task2,3,4

March 23, 2025

1 Task 2: Multi-Task Learning Expansion

[1]:
```
import numpy as np
import torch
import torch.nn as nn
from torch.utils.data import Dataset, DataLoader
from transformers import AutoModel, AutoTokenizer
from typing import Any, Optional
```

[2]:
```
class SentenceTransformerMultiTask(nn.Module):
    def __init__(self,
                 model_name: str = "distilbert-base-uncased",
                 embedding_dim: int = 768,
                 pooling_strategy: str = "mean",
                 max_length: int = 128,
                 num_sentiment_classes: int = 3) -> None:
        super(SentenceTransformerMultiTask, self).__init__()
        self.tokenizer = AutoTokenizer.from_pretrained(model_name)
        self.transformer = AutoModel.from_pretrained(model_name)
        self.pooling_strategy = pooling_strategy
        self.max_length = max_length
        self.transformer_dim = self.transformer.config.hidden_size

        self.use_projection = None
        if embedding_dim != self.transformer_dim:
            self.use_projection = True
        else:
            self.use_projection = False

        if self.use_projection is True:
            self.projection = nn.Linear(self.transformer_dim, embedding_dim)
            self.shared_dim = embedding_dim
        else:
            self.shared_dim = self.transformer_dim

        self.sentiment_classifier = nn.Sequential(nn.Linear(self.shared_dim,␣
↪256), nn.ReLU(), nn.Dropout(0.1), nn.Linear(256, num_sentiment_classes))
```

```
        self.sentiment_labels = ["negative", "neutral", "positive"]

    def _extract_features(self, sentences: list[str]) -> torch.Tensor:
        inputs = self.tokenizer(sentences, return_tensors="pt", padding=True,␣
↪truncation=True, max_length=self.max_length)
        device = next(self.transformer.parameters()).device
        inputs = {k: v.to(device) for k, v in inputs.items()}
        outputs = self.transformer(**inputs)
        hidden_states = outputs.last_hidden_state

        if self.pooling_strategy == "cls":
            pooled = hidden_states[:, 0]
        elif self.pooling_strategy == "max":
            pooled = torch.max(hidden_states, dim=1)[0]
        else:
            attention_mask = inputs["attention_mask"].unsqueeze(-1)
            pooled = torch.sum(hidden_states * attention_mask, dim=1) / torch.
↪sum(attention_mask, dim=1)

        if self.use_projection is True:
            pooled = self.projection(pooled)

        return pooled

    def forward(self, sentences: list[str], task: Optional[str] = None) ->␣
↪dict[str, torch.Tensor]:
        shared_features = self._extract_features(sentences)
        outputs = {}

        if task is None or task == "embedding":
            outputs["embedding"] = shared_features

        if task is None or task == "sentiment":
            sentiment_logits = self.sentiment_classifier(shared_features)
            outputs["sentiment"] = sentiment_logits

        return outputs

    def encode(self, sentences: list[str], batch_size: int = 32, normalize:␣
↪bool = False) -> np.ndarray:
        self.eval()
        all_embeddings = []
        with torch.no_grad():
            for i in range(0, len(sentences), batch_size):
                batch = sentences[i : i + batch_size]
                outputs = self.forward(batch, task="embedding")
                embeddings = outputs["embedding"]
```

```
                if normalize is True:
                    embeddings = nn.functional.normalize(embeddings)
                all_embeddings.append(embeddings.cpu().numpy())
        all_embeddings = np.vstack(all_embeddings)
        return all_embeddings

    def predict_sentiment(self, sentences: list[str], batch_size: int = 32) ->␣
↪dict[str, list[str]]:
        self.eval()
        all_predictions = []
        all_probabilities = []
        with torch.no_grad():
            for i in range(0, len(sentences), batch_size):
                batch = sentences[i : i + batch_size]
                outputs = self.forward(batch, task="sentiment")
                logits = outputs["sentiment"]
                probabilities = torch.softmax(logits, dim=1)
                predictions = torch.argmax(probabilities, dim=1)
                all_predictions.extend(predictions.cpu().numpy())
                all_probabilities.append(probabilities.cpu().numpy())
        prediction_labels = [self.sentiment_labels[pred] for pred in␣
↪all_predictions]
        all_probabilities = np.vstack(all_probabilities)
        results = {"labels": prediction_labels, "probabilities":␣
↪all_probabilities}

        return results
```

[3]:
```
class SentimentDataset(Dataset):
    def __init__(self, texts: list[str], labels: list[int], tokenizer,␣
↪max_length: int = 128) -> None:
        self.texts = texts
        self.labels = labels
        self.tokenizer = tokenizer
        self.max_length = max_length

    def __len__(self) -> int:
        return len(self.texts)

    def __getitem__(self, idx: int) -> dict[str, str]:
        text = self.texts[idx]
        label = self.labels[idx]
        item = {"text": text, "label": label}
        return item
```

[4]:
```python
def train_multitask_model(
    model: SentenceTransformerMultiTask,
    train_texts: list[str],
    train_labels: list[int],
    val_texts: Optional[list[str]] = None,
    val_labels: Optional[list[int]] = None,
    epochs: int = 5,
    batch_size: int = 16,
    learning_rate: float = 2e-5) -> tuple[SentenceTransformerMultiTask,␣
↪dict[str, list[float]]]:

    assert isinstance(epochs, int), "'epochs' must be an integer."
    assert epochs > 0, "'epochs' must be positive."

    train_dataset = SentimentDataset(train_texts, train_labels, model.tokenizer)
    train_loader = DataLoader(train_dataset, batch_size=batch_size,␣
↪shuffle=True)
    if val_texts is not None and val_labels is not None:
        val_dataset = SentimentDataset(val_texts, val_labels, model.tokenizer)
        val_loader = DataLoader(val_dataset, batch_size=batch_size)
    else:
        val_loader = None

    optimizer = torch.optim.AdamW(model.parameters(), lr=learning_rate)
    criterion = nn.CrossEntropyLoss()

    device = next(model.parameters()).device
    model.train()

    training_stats = {
        "train_losses": [],
        "val_losses": [],
        "val_accuracies": []
    }

    for epoch in range(epochs):
        print(f"===== Epoch {epoch + 1} / {epochs} =====")
        model.train()
        train_loss = 0.0

        for batch in train_loader:
            texts = batch["text"]
            labels = batch["label"]
            labels = labels.to(device)
            outputs = model(texts, task="sentiment")
            logits = outputs["sentiment"]
            loss = criterion(logits, labels)
```

```
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()
            train_loss += loss.item()

        avg_train_loss = train_loss / len(train_loader)
        training_stats["train_losses"].append(avg_train_loss)

        print(f"Train Loss: {avg_train_loss:.4f}")

        if val_loader is True:
            model.eval()
            val_loss = 0.0
            correct = 0
            total = 0

            with torch.no_grad():
                for batch in val_loader:
                    texts = batch["text"]
                    labels = batch["label"]
                    labels = labels.to(device)

                    outputs = model(texts, task="sentiment")
                    logits = outputs["sentiment"]
                    loss = criterion(logits, labels)
                    val_loss += loss.item()
                    predictions = torch.argmax(logits, dim=1)
                    total += labels.size(0)
                    correct += (predictions == labels).sum().item()

            avg_val_loss = val_loss / len(val_loader)
            val_accuracy = correct / total
            training_stats["val_losses"].append(avg_val_loss)
            training_stats["val_accuracies"].append(val_accuracy)

            print(f"Validation Loss: {avg_val_loss:.4f}\nAccuracy:␣
↪{val_accuracy:.4f}")

    return model, training_stats
```

[5]:
```
def main() -> None:
    test_model =␣
↪SentenceTransformerMultiTask(model_name="distilbert-base-uncased",␣
↪embedding_dim=384, pooling_strategy="mean", max_length=128,␣
↪num_sentiment_classes=3)
    test_sentences = [
        "This is a simple sentence to encode.",
```

```
        "The quick brown fox jumps over the lazy dog.",
        "I absolutely love this product, it's amazing!",
        "This movie was terrible and a complete waste of time.",
        "The service was okay, nothing special but not bad either."
    ]

    print("GENERATING SENTENCE EMBEDDINGS:")
    test_embeddings = test_model.encode(sentences=test_sentences,␣
↪normalize=True)
    print(f"Generated {test_embeddings.shape[0]} embeddings with dimension␣
↪{test_embeddings.shape[1]}")
    print(f"Sample of the first embedding vector:\n{test_embeddings[0][:10]}")
    print("PREDICTING SENTIMENT:")
    test_sentiment_results = test_model.predict_sentiment(test_sentences)

    for i, (text, label) in enumerate(zip(test_sentences,␣
↪test_sentiment_results["labels"])):
        probs = test_sentiment_results["probabilities"][i]
        print(f"Text: {text}")
        print(f"Predicted sentiment: {label}")
        print(f"Class probabilities: Negative = {probs[0]:.4f}, Neutral =␣
↪{probs[1]:.4f}, Positive = {probs[2]:.4f}\n")

    print("TRAINING EXAMPLE (with dummy data):")
    test_train_texts = [
        "I love this product.",
        "This is terrible.",
        "It's okay I guess.",
        "Best purchase ever!",
        "Complete waste of money.",
        "It works as expected."
    ]
    test_train_labels = [2, 0, 1, 2, 0, 1]
    print("Training the model on sentiment analysis task...")
    test_model, test_stats = train_multitask_model(
        model=test_model,
        train_texts=test_train_texts,
        train_labels=test_train_labels,
        epochs=20,
        batch_size=2)

    print("Re-testing sentiment prediction after training:")
    test_sentiment_results = test_model.predict_sentiment(test_sentences)
    for i, (text, label) in enumerate(zip(test_sentences,␣
↪test_sentiment_results["labels"])):
        probs = test_sentiment_results["probabilities"][i]
        print(f"Text: {text}")
```

```
        print(f"Predicted sentiment: {label}")
        print(f"Class probabilities: Negative = {probs[0]:.4f}, Neutral =␣
 ↪{probs[1]:.4f}, Positive = {probs[2]:.4f}\n")

if __name__ == "__main__":
    main()
```

```
GENERATING SENTENCE EMBEDDINGS:
Generated 5 embeddings with dimension 384
Sample of the first embedding vector:
[-0.07832588 -0.05834525 -0.02594784 -0.06107131 -0.05892345 -0.03642315
  0.00334449 -0.0333345  -0.06209832 -0.01038289]
PREDICTING SENTIMENT:
Text: This is a simple sentence to encode.
Predicted sentiment: neutral
Class probabilities: Negative = 0.3134, Neutral = 0.3492, Positive = 0.3374

Text: The quick brown fox jumps over the lazy dog.
Predicted sentiment: neutral
Class probabilities: Negative = 0.3280, Neutral = 0.3449, Positive = 0.3271

Text: I absolutely love this product, it's amazing!
Predicted sentiment: neutral
Class probabilities: Negative = 0.3333, Neutral = 0.3340, Positive = 0.3327

Text: This movie was terrible and a complete waste of time.
Predicted sentiment: neutral
Class probabilities: Negative = 0.3286, Neutral = 0.3436, Positive = 0.3278

Text: The service was okay, nothing special but not bad either.
Predicted sentiment: neutral
Class probabilities: Negative = 0.3293, Neutral = 0.3395, Positive = 0.3312

TRAINING EXAMPLE (with dummy data):
Training the model on sentiment analysis task…
===== Epoch 1 / 20 =====
Train Loss: 1.0930
===== Epoch 2 / 20 =====
Train Loss: 1.0731
===== Epoch 3 / 20 =====
Train Loss: 1.0200
===== Epoch 4 / 20 =====
Train Loss: 0.9765
===== Epoch 5 / 20 =====
Train Loss: 0.9094
===== Epoch 6 / 20 =====
Train Loss: 0.8373
===== Epoch 7 / 20 =====
```

```
Train Loss: 0.7890
===== Epoch 8 / 20 =====
Train Loss: 0.6959
===== Epoch 9 / 20 =====
Train Loss: 0.6246
===== Epoch 10 / 20 =====
Train Loss: 0.5809
===== Epoch 11 / 20 =====
Train Loss: 0.5192
===== Epoch 12 / 20 =====
Train Loss: 0.4555
===== Epoch 13 / 20 =====
Train Loss: 0.4209
===== Epoch 14 / 20 =====
Train Loss: 0.3558
===== Epoch 15 / 20 =====
Train Loss: 0.2918
===== Epoch 16 / 20 =====
Train Loss: 0.2648
===== Epoch 17 / 20 =====
Train Loss: 0.2414
===== Epoch 18 / 20 =====
Train Loss: 0.2148
===== Epoch 19 / 20 =====
Train Loss: 0.1912
===== Epoch 20 / 20 =====
Train Loss: 0.1768
Re-testing sentiment prediction after training:
Text: This is a simple sentence to encode.
Predicted sentiment: neutral
Class probabilities: Negative = 0.1800, Neutral = 0.6258, Positive = 0.1942

Text: The quick brown fox jumps over the lazy dog.
Predicted sentiment: neutral
Class probabilities: Negative = 0.3186, Neutral = 0.3665, Positive = 0.3149

Text: I absolutely love this product, it's amazing!
Predicted sentiment: positive
Class probabilities: Negative = 0.0785, Neutral = 0.0801, Positive = 0.8414

Text: This movie was terrible and a complete waste of time.
Predicted sentiment: negative
Class probabilities: Negative = 0.8016, Neutral = 0.0925, Positive = 0.1058

Text: The service was okay, nothing special but not bad either.
Predicted sentiment: neutral
Class probabilities: Negative = 0.1670, Neutral = 0.6690, Positive = 0.1640
```

1.1 Description for Task 2

I've built task 2 on top of the code from task 1 to handle multi-task learning by implementing sentiment analysis on the embeddings. Sentiement analysis is a concept in NLP that allows us to analyze and classify the emotional states from text or voice inputs. In this case, we are focusing on the text part.

The fundamental part of building the multi-task learning expansion is the creation of a **shared representation layer**. It captures the natural language understanding (NLU), while also turned the transformer backbone and pooling layers into shared feature extractors. All the task-specific heads are able to use the shared features. More importantly, this approach explains the principle that learning multiple related tasks simultaneously can improve the quality of the shared representation layer.

For each of the two tasks (embeddings and sentiment analysis), components were added to adjust and reinforce their features. As for encoding embeddings in the `encode` function, it now uses the shared representations directly on top of its original functionality. As for the sentiment analysis, a new two-layer neural network (NN, using `torch.nn`) was created with 256 hidden units and ReLU activation. A dropout regularization rate was set to 0.1 to prevent overfitting. The neural network also has an output layer producing logits for 3 sentiment classes (negative, neutral, positive).

By adding more features to the model, it now allows users to select which task to execute. This is seen by the `task` parameter in the `forward` function, which takes a string. Additionally, the outputs are stored in a dictionary (variable `outputs`) with keys for each task.

Two essential new functions were added to the model. `predict_sentiment` is the function that predicts the sentiment in a given sentence. It's a dedicated inference method for sentiment analysis that returns both predicted labels and class probabilities. It's also capable of performing batch processing for efficient inference on multiple sentences, as well as converting numeric predictions to pre-defined string labels. `train_multitask_model` is the main function for the training process. It sets up the optimization process by using the AdamW optimizer in PyTorch and contains the implementation of the training and validation loops. It allows users to input the number of epochs for training and keeps track of the loss and accuracy values.

Finally, to support the training process, a `SentimentDataset` class was created by using the PyTorch Dataset interface with inheritance.

2 Task 3: Training Considerations

2.1 1. If the entire network should be frozen.

If the entire network is frozen, then we are only left with using the model for inference without any adaptation to new data.

2.1.1 Implications:

- Models relies entirely on knowledge from pre-trained weights
- Sentiment analysis tasks would be limited by the rate of alignment between the pre-trained model's representation space and the sentiment concepts
- The quality of embeddings would remain constant, thus preserving the original semantic relationships

2.1.2 Advantages:

- Very fast execution time during the training phase (since there's no actual training)
- No risk of losing pre-trained knowledge
- Consistent and predictable behavior across different datasets
- Minimum requirements for computational power

2.1.3 We should use this method if:

- We have extremely limited computational power
- Our domain is almost identical to the pre-trained data
- We need absolute consistency in representations

2.2 2. If only the transformer backbone should be frozen.

2.2.1 Implications:

- Shared natural language understanding (NLU) remains fixed
- Model learns how to interpret the existing representations for specific tasks
- Creates a difference between "knowledge for natural languages" and "knowledge for specific tasks"

2.2.2 Advantages:

- Preserves knowledge related to linguistics in pre-trained model
- Significantly reduces execution time during the training process and requirements for computational power
- Reduces the risk on overfitting for small datasets

2.2.3 We should use this method if:

- We are low on computational power (but more than that stated in 1.)
- We have some data on task-specific tasks, but it's not enough to fine-tune a model safely
- We want to frequently add new tasks but don't want to retrain all data
- We want to quickly adapt to new tasks while maintaining a consistent representation space

2.3 3. If only one of the task-specific heads should be frozen.

2.3.1 Implications:

- Model would optimize its representations primarily for the embedding task
- Classifier for sentiment analysis needs to adapt to shifting representations as the training elapses
- Fixed classifier enforces constraints on the amount which the shared representation can drift

2.3.2 Advantages:

- Optimization on backbone particularly for high-quality embeddings
- May improve embedding performance
- Maintains some level of functionality for sentiment analysis

2.3.3 We should use this method if:

- Our main goal is to optimize the quality for embeddings
- Sentiment analysis is not our main goal
- We have a well-trained sentiment classifier which we want to maintain

2.4 1. The choice of a pre-trained model.

First, I would choose a model such as RoBERTa-base or MPNet-base for the following reasons:

- RoBERTa-base could make improvements on BERT since it contains more robust training methodology and larger data for training
- MPNet-base combines the strengths of masked language modeling and permuted language modeling, which makes it beneficial in transfer training
- Both models have shown their strengths in natural language understanding (NLU)
- Both models provide a balance between performance and computational burden

2.5 2. The layers you would freeze/unfreeze.

I would use a gradual unfreezing strategy for the transfer learning process. At the beginning, I would freeze the transformer backbone and run training tasks on task-specific heads for less than 10 epochs. Then, I would unfreeze the top layers of the transformer and continue training with a reduced learning rate. Finally, I would gradually unfreeze more layers and train them with smaller learning rates.

2.6 3. The rationale behind these choices.

This gradual unfreezing strategy that I intended to use contains the following benefits:

- Specialization on layers' position: In a transformer model, layers located at the front would capture more patterns on linguistic characteristics, whereas layers at the back would encode information more on specific tasks. By adopting this gradual unfreezing strategy from top to bottom, we could make the most task-relevant layers adapt first.
- Prevention for knowledge loss: The gradual unfreezing strategy could help prevent the model from losing pre-trained knowledge in the middle of the training process.
- Increase of efficiency on training data: This strategy also helps to increase the use of limited task-specific data by utilizing the parameters related to adaptation at the beginning.
- Balance on multi-tasking: Being able to control the amount of change on shared representations allows us to create a balance in performance among multiple tasks. If one tasks is consuming an excessive amount of computational power, the learning rates can be adjusted to resolve the situation.

3 Task 4: Training Loop Implementation (BONUS)

3.1 1. Handling of hypothetical data

Compare to the single-task learning model I created in task 1, the multi-task learning model in task 2 is more complex and requires more consideration on handling data. First, it would require data balance between tasks. If we were to train this model, balancing the training signal between tasks if one has significantly more data would be one of the points we need to consider. The current

architecture assumes that learning to perform sentiment analysis and generating quality embeddings complement each other and can benefit from shared representations. Lastly, to maintain the quality of the embeddings, we might need to develop a back-translation technique to boost the semantic meaning for emotional words in sentences.

3.2 2. Forward pass

The `forward` function I implemented takes a `task` parameter as a string. Depending on the content, it can tell the `forward` function to perform different tasks:

- `None`: run all tasks
- `"embedding"`: generate embeddings only
- `"sentiment"`: run sentiment analysis only

By designing the `forward` function like this, we could increase the overall efficiency by running tasks that are needed by the user, while also enabling joint training where all tasks are running simultaneously.

Also, during backpropagation, the gradients would flow through different channels. If only the sentiment analysis task is run, gradients would flow through the sentiment classifier and then into the shared layers; however, if both tasks are run, gradients from both tasks influence the shared representations. By doing so, we create an implicit weighting of tasks based on their loss magnitudes.

3.3 3. Metrics

My implementation also tracks metrics separately for each task. In sentiment analysis, we track the values for **loss and accuracy**. Currently, there's no track being tracked when generating embeddings, but if we were to turn the code into a real implementation, we would need to track metrics that are related embeddings as well, including sentiment accuracy, cosine similarity between semantically related sentences, correlation in performance between one task and another, etc. The current training loop for the Multi-Task Learning Expansion focuses primarily on the **classification** of sentences. But in a real-world implementation, we would need to collect the total loss by calculating the summation from sentiment loss and embedding loss.

Another feature that I didn't demonstrate is validation. In a real-world machine learning pipeline, validation data needs to be supplied from a reliable source and cannot be fabricated or made up. The validation set serves as a critical independent check on a model's performance and helps to detect issues like overfitting. What makes this even more challenging is the fact that different tasks could reach their respective point of best performance at different time. The current code contains validation metrics for sentiment analysis, but a real-world implementation would need to have validation metrics for all tasks.

[ ]: