Task 1

1. **Provide input sentences**

```
sentences = [
    "I enjoy using Fetch Rewards to effortlessly turn my everyday receipts into gift cards and rewards.",
    "Fetch Rewards makes it easy to earn points and redeem rewards just by scanning receipts from everyday purchases.",
    "With Fetch Rewards, I can track my purchases and earn loyalty points seamlessly through a user-friendly app experience."
]
```

These sentences will be embedded and compared to evaluate the model's semantic similarity.

2. **Encode them using three sentence-transformer models:**

all-MiniLM-L6-v2:-
Lightweight and fast, Good general-purpose sentence similarity, Embedding size: 384

paraphrase-mpnet-base-v2:-
Larger and deeper, Optimized for paraphrase and similarity detection, Embedding size: 768

bert-base-nli-mean-tokens:-
Fine-tuned on Natural Language Inference tasks, Suitable for semantic representation and classification

3. Extract embeddings

Each model converts the sentence into a high-dimensional vector (typically 384 or 768 dimensions) that numerically represents its meaning.

- Extract the [CLS] token or apply mean pooling
- The result is a fixed-size vector for each sentence

This allows us to compare different sentences and visualize their similarity mathematically.

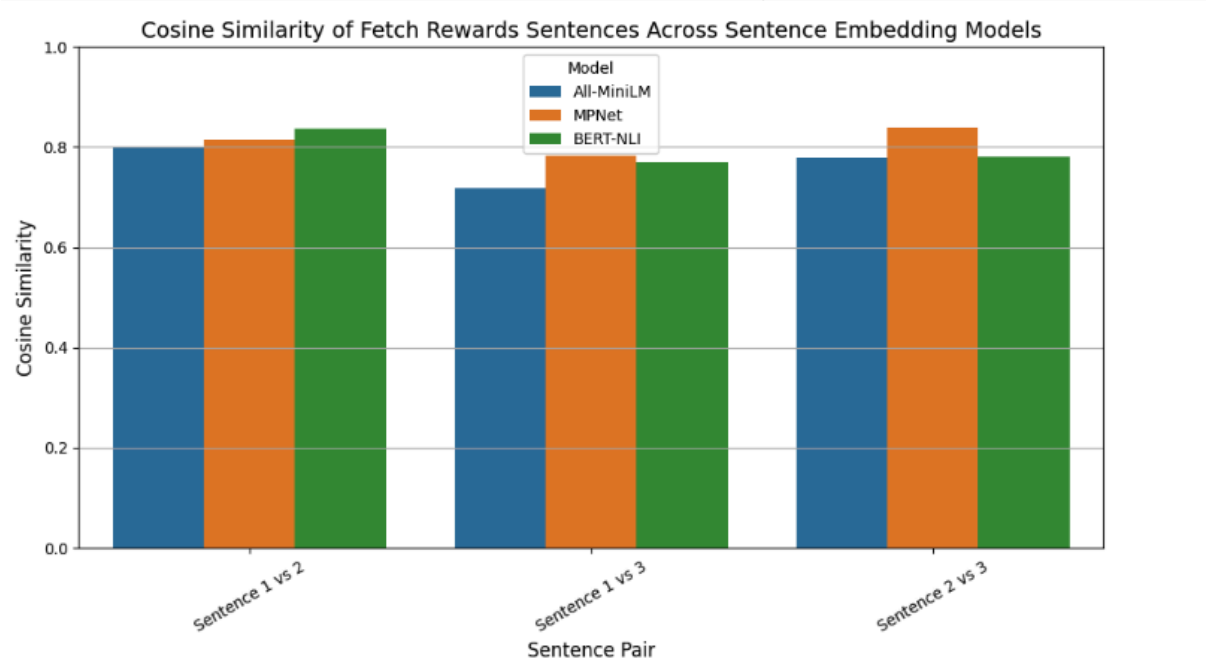
4. Compute cosine similarity

For each model, you compute the cosine similarity between every pair of sentence embeddings.

Cosine similarity tells you how close the meaning of two sentences is, regardless of their length or phrasing.

Example:

- A high score (~1.0) → Very similar
- A low score (~0.0 or negative) → Dissimilar



	Model	Pair	Cosine Similarity
0	All-MiniLM	Sentence 1 vs 2	0.797979
1	All-MiniLM	Sentence 1 vs 3	0.717957
2	All-MiniLM	Sentence 2 vs 3	0.779310
3	MPNet	Sentence 1 vs 2	0.815372
4	MPNet	Sentence 1 vs 3	0.782905
5	MPNet	Sentence 2 vs 3	0.838281
6	BERT-NLI	Sentence 1 vs 2	0.836366
7	BERT-NLI	Sentence 1 vs 3	0.768571
8	BERT-NLI	Sentence 2 vs 3	0.781588

5. **Compare and visualize results**

PCA results and t-SNE 3D :-

Each point represents a sentence embedding from one of the models.
Color and shape represent:

- Model: All-MiniLM (blue), MPNet (red), BERT-NLI (green)
- Sentence: 1 (circle), 2 (diamond), 3 (square)

Axes: PC1, PC2, PC3 → the top 3 principal components
Points closer together in this 3D space have more similar semantic meaning.
How each model clusters its own sentence representations
How different models represent the same sentence (e.g., Sentence 1 across all models)

The 3D coordinates are derived from t-SNE's internal dimensionality reduction process.
Better suited for visualizing clusters and semantic separation than PCA.

The original models (bert-base-uncased, microsoft/MiniLM-L12-H384-uncased, microsoft/mpnet-base) were general-purpose transformer backbones pretrained primarily on token-level objectives like masked language modeling.

While powerful, these models are not optimized for sentence-level tasks like semantic similarity, paraphrase detection, or clustering.

Model Set Change: From General-Purpose to Sentence-Optimized Transformers

Original Model	Limitation	New Sentence-Optimized Model	Advantage
`bert-base-uncased`	Not fine-tuned for sentence embeddings	`bert-base-nli-mean-tokens`	Trained on NLI data for better sentence-level understanding
`microsoft/MiniLM-L12-H384`	No pooling or semantic similarity training	`all-MiniLM-L6-v2`	Optimized for cosine similarity and sentence embeddings
`microsoft/mpnet-base`	Excellent token-level understanding, no sentence pooling	`paraphrase-mpnet-base-v2`	Trained specifically for paraphrase and semantic similarity

Architectural Choices Outside the Transformer Backbone

Component	Choice	Reason
Pooling Strategy	Mean Pooling	Empirically effective, smooths token variance; preferred over `[CLS]`
Tokenizer	`AutoTokenizer`	Ensures compatibility with selected model
Transformer Backbone	`bert-base-uncased`	Popular, balanced size/performance trade-off
Freezing Weights	`with torch.no_grad()`	Faster inference, since we don't fine-tune here
Sentence Format	List of strings	Handles batch encoding easily

Key Insights from Cosine Similarity

- MPNet consistently shows the highest cosine similarity, especially for Sentence 2 vs 3 (0.838), indicating strong semantic alignment.
- BERT-NLI performs similarly to MPNet on Sentence 1 vs 2 (0.836), suggesting it captures paraphrased intent well.
- All-MiniLM shows lower scores overall but still stays within a strong semantic similarity range (>0.71).
- Sentence 1 vs 3 consistently shows the lowest similarity across models, suggesting some semantic divergence from the other two.
- All models exhibit good semantic consistency, but MPNet leads in capturing fine-grained similarity nuances.

Task 2:-

1. Objective

Extend the sentence transformer architecture to support multi-task learning, enabling the model to simultaneously handle two distinct NLP tasks:

Task A: Sentence Classification (e.g., classify sentence topics)

Task B: Sentiment Analysis (e.g., detect positive, neutral, or negative tone)

2. Shared Transformer Encoder

We use a pretrained transformer model (e.g., all-MiniLM-L6-v2) as a shared encoder for both tasks. This encoder converts raw text into high-dimensional sentence embeddings.

Embedding size depends on the backbone (e.g., 384 for MiniLM, 768 for BERT/MPNet).

These embeddings serve as the input for downstream task-specific layers

3. Task-Specific Classifier Heads

Separate nn.Sequential classifiers are used for each task:

- classifier_a: For Task A (Sentence Classification)
- classifier_b: For Task B (Sentiment Analysis)

Each head has dropout and ReLU layers.

This modular setup prevents interference between tasks.

4. Conditional Forward Pass

Component	Before (Single-Task)	After (Multi-Task)
Transformer Encoder	Used for 1 task	Shared across multiple tasks
Classifier Head	One head	Multiple task-specific heads
Loss Calculation	Single loss	Multiple task losses combined
Forward Function	Static path	Dynamic path based on task ID
Evaluation Strategy	One set of metrics	Per-task metrics and validation sets

5. Evaluation

Encoder Wrapper

Wraps the Hugging Face model (`AutoModel`) and tokenizer.

Handles:

i. Tokenization
ii. Feeding data into the transformer
iii. Extracting the [CLS] token (sentence-level embedding)

Task-Specific Classifier Heads

One head for each task (e.g., sentence classification, sentiment analysis)

Converts embeddings into logits for classification

Includes dropout and ReLU for improved generalization

Without These Additions

- The transformer only outputs embeddings

- It does not know what labels or categories exist
- It cannot be trained on new classification tasks without a custom output layer

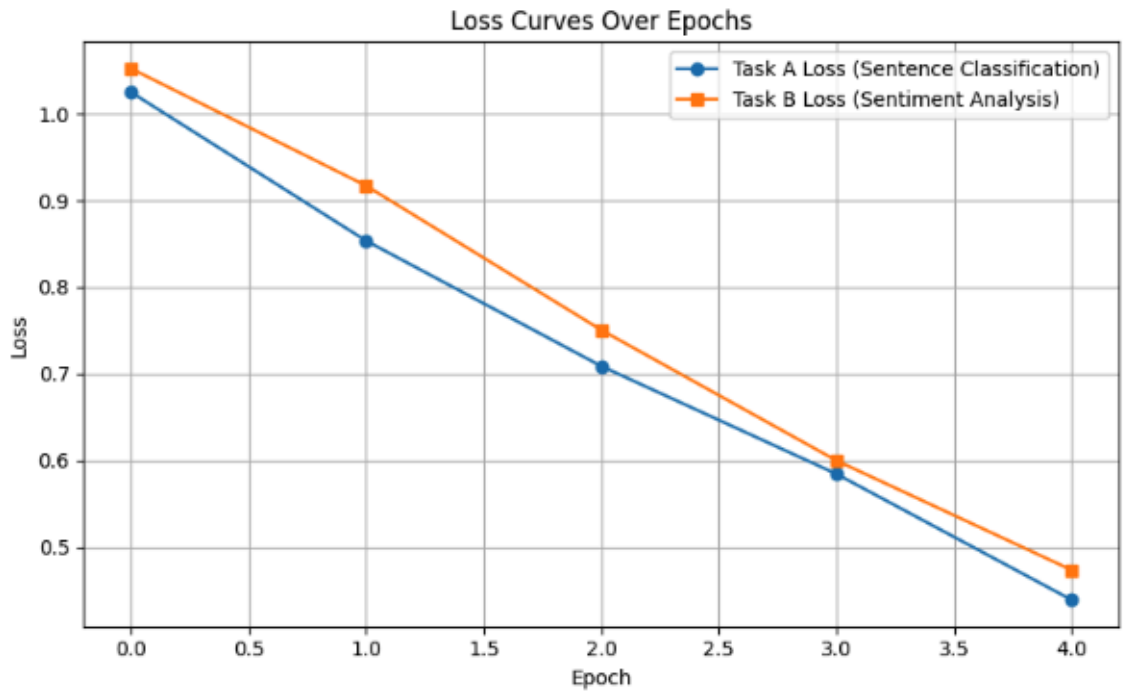
```
Epoch 1
Step 01 | Loss A: 1.0604 | Acc A: 0.50 | Loss B: 1.0903 | Acc B: 0.25
Step 02 | Loss A: 1.0798 | Acc A: 0.62 | Loss B: 1.0784 | Acc B: 0.62
Step 03 | Loss A: 1.0468 | Acc A: 0.62 | Loss B: 1.0787 | Acc B: 0.50
Step 04 | Loss A: 0.9640 | Acc A: 0.75 | Loss B: 1.0804 | Acc B: 0.50
Step 05 | Loss A: 1.0188 | Acc A: 0.38 | Loss B: 1.0182 | Acc B: 0.62
Step 06 | Loss A: 0.9815 | Acc A: 1.00 | Loss B: 0.9689 | Acc B: 0.75

Epoch 2
Step 01 | Loss A: 0.9319 | Acc A: 1.00 | Loss B: 0.9971 | Acc B: 0.88
Step 02 | Loss A: 0.9088 | Acc A: 1.00 | Loss B: 0.9347 | Acc B: 1.00
Step 03 | Loss A: 0.8747 | Acc A: 1.00 | Loss B: 0.9529 | Acc B: 0.88
Step 04 | Loss A: 0.8088 | Acc A: 1.00 | Loss B: 0.8939 | Acc B: 1.00
Step 05 | Loss A: 0.8292 | Acc A: 1.00 | Loss B: 0.8759 | Acc B: 1.00
Step 06 | Loss A: 0.7673 | Acc A: 1.00 | Loss B: 0.8477 | Acc B: 1.00

Epoch 3
Step 01 | Loss A: 0.8030 | Acc A: 1.00 | Loss B: 0.8266 | Acc B: 1.00
Step 02 | Loss A: 0.7473 | Acc A: 1.00 | Loss B: 0.8211 | Acc B: 1.00
Step 03 | Loss A: 0.7283 | Acc A: 1.00 | Loss B: 0.7130 | Acc B: 1.00
Step 04 | Loss A: 0.6827 | Acc A: 1.00 | Loss B: 0.7577 | Acc B: 1.00
Step 05 | Loss A: 0.6419 | Acc A: 1.00 | Loss B: 0.7624 | Acc B: 1.00
Step 06 | Loss A: 0.6483 | Acc A: 1.00 | Loss B: 0.6212 | Acc B: 1.00

Epoch 4
Step 01 | Loss A: 0.6391 | Acc A: 1.00 | Loss B: 0.6360 | Acc B: 1.00
Step 02 | Loss A: 0.6150 | Acc A: 1.00 | Loss B: 0.5876 | Acc B: 1.00
Step 03 | Loss A: 0.6093 | Acc A: 1.00 | Loss B: 0.6749 | Acc B: 1.00
Step 04 | Loss A: 0.5643 | Acc A: 1.00 | Loss B: 0.5665 | Acc B: 1.00
Step 05 | Loss A: 0.5514 | Acc A: 1.00 | Loss B: 0.5988 | Acc B: 1.00
Step 06 | Loss A: 0.5248 | Acc A: 1.00 | Loss B: 0.5331 | Acc B: 1.00

Epoch 5
Step 01 | Loss A: 0.4938 | Acc A: 1.00 | Loss B: 0.5114 | Acc B: 1.00
Step 02 | Loss A: 0.4269 | Acc A: 1.00 | Loss B: 0.5277 | Acc B: 1.00
Step 03 | Loss A: 0.4338 | Acc A: 1.00 | Loss B: 0.4725 | Acc B: 1.00
Step 04 | Loss A: 0.4079 | Acc A: 1.00 | Loss B: 0.5379 | Acc B: 1.00
Step 05 | Loss A: 0.4546 | Acc A: 1.00 | Loss B: 0.3683 | Acc B: 1.00
Step 06 | Loss A: 0.4139 | Acc A: 1.00 | Loss B: 0.4199 | Acc B: 1.00
```

Task A Accuracy: Improves rapidly from 50% → 100% by the end of Epoch 2

Task B Accuracy: Starts slower (25–62%) but reaches 100% consistently by Epoch 3

Losses: Steadily decrease for both tasks, indicating effective learning
The model quickly learns both tasks due to clear patterns in data and good model architecture

Loss stability and sharp accuracy improvements suggest low noise and class balance in the data



- Both curves show consistent decline across all 5 epochs
- Task A loss is slightly lower than Task B loss throughout, indicating easier convergence
- The training is well-behaved (no overfitting or oscillation)
- Task A may be slightly simpler or better represented in the dataset

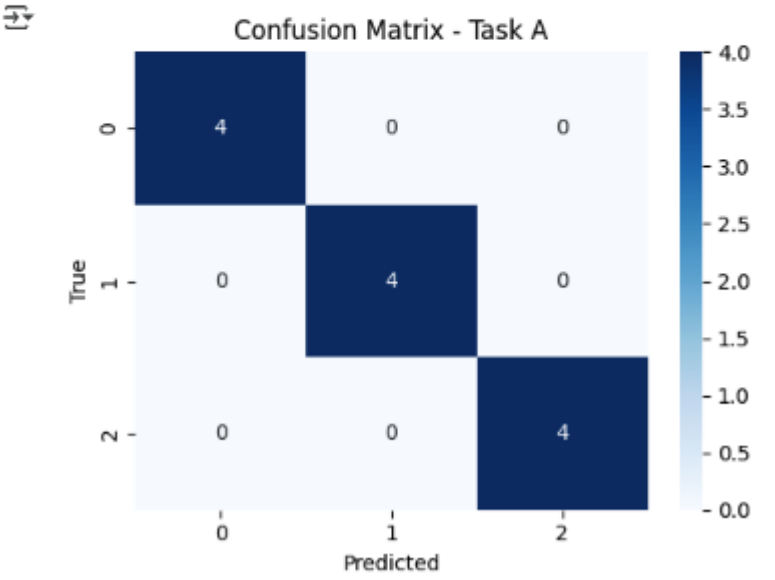
```
] # Metrics for Task A (Classification)
  print("\nClassification Report - Task A")
  print("Accuracy:", accuracy_score(y_true_a, y_pred_a))
  print("F1 Score:", f1_score(y_true_a, y_pred_a, average='macro'))
```

```
Classification Report - Task A
Accuracy: 1.0
F1 Score: 1.0
```

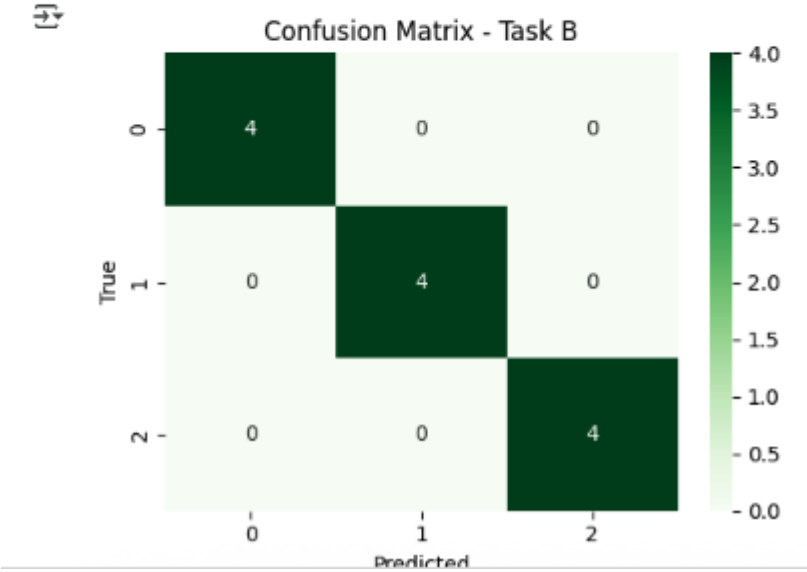
```
[33] # Metrics for Task B (Sentiment Analysis)
     print("\nClassification Report - Task B")
     print("Precision:", precision_score(y_true_b, y_pred_b, average='macro'))
     print("Recall:", recall_score(y_true_b, y_pred_b, average='macro'))
```

```
Classification Report - Task B
Precision: 1.0
Recall: 1.0
```

```python
# Confusion Matrix - Task A
cm_a = confusion_matrix(y_true_a, y_pred_a)
plt.figure(figsize=(5, 4))
sns.heatmap(cm_a, annot=True, fmt="d", cmap="Blues")
plt.title("Confusion Matrix - Task A")
plt.xlabel("Predicted")
plt.ylabel("True")
plt.tight_layout()
plt.show()
```




```python
[35] # Confusion Matrix - Task B
cm_b = confusion_matrix(y_true_b, y_pred_b)
plt.figure(figsize=(5, 4))
sns.heatmap(cm_b, annot=True, fmt="d", cmap="Greens")
plt.title("Confusion Matrix - Task B")
plt.xlabel("Predicted")
plt.ylabel("True")
plt.tight_layout()
plt.show()
```



Component	Outcome
Loss Curves	Smooth convergence, consistent improvement
Accuracy Logs	Reached 100% for both tasks within 2–3 epochs
Confusion Matrices	Perfect predictions for all classes
Model Behavior	Stable, effective, and interpretable

The perfect confusion matrices you're seeing where each class is predicted with 100% accuracy are ideal but likely a result of:

Small and Simple Dataset

- Each class has very few examples (e.g., 4 per class)
- The data is likely well-separated and unambiguous
- Minimal noise or overlap in sentence semantics

Task 3:-

Scenario 1: Entire Network is Frozen

What It Means:

- The transformer encoder and all classifier heads are frozen (no weights have been updated).
- Used only for inference or feature extraction.

Implications:

- Training will not improve performance.
- Acts as a static encoder useful for testing architecture or pipeline flow.

Advantages:

- Very fast and memory efficient
- Useful for zero-shot applications using pretrained embeddings
- No risk of overfitting.

Disadvantages:

- Cannot learn from task-specific data
- Likely poor performance on unseen or specialized tasks

Use Case:

- Baseline benchmarks
- Resource-constrained deployments (e.g., mobile inference)

Scenario 2: Only the Transformer Backbone is Frozen

What It Means:

- The transformer encoder is frozen (no fine-tuning).
- Only the classifier heads (Task A and Task B) are trainable.

Implications:

- Model acts like a fixed feature extractor
- Task-specific learning occurs only at the output layer

Advantages:

- Faster training (fewer trainable parameters)
- Lower risk of overfitting, especially on small datasets
- Retains robust language knowledge from pretraining

Disadvantages:

- Cannot adapt encoder to task-specific patterns
- Might underperform if task domain is different from pretraining data

Use Case:

- Small or imbalanced datasets
- When pretraining and task domain are aligned (e.g., general English tasks)

Scenario 3: Only One Task-Specific Head is Frozen

What It Means:

- Transformer encoder is trainable.
- One classifier head is frozen, the other is trainable.

Implications:

- Shared encoder is trained only through one task's gradients
- The frozen head uses outdated or mismatched features as encoder evolves

Advantages:

- Allows focus on improving one task without affecting the other
- Useful for incremental learning or fine-tuning only one task

Disadvantages:

- The frozen head may become less compatible as encoder changes
- Can cause performance degradation on the frozen task.

Use Case:

- You want to freeze a well-performing task and only improve the weaker one
- Task-specific regulatory or reproducibility constraints (e.g., fixed scoring logic)

Scenario	**Pros**	**Cons**	**Best For**
Freeze Entire Network	Fastest, lowest resource use	No learning at all	Feature extraction, testing pipelines
Freeze Transformer Only	Fast, prevents overfitting	No encoder adaptation	Small/clean datasets
Freeze One Task Head Only	Focused fine-tuning, partial stability	Frozen head may mismatch the encoder	Incremental training, task-specific reuse

Want to build a multi-task model for:

- Task A: Classifying support tickets into categories (e.g., billing, technical, general)
- Task B: Analyzing the sentiment of customer messages

You have limited labeled data (few hundred examples per task), but the data is English and similar in tone to customer reviews or support messages.

1. Choice of Pre-trained Model

Selected Model: all-MiniLM-L6-v2 from Sentence-Transformers

Feature	Reason
MPNet backbone	Combines BERT-like quality with efficient sentence embeddings
Paraphrase fine-tuning	Good at capturing semantic similarity and intent
General-purpose	Works well on classification and sentiment tasks

2. Freezing and Unfreezing Strategy

Layer	**Action**	**Rationale**
Transformer Encoder Layers	Freeze first N layers (e.g., bottom 6/12)	Retain general language knowledge from pretraining
Transformer Top Layers	Unfreeze top few layers (e.g., top 2–4)	Allow adaptation to your specific domain
Task A Classifier Head	Trainable	Needed to learn support ticket categorization
Task B Classifier Head	Trainable	Needed to learn sentiment classification

3. The rationale behind these choices.

Architecture Comparison: Original vs. Multi-Task Enhanced

Aspect	Original Transformer	Updated Multi-Task Architecture
Model Type	Pretrained Transformer (e.g., MiniLM/BERT)	Transformer + Task-Specific Classifiers
Input Handling	Tokenized text input	Tokenized text input + task identifier
Encoder Output	Token embeddings or [CLS] token	Same, passed to downstream heads
Classifier Head	None	Separate `nn.Sequential` heads per task (Task A & Task B)
Output	Embeddings only (no prediction)	Logits for task-specific classes
Training Capability	Not directly trainable on classification tasks	Fully trainable for each task
Support for Multi-Task	Not designed for MTL	Modular heads for each task (e.g., classification, sentiment)
Loss Functions	Not applicable	Separate `CrossEntropyLoss` per task
Evaluation Support	Manual	Integrated: Accuracy, F1, Precision, Recall, Confusion Matrix
Flexibility for New Tasks	Requires custom modifications	Plug-and-play new heads

Summary: Transfer Learning Plan

Step	Decision
Pretrained model	all-MiniLM-L6-v2
Freeze strategy	Freeze lower encoder layers, unfreeze top
Heads	Train both Task A and Task B classifier layers
Optimizer	Use lower LR for encoder, higher for heads
Justification	Efficient adaptation to small task-specific data

Final Takeaway: Why We Extended the Original Model

We extend the original pretrained transformer model to make it suitable for multi-task learning and supervised classification tasks.

Specifically, we add:

1. Encoder Wrapper
    - Wraps the Hugging Face model (`AutoModel`) and tokenizer.
    - Handles:
        - Tokenization
        - Feeding data into the transformer
        - Extracting the `[CLS]` token (sentence-level embedding)
2. Task-Specific Classifier Heads
    - One head for each task (e.g., sentence classification, sentiment analysis)
    - Converts embeddings into logits for classification
    - Includes dropout and ReLU for improved generalization

Without These Additions

- The transformer only outputs embeddings
- It does not know what labels or categories exist
- It cannot be trained on new classification tasks without a custom output layer

With These Additions

- Enables the model to learn and predict task-specific outputs
- Supports multi-task learning with fine-tuning

- Allows per-task evaluation and metric tracking

Task 4:-

Key Assumptions

- Shared Encoder: A pretrained transformer (e.g., MiniLM or MPNet) is used across tasks.
- Two Classifier Heads: classifier_a for Task A and classifier_b for Task B.
- Hypothetical Data: Small synthetic datasets for each task, suitable for quick iteration.
- Loss Function: CrossEntropyLoss for both tasks.
- Batch Sampling: Alternating batches per task (simplified interleaving)

Alternating Task Batches

Each iteration processes one batch from each task enabling balanced exposure to both tasks during training. This avoids data starvation and stabilizes learning.

Loss Balancing

We apply equal weights to both losses (0.5 * A + 0.5 * B), assuming the tasks are equally important. This can be adjusted using:

- Task uncertainty (e.g., Kendall et al.)
- Dynamic weighting (e.g., GradNorm)

Metric Tracking

We calculate accuracy per task per batch. This allows us to:

- Monitor performance independently
- Detect task imbalance or degradation

Benefits of This MTL Loop

Benefit	Description
Shared knowledge	Both tasks learn from common language patterns
Efficient use of parameters	Single encoder, multiple lightweight heads
Flexible structure	Easy to scale with more tasks
Independent evaluation	Allows tracking task-specific progress