Please write a report for this project.

We will be building and training a basic character-level RNN and LSTM to classify words. A character-level RNN reads words as a series of characters—outputting a prediction and "hidden state" at each step, feeding its previous hidden state into each next step. We take the final prediction to be the output, i.e. which class the word belongs to. Specifically, we'll train on a few thousand surnames from 18 languages of origin, and predict which language a name is from based on the spelling:

```
$ python predict.py Hinton
(-0.47) Scottish
(-1.52) English
(-3.57) Irish

$ python predict.py Schmidhuber
(-0.19) German
(-2.48) Czech
(-2.68) Dutch
```

Step 1. Preparing data. Download the data from here "`https://download.pytorch.org/tutorial/data.zip`" and extract it to the current directory. Included in the data/names directory are 18 text files named as "[Language].txt". Each file contains a bunch of names, one name per line, mostly romanized (but we still need to convert from Unicode to ASCII). We will end up with a dictionary of lists of names per language, {language: [names ...]}. The generic variables "category" and "lines" (for language and name in our case) are used for later extensibility. In the code "RNN_vs_LSTM.py", I have implemented the above procedure, which you can start from.

Step 2. Turning Names into Tensors. Now that we have all the names organized, we need to turn them into tensors to make any use of them. To represent a single letter, we use a "one-hot vector" of size <1 x n_letters>. A one-hot vector is filled with 0s except for a 1 at index of the current letter, e.g. "b"=<0 1 0 0 0 $\cdots$>. To make a word we join a bunch of those into a 2D matrix <line_length x 1 x n_letters>, where the extra 1 dimension is because PyTorch assumes everything is in batches–we are using a batch size of 1 here.

Step 3. Create the Network. We create a recurrent neural network in PyTorch involved cloning the parameters of a layer over several timesteps. The layers held hidden state and gradients which are now entirely handled by the computational graph itself. Try to understanding the RNN implementation in the code "RNN_vs_LSTM.py".

Step 4. Training. We have implemented a very simple optimization algorithm for training RNNs in the code "RNN_vs_LSTM.py".

Step 5. Evaluating the Results. On the one hand, we can plot iterations vs. training loss. On the other hand, to see how well the network performs on different categories, we will create a confusion matrix, indicating for every actual language (rows) which language the network guesses (columns). Moreover, we can input a last name to the trained model and evaluate how accurate the prediction is.

Your tasks:

1. Given an input sequence $\{\boldsymbol{x}_1, \boldsymbol{x}_2, \cdots\}$, recall that the hidden state of RNNs updates as follows

$$\boldsymbol{h}_t = \sigma(\boldsymbol{U}\boldsymbol{h}_{t-1} + \boldsymbol{W}\boldsymbol{x}_t),$$

   where $\sigma(\cdot)$ is a nonlinear activation function, e.g. sigmoid and tanh, and $\boldsymbol{U}$ and $\boldsymbol{W}$ are the matrices of trainable weights.

   In class, we also introduced a new model called MomentumRNN to mitigate the vanishing gradient issue of training RNNs. The hidden state of MomentumRNN follows the following dynamics

$$\boldsymbol{v}_t = \mu\boldsymbol{v}_t + s\boldsymbol{W}\boldsymbol{x}_{t-1}; \quad \boldsymbol{h}_t = \sigma(\boldsymbol{U}\boldsymbol{h}_{t-1} + \boldsymbol{v}_t),$$

   where $\mu, s \geq 0$ are two tunable hyperparameters, and both of them are scalars. Can you derive the MomentumRNN by inspiration from the gradient descent vs. the heavy-ball method? And further show that why MomentumRNN can overcome the vanishing gradient problem.

2. Implement MomentumRNN for the above character-level RNN task, and compare the performance of MomentumRNN with RNN.

3. Can you integrate momentum into LSTM in the same manner as deriving the MomentumRNN? Moreover, can you implement the resulting model, called MomentumLSTM, and compare its performance with LSTM?