DSC 275/475: Time Series Analysis and Forecasting (Fall 2023)

Project 3.1 – Sequence Classification with Recurrent Neural Networks

(Total points: 60 – Undergraduate Students; 70 (Graduate students; Extra credit for Undergraduate students))

Overview

In Project 3, you will work with recurrent neural networks on real-world data. The goal of Project 3.1 is to train an RNN from scratch on a set of data containing over 20 thousand Last Names and their respective Country of Origin. The network should thus be able to predict the country of origin correctly based on a given last name by evaluating the sequence of characters in the given last name. You will implement progressively complex training and testing/validation scenarios for RNNs.

Points highlighted in red denote problems optional for undergraduate students and earns extra credit.

For the submission, please make sure to hand in the following:

- A document (PDF, Word etc) that captures your responses to the questions below *separately* from the code in order to facilitate grading.
- Your code files

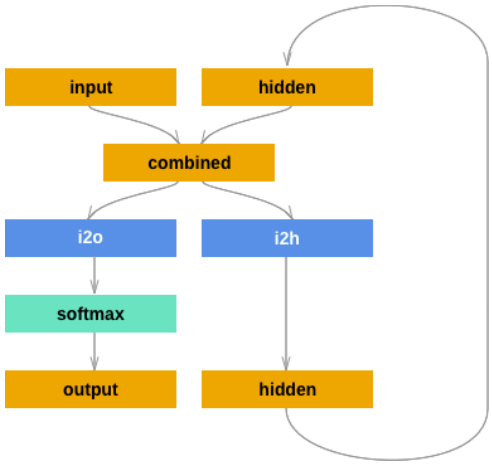
INSTRUCTIONS:

- You are welcome to work on this project individually or in teams (up to 2 members in each team max).
- If you plan to use PyTorch, a good resource is to review and modify the example code provided for each problem. We plan to review this example code in class as well.

1. Hidden States and Systematic Training

Script `char_rnn_classification_Project3p1_Q1.py`[1] implements a vanilla RNN from scratch and trains it on a set of data containing over 20 thousand last names and their respective country of origin. The number of possible countries, or classes, is 18. All of the data for this project is in the file: data.zip. Each country of origin is contained in a separate file with the corresponding last names. A diagrammatic view of the structure of the implemented RNN is as follows:

(1 Modified from https://pytorch.org/tutorials/intermediate/char_rnn_classification_tutorial.html



The architecture is such that the input (i.e. characters of the Last Name in this example) to the network is concatenated/stacked with the current hidden state to form a 'combined' stacked tensor *(Refer to Lines 72-86 in the example code to see the structure of the tensors).* The combined tensor is then fed through two different paths, i2h which updates the hidden state, and i2o which produces an output. Although outputs are produced at each time step, given that we are dealing with a classification task, only the output at the last time step is used.

1.1. (15 points) *Effect of hidden state length* - run the script for hidden state sizes of 2, 8 and 32 by modifying the value of variable `n_hidden`.

What is the Accuracy yielded for different hidden state sizes? Also, include a graph of the loss function and the confusion matrix for each case.

Note that you will have to modify the existing evaluation function, as it measures accuracy from a randomly sampled population, which could lead to biased results:

```
for i in range(n_confusion):
    category, name, category_tensor, name_tensor =
randomTrainingExample()
    output = evaluate(name_tensor)
    guess, guess_i = categoryFromOutput(output)
    category_i    =    languages.index(category)
    confusion[category_i][guess_i] += 1
```

You will want to measure performance on *every* sample in the dataset by comparing the label of the sample to the output of your trained network, and report the average accuracy.

With hidden state sizes 2, the accuracy is 42.72%.
With hidden state sizes 8, the accuracy is 48.999%.
With hidden state sizes 32, the accuracy is 58.64%.

1.2. (20 points) *Effect of systematic training* - the script trains the network by going through 100 thousand data samples one by one in a random manner:

(1 Modified from
https://pytorch.org/tutorials/intermediate/char_rnn_classification_tutorial.html

```
for iter in range(1, n_iters + 1):
    category, name, category_tensor, name_tensor =
randomTrainingExample()
    output,   loss   =train(category_tensor,name_tensor)
    current_loss += loss
```

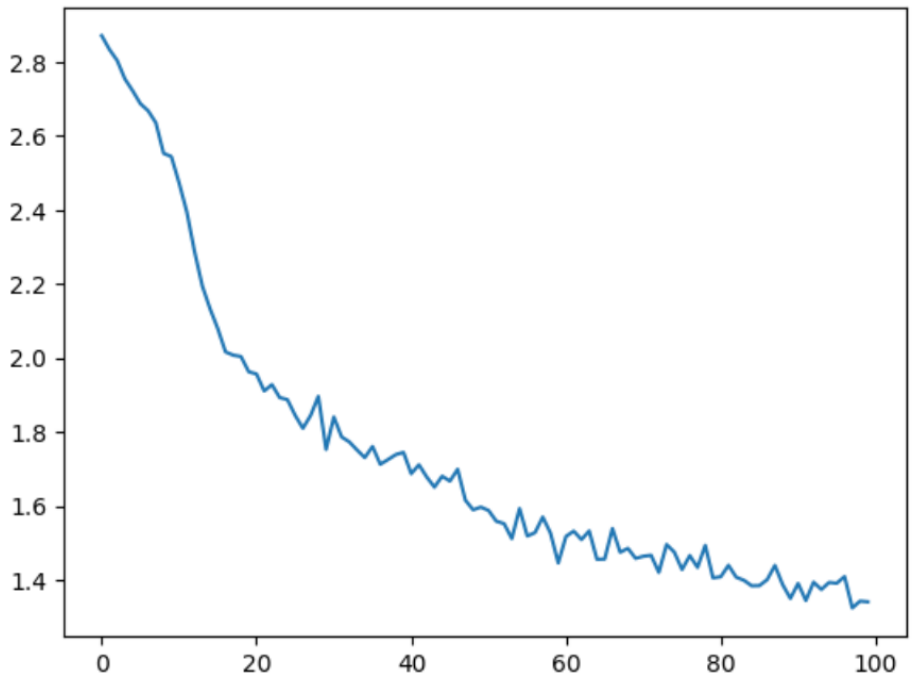
The network parameters are updated by backpropagating losses computed on a per-sample basis. Also note that the there is only 1 training epoch in the example provided.

Modify the script so that, instead of picking each training sample randomly, it goes through every available sample exactly once per training epoch. Randomize the order of the samples within each epoch. Train the network for five epochs, and report results as you change the hidden state size as in Problem 1.1 above. Note that, since the dataset comprises around 20000 datapoints, the total number of data passes for this modified training process are like those required by Problem 1.1.

What is the accuracy thus trained and evaluated for the different values of hidden state size (2,8,32)? Also, include a graph of the loss function and the confusion matrix for each case.

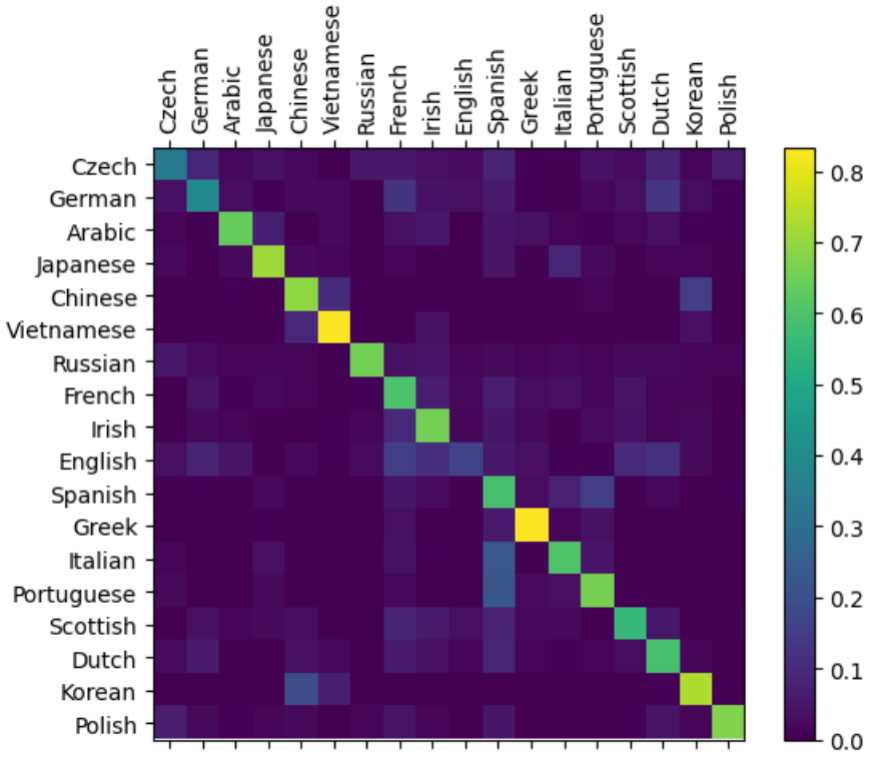
With hidden state sizes 2, the accuracy is 43.95%.
With hidden state sizes 8, the accuracy is 54.33%.
With hidden state sizes 32, the accuracy is 60.68%.

Graph of loss function:

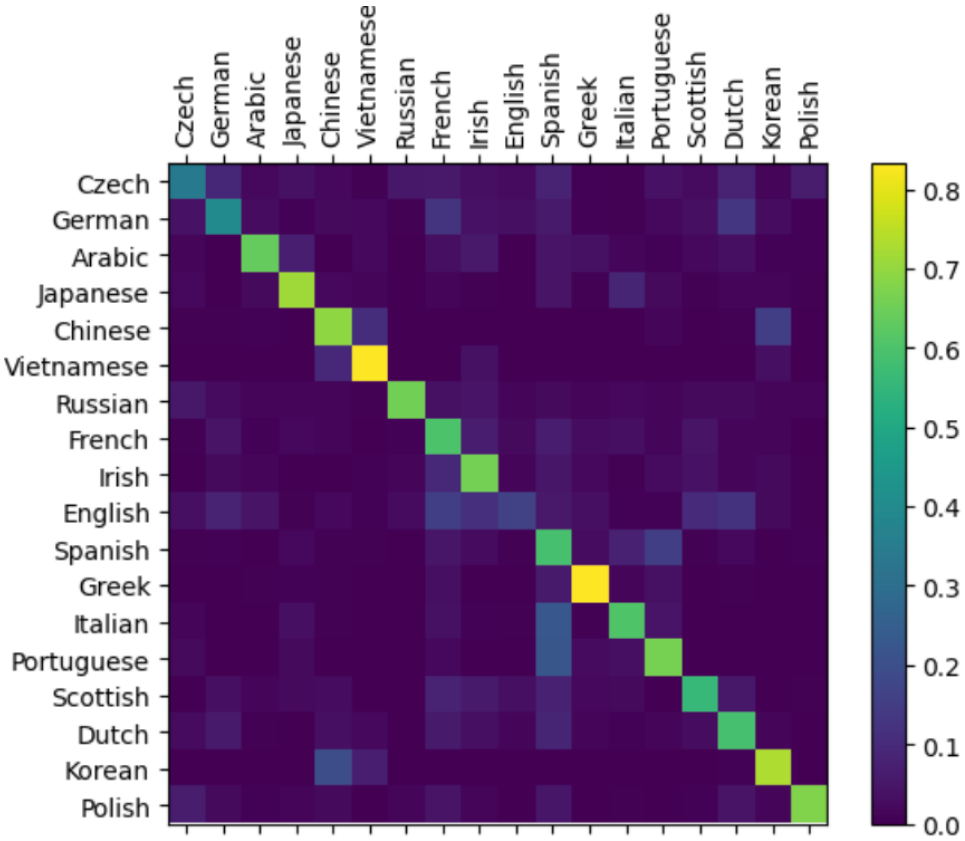


(1 Modified from
https://pytorch.org/tutorials/intermediate/char_rnn_classification_tutorial.html

Hidden layer 2:

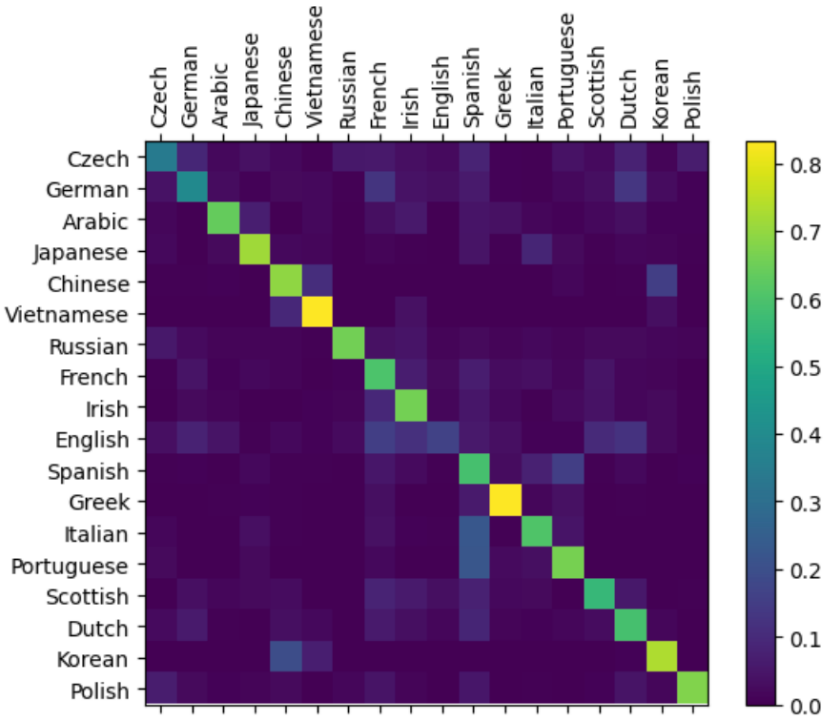


Hidden layer 8:



(1 Modified from
https://pytorch.org/tutorials/intermediate/char_rnn_classification_tutorial.html

Hidden layer 32:



2. Batch training of data

The training process in Problem 1 above is still suboptimal since the gradient estimates are computed on a sample-by-sample basis (i.e., a batch size of 1). In this question, you are exposed to batch processing.

For this question, you can use the code sample *"char_rnn_classification_Project3p1_Q2.py"* which provides you an outline of how the code can be structured to meet the goals of this question. Of course, you are welcome to create your own custom code from scratch!

2.1. **(25 points)** Modify the implementation of the network to leverage the RNN subclass of module `torch.nn`, which readily incorporates support for batch training. Note that the "nn.RNN" class is modified to operate in a batch mode.

Set the hidden state size to 128 and train the network through five epochs with a batch size equal to the total number of samples. Note that, since the data samples are of different lengths, you will need to pad the length of the samples to a unique sequence length (e.g., at least the length of the longest sequence) in order to be able to feed the batch to the network.
This is because RNN expects the input to be a tensor of shape (*batch, seq_len, input_size*). It is best to manually pad with 0s, or you can use built-in functions such as `torch.nn.utils.rnn.pad_sequence` to perform the padding.

(1 Modified from
https://pytorch.org/tutorials/intermediate/char_rnn_classification_tutorial.html

Report the accuracy yielded by this approach on the full training set after training for 5 epochs.

The code below indicates the modified section of nn.RNN class. This section is included in the example script.

```
class  RNN(nn.Module):
    def_init__(self):
        super(RNN, self)._init_()

        self.rnn     =     nn.RNN(
            input_size           =
            INPUT_SIZE,
            hidden_size = HIDDEN_SIZE,          #   number   of   hidden
            units num_layers = N-LAYERS,        #   number   of   layers
            batch_first = True,         # If your input data is of shape
(seq_len, batch_size, features) then you don't need batch_first=True and your
RNN will output a tensor with shape (seq_len, batch.
            #If your input data is of shape (batch_size, seq_len, features)
then you need batch_first=True and your RNN will output a tensor with shape
(batch_size, seq_len, hidden_size).
        )
        self.out = nn.Linear(HIDDEN_SIZE, OUTPUT_SIZE)

    def forward(self, x):
        # r_out, (h_n, h_c) = self.rnn(x, None)    #   None   represents
zero initial hidden state
        r_out, h = self.rnn(x, None)    # None represents zero initial hidden
state

        # choose last time step of output
        out = self.out(r_out[:, -1, :])
        return out
```

Recall that `input_size` refers to the size of the features (in this case one-hot encoded representation of each letter). Hidden size is a hyper parameter you can adjust (we will keep it fixed at 128 in this question). The comments in the code above explain how to format your batch data, depending on the value of `batch_first`. Lastly, `OUTPUT_SIZE` denotes the number of classes.

Epoch: 1 | accuracy: 72.80%
Epoch: 2 | accuracy: 71.85%
Epoch: 3 | accuracy: 72.56%
Epoch: 4 | accuracy: 71.62%
Epoch: 5 | accuracy: 70.66%

2.2. **(10** points**)** Modify the implementation from 2.1 to support arbitrary mini-batch sizes. In this case, instead of padding to a unique sequence length, adaptively pad the length of the mini batch to the length of the longest sample in the mini batch itself. Report the accuracy

(1 Modified from
https://pytorch.org/tutorials/intermediate/char_rnn_classification_tutorial.html

number (on the full training set) yielded by this approach on mini batch sizes of 1000, 2000, 3000 after five epochs of training.

Note that since these problems only ask you to train for five epochs it won't be graded based on performance (unless you get significantly smaller numbers than what's reasonable for five epochs of training).

Epoch: 1 | Mini-batch size: 1000 | accuracy: 0.7098
Epoch: 2 | Mini-batch size: 1000 | accuracy: 73.05%
Epoch: 3 | Mini-batch size: 1000 | accuracy: 71.63%
Epoch: 4 | Mini-batch size: 1000 | accuracy: 71.20%
Epoch: 5 | Mini-batch size: 1000 | accuracy: 70.34%
Epoch: 1 | Mini-batch size: 2000 | accuracy: 70.26%
Epoch: 2 | Mini-batch size: 2000 | accuracy: 70.75%
Epoch: 3 | Mini-batch size: 2000 | accuracy: 70.27%
Epoch: 4 | Mini-batch size: 2000 | accuracy: 69.94%
Epoch: 5 | Mini-batch size: 2000 | accuracy: 69.35%
Epoch: 1 | Mini-batch size: 3000 | accuracy: 65.85%
Epoch: 2 | Mini-batch size: 3000 | accuracy: 66.51%
Epoch: 3 | Mini-batch size: 3000 | accuracy: 67.39%
Epoch: 4 | Mini-batch size: 3000 | accuracy: 66.34%
Epoch: 5 | Mini-batch size: 3000 | accuracy: 65.61%

(1 Modified from
https://pytorch.org/tutorials/intermediate/char_rnn_classification_tutorial.html