Assignment 4

COMP-599 and LING-782/484

For this assignment, you will need to complete different sections depending on your enrollment.

COMP-599: Complete everything
LING-484/782: Complete section 1 and 2

Please choose the correct option in Gradescope when submitting, or you may be penalized. The difference is due to estimated workload for students enrolled in the 3-credit version of this course (LING) vs 4-credit (COMP). Students taking the 3-credit version are welcome to complete section 3 for practice as the difficulty should be similar to the rest of the assignment, but it will not be graded nor will there be bonus marks.

1. Finetune DistilBERT for classification (40 pts)

In this part, you will use the NLI training data (same as A1) to finetune a DistilBERT model and predict whether a premise entails a hypothesis or not. Just like the first assignment, you will have to implement various parts of a custom nn.Module that loads a **pretrained** DistilBERT from Huggingface transformers. You can learn more about DistilBERT here, but you can just assume it's a smaller version of BERT that remains fairly accurate.

1.1 Initialize CustomDistilBert (15 pts)

You will have to implement the init function of the CustomDistilBert class. You will need to initialize the following attributes:

- self.distilbert
- self.tokenizer
- self.pred_layer
- self.sigmoid
- self.criterion

For distilbert and tokenizer, you will need to use `transformers`, whereas pred_layer, sigmoid, and criterion require `torch` and correspond to questions you have previously answered in A1.

CustomDistilBert.__init__

Note:

- Load the DistilBERT model's pretrained "base uncased" weights from the Huggingface repository. We want the bare encoder outputting hidden-states without any specific head on top.
- Load the corresponding pre-trained tokenizer using the same method.

- self.pred_layer takes the output of the model and predicts a single score (binary, 1 or 0), then pass the output to the sigmoid layer
- self.sigmoid should return torch's sigmoid activation.
- self.criterion should be the binary cross-entropy loss. You may use torch.nn here.

Please take a look at the docs.

CustomDistilBert.assign_optimizer

This assigns the Adam optimizer to this model's parameters (self) and returns the optimizer.

Parameter	Type	Description
**kwargs		The arguments passed to the optimizer.

Returns	Description
torch optimizer	An Adam optimizer *bound* to the model (think of it as being aware of the model's parameters being changed)

1.2 Slicing CLS's hidden state (5 pts)

Edit the method ***CustomDistilBert.slice_cls_hidden_state***. This is a helper method that will be used inside forward, and will convert the output of your transformer model to something that can be input in the prediction layer.

CustomDistilBert.slice_cls_hidden_state

Using the output of the model, return the last hidden state of the CLS token.

Parameter	Type	Description
x	BaseModelOutput	The output of the distilbert model. You need to retrieve the hidden state of the last output layer, then slice it to obtain the hidden representation. The last hidden state has shape: [batch_size, sequence_length, hidden_size]

Returns	Description
Tensor[batch_size, hidden_size]	The last layer's hidden state representing the [CLS] token. Usually, CLS is the first token in the sequence.

1.3 Custom tokenize function (5 pts)

Use the get_tokenizer function implemented in 2.1 to write the method ***CustomDistilBert.tokenize***. That method is specifically to help you understand how the tokenizer works, and should be fairly straightforward.

CustomDistilBert.tokenize

This function will be applied to the premise and hypothesis (list of str) to obtain the inputs for your model. You will need to use the Huggingface tokenizer returned by get_tokenizer().

Parameter	**Type**	**Description**
premise	list of str	The first text to be input in your model.
hypothesis	list of str	The second text to be input in your model.

For the remaining params, see documentations.

Returns	**Description**
BatchEncoding	A dictionary-like object that can be given to the model (you can find out how by reading the docs)

1.4 Forward pass (10 pts)

Given the output of your tokenizer (a BatchEncoding object), you will have to pass through your custom DistilBert model and output a score between 0 and 1 for each element in your batch; this score represents whether there's an entailment or not.

CustomDistilBert.forward

Note: In the original BERT paper, the output representation of CLS is used for classification. You will need to slice the output of your DistilBERT to obtain the representation before giving it to the last layer with sigmoid activation.

Parameter	**Type**	**Description**
inputs	BatchEncoding	The input ingested by our model. Output of tokenizer for a given batch

Returns	**Description**
Tensor[batch_size]	The output prediction for each element in the batch, with sigmoid

	activation. Make sure the shape is not [batch_size, 1]

1.5 Train your model and report results (5 pts)

You will need to answer this part on Gradescope in “A4 (Report)”. You will need to use the code that has been given to you at the end of the script, and run that to train the model. It will use the methods of CustomDistilBert you have written in 1.1-1.4, and evaluate it on the F1 score.

For the remaining instructions, please see Gradescope.

2. Prompt Tuning/Engineering (15 pts)

Following Lester et al (2021), please implement their prompt tuning method (they call it "soft prompts"). Please read relevant parts of the paper before starting this section.

2.1 Freezing parameters (1 pts)

freeze_params

Before starting, you will need to freeze all the parameters (including the embedding!). This is because prompt tuning relies on tuning a very small number of fixed parameters (aka "prompts", since they are inserted as input embeddings to the model). Thus, everything else, including the input word embeddings, are not trainable.

2.2 Implement prompt tuning (4 pts)

pad_attention_mask

Pad the start of the sequence p times of the `attention_mask` because the sequence length has changed. Find the correct value based on Huggingface documentations.

SoftPrompting.__init__

Parameter	**Type**	**Description**
p	int	The length of the prompt, which is a hyperparameter in the paper.
e	int	The embedding dimension, corresponds to E

SoftPrompting.forward

This takes the output of model.embeddings and adds the soft prompts, as described in the paper. The prompts must be added at the start of the sequence.

Parameter	**Type**	**Description**
embedded	Tensor[B, L, E]	This corresponds to model.embeddings (where model is a Huggingface transformer) ● B: Batch size ● L: Sequence Length ● E: Embedding dimension (same as `e`)

Returns	Description
Tensor[B, L+p, E]	The `input_embed` to be given to the model, but with the added

2.3 Prompt tuning DistilBERT for NLI classification (5 pts)

Use your previous training loop for distilbert, but this time use prompt tuning, so you might need to make modifications.

Please see Gradescope to see how to report your results.

2.4 Engineering prompts to use GPT-3 for classification (5 pts)

Now, you will need to create an account on openai.com, and use the playground to generate prompts. When you create an account, you will be given free credits for 3 months, but they are limited, so make sure you don’t run out before completing the assignment. Moreover, different sizes are available, so please read the documentation on that.

Please see Gradescope to see how to report your results.

3. Open-domain question answering (45 pts)

In this part, you will need to implement a model similar to Dense Passage Retrieval for Open-Domain Question Answering by *Karpukhin et al (2020)*. Before starting, please go over the paper, and optionally read this blog post section on open-domain QA and skim over the DPR code repository.

For the assignment, you are given CSV files in data/qa that correspond to the training Q&A pairs, validation pairs, test questions, and answers.csv that contains all the answers. The data is taken from a Q&A forum about cooking, and at the end of this section you will have built a model that can automatically retrieve (i.e. search) relevant answers for any question about cooking.

3.1 Load model and tokenizers (5 pts)

In part 1, you used an object-oriented approach to load a custom distilbert model. This time, you need to load two models (one for encoding the questions, another for encoding the candidate answers, and they have separate weights but same embedding dimensions). You will use a simpler approach this time; simply build functions that load and return the models, perform slicing, and generate tokens, all separate from OOP.

load_models_and_tokenizer

For this, we will be testing with the string `'google/electra-small-discriminator'` but your function should work with other names as well. You do not review ELECTRA, other than knowing that this is a pretrained model that performs well but is much smaller than DistilBERT and has smaller embedding size. You should be comfortable loading other models from Huggingface, as hundreds of them exist.

Parameter	**Type**	**Description**
q_name	str	Name of the model that will be passed to `transformers` to **auto**matically load the pre-trained version, and used for encoding questions. It must be something that can be found on Huggingface Hub, just like you've previously done.
a_name	str	Name of the model encoding the answers.
t_name	str	Name of the tokenizer, which will be passed to `transformers` to **auto**matically load the corresponding pre-trained tokenizer.
device	str	"cuda" or "cpu", the models will be loaded here.

Returns	**Type**	**Description**

q_enc	`transformers` Model	The question encoder model, which is a huggingface transformer. This corresponds to q_name
a_enc	`transformers` Model	The candidate answers the encoder model, which is a huggingface transformer. This corresponds to a_name
tokenizer	`transformers` tokenizer	The tokenizer that will be used for each of the models. For simplicity, it will be shared, but in theory you could have separate tokenizers if you use different models for encoding questions and answers, but this is rare in practice.

3.2 Tokenize batch and get class output (7 pts)

tokenize_qa_batch

Tokenize question titles/bodies and answers into two input batches. The following conditions apply:

- The input_ids returned should be Pytorch tensors
- The content should be truncated if it exceeds max_length
- You should only pad everything to the longest sequence in a batch
- The title and body should be tokenized together as a pair, separated with a [SEP] token.
- The answers should be tokenized separately
- q_titles,q_bodies, answers are all lists of the same length

Parameter	**Type**	**Description**
tokenizer	`transformers` tokenizer	A huggingface tokenizer returned by your previous function.
q_titles	list of str	The list of titles of the questions
q_bodies	list of str	The list of questions bodies (actual content)
answers	list of str	The contents of the corresponding answers
max_length	int	The maximum length of the tokens, after which it is truncated

Returns	**Type**	**Description**
q_batch	BatchEncoding	A "BatchEncoding" (inherited from a dict) containing the question titles and bodies, which can be used as the input of a transformer model (which means the input IDs are PyTorch tensors)

a_batch	BatchEncoding	A "BatchEncoding" (inherited from a dict) containing the answer bodies, which can be used as the input of a transformer model.

get_class_output (ungraded)

Since this is similar to a previous question, it is left ungraded.

Returns	**Description**
BaseModelOutput	The output representation of the class token (for example [CLS]) after encoding the tokenized text through your model.

3.3 Batch: Implement in-batch negative sampling (7 pts)

The in-batch negative sampling method uses the answers from other questions in the same batch as the negative examples (because they are unrelated as they are randomly taken from the training set). Please read the relevant section in the paper and implement it accordingly.

inbatch_negative_sampling

This function should take the tensors of questions Q and passages P, and use the in-batch negatives (as described in the paper) to compute a similarity matrix evaluated on each of the N questions with the M passages. Although we call it “sampling”, you are computing it over all passages in a batch rather than taking a subsample of the batch. You can find a way to do both the “negative sampling” and computing similarity at the same time; you can read the paper to find out how to do that.

Parameter	**Type**	**Description**
Q	Tensor[N, E]	The output representation of N question titles+bodies
P	Tensor[M, E]	The output representation of M answers (aka passages)
device	str	“cuda” or “cpu”, the models will be loaded here.

Returns	**Description**
Tensor[N, M]	The matrix of similarity score that results from in-batch negative sampling.

3.4 Implement contrastive loss (7 pts)

Contrastive loss is different from the loss functions we have previously been exposed to. You have previously seen loss functions for classification and for text generation, but retrieval requires something different. You will need to implement the loss function described in the DPR paper (please read the equations carefully), and the official source code shows how to implement it in Pytorch.

contrastive_loss_criterion

Parameter	Type	Description
S	Tensor[N, M]	The matrix of similarity score that results from in-batch negative sampling.
labels	Tensor[N]	The optional label indices between 0-M. For example [0, 2, M,..., 1] respectively indicate that the ● Passage #0 is the answer for Question #0 ● Passage #2 is the answer for Question #1 ● Passage #M is the answer for Question #2 ● … ● Passage #1 is the answer for Question #N If labels=None, simply return a tensor with values such at Passage #0 corresponds to Question #0, P1 with Q1, etc.
device	str	“cuda” or “cpu”, the models will be loaded here.

Returns	Description
Tensor(1)	A scalar tensor on which you can call backward to initiate the backprop process. The value represents the loss, which means a lower value means the model has a low error when matching the questions with the passages.

3.5 Implement functions to run retrieval on list of passages (7 pts)

get_topk_indices

Compute the dot-product similarity score (without normalizing or cosine scaling) and return the indices and scores for the top-k candidate answers for each question in Q.

Parameter	Type	Description
Q	Tensor[N]	The output class representation for question title+body
P	Tensor[M]	The output class representation for answer (aka passage)
k	int	The number of indices to return based on the similarity.

		When k=None, simply return everything in the sorted order.

Returns	**Type**	**Description**
indices	Tensor[N, k] of int	The sorted indices of the most similar answers for each of N questions (from largest to smallest magnitude)
scores	Tensor[N, k] of int	The dot-product similarity score for the corresponding indices

select_by_indices

Parameter	**Type**	**Description**
indices	Tensor[N, k] of int	The sorted indices of the most similar answers for each of N questions (from largest to smallest magnitude)
passages	list of str	A list of answers in the original textual format (before tokenization). The length should be greater than the largest index in indices (i.e. M).

Returns	**Description**
list of list of str	● The outer lists correspond to answers for the the N questions ● The inner lists contain the k sorted answers in original text format

embed_passages

First, set the model into the evaluation mode and disable gradients, then embed a list of passages.

Parameter	**Type**	**Description**
passages	list of str	A list of answers in the original textual format (before tokenization). The length should be greater than the largest index in indices (i.e. M).
model		The Huggingface transformer model used to encode the text
tokenizer		The Huggingface tokenizer used to encode the passages

Returns	**Description**

Tensor[N]	The output class representation for answer (aka passage)

embed_questions

For embed_questions, it's very similar except you now to embed titles and bodies at the same time (separated by a sep token, e.g. "<title> [SEP] <body>". Keep that you are using a different model too.

3.6 Implement evaluation metrics (7 pts)

recall_at_k

You can review the recall score here. Note that since we are returning k elements, we want to know if the correct answer is in one of the k elements, hence we are calling this the "recall" score, though it might feel different from how we use recall for binary classification.

Parameter	**Type**	**Description**
retrieved_indices	list of list of int	outer list: The retrieved results for each of N questions inner list: The k indices ranked in order of estimated relevance
true_indices	list of int	The correct index for each of N questions, same length as retrieved_indices's outer list
k	int	The number of inner items in retrieved_indices to consider; k must be smaller than the length of the inner lists

Returns	**Description**
float	A single score representing the recall at k score

mean_reciprocal_rank

Please refer to `recall_at_k`, except you are now returning the Mean Reciprocal Rank, and there's no parameter k. You can read more about this on Wikipedia.

3.7 Train model and report results (5 pts)

You now have all the different functions needed to build your own retrieval model! It is now time to train your model on the actual task, and report your results in the validation set. Note that this is a bit different since the answers.csv file contains all answers, including ones that might be associated with the test set. Also, it means that you might not be able to evaluate over all the answers after every epoch, so your evaluation metric might not necessarily give you the right results.

Please see Gradescope to see how to report your results.