NNFL Assignment 2 Readme

PART 1

Aim: To familiarise students with deep neural architectures and coding them, in PyTorch. Additionally, to create an exercise to simulate the typical workflow that follows in a research paper implementation in order to prepare students for the next assignment.

Objective: To build a multi-class classifier for Fake News Detection on the LIAR-PLUS dataset.

Note: It is recommended that you skim through the paper - "Where is your Evidence: Improving Fact-checking by Justification Modelling". This will provide a comprehensive overview of what the data looks like and its purpose. You don't need to get an in-depth understanding of the methods or architecture that they have used. The goal is to simply understand the problem statement and dataset.

Instructions: This file is a readme for the assignment. All functions and classes have been sufficiently described here. Kindly go through this file carefully and follow the step-wise instructions to successfully understand and implement this assignment. There are multiple fill in the blanks throughout this assignment that you will have to complete which will be evaluated disjointly.

Note: We DO NOT expect you to train the model. The evaluation will be based on function-specific test cases and code. However, you are free to try and run the code if you have the required computation power and time.

Important: DO NOT import any extra modules, change any lines of code, rename or edit function prototypes, change class names or default values etc. unless specifically asked to do so in the evaluative parts. Changing other code may cause the code to break.

System Requirements - Pytorch - Minimum version 1.5.0, 1.6.0 preferable.

Nltk (pip3 install nltk). Run nltk.download('punkt') before starting.

Directory Structure (Top-down):

1. **main.py**: Contains all code to instantiate appropriate classes and call appropriate functions in the correct order from all other files. This will have evaluative parts, further instructions will follow.

2. **test.py**: contains set of test cases to evaluate the different functions. This file is **non-evaluative**, as in you are not expected to write/change any code in this file. The sample cases are for your convenience to test your code. There will be additional hidden test cases which will be used to grade the other evaluative functions.

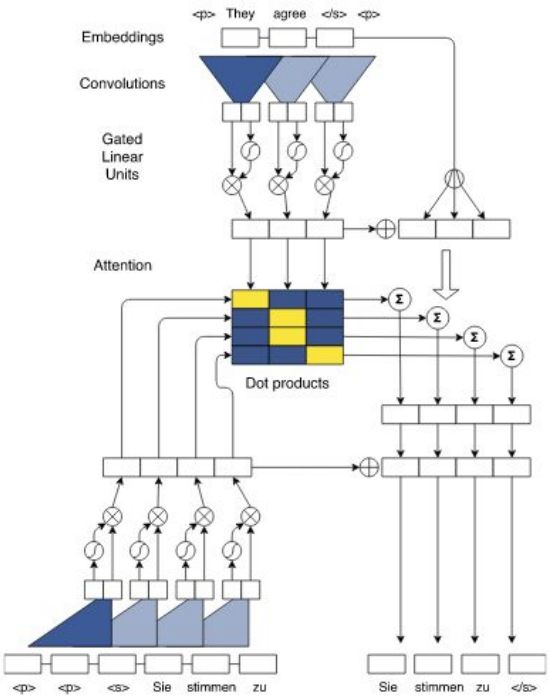
3. **trainer.py**: contains code to train the model. This file is partially evaluative.

4. **LiarLiar.py**: class that defines the network architecture. This file is partially evaluative.

5. **Encoder.py**: defines the encoder used in the network architecture. Partially evaluative.

6. **Attention.py**: defines the Multi-head Attention and Position Feed Forward networks used in the encoder. Partially evaluative.

7. **Conv2S2S.py**: defines the convolutional encoder used in the encoder network architecture. Partially evaluative. This convolutional encoder is based on the paper: "Convolutional Sequence to Sequence Learning - https://arxiv.org/pdf/1705.03122.pdf". *NOTE*: you do not need to read or understand the entire paper. The architecture used in this paper is roughly similar to the figure as shown below, we need only the encoder part of it.



8. **Datasets.py**: code to load and pre-process the dataset. The end objective is to obtain vectors for each row in the dataset so that it can be used in the neural model for training and testing. partially evaluative.

Note - The docstring of “datasets.py” contains instructions to download the dataset and the Glove vectors.

9. **Utils.py**: some extra utility functions, partially evaluative.

Implementation (Bottom-Up):

In the implementation part, we will follow a bottom-up approach. We will start with implementing the utility functions and functions for data processing. This will be followed by implementation of parts of the network architecture including the convs2s layers, attention layers and feed-forward

network and putting it all together in the encoder. Finally, we will write the train function and main function that will complete the implementation.

1. Utils.py:

Non-evaluative part: the functions - visualize_Attenion(...) and infer(...) are non-essential. If someone does train the model, they can uncomment these and use them to infer and visualise attention.

Following are the definitions of the functions you need to implement for evaluation:

I. *get_max_length(dataframe: pandas.DataFrame, column_number: int)*

This function is supposed to return the maximum sentence length for all sentences in the specified column (specified by column_number) for the given data frame. When the function will be called this column number will indicate either the “statement” column or the “justification” column of the LIAR-PLUS dataset which contains the sentences corresponding to the

The length of a sentence is defined as the length of the list of tokens returned by nltk.tokenize.word_tokenize applied on the sentence after lowercasing.

II. *create_glove_dict(path_to_text: str)*

This function reads the (text) file at the path which contains the glove embeddings for words. The format of the file is: every line in the text file has a word followed by a fixed-length (200) integer vector which is the static word embedding for the word. The function should finally return a dictionary for which the key is the word and the value is its corresponding word vector. Eg of return value: {"ball": [1,2,3], "bat": [7,8,9] ...}

2. Datasets.py:

Beware of NaN values, drop them.

Non-evaluative: __init__(), __len__(), get_max_lenghts(), get_Data_shape().

Note: You can (and should) use the variables that have been defined in the _init_() for writing the evaluative functions. So go through the _init_ function carefully and understand what each line of code is defining.

Following are the definitions of the functions you need to implement for evaluation:

I. ***__getitem__****(self, idx)*

This function gets the data from a specific row in the dataframe (specified by idx) and processes and finally converts it to torch tensors which will be usable by the neural model.

The steps to be followed for this processing are as follows:

1. Get the list of tokens for strings corresponding to statement and justification columns of the data after lower-casing (note- use word_tokenize from nltk).

2. get the label for the row and convert it to a torch tensor.

3. Initialise two numpy matrices to zeros. These will be later populated with the word vectors. The dimensions should be embedding_dimension x max_len of all (statement/justification) [Note: use variables from init to get appropriate dimensions]. The matrix will eventually be such that each column represents a word (by its vector). For eg: "ball bat" would be represented as:

```
[       1       7
        2       8
        3       9       ]
```

4. Traverse the list of tokens of statement obtained in step 1 and populate the initialised matrix with word vectors for each word (obtained from the glove_dict) at its appropriate position such that the output is how it has been defined in step 3. In case a word is not present in the glove embeddings, leave the column as zeros.

5. Repeat the same process to obtain vectorized justifications.

6. Convert the vectorized statement and justification to torch tensors.

7. Create a torch tensor for credit history. Essentially an array of size 5 with the counts in order - mostly_true, half_true, barely_true, false, pants_on_fire.

8. Return a dictionary with all the torch tensors. Essentially the keys and values of this dictionary should be:

```
{
        "statement": <torch_tensor_of_vectorised_statement>
        "justification": <torch_tensor_of_vectorised_statement>
        "label": <torch_tensor_for_label>
        "credit_history": <torch_tensor_for_credit_history>
}
```

3. ConvS2S.py

Non-evaluative: __init__()

Following are the definitions of the functions you need to implement for evaluation:

I. get_convolutions(self, input_dim, num_layers=3)

Returns a torch.nn.ModuleList() object which has num_layers of torch.nn.Conv1d(..) layers stacked (appended). For each Conv1d layer, input channels are equal to input_dim, the output channels is twice the number of input channels, kernel size is 3, stride is 1 and padding is 1.

II. forward(self, source)

Forward function for the layer. For each convolution in the stacked convolutions, apply conv() on the source, followed by GLU activation (grated linear unit - available in torch.nn.functional) with dimension 1, add the output of this activation to the input source, let's call this obtained_outpout. Finally, the input source for the next convolution should be equal to the above obtained_output.

4. Attention.py

Non-evaluative: __init__() of both the MultiheadAttention and PositionFeedForward classes

Following are the definitions of the functions you need to implement for evaluation:

I. *PositionFeedForward.get_layers(self, hid_dim, feedForward_dim)*

Returns [conv1, conv2]. conv1 is a torch.nn.Conv1d object with input channels equal to hidden dimensions, output channels equal to feed_forward_dimension, kernel size as 3, and stride and padding is 1. conv2 is a torch.nn.Conv1d object with input channels equal to feed_forward_dimension dimensions, output channels equal to hidden dimensions, kernel size as 3, and stride and padding is 1. Check _init_() function to see in which variables these layers are stored.

II. PositionFeedForward.forward(self, x)

save the value of the input (x) in a new variable, this is the residual. apply conv1 followed by conv2 on x (conv1 and conv2 are defined in _init_()). Add the resultant output to the residual to get the final output. Finally, the return value is a layer_normalised version of this final output, the return statement is already written for you (final output has been stored in 'x').

III. MultiHeadAttention.get_layers(self, hid_dim)

Returns 4 different conv1d layers in order - conv_query, conv_key,conv_value, and conv_output. conv_query, con_key, conv_value is a torch.nn.Conv1d object with input and output channels equal to hidden dimensions, kernel size as 3, and stride and padding is 1. conv_output is the same as the three previous layers, except padding is false (0). Initialise these objects and return them in the order as mentioned above. Check _init_() function to see in which variables these layers are stored.

5. Encoder.py

Evaluative: __init__()

Instantiate the required Classes in the __init__() method. The classes are ConvEncoder, MultiheadAttention and PositionFeedForward.

Following are the definitions of the functions you need to implement for evaluation:

I. forward(self, input)

On input first apply convolutional encoder, followed by MultiHeadAttention (query, key and value are equal to output of the previous step) and finally the PositionFeedForward layer. Use appropriate functions as defined in _init_() with appropriate dimensions. Return the final output from the feed-forward layer.

6. LiarLiar.py

Non-evaluative: __init__()

Following are the definitions of the functions you need to implement for evaluation:

I. get_convolutions(self, input_dim, hidden_dim)

Returns in order - upsacle_conv, first_cov and flatten_conv. upscale_conv is a torch.nn.Conv1d object with input channels equal to input_dims and output channels equal to hidden dimensions, kernel size as 1, and stride as 1. first_cov is a torch.nn.Conv1d object with input channels equal to output channels of the previous layer and output channels equal to half of the input channels (integer division with 2), kernel size as 3, and stride and padding as 1. flatten is a torch.nn.Conv1d object with input channels equal to output channels of the previous layer and output channels equal to 1, kernel size as 5, and stride as 1 and padding as 2. Return these three convolutions in order as mentioned previously.

II. get_linear_layers (self, max_length_sentence)

Returns in order - linear1, linear2, bilinear, and classifier layers. linear1 is a Linear layer with input_features equal to maximum sentence length and output features is equal to integer division of input features by 4. linear2 s a Linear layer with input_features equal to output features of the previous linear layer and output features is equal to number of classes in the dataset. bilinear is a Bilinear layer with first input features equal to number of output features of linear2, second input features is equal to size of credit_history array from data, output_features equal to 12 and bias is True. classifier is a Linear layer with 12 input features and output features equal to number of classes. The Linear and Bilinear layer classes are available in torch.nn and have to be returned in the specified order.

III. forward(self, sentence, justification, credit_history)

This is the forward function for the entire architecture. Look into _init_() of this class to appropriately understand the variables and layers and use them in this function. Initial part of the function is written, write the rest of the function following the steps described as follows:

1. encode the sentence and justification using sentence_encoder and explanation_encoder respectively.

2. define attention_output as the output from the attention layer (self.attention()), with query as encoded_sentence and key and value both as encoded_justification.

3. apply positional feed forward layer on the attention_output.

4. apply first_conv followed by relu activation on the output of step 3

5. apply flattened convolution of the output of step 4.

6. Flatten the output from the last layer so that we can pass the output to a linear layer.

7. apply linear1, linear2 and bilinear layers in this order on the final output of step 6. For bilinear layer, the second input is the credit history.

8. apply the classifier layer on the output of step 7 and return the result as the final return value of the forward function.

7. trainer.py

This file has the trainer function for the model and is entirely evaluative. Function definition -

trainer(model, train_dataloader, val_dataloader, num_epochs, path_to_save='/home/atharva', checkpoint_path='/home/atharva', checkpoint=100, train_batch=1, test_batch=1, device='cuda:0').

You can edit the default values of path_to_save and checkpoint_path appropriately, or pass it when the function is called.

The training by default gets shifted to GPU. We DO NOT expect you to train the model or report the accuracy, but should you wish to do this, change the device as per your system configuration as instructed in the comments in this function. If you wish to test run it on your system and do not have a GPU, pass the `device` parameter as `cpu`.

Steps for this implementation:

1. Train the model on the train_dataloader. (Remember we are receiving dictionaries from our dataloader.)

2. Calculate the required metrics, that is the loss and accuracy for the training phase per epoch and store them in a list. Training accuracies to be stored in a list names training_acc, training loss to be stored in training_loss. Please do not change the name of these lists.

3. Calculate the aforementioned metrics on the validation dataloader. Store these metrics too, validation accuracy in val_acc and validation loss in val_loss. Please do not change the name of these lists.

4. Save your model at the maximum validation accuracy obtained till the latest epoch.

5. Checkpoint at the 100th epoch

Note - if you want to shift the tensors to the mentioned device parameter, use the “to” method. For example if you have a tensor “torch_tensor” and want to shift it to the GPU or the CPU do the following- torch_tensor = torch_tensor.to(“device”) where the “device” is the one which you passed to the trainer function(the device parameter.)

7. main.py

All classes have to be instantiated here. Following are the definitions. Keep the variable names of the objects same as mentioned here (and the module_list defined in the main.py).

1. liar_dataset_train and liar_dataset_val defined as datasets.dataset() with appropriate value in prep_data_from argument to prepare data. sentence and justification length are both defined as liar_dataset_train.get_max_length(). Instantiate dataloader_train and dataloader_val on train and val dataset (suggested batch size 50 and 25 respectively).

2. statement_encoder and justification_encoder defined as instances of Encoder class with hidden_dims = 512 and conv_layers = 5.

3. multiHeadAttention and positionFeedForward are instances of the respective classes. hidden_dims in both as 512, number of heads in multiHeadAttention is 32 and feedForwardDims in positonFeedForward is 2048.

4. model is an instance of arePantsOnFire class with all arguments populated appropriately from the variables defined in the above steps and hidden_dims = 512

5. write a call to the trainer function with appropriate arguments. (number of epochs - 1, train_batch = 1, test_batch = 1)

6. define liar_data_test as datasets.dataset with test data and test_dataloader on this dataset with batch_size = 1. DO NOT PASS THE num_workers parameter.

7. write a function call to the infer function from utils.

NOTE: We do not expect you to train the model completely or report the accuracies, but we will test the trainer function and the main.py to see if everything is called properly.

PS - If you run to see if everything is working fine or not, but run out of memory, decrease the dimension and the number of attention heads or number of layers so that they consume lesser memory. But the file you submit, we expect you instantiate the classes with the aforementioned parameters.

PART 2

Aim: To help students understand the working of Particle Swarm Optimization and its basic implementation in python.

Background: What is Particle Swarm Optimization?

It's a powerful evolutionary optimization algorithm, known for its utility and simpleness. If there's a predefined solution space and a function to optimise, it gives really good results.

Working:

Given a cost function to optimise, we try to search for the best possible solution using M particles, where M is known as the swarm size.

The position of a particle depends on:

1. Its own current velocity
2. Its own best position
3. The swarm's best global position

The fitness of a particle is based on its cost at its current position. In an iterative manner, the swarm particles are made to converge towards a more optimal solution, to get the global best position.

Instructions:

1. There are 6 functions to be written in this section, all of which are included in the file pso.py
2. DO NOT import any other module, or change the function prototypes
3. The sample test cases have been given in pso_sample_tests.py. Uncomment the functions that you want to test.

Brief description of functions: (Complete details in the file pso.py in comments)

1. **cost_function(X)**: Takes a list of values and calculates its cost as defined in the comments
2. **initialise(length):** Takes the length of list to be optimised. Initialises velocity, position, best position and best cost
3. **assess(position, best_position, best_cost, func):** Evaluates the position based on the func parameter, which is the cost function. Updates best position list if a better cost is achieved and returns the better cost.
4. **velocity_update(w, c1, c2, velocity, position, best_position, best_group_position):** Updates the velocity based on the formula given in comments(current + cognitive + social)
5. **position_update(position, velocity, limits):** Updates the position by adding velocity. Position has to lie between the limits(min, max) given.
6. **optimise(vector_length, swarm_size, w, c1, c2, limits, max_iterations, initial_best_group_position=[], initial_best_group_cost=-1):** Uses all the above functions. First initialises a swarm, each particle having a position, velocity, best_position([ ]), and best_cost. Runs for max_iterations. In each iteration, evaluates the cost of each particle and updates the group best position accordingly. At the end of each iteration, it updates first the velocity and then the position of each particle.

Submission Instructions:

- All the completed python files WITHOUT CHANGING ORIGINAL NAMES of the Files should be kept in a single folder.
- DO NOT include the glove file or data files in this folder.
- The name of this folder should be your id number as: “2017A7PS0040P” and zipped as .zip file (Do not use other zip formats).
- Follow the submission instructions very carefully, otherwise the code may not be appropriately evaluated. We WILL NOT CONSIDER RECHECKS if code is not evaluated due to improper submission, changed name files or changed function prototypes etc.