Assignment 1

COMP-545 and LING-545

Overview

The purpose of this assignment is to help you become familiar with building ML models from start to finish using Pytorch. You will be given a Natural Language Inference dataset with binary labels (0 for entailment, 1 for no entailment). You will have to handle various preprocessing steps (batching, shuffling, converting to tensor), then design and train various types of neural networks to accurately predict the labels. Once trained, you can evaluate your models on the validation split using the F1 score.

Natural Language Inference (NLI): In this task, you are given a sentence called the “premise”, and you want to predict if another sentence, the “hypothesis”, either *entails* or *does not entail*. Saying that the premise entails a hypothesis means that, if I read the premise, then I would infer that the hypothesis is true. For example, if my premise is “*Two women are embracing while holding to go packages*”, then it would **entail** the hypothesis that “*Two women are holding packages*”, but **does not entail** the hypothesis that “*The men are fighting outside a deli*”.

Points breakdown: The assignment will be broken into the following parts:

1. Write functions for data processing, batching, text-to-vector embedding **(30 points)**
2. Design a baseline pytorch model, select an optimizer and train it **(50 points)**
3. Experiment with more sophisticated models **(20 points)**
4. Report your results (ungraded, see instructions on gradescope).

Grading: We will use automatic grading via Gradescope. You will have to sign up using your McGill email, Student ID, and the course code provided in MyCourses’ content tab. You will have to write your code in the provided ***code.py*** file before uploading it (make sure the file name remains the same!).

Submission: Make sure to not change the function names or parameters as they will be needed in order to automatically evaluate your code. For each question, you will be given function signatures for which you will have to fill in the blanks. If you have supplementary code (e.g. to test or train your model), please move them to the ***if __name__ == “__main__”*** scope (see at the end of the provided python file) or define them with new function names. This is important because if your file takes a long time to run because it executes code, then your code might not be correctly graded.

Pytorch: This assignment uses Pytorch. To install it, run: `pip3 install torch==1.10.*`. Pytorch is a library for building neural networks and training them using backpropagation. It contains useful functionality like [autograd](). If you have never used Pytorch before, please go over the [official quickstart tutorial]() before starting this assignment.

Compute: It is possible to complete this assignment on your personal computer, even without access to a GPU. You will need to use Python 3.7 or Python 3.8. You will also need to install Pytorch by running
Moreover, you can also complete this assignment without any local setup while accelerating your training code with free GPU access through Kaggle. For alternatives, please look at *Compute Resources* tab in MyCourses's contents.

Part 1

In this section, you will implement a series of functions that will be useful for converting the initial data into batches of torch tensors. To help you get started, you are given a few helper functions to load the data, apply tokenization and convert to indices (e.g. "hello" → 5, where 5 is the index where the embedding for "hello" is located). For more information on how to use them, look at the docstring or the starter code immediately after ***if __name__ == "__main__"***.

1.1 *build_loader* - Batching, shuffling, iteration [20]

The first task will be to handle the process of creating batches, shuffling the training set, and being able to iterate through the entire dataset. Note that the dictionary contains the data for an entire split, whereas the loader enables access to a single batch (which might or might not be shuffled, and might have an arbitrary size) in an iterative fashion.

build_loader(...) specifies what type of loader you want, and the output is itself a function that, when called, returns a *generator*. You can iterate over that *generator* to get a batch of data, which is a dictionary with the same keys, and values are lists of length batch_size (the last batch may be shorter since you only need to include the remaining samples). Do not use Pytorch's data loader.

Notes:

- It's possible to implement this function such that data_dict could have arbitrary keys as long as they are all lists of the same length.
- You should have the option to shuffle the inputs before starting.
- Do not use Pytorch's data loader

Parameter	**Type**	**Description**
data_dict	dict	A dictionary with keys 'premise', 'hypothesis', and potentially 'label', all of which are lists of the same length.
batch_size	int, optional	The size of the batch. The length of the list in the batch yield by loader will be equal to batch_size, except for the last batch, which may be shorter (since it contains the remaining samples).

shuffle	bool, optional	Whether to shuffle the dataset. When shuffle=True, then every time you iterate through the loader, the batches will contain different samples. This means the order of the training set is randomized every time you call `for batch in loader()`. At the end of the loop, all the data in the training set must have appeared exactly once in a randomized order.

Returned	Description
function	The “loader” is a generator function that, when iterated with a for loop, yields a dictionary with the same keys as `data_dict`, but with values of length corresponding to `batch_size` (or, in the case of the last batch, can be shorter if it has fewer samples).

Example

```
>>> # let's assume 200 samples
>>> num_epochs = 1
>>> loader = build_loader(data)  # only do this once in entire
training
>>> for x in num_epochs:
>>>     # This will have different order each time
>>>     for batch in loader():
...         premise = batch['premise']
...         label_batch = batch['label']
...         # do something with batch here
...         print(len(premise))
64
64
64
8
```

1.2 *convert_to_tensors* - Converting a batch into inputs [10]

From above, we now have a batch that we can use when iterating through our loader. However, the batch is a nested list. We now want to convert that into a torch tensor of integers (representing the indices), which will require us to pad it with 0’s. The function you design here will be applied to the premise, or to the hypothesis, but not to the label (you can handle that easily using existing torch functions). Edit the function named ***convert_to_tensors.***

Parameter	Type	Description
text_indices	list of list of int	A list of token indices, which can be either the premise or hypothesis from a batch yield by loader().

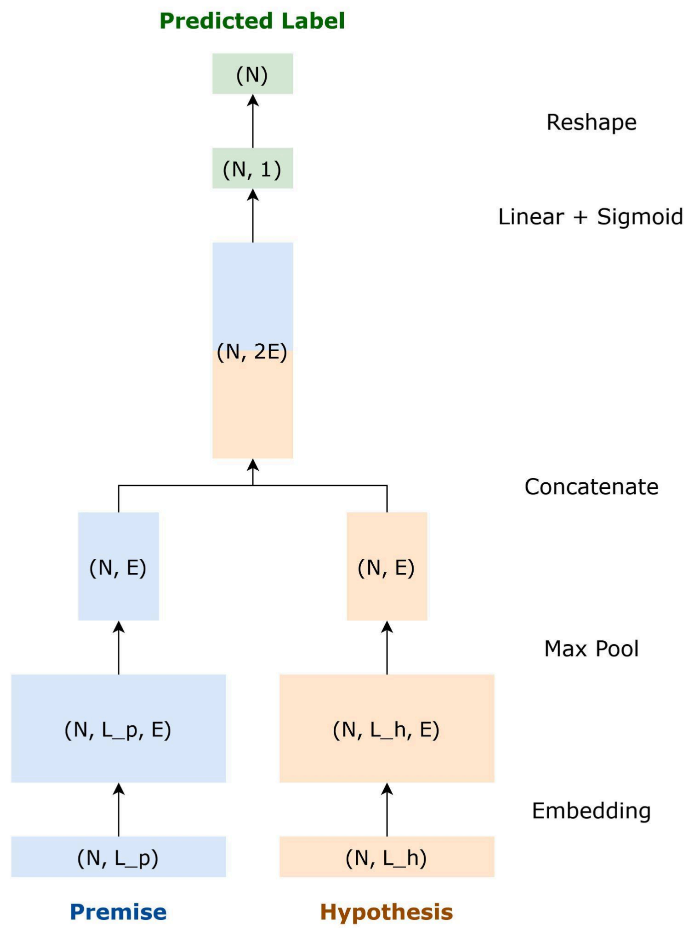
Returns	Description
Tensor of int32	A Torch tensor of shape (N, L) where L is the length of the longest inner list, and N is the length of the outer list.

Part 2

In this section, you will implement the full training procedure. First, you will need to design a very simple neural network. Afterward, you will need to handle various aspects of training: loss function, optimizer, evaluation metric and training loop.

2.1 Design a logistic model with embedding and pooling [20]

Use Pytorch's nn.Module to build a simple logistic regression model (in other words, a feed-forward layer with a single output between 0 and 1). Although the architecture will be simple, there are a few steps involved: activation, concatenation and pooling. Edit the function named ***max_pool*** and class named ***PooledLogisticRegression.*** Here's a diagram (N is batch size, L is sequence length, E is embedding dimension)



max_pool

Take the max pooling over the second dimension, i.e. a (N, L, D) -> (N, D) transformation where D is the `hidden_size`, N is the batch size, L is the sequence length.

PooledLogisticRegression

When called this simple linear model will do the following:

1. Individually embed a batch of premise and hypothesis (token indices)
2. Individually apply max_pool along the sequence length (L_p and L_h)
3. Concatenate the pooled tensors into a single tensor
4. Apply the logistic regression to obtain prediction (aka `layer_pred` in the code)

Parameter	**Type**	**Description**
embedding	nn.Embedding	The embedding layer you created using the size of the word index. You can create it outside of this module. The transformation is (N, L) -> (N, L, E) where E is the initial embedding dimension, and L is the

		sequence length.

PooledLogisticRegression.forward(...)

Parameter	Type	Description
premise	torch.Tensor[N, L_p]	The premise tensor, where L_p is the premise sequence length and N is the batch size.
hypothesis	torch.Tensor[N, L_h]	The hypothesis tensor, where L_h is the hypothesis sequence length.

Returns	Description
torch.Tensor[N]	The predicted score for each example in the batch.

Note that the returned tensor is of shape N, not (N, 1). You will need to reshape your tensor to get the correct format.

2.2 Choose an optimizer and a loss function [5]

There are many torch optimizers you can use from torch.optim; you can start with SGD or something else you prefer. Make sure you have applied the optimizer to your model. As for loss, you will have to implement binary-cross entropy using basic torch math functions (without using torch's BCE loss). Edit the functions named ***assign_optimizer*** and ***bce_loss.***

assign_optimizer

Parameter	Type	Description
model	nn.Module	The model to optimize.
**kwargs	dict	The keyword arguments that will be passed to the optimizer (a ** performs packing and unpacking). This will vary depending on the optimizer, but the most common one is `lr`.

Returned	Description
torch.optim.Optimizer	The optimizer that you will use during the model training.

There's many optimizers in PyTorch. You can start with SGD, but it's recommended to try other popular options.

bce_loss

The binary cross entropy loss, implemented from scratch using torch Do not use torch.nn, but you may compare your implementation against the official one.

Parameter	**Type**	**Description**
y	torch.Tensor[N]	The true labels
y_pred	torch.Tensor[N]	The predicted labels

Returned	**Description**
torch.Tensor	The binary cross entropy loss (averaged over N).

2.3 Forward and backward pass [10]

Implement a function that performs one step of the training process. It should take in your network, a batch, an optimizer, and make sure that the loss is back-propagated, the weights are updated by your optimizer, and the gradients are cleared at the end of the step. Edit the functions named ***forward_pass*** and ***backward_pass.***

forward_pass

Implement a function that performs one step of the training process. Given a batch and a model, this function should handle the text to tensor conversion and pass it in a model.

Parameter	**Type**	**Description**
model	nn.Module	The model you will use to perform the forward pass.
batch	dict of list	A dictionary with 'premise' and 'hypothesis' keys (lists of same size).
device	str	The device you want to run the model on. This is usually 'cpu' or 'cuda'.

Returned	**Description**
torch.Tensor	The y value predicted by the model.

backward_pass

This function takes in the optimizer, the true labels, and the predicted labels, then computes the loss and performs a backward pass before updating the weights.

Parameter	Type	Description
optimizer	optim.Optimizer	The optimizer you will use to perform the backward pass.
y	torch.Tensor[N]	The true labels.
y_pred	torch.Tensor[N]	The predicted labels.

Returned	Description
torch.Tensor	The loss value computed with bce_loss()

2.4 Evaluation [5]

Implement F1 scoring from scratch with torch operations (do not use external F1 implementation) to evaluate the performance of your model on the validation split. Make sure to set the network on evaluation mode and not to backpropagate gradients. Edit the function named ***f1_score.***

f1_score

Apply the threshold (if it is not None), then compute the F1 score from scratch (without using external libraries).

Parameter	Type	Description
y	torch.Tensor[N]	The true labels.
y_pred	torch.Tensor[N]	The predicted labels.
threshold	float, default 0.5	The threshold to use to convert the predicted labels to binary. If set to None, y_pred will not be thresholded (in this case, we assume y_pred is already binary).

Returned	Description
torch.Tensor[1]	The F1 score.

2.5 Training loop [10]

Create a complete training loop using everything from above. For every epoch, it will train your network on each batch until you have made an entire pass through the training set. At the end of the epoch, you will evaluate your model on both the training and validation sets with the `eval_run` function (without computing the gradients!) and return the validation F1 score; you can print the score as you train. You might want to save the results to disk so you can review them later! Edit the functions named `eval_run` and `train_loop`.

`eval_run`

Iterate through a loader and predict the labels for each example, all while collecting the original labels.
Note: You can use the `forward_pass` function to get the predicted labels. Don't forget to disable the gradients for the model and to turn your model into evaluation mode.

Parameter	**Type**	**Description**
model	nn.Module	The model you will use to perform the forward pass.
loader	function	The loader function that will yield batches.
device	str	The device you want to run the model on. This is usually 'cpu' or 'cuda'.

Returned	**Type**	**Description**
y_true	Tensor[D] of float	The true labels in float form (either 0.0 or 1.0) extracted from the loader. D refers to total dataset size.
y_pred	Tensor[D] of float	The output score between 0.0 and 1.0 predicted by the model.

`train_loop`

Train a model for a given number of epochs.

Note: This function is left open-ended and is strictly to help you train your model. You are free to implement what you think works best, as long as it runs on the training and validation data and return a list of validation score at the end of each epoch.

Parameter	**Type**	**Description**
model	nn.Module	The model you will use to perform the forward pass.
train_loader	function	The loader function that will yield shuffled batches of training data.
valid_loader	function	The loader function that will yield non-shuffled batches of validation data.
optimizer	optim.Optimizer	The optimizer you will use to perform the backward pass.
n_epochs	int	The number of epochs you want to train your model
device	str	The device you want to run the model on. This is usually 'cpu' or 'cuda'.

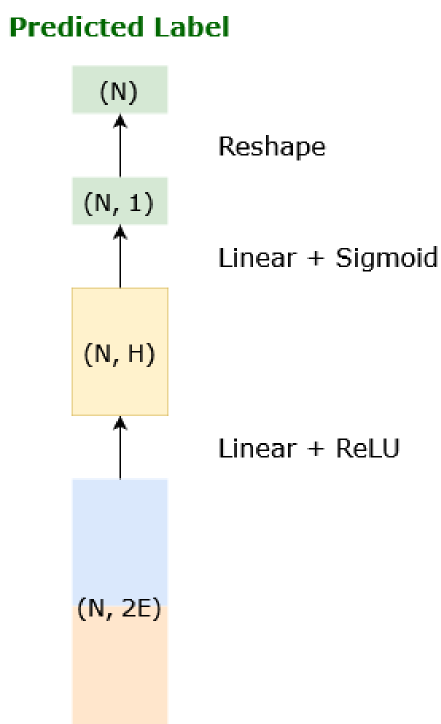
Returned	Description
list	A list of f1 scores evaluated on the valid_loader at the end of each epoch.

Part 3

Now that you have the full training procedure ready, you can experiment with different architectures! For each new architecture, you can train your model for a few epochs (it should be very fast if you run it on Kaggle or Colab).

3.1 Design a shallow neural network with activation [10]

This follows the same idea as the logistic model, but before passing your concatenated tensor (with shape N, 2E) to the logistic regression layer, you should add a single feed-forward layer, and apply a ReLU activation (see "Linear + ReLU" in the diagram below). Write your code inside the class named ***ShallowNeuralNetwork***. The diagram below is a truncated version of the previous diagram, with the added "Linear + ReLU" and a yellow block which represents what you will implement. H stands for the hidden size of the output of your feedforward (aka linear) layer.



ShallowNeuralNetwork

When called this simple linear model will do the following:

1. Individually embed a batch of premise and hypothesis (token indices)
2. Individually apply max_pool along the sequence length (L_p and L_h)
3. Concatenate the pooled tensors into a single tensor
4. Apply one feedforward layer to the tensor resulting from the concatenation
5. Use the ReLU on the outputs of your layer
6. Apply sigmoid layer to obtain prediction

Parameter	**Type**	**Description**
embedding	nn.Embedding	The embedding layer you created using the size of the word index.
hidden_size	int	The size of the hidden layer.

ShallowNeuralNetwork.forward

Parameter	**Type**	**Description**
premise	Tensor[N, L_p]	The premise tensor, where N is the batch size and L_p is the premise sequence length.
hypothesis	Tensor[N, L_h]	The hypothesis tensor, where L_h is the hypothesis sequence length.

Returned	**Description**
Tensor[N]	The scores for each example in the batch.

3.2 Create deeper networks [10]

Modify your model to accept an arbitrary number of layers. The ReLU activation will be applied after every intermediate layer. Write your code inside the class named ***DeepNeuralNetwork***.

DeepNeuralNetwork

When called this simple linear model will do the following:

1. Individually embed a batch of premise and hypothesis (token indices)
2. Individually apply max_pool along the sequence length (L_p and L_h)
3. Concatenate the pooled tensors into a single tensor
4. Apply one feedforward layer to the tensor resulting from the concatenation
5. Use the ReLU on the outputs of your layer, repeat (4) for `num_layers` times.
6. Apply sigmoid layer to obtain prediction

Note: You will need to use nn.ModuleList to track your layers.

Parameter	Type	Description
embedding	nn.Embedding	The embedding layer you created using the size of the of the word index. You can create it outside of this module. The transforma dimensions is (N, L) -> (N, L, E) where E is the initial embedding dimension, and L is the sequence length.
hidden_size	int	The size of the hidden layer.
num_layers	int, default 2	The number of hidden layers in your deep network. Each layer must be activated with ReLU.

DeepNeuralNetwork.forward

Parameter	Type	Description
premise	Tensor[N, L_p]	The premise tensor, where N is the batch size and L_p is the premise sequence length.
hypothesis	Tensor[N, L_h]	The hypothesis tensor, where L_h is the hypothesis sequence length.

Returned	Description
Tensor[N]	The scores for each example in the batch.