ML Apprentice Exercise

Mudit Jindal

Task 1:

Describe any choices you had to make regarding the model architecture outside of the transformer backbone.

The backbone of the sentence transformer model is the TransformerEncoder layers. This is a standard transformer architecture which includes:

- **MultiHead Attention (Self):** The input sequence is split along the *d_model* dimension and the splits are passed into different heads. This allows the model to attend to different parts of the input sequence. It must be noted that the *num_heads* should divide the *d_model* perfectly.
- **Positional Encoding:** The output from the embedding layer is enhanced by adding in positional encoding to it. Sin curve is used to encode the even indices whereas Cos curve is used to encode the odd indices. This injects information about the order of tokens in the sequence
- **FeedForward Network:** This is used to add non-linearity and allows the model to learn more complex representations. This comprises two linear layers with Relu as the activation function.
- **Residual Connections and Layer Normalization:** These are introduced to stabilize training and improve performance.

This is in line with the original transformer model. However, I think there are several design choices beyond the standard transformer encoder layers. Here they are:

- **Embedding Layer:** Here I have used randomly initialized embedding layers rather than pre trained ones. This will force the model to learn token representations from scratch during the training. The size of the embedding is controlled by the hyperparameter of *d_model*. This needs to be chosen carefully in my opinion. A larger *d_model* will allow the model to capture more information but it also leads to an increase in the number of parameters. The number of parameters in this layer also depend on the vocab size. It must also be noted that the vocab size in this case is solely dependent on the example sentences that I use to test out the model.
- **Positional Encoding:** The positional encodings are sinusoidal functions to encode information about the order of tokens just like the original transformer paper. However, many modern models also use learned positional embeddings, so it's worth mentioning as an alternative. In this model, the positional encoding is based on the *MAX_LEN* parameter which is equal to the maximum length of sentence that I have in my example sample sentences.
- **Pooling Strategy:** I have used Adaptive Average Pooling 1d over the sequence dimension to get fixed size sentence embeddings. The original transformer model did not use pooling for sentence level representations as it was designed for sequence to

sequence tasks like translation. Here they had a decoder that generated the outputs autoregressive. There is no extraction of an explicit sentence embedding. The average pooling layer in this model receives the shape of *(batch_size, d_model, seq_len)* and converts it to *(batch_size, d_model, 1*) which we can squeeze to get *(batch_size, d_model)*. Max pooling could have also been used in this case but I went ahead with Average pooling. This is because of the assumption that all tokens contribute equally to the sentence's meaning as average pooling balances information across the sequence.

- **Final Output Shape:** Since this is a sentence transformer the final output is of the shape of *(batch_size, d_model)*. This is a fixed dimensional representation of the entire sentence. Even if the input sentences are of variable length each sentence will have a fixed length representation. The output is controlled by the *d_model* hyperparameter.

The hyperparameters decide how big and fast the model is. The 128 dimensional embedding *(d_model)* keeps sentence representations small and efficient, while 8 attention heads help the model focus on different parts of the sentence at once. With 4 Transformer layers, the model can understand context well. The 512 dimensional feedforward network adds a bit of complexity. The model is designed to be small and quick but we can change this by changing the different hyperparameters. The model has a total of 795,648 trainable parameters.

Overall, the model is quite different from the original transformer architecture which processes entire sequences for tasks like machine translation. The SentenceTransformer is designed to generate fixed-size sentence embeddings. It only uses an encoder stack instead of an encoder-decoder setup. While the original Transformer outputs per-token representations, this model incorporates adaptive average pooling to condense token-wise embeddings into a single vector per sentence. This makes it more suitable for semantic similarity, retrieval, and classification tasks rather than sequential generation.

Task 2:

Describe the changes made to the architecture to support multi-task learning.

I've adapted the Sentence Transformer for multi-task learning, specifically for sentence classification (Task A) and named entity recognition (NER, Task B), by adding two distinct linear layer heads. For Task A, the fixed-size sentence embedding, obtained through pooling the encoder's output, is fed into a linear layer with *num_classes_a* output nodes, generating sentence-level classification probabilities. For Task B, per token labeling required for NER. I had to bypass the pooling step so the output tensor from the Transformer encoder layers, shaped *(batch_size, seq_len, d_model)*, is directly passed into a linear layer with *num_classes_b* output nodes. This yields a tensor of shape *(batch_size, seq_len, num_classes_b)*, providing class probability predictions for each token in the input sequence, which is what is needed for the NER. For the sake of simplicity and speed the task heads are just a single linear layer. We can always replace this with a complex dense network depending on the difficulty of the task and the amount of compute available. The model has a total of 796,423 trainable parameters.

Task 3: Training Considerations:

Training Scenarios:

1. **If the entire network should be frozen:**
    a. **Implications:** If the entire model is frozen there is no point in training the model as the parameters would not be updated during the training. The model will retain its weights. The initial model could have randomly initialised weights or pretrained weights. If the weights are randomly initialised the model output will be completely random. In conclusion, the model would not learn from the new data.
    b. **Advantages:** The model cannot undergo training. This sort of a model should be used solely for inference (only if it was pre trained earlier). If pretrained, this model can be used as a backbone for other models.
    c. **Rationale:** This scenario is generally not recommended for training. Training implies adapting the model to new data. If the model is frozen, it will not adapt.
2. **If Only the Transformer Backbone Should Be Frozen**:
    a. **Implications:** In this case only the task specific heads would be trained and the weights in the transformer backbone would remain fixed. The task specific heads will learn to take the output from the backbone and convert it to the task specific output to minimize the loss function.
    b. Advantages: This is quite advantageous as you have to train less number of parameters. The training is a lot quicker and requires less computation. It also reduces the risk of overfitting up to some extent as the weights in the backbone do not get updated. The model essentially leverages the general language understanding capabilities of the backbone.
    c. Rationale: This scenario is really suitable when you have a pre-trained transformer model that has been trained on a large varied corpus. In this case the backbone model has a good general language understanding. So, it is easy to adapt the model to the task that you want with limited data.
3. **If only one of the task-specific heads (either for Task A or Task B) should be frozen:**
    a. Implications: One task's classifier would remain fixed while the other classifier and the sentence transformer will be trained. This would greatly lead to bias towards the trainable task.
    b. Advantages: This can be useful to improve the performance of a specific task. We can also analyze the changes in responses of the frozen task head as it will be impacted by the change in the weights of the backbone.
    c. Rationale: This scenario is only useful in rare specific situations. This could be useful when a task has significantly more data than the other task. You don't want to bias the learning towards the task that has more data so you can freeze that task head. As mentioned in advantages it can also be helpful to analyze the impact of one task on another.

Transfer Learning

1. **The choice of a pre-trained model:** For transfer learning I would generally choose a model that has been pre trained on a large varied corpus (Considering it is a natural language task). In this case, the model has a strong understanding of the general

language. It's also important to consider models that have been pre-trained on tasks similar to the one we want to fine-tune. The key idea is that for related tasks, the features learned in the early layers of the model will be similar, making fine-tuning more effective.

2. **The layers you would freeze/unfreeze:** Starting out, I would freeze the lower layers of the model while keeping the task specific layers unfrozen. The lower layers typically capture general features that are useful across different tasks, so freezing them ensures they remain stable and are not drastically altered by the limited task specific data. A general idea is that you freeze the backbone and train the task specific heads. If I want to train the model a bit further, I will unfreeze some of the upper layers. These layers could then go on to capture more task specific patterns. These patterns still rely on the general features in the lower layers which is why it is important to keep the lower layers frozen. Lastly, if I have enough compute and data, I might also unfreeze the entire network for full fine tuning. This is uncommon because large language models are so massive that they require extensive data and significant compute power for full fine-tuning. In cases where fine tuning a large language model is necessary, parameter efficient fine tuning (PEFT) techniques like LoRA (Low Rank Adaptation) and QLoRA (Quantized Low Rank Adaptation) can be used. These methods reduce the number of trainable parameters by introducing low rank updates to specific layers, allowing the model to adapt to new tasks without modifying the full network.
3. **The rationale behind these choices:** The general idea is that we want to make the most of the pretrained knowledge of the model while adapting it to the task that we want. Freezing layers helps prevent overfitting, especially when task specific data is limited, while also preserving the general understanding of the model from pretraining. We can then gradually unfreeze the model for controlled adaptation of its weights. This ensures that the model refines according to the task without losing valuable pretrained knowledge. This approach also improves computational efficiency by reducing the number of trainable parameters, leading to faster training. Transfer learning also works best when the new task is similar to the one the model was originally trained on. The more alike they are, the easier it is for the model to adapt.

Task 4: Training Loop Implementation (BONUS)

1. **Handling of Hypothetical Data:** In MTL (multi task learning), the tasks share a common underlying representation. In this case, the shared sentence encoder learns a representation of input sentences that benefits both tasks. The training data includes sentences labeled for Task A (sentence classification) with categorical labels Task B (NER) with token level labels. The data is paired, which means each sentence has one label for the whole sentence and separate labels for each word in the sentence. To prepare the data, sentences are tokenized, vocabulary is built, and sequences are padded to ensure uniform sequence length for batch processing. The assumption here is that we will have a large and diverse dataset so that the model can generalize to text it has not seen in the dataset.
2. **Forward Pass:** In the forward pass the preprocessed data is passed through the embedding layer. The positional encoding is added to help the model understand the

order of words in the sentence. The data is then processed through a stack of multi head attention layers that learn contextual relationships between words. Finally, the output is sent to task specific layers, where one layer predicts a label for the entire sentence and another predicts labels for each word in the sentence. The assumption here is that the shared sentence encoder will effectively capture information that is useful for both the tasks.

3. **Metrics and Training Strategy:** For both tasks, the model uses cross entropy loss as the training objective. In Task A (sentence classification), the output passes through a softmax function to convert logits into probabilities, and cross entropy loss measures the difference between predicted and actual labels. In Task B (NER), a softmax layer is applied at the token level, and cross entropy loss is computed for each token's predicted category. The total loss is the sum of both task losses, guiding model updates during training. Once trained, metrics such as accuracy, precision, recall, and F1-score can be used to evaluate the model's performance on each task.

The model now has a total of 799,751 trainable parameters. This is an increase from the model in task 2 (796,423 trainable parameters) because of the increase in vocab size. The vocab size affects the embedding layer in the model, which is a significant contributor to the total parameter count. The larger the vocabulary, the more parameters the embedding layer has.

Note:
The .to(device) function is used to ensure that the code can run on a GPU (CUDA or MPS) if available; otherwise, it falls back to the CPU. This is particularly beneficial when dealing with large models and datasets, as running computations on a GPU can significantly speed up training and inference. However, in our hypothetical case, the dataset is very small, and the model itself is relatively lightweight. Because of this, the time taken to transfer tensors to the GPU is actually greater than the time it takes to run the entire code on the CPU. As a result, the code runs faster without sending data to the GPU.

Despite this, I have still included .to(device) in all functions to maintain good coding practices. This ensures that if we scale up the model or dataset in the future, the code will be ready to utilize GPU acceleration efficiently.

Conclusion:
This assignment was a great learning experience and helped me revise the Transformer architecture in depth. I particularly enjoyed implementing the different components and ensuring a clear understanding of tensor transformations at each step. To make the code easier to follow, I have added thorough comments, especially for tracking the shape of tensors as they pass through various transformations. Additionally, each module of the model was first coded up in a .ipynb notebook, where I tested each component using a random input tensor of the expected shape. This allowed me to verify that the output shape matched expectations before integrating the module into the final implementation. This notebook can be found in the ref folder in the repository.

Additionally, I have packaged the code in a Docker container for better reproducibility and ease of deployment. Overall, this assignment was a valuable hands-on revision of transformers and multi-task learning.