Task 1:

Outside of the BERT transformer backbone, the primary architectural changes to generate fixed-length sentence embeddings revolve around the **pooling strategy** applied to the token embeddings. The transformer backbone outputs a sequence of token embeddings, and these need to be aggregated into a single, fixed-size vector representing the entire sentence.

Here are the specific choices related to the pooling strategy:

1. **Pooling using the ‘CLS’ token:**
   This strategy selects the embedding of the special ‘CLS’ token (at index 0) as the sentence embedding. The ‘CLS’ token is prepended to the input sequence during tokenization and represents the aggregate classification information from the entire sentence during the transformer's forward pass.

2. **Mean pooling:**
   This strategy calculates the element-wise mean of all the token embeddings in the sequence, excluding the padding tokens. I handle padding by using the BERT attention mask to selectively weigh the embeddings of actual tokens while assigning padding tokens a weight of zero before averaging.

3. **Max pooling:**
   This strategy takes the element-wise maximum of all the token embeddings in the sequence, again ignoring padding tokens by using the BERT attention mask. Rather than zeroing out the embeddings of padding tokens (like I did for mean pooling), I replace each value in the embeddings with a large negative value (-1e9) before performing the maximum operation. This ensures that the padding tokens do not influence the resulting maximum embedding values for the non-padding tokens.

4. **Optional Embedding Normalization:**
   I also added functionality for L2 normalization of the final sentence embeddings. The embeddings are normalized before applying similarity comparisons (e.g., cosine similarity), as it ensures that the length of the vectors does not disproportionately influence the similarity score. An application of cosine similarity would be to output how “closely related” 2 sentences are on a scale from 0 to 1 (after scaling), allowing you to build a dataset around sentences that feature similar content.

Task 2:

The changes made to the architecture to support multi-task learning are as follows:

1. **Transformer Backbone Reuse:**
   The sentence transformer from Task 1 is used as the backbone for the multi-task transformer for Task 2. Thus, I rely on the original sentence transformer for both the classification and sentiment analysis tasks because the sentence embeddings serve as the input for the 2 task-specific heads.

2. **Task-Specific Output Heads:**
   The multi-task transformer incorporates separate heads for each task:

   - Classification head: The classification task determines a relevant topic that a sentence belongs to based on the following topics: *"Technology", "Sports", "Politics", "Entertainment", "Health"*. The classification head is a sequence of linear layers appended with activation and dropout that takes the shared sentence embedding as input and outputs logits for the classification task. The output dimension of the final linear layer in this head per sentence is equal to the number of classes for the classification task (5), with each value in the output representing the likelihood that the sentence belongs to that respective class.

   - Sentiment head: The sentiment analysis task determines whether a sentence has one of the following tones: "Negative", "Neutral", "Positive". The sentiment head is a sequence of linear layers appended with activation and dropout that takes the shared sentence embedding as input and outputs logits for the classification task. The output dimension of the final linear layer in this head per sentence is equal to the number of classes for the classification task (3), with each value in the output representing the likelihood that the sentence belongs to that respective class.

3. **Forward Pass Modification:**
   The forward pass of the multi-task transformer is designed to:
   1. Pass tokenized sentences through the sentence transformer to obtain a single sentence embedding

   2. Feed this sentence embedding into both the classification head and the sentiment head independently.

   3. Return the logits from both heads

4. **Observe the predicted classes/sentiments:**
    - Using the logits from both heads, take the softmax over each of the logits to observe the category-wise prediction probabilities for each sentence for each task

5. **Freezing the Transformer Backbone (Optional):**
   The multi-task transformer class allows for freezing the weights of the underlying sentence_transformer since I utilize a pre-trained BERT backbone to generate sentence embeddings. If set to *True*, the parameters of the pre-trained sentence transformer are not updated during the multi-task training (implemented in Task 4). This weight freezing strategy allows me to leverage the knowledge learned during pre-training and focus the training on the task-specific heads.

This design for multi-task learning allows the model to learn relationships and dependencies between the two tasks, potentially leading to improved performance on both compared to training individual models for each task. The shared representation can capture general linguistic features relevant to classification and sentiment analysis via a pre-trained BERT backbone, while the task-specific heads can learn the nuances required for each specific task.

Task 3:

1. **If the entire network is frozen:**
    a. **Advantages:**
        i. Fast training convergence since only the final classification layer needs to be trained
        ii. Prevents overfitting when working with very small datasets
        iii. Memory efficient as no gradients need to be stored for most parameters
        iv. Maintains all knowledge from pre-training (however, the task-specific heads in my implementation are not pre-trained)
    b. **Disadvantages:**
        i. Limited adaptation to domain-specific language patterns (the frozen task-specific heads cannot adapt to learn the specific patterns relevant to classification and sentiment analysis respectively)
        ii. Fixed feature extraction is not optimal for the target task (likely to underfit to the task)
        iii. No fine-tuning of contextualized representations

2. **If only the transformer backbone should be frozen (Utilized in Task 4):**
    a. **Advantages:**
        i. Task-specific heads can adapt to the classification task
        ii. Better performance than freezing everything
        iii. Less likely to underfit to the tasks than freezing everything
        iv. Prevents overfitting when working with smaller datasets (not few-shot/single-shot datasets)
        v. Efficient training compared to full fine-tuning
        vi. Preserves the general language understanding capabilities of the frozen pre-trained model
    b. **Disadvantages:**
        i. Limited adaptation of the transformer's contextual representations

3. **If only the transformer backbone should be frozen (Utilized in Task 4):**
    a. **Advantages:**
        i. Allows the shared backbone to adapt to both tasks
        ii. Prevents previously learned task (frozen head) from being forgotten
        iii. Efficient for continual learning scenarios
        iv. Potentially enables positive knowledge transfer between tasks
    b. **Disadvantages:**
        i. More complex training dynamics (balancing learning for both tasks)

ii. Possibility of negative knowledge transfer if tasks are too different
iii. Increased computational requirements for training compared to freezing everything
iv. More likely to overfit to smaller datasets

Transfer Learning Approach + Rationale

1. **Choice of Pre-trained Model:**
   For sentiment analysis, I chose the pre-trained 'bert-base-uncased' model and utilized the HuggingFace transformers library to instantiate and deploy it. I chose this model because it was pre-trained on a large corpus of data and is frequently fine-tuned for tasks like classification, tokenization, and question answering. Moreover, I chose the case-insensitive version because of its reduced complexity, potentially allowing the model to better fit to my limited training data, learn more general sentiment patterns (independent of capitalization), and generalize better to unseen data. Moreover, I utilized a local anaconda environment to solve each task, and my limited computational resources motivated my model selection process.

2. **Layers to Freeze/Unfreeze:**
   I froze the transformer backbone and trained the 2 task-specific heads to convergence. I used this approach because it is not susceptible to overfitting to my small training and testing dataset, allows for computationally efficient training, and maintains the general language understanding of pre-trained BERT, thus enabling each task-specific head to adjust itself based on BERT's embeddings. The training loop could be further modified to freeze each head until convergence, beginning with the sentiment analysis head frozen and ending with the (now-trained) classification head frozen and the sentiment analysis head unfrozen until convergence. This approach can add stability to the training loop and ensure that negative knowledge transfer does not degrade the performance of one head at the expense of improving the other. If each head featured greater complexity, I would enhance the transfer learning strategy by gradually unfreezing each task-specific head from top to bottom using smaller learning rates. This prevents each task-specific head from forgetting useful pre-trained knowledge, especially in lower layers, and mitigates drastic changes in the weights of the head that could destroy useful knowledge.

Task 4:

Assumptions and Decisions:

1. **Model Architecture and Forward Pass:**
    - **Assumption:** The architecture is designed around a shared transformer backbone connected to 2 task-specific heads. One task-specific head is responsible for classifying a sentence into a ‘topic’, and the other is responsible for determining its sentiment. I assumed that this would be the procedure used by the multi-task transformer:
        - A pre-trained BERT model is the backbone for encoding the input sentences into a fixed-length sentence embedding.
        - The forward pass takes the tokenized sentence input (input IDs and attention mask), gets the sentence embeddings, feeds the sentence embeddings into the task-specific heads, and outputs the logits for both the classification and sentiment tasks.
        - The combined training loss of both heads is backpropagated through the trainable layers (each head), and each head is updated concurrently to maximize shared performance on the 2 tasks.
    - **Decision:** The design choice to leverage a pre-trained sentence transformer to obtain meaningful representations allows me to harness the power of transfer learning to perform well over a small dataset. Moreover, these design decisions align with the core idea of multi-task learning – leveraging shared representations to improve performance and efficiency.
2. **Data Handling:**
    - **Assumption:** The training and testing data are provided as lists of strings (sentences), and the corresponding labels (for classification and sentiment) are provided as lists of numerical indices. The order of sentences aligns with the order of their respective labels. To demonstrate the effectiveness of transfer learning, I provided 10 total training and testing sentences to evaluate my approach.
    - **Decision:** The code iterates through the training data in batches. For each batch:
        - It tokenizes the sentences using the pre-trained BERT base tokenizer from the transformers library.
        - The tokenized inputs (input IDs and attention mask) are moved to the specified device (GPU if available, otherwise CPU).

        - The corresponding batch of labels and sentiment scores is also converted to PyTorch tensors and moved to the same device.

            1. For larger datasets and more complex tasks, I would have utilized the PyTorch DataLoader class, where the train and test labels would be effectively collated and shuffled to prevent a complex model from memorizing the input sequence.

3. **Loss Calculation and Training:**
    - **Decision:** Separate losses are computed for each task: CrossEntropyLoss() is used for both classification and sentiment analysis because both tasks are framed as multi-class classification problems.
    - **Assumption:** The target labels for each task are integer class indices, which is the expected input format for cross-entropy loss.
    - **Decision:** The losses for both tasks are calculated independently for each batch. A combined loss is then computed by simply adding the individual task losses, assuming equal weighting of the two tasks in the overall optimization objective. More sophisticated training strategies (freezing one head while training the other to convergence) or employing separate weighting strategies if one task is assigned greater importance can provide additional performance gains.
    - **Backpropagation:** The total loss is backpropagated through the network, allowing the weights of each head to be updated based on the gradients from both tasks. This shared gradient update enables the model to learn representations beneficial for both classification and sentiment analysis.
4. **Evaluation:**
    - **Decision:** During evaluation, the model is set to eval() mode, which disables dropout and batch normalization's training behavior. Moreover, none of the trained model's weights are updated (no gradients are computed or backpropagated) during evaluation to demonstrate whether the model effectively fits to data it has not trained on.
    - **Handling of Hypothetical Data:** The evaluation loop takes in the same format of data as the training loop (lists of sentences and their corresponding labels for classification and sentiment analysis), but processes each test sentence individually. It tokenizes each sentence, performs a forward pass to get the logits for both tasks from the trained model, and then outputs the predicted class and sentiment labels.

- **Assumption:** The test data format matches the training data format and represents the same set of tasks the model has been trained on.

5. **Metrics:**
    - **Metrics Provided:** I used the classification_report() function from scikit-learn, which provides precision, recall, F1-score, and support for each class.
        - **Precision:** Answers the question: "Of all instances predicted as class X, what percentage belongs to class X?" High precision means the model rarely mislabels text as belonging to a category (topic or sentiment) when it doesn't.
        - **Recall:** Answers the question "Of all actual instances of class X in the dataset, what percentage did the model correctly identify?" High recall ensures the model doesn't miss sentences belonging to important categories.
        - **F1-score:** The harmonic mean of precision and recall, providing a balance between the two. F1-score is particularly valuable when classes are imbalanced, as it prevents the model from appearing effective by simply predicting the majority class, a pitfall that plain accuracy is susceptible to.
        - **Support:** The number of actual occurrences of each class in the test dataset, providing context to the metrics by showing class distribution. If the dataset was unbalanced, I could identify if poor performance for a particular category stems from insufficient training examples.
    - **Assumption:** I used a default value of 0 to handle cases where precision would encounter a divide-by-zero error. This occurs when a class has no predicted instances, a scenario I encountered occasionally when running my models.