Module 44 : Sequence models in Pytorch

Outline : i) introduce dataset — preparing the data.

ii) data processing : train-test split, encoding, visualization

iii) basic RNN - testing/inference — to check if everything is working

iv) evaluation and training

v) LSTM

vi) GRU

vii) Exercises

*) importing libraries : from io import open

```
import os, string, random, time, math
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np

from sklearn.model_selection import train_test_split

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

device_gpu = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

from IPython.display import clear_output

↓

used to clear o/p of cell

↓

for ex. needing to show multiple plots, but retaining only the last instance, then this construct can be used to clear the intermediate o/ps programmatically

*) Task: Given a name, o/p the language of the name typically.

↓

name: varying length string

multi classification task. → softmax based classifier

↓

sequence to class task. (sequence labelling task)

*) Dataset:

languages = [ ]

data = [ ]

X = [ ]

Y = [ ]

```
with open ('name2lang.txt', 'r') as f:
    for line in f:
        line = line.split(',')
        name = line[0].strip()
        lang = line[1].strip()
        if not lang in languages:
            languages.append(lang)
        X.append(name)
        Y.append(lang)
        data.append((name, lang))
```

storing it in diff formats.

```
n_languages = len(languages)

print(languages)
print(data[0:10])
```

→ train-test split

```
X_train, X_test, Y_train, Y_test =
        train_test_split(X, Y, test_size = 0.2
                         , random_state = 0
                         , stratify = y)
```

*) encoding names and languages: languages basically represent classes, so we encode them using one hot encoding, whereas the input strings will require an encoding of each character, and then stacking of these vector encodings as one entity. Characters are encoded since they are the one sent to the model one after the other.

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

↓

used to dimension our sequence models

to accepts i/ps of upto n letters

↓

one hot encoding for each character

→ encoding fn for string

```
def name_rep(name):
    rep = torch.zeros(len(name), 1, n_letters)
    for index, letter in enumerate(name):
        pos = all_letters.find(letter)
        rep[index][pos] = 1
    return rep
```

```
def lang_rep(lang):
    return torch.tensor([languages.index(lang)],
                        dtype = torch.long)
```

↓

index label corresponding to the language.

→ Basic visualization

```
count = {}
for l in languages:
    count[l] = 0
for d in data:
    count[d[1]] += 1
```

```
plt_ = sns.barplot(list(count.keys()),
                   list(count.values()))
```

```
plt_.set_xticklabels(plt_.get_xticklabels(),
                     rotation = 90)
plt.show()
```

→ rotate the labels to avoid visual overlap.

figuring out the baseline accuracy (accuracy on random class selection) for a dataset.

*) Basic net and testing inference

```
class RNN_net (nn.Module):
```

corresponding to size of 1-hot encoded vector

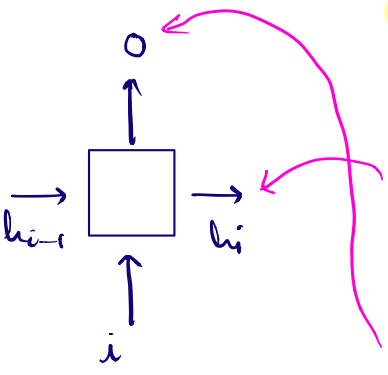
```
    def __init__(self, input_size, hidden_size,
                 output_size):
```

output_size → no. of classes

```
        super(RNN_net, self).__init__()
```

→ parent module's 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)
```



i/p vector

initial hidden state

```
    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
```

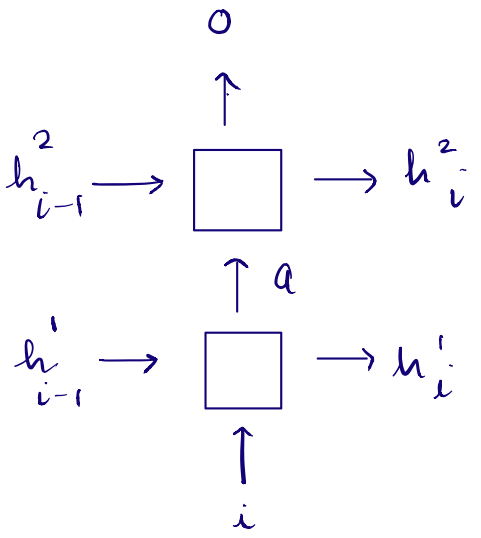
concatenating about a particular axis

→ can be avoided for inference, since softmax is exp.

↓ can be done by defining own boolean.

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

•) Extending the idea to 2 layers.



(net i/p of this layer will be concatenation of a and h^2_{i-1}) & so on.

(net i/p of this layer will be concatenation of i & h^1_{i-1})

→) hidden_size is a hyperparameter, and is to be tuned based on the complexity of the task and the amount of data at hand.

→ greater the value, more is the capacity for complexity with the chance of overfitting

```
n_hidden = 128
net = RNN_net(n_letters, n_hidden, n_languages)
```

→ a little non trivial as compared to DNN, or CNN.

```
def infer(net, name):
    net.eval()
    name_ohe = name_rep(name)
```

→ one hot encoding repr. of the i/p.

```
hidden = net.init_hidden()
```
→ initializing hidden unit

→ one character at a time

```
for i in range(name_ohe.size()[0]):
    output, hidden = net(name_ohe[i], hidden)
```

return output → all o/ps but the last one ignored, where the last one returns the probability distribution corresponding to the labels of the task.

*) name_ohe[i] represents a vector corresponding to (1, n_letters). The reason we had the middle dimension, is because we always want to give an i/p to the RNN which is (1, x).

```
output = infer(net, "Adam")
index = torch.argmax(output)
print(output, index)
```
→ inference run

→ fetching class with max probability

*) evaluate model

→ to remove bias due to correlation between samples being in a sequence.

```
def dataloader(npoints, X_, Y_):
    to_ret = []
    for i in range(n_points):
```
→ no. of samples to sample from dataset

```
        index_ = np.random.randint(len(x_))
        name, lang = x_[index_], y_[index_]
        to_ret.append((name, lang, name_rep(name),
                       lang_rep(lang)))
    return to_ret
```

one hot encodings

```
dataloader(2, x_train, y_train)
```

top k metric

```
def eval(net, n_points, k, x_, y_):
    data_ = dataloader(n_points, x_, y_)
    correct = 0

    for name, language, name_one, lang_rep in data_:
        output = infer(net, name)
        val, indices = output.topk(k)

        if lang_repr in indices:
            correct += 1

    accuracy = correct / n_points
    return accuracy

eval(net, 1000, 3, x_test, y_test)
```

*) Training

computation for one batch.

```
def train(net, opt, criterion, n_points):
    opt.zero_grad()
    total_loss = 0

    data = dataloader(n_points, X_train, Y_train)

    for name, lang, name_ohe, lang_rep in data:
        hidden = net.init_hidden()
        for i in range(name_ohe.size[0]):
            output, hidden = net(name_ohe[i],
                                 hidden)

        loss = criterion(output, lang_rep)
    loss.backward(retain_graph=
                  True)

    total_loss += loss

    opt.step()

    return total_loss / n_points
```

Creates the compute graph

→ backpropagation through the dynamically created compute graph

```
criterion = nn.NLLLoss()
opt = optim.SGD(net.parameters(), lr=0.01,
                momentum=0.9)

%%time
train(net, opt, criterion, 200)
eval(net, 1000, 1, x_test, y_test)
```

good practice to have separate infer, train & training setup

*) full training setup : multiple batches, observing how loss is changing.

```
def train_setup(net, lr=0.01, n_batches=100,
                batch_size=10, momentum=0.9,
                display_freq=5):

    criterion = nn.NLLLoss()
```

→ can be cross entropy if softmax not applied on o/p. → beneficial for inference separation for speedup.

```
    opt = optim.SGD(net.parameters(), lr=lr,
                    momentum=momentum)

    loss_arr = np.zeros(n_batches+1)
```

→ this workflow can also contain checkpointing and saving of models.

```
    for i in range(n_batches):
        loss_arr[i+1] = (loss_arr[i]*i
                         + train(net, opt, criterion, batch_size))/(i+1)
```

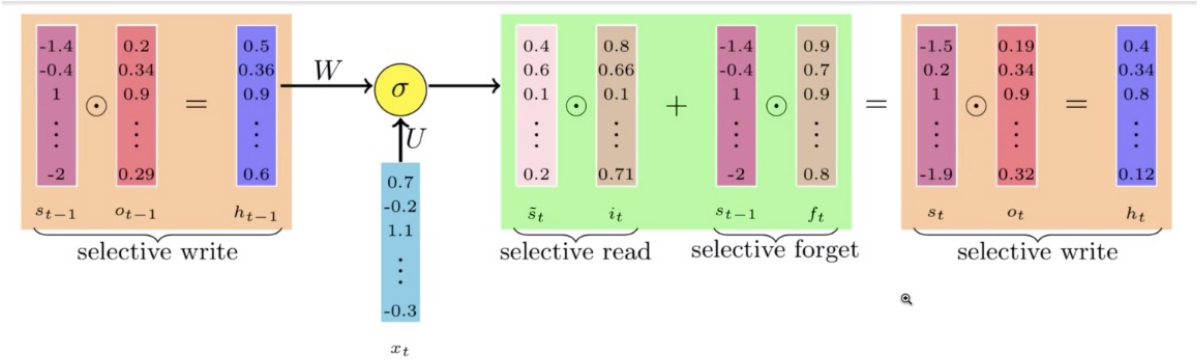
```
if i%. display_freq == display_freq - 1:
    clear_output (wait = True)
    print ('Iteration', i, 'Top-1:',
eval (net, len( X_test), 1, X_test, y_test),
    'Top-2:', eval (net, len(X_test),
                    2, X_test, y_test)

    plt.figure ()
    plt.plot (loss_arr [1:i], '-*')
    plt.xlabel ('Iteration')
    plt.ylabel ('Loss')
    plt.show ()
    print ('\n\n')
```

→ n_hidden = 128

net = RNN_net (n_letters, n_hidden, n_languages)

train_setup (net, lr = 0.0005, n_batches = 100,
batch_size = 256)

*) LSTM



CLASS `torch.nn.LSTM(*args, **kwargs)` [SOURCE]

Applies a multi-layer long short-term memory (LSTM) RNN to an input sequence.

For each element in the input sequence, each layer computes the following function:

$$
\begin{aligned}
i_t &= \sigma(W_{ii}x_t + b_{ii} + W_{hi}h_{(t-1)} + b_{hi}) \\
f_t &= \sigma(W_{if}x_t + b_{if} + W_{hf}h_{(t-1)} + b_{hf}) \\
g_t &= \tanh(W_{ig}x_t + b_{ig} + W_{hg}h_{(t-1)} + b_{hg}) \\
o_t &= \sigma(W_{io}x_t + b_{io} + W_{ho}h_{(t-1)} + b_{ho}) \\
c_t &= f_t * c_{(t-1)} + i_t * g_t \\
h_t &= o_t * \tanh(c_t)
\end{aligned}
$$

→ LSTM block

fn. handles major computations.

where h_t is the hidden state at time *t*, c_t is the cell state at time *t*, x_t is the input at time *t*, $h_{(t-1)}$ is the hidden state of the layer at time *t-1* or the initial hidden state at time 0, and i_t, f_t, g_t, o_t are the input, forget, cell, and output gates, respectively. σ is the sigmoid function, and $*$ is the Hadamard product.

In a multilayer LSTM, the input $x_t^{(l)}$ of the l -th layer ($l >= 2$) is the hidden state $h_t^{(l-1)}$ of the previous layer multiplied by dropout $\delta_t^{(l-1)}$ where each $\delta_t^{(l-1)}$ is a Bernoulli random variable which is 0 with probability `dropout`.

In terms of fun signature, tried to keep things same to as they were for RNN.

```
class LSTM_net(nn.Module):

    def __init__(self, input_size, hidden_size, output_size):
        super(LSTM_net, self).__init__()
        self.hidden_size = hidden_size
        self.lstm_cell = nn.LSTM(input_size,
                                 hidden_size)
        self.h2o = nn.Linear(hidden_size,
                             output_size)
        self.softmax = nn.LogSoftmax(dim=2)
```

to take benefit of vectorization

multiple i/ps at same time

lstm cell allows for multiple layer of transformations

```
    def forward(self, input, hidden):
        out, hidden = self.lstm_cell
        ← (input.view(1, 1, -1), hidden)
```

to get performance benefit

batches

no. of levels

↓ comprises of the hidden + cell state

```
output = self.h2o(hidden[0])
output = self.softmax(output)
return output.view(1, -1), hidden
```

→ to be sent for the next character

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

↓

different from RNN, now we have a tuple of two tensors. one for hidden state and one for cell state.

```
n_hidden = 128
net = LSTM_net(n_letters, n_hidden, n_languages)
train_setup(net, lr = 0.0005, n_batches = 100, batch_size = 256)
```

*) GRU cell

GRU

CLASS `torch.nn.GRU(*args, **kwargs)` [SOURCE]

Applies a multi-layer gated recurrent unit (GRU) RNN to an input sequence.

For each element in the input sequence, each layer computes the following function:

$$
\begin{aligned}
r_t &= \sigma(W_{ir} x_t + b_{ir} + W_{hr} h_{(t-1)} + b_{hr}) \\
z_t &= \sigma(W_{iz} x_t + b_{iz} + W_{hz} h_{(t-1)} + b_{hz}) \\
n_t &= \tanh(W_{in} x_t + b_{in} + r_t * (W_{hn} h_{(t-1)} + b_{hn})) \\
h_t &= (1 - z_t) * n_t + z_t * h_{(t-1)}
\end{aligned}
$$

→ only state to save

where h_t is the hidden state at time t, x_t is the input at time t, $h_{(t-1)}$ is the hidden state of the layer at time t-1 or the initial hidden state at time 0, and r_t, z_t, n_t are the reset, update, and new gates, respectively. σ is the sigmoid function, and $*$ is the Hadamard product.

In a multilayer GRU, the input $x_t^{(l)}$ of the l -th layer ($l >= 2$) is the hidden state $h_t^{(l-1)}$ of the previous layer multiplied by dropout $\delta_t^{(l-1)}$ where each $\delta_t^{(l-1)}$ is a Bernoulli random variable which is 0 with probability `dropout`.

```
class GRU_net (nn.Module):
    def __init__ (self, input_size, hidden_size, output_size):
        super (GRU_net, self).__init__()
        self.hidden_size = hidden_size
        self.gru_cell = nn.GRU (input_size, hidden_size)
        self.h2o = nn.Linear(hidden_size, output_size)
        self.softmax = nn.LogSoftmax (dim=2)

    def forward (self, input_, hidden):
        out, hidden = self.gru_cell (input.view(1,1,-1),
                                     hidden)
        output = self.h2o(hidden)
        output = self.softmax (output)
        return output.view(1,-1), hidden

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

n_hidden = 128
net = gru_net (n_letters, n_hidden, n_languages)
train_setup (net, lr= 0.0005, n_batches = 100, batch_size = 256)
```

topk accuracy may be misleading

•) exercises: 0. compute language wise accuracy, confusion matrix

i, J: if ground truth was i, how many times I am classifying it as J

1. play with hyper-parameters (including hidden layer size) & trying to get better results.
2. observe gradient explosion at higher learning rates
3. increase performance by moving to GPUs
4. Think on how to increase performance further (benefit of batching)

*) SIMD: Single instruction multiple data (benefit of batching on GPU)