**Task 1:**

For task 1, BERT-base-uncased was chosen as it outputs sentence embeddings and is a popular model for fine tuning to perform downstream tasks. Furthermore, it is small enough to run easily on my laptop. Also, lowercase/uppercase distinctions are not important to me, leading me to go with the uncased model rather than BERT-base-cased.

Also, as you can see from the code, I implement CLS-pooling. I do this because BERT makes it very easy to pull this (simply retrieving index 0 of the last hidden state), and because it readily represents the contextual summary vector of the sentence.

I decide not to implement any normalization. Normalization seems unnecessary, given we aren’t using the embeddings for something like semantic search and given it is perfectly fine to feed raw embeddings into our later dense linear layer + softmax.

**Regarding the inputs and outputs:**

I input a batch of 4 sentences. The output is 4 different 768-length vectors. This makes sense as each token in a sequence is “converted” to a 768 length vector, and we are taking the 768-length vector associated with the CLS token.

For the "There is a stack of papers on the table.” input, Most of the values look centered around 0, in range of -1 to 1, and on the order of magnitude of 10^-1. These are expected for BERT.

Note that there are some outliers such as -3, 4, and -8. The presence of outliers isn’t necessarily atypical (especially if certain neurons fired more strongly due to certain semantic features of the sentence). Note that it is, of course, not possible to easily map specific values to meaning

For the "The largest mountain in the world is Mount Everest." input, like before, the values look centered around 0, in range of -1 to 1, and on the order of magnitude of 10^-1. Like before, there are some outliers.

**Task 2:**

For task 2a, I will be implementing domain classification for the following categories: history, geography, health, technology

For task 2b, I will be implementing sentence classification

For both, I attach a Linear (fully-connected) layer on top of the pooled CLS embedding to produce raw logits, and train with CrossEntropyLoss (which applies softmax internally).

The dropout layer helps with preventing overfitting, as 0’ing out some of the 768 values forces the weights to be more robust and less dependent on any 1 weight.

The linear layer is standard-practice for BERT-fine tuning, converges fast due to its small size, and cross entropy is the gold standard for multi class classification.

**Task 3:**

**Scenario 1:**

I would be doing no training of any weights, so my system would be purely for inference.

If I did scenario 1 using my randomly-initialized weights, I would receive nonsense outputs.

However, this methodology could work if I imported already-trained heads and attached these on. Furthermore, in this case, I would also need no compute for training, so my compute costs would be lower.

**Scenario 2:**

In this scenario, I would be freezing BERT’s layers during training and thus just updating the weights of the classification heads.

This approach could work, as out-of-the-box BERT has existing hidden representations that would be transferrable to our problem, and the training of the weights of the initially-random classification heads is the most important endeavor.

It would be nice to be able to tweak BERT’s weights (in an alternate scenario) though, as the overall system would likely have superior performance on the tasks.

This scenario would make sense if you are compute constrained (you are optimizing way fewer weights due to freezing the millions of weights in BERT), or if you have a small dataset of less than 100 examples per task (there wouldn’t be enough data to effectively tune both BERT and train the classification heads).

**Scenario 3:**

In this scenario, I would be tweaking BERT’s weights and training 1 head, while not training the other head.

This scenario would only make sense if the frozen head was already pre-trained and ready to go. If we did have this available, then it could work, and I would also be saving on compute.

Otherwise, if I’m dealing with a frozen head that is a randomly initialized layer, I would be dealing with nonsense outputs for that head, therefore nonsense results for that task.

Furthermore, I would effectively be training my system to do a single task (the task associated with the unfrozen head).

**Transfer learning:**

1 ) If I wanted a general-purpose model that could transfer to a wide variety of tasks, I could choose a good generalist such as BERT-base-uncased/cased. If I wanted a more domain-specific model, I could go with something such as BioBERT

2 ) I would adopt a graduated approach.

First I could try just freezing all but the last couple layers (these last couple layers are more task-specific), and seeing whether I get good validation/test set performance. This approach limits compute (as I’m only updating a small subsection of weights) and mitigates overfitting due to keeping most of the general-performance model weights intact. If performance is sufficiently high, great. Otherwise, continue to the next phase:

Next, I can try to unfreeze the middle layers in addition to the last couple layers. Those middle layers are abstract deep representations of the state space.

**Overall:**

Overall, for MTL and transfer learning, freezing more layers generally reduces compute but hampers classification performance

**Task 4:**

**Regarding datasets:**

I assume two csv files: the first CSV file domain.csv has two columns: sentence and domain (the class). The second CSV file sentiment csv is similarly structured. I ultimately load these into a PyTorch DataLoader.

**Regarding the core forward pass logic, I describe this below:**

Take your batch of input sentences

Tokenize (including padding tokens due to working with a batch)

Feed that into BERT, along with attention mask (which allows for ignoring of padding tokens)

Forward pass through BERT’s feed forward layers;

Get the CLS-pooled 768-length vector

Pass it through the 10% dropout layer

Feed this into linear layer. Output is a 4-length vector of logits

PyTorch’s CrossEntropy method applies softmax then calculates cross entropy loss.

**Regarding batch sizes and dual training:**

I assume the datasets will be small, so I stick with small batch sizes. Smaller batches tend to add noise to the gradient estimate, which can help in getting to the absolute minimum / not getting stuck in a local minimum

So, I do size-16 batches for both datasets.

Note that in my MTL setup, I pull both size-16 batches from both datasets, calculate their respective loss functions, and simply add them together (though I do leave an option to weight the losses differently)

Also note that if one dataset was significantly larger than the other, one could have a larger batch size for the bigger dataset.

**Regarding loss:**

For now, I just do a simple arithmetic sum of losses. However, I leave the option, given my weights w1 and w2, of potentially differently weighting the losses. Factors that would determine weighting would be greater importance of one task vs the other, a difference in dataset sizes, or a difference in task difficulty.

**Regarding epochs:**

I don’t expect much training, given we are fine tuning from a well-pretrained BERT model, so 3 epochs should be sufficient for now. If convergence is an issue, then one could bump that number up.

**Regarding learning rate:**

Given we are not doing extensive training, a smaller leraning rate seems appropriate, as the drawback of longer time to convergence is not a big deal anymore. We get the benefit of greater precision in reaching our optimal weights, and less likelihood of overshooting that optimum.

**Overall:**

So, overall, what we are doing in task 4 is a training loop that, within each epoch, updates the weights for BERT + both heads using the summed loss function.