Task 3: Training Considerations

Possible scenarios to train the multi-task model are:

1. **Freezing the whole model**:
   Useful for inference or when the model is already trained on a very similar domain. But the model can't adapt to the new task/data, especially since the task-specific heads are initialized randomly.

2. **Freezing only the transformer**:
   This lets the model benefit from BERT's general language understanding while still learning the task-specific heads. It's a good tradeoff when data is limited and avoids overfitting. I went with this in my code.

3. **Freezing one task head**:
   This would make sense if one task is already well-trained or stable, and we want to transfer that knowledge while focusing on the other. For example, freezing the classification head while improving NER performance.

For **transfer learning**, I used *bert-base-uncased* from HuggingFace. It provides a solid backbone for both sentence-level and token-level tasks. For now, I froze the backbone and only kept the heads trainable.

If training stalls, I'd gradually unfreeze the last few layers of the BERT backbone. These layers capture more task-specific patterns, while earlier layers retain general syntax/semantics. This staged approach is common in transfer learning and helps balance stability and adaptation.

Task 4: Training Loop Implementation

I implemented a multi-task training loop with dummy data to show how the model would be trained.

- **Data**:
  - Made-up sentences and corresponding labels for:
    - Task A: sentence classification (3 categories - Sports, Jobs, Travel).
    - Task B: named entity recognition (4 categories - Person, Location, Organization, Other)
  - Using PyTorch's data utilities, defined a custom MTLDataset class and a corresponding collate function to be used with PyTorch's DataLoader.

- **Forward pass**:
  The shared transformer backbone produces hidden states. For Task A, I used the [CLS] token output, and for Task B, I used the full sequence output. Each task has its own head.
- **Losses**:
  Both tasks use *CrossEntropyLoss*. For token-level loss, I ignored padding tokens (ignore_index = -100). I combined the two losses using an unweighted sum for now (loss = loss_a + loss_b). If needed, the tasks can be assigned different weights later.
- **Metrics**:
  Accuracy for both tasks. Other metrics (like precision, recall, F1) could be added if needed, but for now I focused on getting the loss computation and structure right.

This setup allows for efficient joint training, with the transformer either kept frozen (or partially frozen) or learning shared features and the heads adapting to each task. The design is modular, so it's easy to extend or tweak individual tasks.