2.0214
Epoch [55/100], Loss: 1.9504
Epoch [60/100], Loss: 1.8828
Epoch [65/100], Loss: 1.8189
Epoch [70/100], Loss: 1.7585
Epoch [75/100], Loss: 1.7015
Epoch [80/100], Loss: 1.6475
Epoch [85/100], Loss: 1.5961
Epoch [90/100], Loss: 1.5474
Epoch [95/100], Loss: 1.5008
Epoch [100/100], Loss: 1.4562
```

In [159…
```
def predict(name):
    indices = name_to_indices(name, max_length_of_name)
    test_tensor = torch.tensor([indices], dtype=torch.long)

    model.eval()
    with torch.no_grad():
        output = model(test_tensor)

    scores = output[0]
    top_values, top_indices = torch.topk(scores, k=3)
    print(f"name = '{name}'\n")
    for i in range(3):
        score = top_values[i].item()
        idx = top_indices[i].item()
        nat = index_to_nationality[idx]
        print(f"({score:.2f}) {nat}")
```

In [160…
```
predict("karima")
```

```
name = 'karima'

(2.68) Arabic
(0.87) Japanese
(0.63) Spanish
```

Metrics

In [164…
```
model.eval()
correct = 0
total = 0

with torch.no_grad():
    outputs = model(X_test)
    predicted = torch.argmax(outputs, dim=1)
    for i in range(len(y_test)):
        if predicted[i] == y_test[i]:
            correct = correct + 1
        total = total + 1
accuracy = (correct / total) * 100
print(f"Accuracy on Test Set: {accuracy:.2f}%")
```

```
Accuracy on Test Set: 58.06%
```