Task 3: Training Considerations

(Multi Task Learning)

In this section, I evaluate different training strategies for multi-task learning (MTL), analyzing their implications, advantages, and trade-offs. Additionally, I thought it would be insightful to incorporate my thoughts on how these strategies apply to our current Fetch use case.

Scenario 1: Freezing the Entire Network

Implications:

Freezing both the transformer backbone and task-specific heads prevents any learning.

If the heads are untrained, they will produce random outputs, leading to poor performance.

Advantages:

- Computational efficiency - no additional training required.
- Preserves pre-trained knowledge, avoiding catastrophic forgetting.

Disadvantages:

- No task-specific adaptation, which could make it suboptimal.
- The model acts only as a feature extractor, limiting flexibility.

Rationale:

This approach works well when the pre-trained model is already well-aligned with the downstream tasks, such as when it has been fine-tuned on a highly similar dataset. However, if the target tasks require further adaptation, freezing the entire network prevents the model from learning task-specific patterns, leading to suboptimal performance and poor generalization. Also, going by the definition of MTL, **freezing the entire network wouldn't involve any learning at all**, as there would be no parameter updates for either the backbone or the task-specific heads.

Relevance to Fetch Use Case:

This isn't a suitable approach for Fetch. The model needs to adapt to receipt-specific patterns, and freezing the entire network would lock in errors from Task 2, making accurate classification and quantity extraction impossible.

Scenario 2: Freezing Only the Transformer Backbone

Implications:

The transformer backbone remains frozen, but the task-specific heads are trained.

The model leverages pre-trained embeddings while adapting to task-specific needs.

Advantages:

- Computationally efficient - only training the heads speeds up learning.
- Retains general NLP knowledge, reducing overfitting on small datasets.
- Balances efficiency and adaptability for multi-task learning.

Disadvantages:

- If the backbone isn't well-aligned with receipts, it could limit performance.
- Task-specific heads may struggle to compensate for domain differences.

Rationale:

This approach is effective when the backbone already captures general representations that can be applied across multiple tasks. By keeping the backbone frozen, the model remains computationally efficient while still allowing the task-specific heads to adapt. However, if the backbone's representations do not align well with the target domain, fine-tuning may be necessary to ensure better task-specific performance. Also, since only the heads are being trained, the backbone remains static, meaning the model's core feature extraction does not evolve with the new tasks.

Relevance to Fetch Use Case:

This is the best approach for Fetch. Training the heads allows the model to learn receipt-specific structures while keeping inference fast, which is crucial for Fetch's high-volume processing. Since Task 2 currently produces inaccurate predictions, fine-tuning the heads can correct these errors without retraining the entire model.

Scenario 3: Freezing One Task-Specific Head

Implications:

One task-specific head remains frozen, while the backbone and the other head are trained.

Advantages:

- Helps prioritize adaptation for one task while keeping the other stable.
- Reduces overfitting risk if one task is already optimized.

Disadvantages:

- Creates imbalanced learning (one task evolves, while the other remains static).
- The frozen head may misalign with the evolving backbone, leading to inconsistencies.

Rationale:

This strategy is useful when one task is already well-optimized and doesn't require further fine-tuning. This will allow the weaker head to utilize more computational resources for training. However, if the frozen task needs to evolve, inconsistencies can emerge.

Relevance to Fetch Use Case:

This isn't ideal for Fetch. Both classification and quantity extraction require improvement (as we see from the output of task 2), so freezing one would lock in existing errors rather than correcting them. Additionally, training the backbone might contradict Fetch's requirement for computational efficiency.

Transfer Learning Considerations

When is Transfer Learning Beneficial?

Through my research reading various research papers & watching Andrew Ng's YouTube videos, I found that transfer learning is particularly useful when:

- **Labeled data is limited**, making it impractical to train a model from scratch.
- **A pre-trained model already captures useful representations**, reducing the need for extensive task-specific training.
- **Computational efficiency is a concern**, and full retraining would be too expensive.

In Fetch's case, receipt data is usually semi-structured, containing a mix of text, numbers, and positional information that general NLP models trained on unstructured text may not handle well. This means that instead of relying on a standard MTL setup, transfer learning could be used to align the model with the structure of receipts.

Scenario Where Transfer Learning is More Useful than MTL

Multi-task learning works best when different tasks share a common structure, but if the pre-trained model was never trained on structured text like receipts, MTL alone will not be enough to bridge the gap.

For example, general NLP models like MiniLM, DistilBERT, or T5 are trained on unstructured text, such as books, Wikipedia, and web data. Applying them to structured data like receipts or invoices might result in poor performance because they do not inherently recognize structured formats, numeric values, or layout information.

A better approach is to use transfer learning with a model already trained on structured documents, such as LayoutLM, which is pre-trained on scanned invoices, receipts, and structured financial documents. This means it already understands how key information is positioned, making it a much better fit than a general NLP model.

Choosing a Pre-Trained Model

***I compared two model types**:*

- General NLP Models (MiniLM, DistilBERT, T5): These are widely used but seem to lack structured text awareness and would a lot of training data and heavy fine-tuning to work well for receipts.
- Document-Aware Models (LayoutLM): These are pre-trained on documents, invoices, and receipts, making it ideal for structured text processing.

Final Choice: LayoutLM because:

a) It captures both text and layout structure, making it more effective for receipts than general NLP models.
b) It has been pre-trained on scanned financial documents, so it already recognizes itemized lists, pricing formats, and structured information.
c) If computational efficiency is a concern, MiniLM could still be used, but it would require significantly more fine-tuning on receipt-specific data.

Layers to Freeze/Unfreeze: A Phased Approach

Instead of fine-tuning the entire model, which would erase useful pre-trained knowledge, I would follow a phased fine-tuning strategy:

Phase 1: Freeze the Backbone, Train the Task-Specific Heads

- **Frozen:** The transformer backbone (to retain its pre-trained knowledge).
- **Trainable:** The classification and quantity extraction heads.

- **Why?**

a) Keeps training efficient, allowing the model to adapt without modifying the backbone.
b) Ensures that task-specific heads learn to map pre-trained embeddings to the Fetch receipt tasks.

Phase 2: If Performance is Suboptimal, Unfreeze the Top Layers

- **Unfreeze:** The top 1–2 layers of the transformer backbone.
- **Keep Frozen:** The lower layers, which contain fundamental language patterns.
- **Why?**

a) If receipt structure is too different from what the model was trained on, fine-tuning the top layers helps it specialize.
b) Keeps computational cost down while improving domain adaptation.

Phase 3: Full Fine-Tuning (Only If Necessary)

- **Unfreeze:** More layers progressively if further adaptation is needed.
- **Why?**

a) Ensures better adaptation to receipt data while avoiding overfitting.
b) Only necessary if previous steps don't sufficiently align the model with Fetch's needs.

My Rationale Behind These Choices:

Fine-tuning in phases is the best approach for structured text tasks because it:

- **Saves computing** - Training only the heads first avoids unnecessary fine-tuning.
- **Adapts gradually** - Unfreezing layers in stages prevents overfitting and ensures stability.
- **Preserves useful pre-trained knowledge** - A full fine-tune might erase too much of what the model already learned.

For Fetch's receipt processing, this phased approach would allow the model to classify receipt items and extract quantities more accurately while keeping training efficient.