

Sheet 2.4: Character-level sequence modeling w/ RNNs

Author: Michael Franke

The goal of this tutorial is to get familiar with simple language models. To be able to have a manageable (quick to train, evaluate, inspect) case study, we look at character-level predictions of surnames from different languages. (The inspiration and source of this notebook is [this tutorial](#) from PyTorch's documentation.)

Packages & global parameters

In addition to the usual packages for neural network modeling, we will also require packages for I/O and string handling.

```
#####  
## import packages  
#####  
  
from __future__ import unicode_literals, print_function, division  
import json  
import pandas  
import string  
import torch  
import urllib.request  
import numpy as np  
import torch.nn as nn  
import random  
import time  
import math  
import matplotlib.pyplot as plt  
import warnings  
  
warnings.filterwarnings("ignore")
```

Loading & inspecting the data

Our training data are lists of surnames from different countries. We will use this data set to train a model that predicts a name, given the country as a prompt.

The (pre-processed) data is stored in a JSON file. We load it and define a few useful variables for later use.

```
#####  
## read and inspect the data
```

```
#####
# with urllib.request.urlopen(
#
# "https://raw.githubusercontent.com/michael-franke/npNLG/main/neural_pr
# agmatic_nlg/05-RNNs/names-data.json"
# ) as url:
#     names_data = json.load(url)

# # If unable to download, use local import
with open('/Users/ychen/Documents/CL/- Understanding LLM/names-
data.json') as dataFile:
    names_data = json.load(dataFile)

categories = list(names_data.keys())
n_categories = len(categories)

# Use all ASCII letters as vocabulary (plus tokens [EOS], [SOS])
all_letters = string.ascii_letters + " .,:'-"
n_letters = len(all_letters) + 2 # all letter + [EOS] and [SOS]
SOSIndex = n_letters - 1
EOSIndex = n_letters - 2
```

The data consists of two things: a list of strings, called “categories”, contains all the categories (languages) for which we have data; a dictionary, called “names_data”, contains a list of names for each category.

Exercise 2.4.1: Inspect the data

1. [Just for yourself.] Find out what’s in the data set. How many different countries do we have? How many names per country? Are all names unique in a given country? Do the names sound typical to your ears for the given countries?

```
# If I use codes to discover:
import multiprocessing

def names_per_country(country):
    names = names_data[country]
    num_names = len(names)
    unique_names = len(set(names))
    return country, num_names, num_names == unique_names

# Create a pool of processes
pool = multiprocessing.Pool(processes=multiprocessing.cpu_count())

# Map the function over the list of categories (countries)
results = pool.map(names_per_country, categories)

# Close the pool to release resources
pool.close()
pool.join()
```

```
# Print the results
for country, num_names, are_unique in results:
    print(f"Country: {country}, Number of names: {num_names}, All
names are unique: {are_unique}")
```

Train-test split

We will split the data into a training and a test set. Look at the code and try to answer the exercise question of how this split is realized.

```
#####
## make a train/test split
#####

train_data = dict()
test_data = dict()
split_percentage = 10
for k in list(names_data.keys()):
    total_size = len(names_data[k])
    test_size = round(total_size / split_percentage)
    train_size = total_size - test_size
    print(k, total_size, train_size, test_size)
    indices = [i for i in range(total_size)]
    random.shuffle(indices)
    train_indices = indices[0:train_size]
    test_indices = indices[(train_size + 1) : (-1)]
    train_data[k] = [names_data[k][i] for i in train_indices]
    test_data[k] = [names_data[k][i] for i in test_indices]
```

```
Czech 519 467 52
German 724 652 72
Arabic 2000 1800 200
Japanese 991 892 99
Chinese 268 241 27
Vietnamese 73 66 7
Russian 9408 8467 941
French 277 249 28
Irish 232 209 23
English 3668 3301 367
Spanish 298 268 30
Greek 203 183 20
Italian 709 638 71
Portuguese 74 67 7
Scottish 100 90 10
Dutch 297 267 30
Korean 94 85 9
Polish 139 125 14
```

Exercise 2.4.2: Explain the train-test split

1. How is the original data information split into training and test set? (E.g., what amount of data is allocated to each part?; is the split exclusive and exhaustive?; how is it determined which item goes where?)

Defining the model

The model we use is a (hand-crafted) recurrent neural network. The architecture follows [this tutorial](#), from where we also borrow the following picture:

img{width=550px}

The model makes consecutive predictions about the next character. It is conditioned on three vectors:

1. 'category' is a one-hot vector encoding the country
2. 'input' is a one-hot vector encoding the character
3. 'hidden' is the RNN's hidden state (remembering what happened before)

These vectors are first combined and then used to produce a next-character probability distribution *and* the hidden state to be fed into the next round of predictions.

Next to the usual functions (initialization and forward pass), there is also a function that returns a blank 'hidden state'. This will be used later during training and inference, because at the start of each application (training or inference) the RNN should have a blank memory. (It makes sense to include this function in the definition of the module because it depends on the module's parameters (size of the hidden layer).)

Notice that the architecture features a dropout layer, which randomly sets a fixed proportion of units to 0. The inclusion of dropout introduces a random element in the model during training and inference.

```
#####  
## define RNN  
#####  
  
class RNN(nn.Module):  
    def __init__(self, input_size, hidden_size, output_size,  
dropout=0.1):  
        super(RNN, self).__init__()  
        self.hidden_size = hidden_size  
        self.i2h = nn.Linear(n_categories + input_size + hidden_size,  
hidden_size)  
        self.i2o = nn.Linear(n_categories + input_size + hidden_size,  
output_size)  
        self.o2o = nn.Linear(hidden_size + output_size, output_size)  
        self.dropout = nn.Dropout(dropout)  
        self.softmax = nn.LogSoftmax(dim=1)  
  
    def forward(self, category, input, hidden):  
        input_combined = torch.cat((category, input, hidden), 1)
```

```

        hidden = self.i2h(input_combined)
        output = self.i2o(input_combined)
        output_combined = torch.cat((hidden, output), 1)
        output = self.o2o(output_combined)
        output = self.dropout(output)
        output = self.softmax(output)
        return output, hidden

    def init_hidden(self):
        return torch.zeros(1, self.hidden_size)

```

Exercise 2.4.3: Inspect the model

1. [Just for yourself.] Make sure that you understand the model architecture and its implementation. E.g., do you agree that this code implements the model graph shown above? Can you think of slight alterations to the model which might also work?

Helper functions for training

For training, we will present the model with randomly sampled single items. This is why we define a 'random_training_pair' function which returns, well, a random training pair (category and name).

```

#####
## helper functions for training
#####

# Random item from a list
def random_choice(l):
    return l[random.randint(0, len(l) - 1)]

# Get a random category and random line from that category
def random_training_pair():
    category = random_choice(categories)
    line = random_choice(train_data[category])
    return category, line

```

We also need to make sure that the training and test data are in a format that the model understands. So, this is where we use vector representations for the categories and sequences of characters. For sequences of characters we distinguish those used as input to the model ('input_tensor') and those used in training as what needs to be predicted ('target_tensor').

```

# One-hot vector for category
def category_tensor(category):
    li = categories.index(category)
    tensor = torch.zeros(1, n_categories)
    tensor[0][li] = 1

```

```

    return tensor

# One-hot matrix of first to last letters (not including [EOS]) for
input
# The first input is always [SOS]
def input_tensor(line):
    tensor = torch.zeros(len(line) + 1, 1, n_letters)
    tensor[0][0][SOSIndex] = 1
    for li in range(len(line)):
        letter = line[li]
        tensor[li + 1][0][all_letters.find(letter)] = 1
    return tensor

def target_tensor(line):
    letter_indexes = [all_letters.find(line[li]) for li in
range(len(line))]
    letter_indexes.append(EOSIndex)
    return torch.LongTensor(letter_indexes)

```

Finally, we construct a function that returns a random training pair in the proper vectorized format.

```

# Make category, input, and target tensors from a random category,
line pair
def random_training_example():
    category, line = random_training_pair()
    category_tensor_ = category_tensor(category)
    input_line_tensor = input_tensor(line)
    target_line_tensor = target_tensor(line)
    return category_tensor_, input_line_tensor, target_line_tensor

```

Exercise 2.4.4: Understand the representational format

1. Write a doc-string for the function 'random_training_example' that is short but completely explanatory regarding the format and meaning of its output.

We use this timing function to keep track of training time:

```

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

```

Training the network

This function captures a single training step for one training triplet (category, input representation of the name, output representation of the string).

What is important to note here is that at the start of each “name”, so to speak, we need to supply a fresh ‘hidden layer’, but that subsequent calls to the RNN’s forward pass function will use the hidden layer that is returned from the previous forward pass.

```
#####  
## single training pass  
#####  
  
def train(category_tensor, input_line_tensor, target_line_tensor):  
    # reshape target tensor  
    target_line_tensor.unsqueeze_(-1)  
    # get a fresh hidden layer  
    hidden = rnn.init_hidden()  
    # reset cumulative loss  
    optimizer.zero_grad()  
    loss = 0  
    # zero the gradients  
    # sequentially probe predictions and collect loss  
    for i in range(input_line_tensor.size(0)):  
        output, hidden = rnn(category_tensor, input_line_tensor[i],  
hidden)  
        l = criterion(output, target_line_tensor[i])  
        loss += l  
    # perform backward pass  
    loss.backward()  
    # perform optimization  
    optimizer.step()  
    # return prediction and loss  
    return loss.item() # / input_line_tensor.size(0)
```

The actual training process is furthermore not very special.

```
#####  
## actual training loop  
## (should take about 2-4 minutes)  
#####  
  
# instantiate the model  
rnn = RNN(n_letters, 128, n_letters)  
# training objective  
criterion = nn.NLLLoss()  
# learning rate  
learning_rate = 0.0005
```

```

# optimizer
optimizer = torch.optim.Adam(rnn.parameters(), lr=learning_rate)
# training parameters
n_iters = 100000
print_every = 5000
plot_every = 500
all_losses = []
total_loss = 0 # will be reset every 'plot_every' iterations

start = time.time()

for iter in range(1, n_iters + 1):
    loss = train(*random_training_example())
    total_loss += loss

    if iter % plot_every == 0:
        all_losses.append(total_loss / plot_every)
        total_loss = 0

    if iter % print_every == 0:
        rolling_mean = np.mean(all_losses[iter - print_every * (iter
// print_every) :])
        print(
            "%s (%d %d%%) %.4f"
            % (time_since(start), iter, iter / n_iters * 100,
rolling_mean)
        )

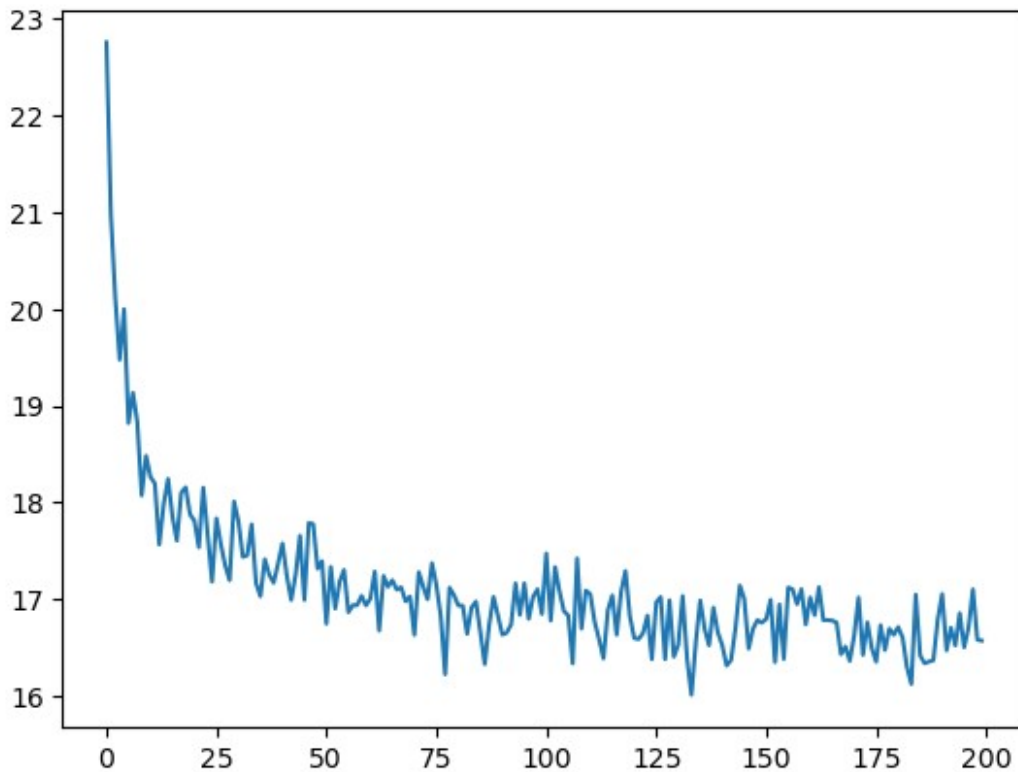
```

0m 9s (5000 5%) 19.6658
0m 18s (10000 10%) 18.8256
0m 26s (15000 15%) 18.4274
0m 35s (20000 20%) 18.1675
0m 43s (25000 25%) 18.0140
0m 52s (30000 30%) 17.8482
1m 0s (35000 35%) 17.7381
1m 9s (40000 40%) 17.6431
1m 18s (45000 45%) 17.5494
1m 26s (50000 50%) 17.4842
1m 35s (55000 55%) 17.4396
1m 44s (60000 60%) 17.3909
1m 52s (65000 65%) 17.3364
2m 1s (70000 70%) 17.2860
2m 10s (75000 75%) 17.2455
2m 18s (80000 80%) 17.2208
2m 27s (85000 85%) 17.1925
2m 35s (90000 90%) 17.1608
2m 44s (95000 95%) 17.1264
2m 52s (100000 100%) 17.1055

Here is a plot of the temporal development of the model's performance during training:


```
#####
## monitoring loss function during training
#####

plt.figure()
plt.plot(all_losses)
plt.show()
```



Exercise 2.4.5: Investigate the training regime

1. What exactly is the loss function here? What are we training the model on: perplexity, average surprisal, or yet something else?

Evaluation & inference

Let's see what the model has learned and how well it does in producing new names.

Here are some auxiliary functions to obtain surprisal values and related notions for sequences of characters. We can use them to compare the model's performance on the training and test data set.

```
#####
## evaluation
#####
```

```

def get_surprisal_item(category, name):
    category_tensor_ = category_tensor(category)
    input_line_tensor = input_tensor(name)
    target_line_tensor = target_tensor(name)
    hidden = rnn.init_hidden()
    surprisal = 0
    target_line_tensor.unsqueeze_(-1)

    for i in range(input_line_tensor.size(0)):
        output, hidden = rnn(category_tensor_, input_line_tensor[i],
hidden)
        surprisal += criterion(output, target_line_tensor[i])
    return surprisal.item()

def get_surprisal_dataset(data):
    surprisal_dict = dict()
    surp_avg_dict = dict()
    perplxty_dict = dict()
    for category in list(data.keys()):
        surprisal = 0
        surp_avg = 0
        perplxty = 0
        # training
        for name in data[category]:
            item_surpr = get_surprisal_item(category, name)
            surprisal += item_surpr
            surp_avg += item_surpr / len(name)
            perplxty += item_surpr ** (-1 / len(name))
        n_items = len(data[category])

        surprisal_dict[category] = surprisal / n_items
        surp_avg_dict[category] = surp_avg / n_items
        perplxty_dict[category] = perplxty / n_items

    return (surprisal_dict, surp_avg_dict, perplxty_dict)

def make_df(surp_dict):
    p = pandas.DataFrame.from_dict(surp_dict)
    p = p.transpose()
    p.columns = ["surprisal", "surp_scaled", "perplexity"]
    return p

surprisal_test = make_df(get_surprisal_dataset(test_data))
surprisal_train = make_df(get_surprisal_dataset(train_data))

print("\nmean surprisal (test):",
np.mean(surprisal_test["surprisal"]))

```

```
mean surprisal (test): 18.06200265467764
mean surprisal (train): 16.530164374188345
```

1. What do these two last numbers represent? What's better: a higher or lower value? What do the two numbers tell us when we compare them?

Note that for each prediction step with the model, the model is embedded under `torch.no_grad()`. This setting means that gradients are not computed during the forward passes through the model. This setting is recommended for *inference* (for any neural net), i.e., when you don't want to train the model anymore (only use it for generating predictions). This allows to increase computational efficiency and save memory.

```
#####  
## prediction function  
#####  
  
max_length = 20  
  
# make a prediction based on given sequence  
def predict(category, initial_sequence):  
  
    if len(initial_sequence) >= max_length:  
        return initial_sequence  
  
    category_tensor_ = category_tensor(category)  
    input_line_tensor = input_tensor(initial_sequence)  
    hidden = rnn.init_hidden()  
  
    name = initial_sequence  
  
    for i in range(input_line_tensor.size(0)):  
        with torch.no_grad():  
            output, hidden = rnn(category_tensor_,  
input_line_tensor[i], hidden)
```

```

# greedy decoding: choosing the most likely guess
topv, topi = output.topk(1)
topi = topi[0][0]

if topi == EOSIndex:
    return name
else:
    name += all_letters[topi]

return predict(category, name)

print(predict("German", "Müll"))
print(predict("German", "Müll"))
print(predict("German", "Müll"))
print(predict("German", "Müll"))

Müllen
Müllen
Müllenger
Müllen

```

You can also probe the model with an empty string:

```

print(predict("Japanese", ""))
print(predict("Japanese", ""))
print(predict("Japanese", ""))
print(predict("Japanese", ""))

Maka
Kana
Maka
Muka

```

Exercise 2.4.7: Explore the model's predictions

1. Play around with these prediction functions for a country or several of which you have robust intuitions about how names from that country might sound. Report on one feature that speaks in favor of the model, and one that suggests that the model is not perfect (or seriously flawed).
2. Is the prediction function as implemented in this subsection stochastic or not?

Inverting the generation model

The model we have trained here could be consider a **speaker model**: the model generates language (very limited in scope but still). Additionally, the model can be used to probe how likely a particular name would be (as a generated string) for a particular category/country. So, essentially we get something like $P_s(\text{name} \mid \text{category})$ as a speaker likelihood function. For instance, we can do this:

```
print(get_surprisal_item("German", "Franke"))  
print(get_surprisal_item("Arabic", "Franke"))  
  
13.716519355773926  
19.072649002075195
```

Exercise 2.4.8: Category prediction with Bayes rule

1. Use the model as a speaker likelihood function to compute, via Bayes rule, the probability $P(\text{category} \mid \text{name})$ for the names "Dovesky", "Jackson" and "Satoshi". Compare the results against those obtained by the RNN-based classifier reported in [this tutorial](#) (all the way at the end).