Table Of Contents

Training Strategy..... 2
Implementation Details:..... 3
KnowledgeDistillationTrainer Class:..... 3
Metrics Tracking.....4
Learning Rate Scheduling..... 5
Gradient Clipping..... 5
Model Tuning.....6
Temperature = 3.....6
Alpha = 0.7.....6
Results Interpretation.....7
Baseline - FineTuned - Supervised Learning.....7
Teacher Model..... 7
Student Model Distilled (Alpha = 0.7, Temp = 3)..... 7
F-1 Score Comparison Across all Models..... 8
Student (distilled) vs Baseline.....8
Student(distilled) vs Teacher.....8
Pipeline & Code Quality:.....9
TextDataset.....9
DataManager Class..... 10
Device Configuration..... 10
Model Manager Class..... 11
KnowledgeDistillationTrainer Class..... 12
KD Pipeline..... 13
Conceptual Alternatives.....14
Alternative Approaches.....14
Feature Based Transfer..... 14
Multi-teacher Distillation.....14
Learnings..... 14
Limitations.....15
Future Work..... 15

Training Strategy

Describe the methodology you researched and implemented to improve the student model using the teacher. What principles or specific techniques of Knowledge Distillation) did you apply? Explain the core mechanics of your implementation and justify your design choices.

The core principle for this training strategy was to maximize the knowledge transfer from the Teacher to the student model. The technique used was **response based distillation**. This allows the student model to learn from the final predictions (responses) of the teacher model rather than from the intermediate features or attention patterns. This is achieved using a loss function called as distillation loss that captures the difference between the logits of the student model and the teacher model. As the loss is minimized over training, the student model becomes better at making the same predictions as the teacher. The core mechanism behind this training strategy is the cumulative loss function.

$$L_{total} = \alpha * L_{CE} + (1 - \alpha) * L_{KD}(T)$$

$L_{KD}(T)$ = Temperature based Kullback-Leibler (KL) divergence loss

α = Distillation weight hyperparameter

L_{CE}= Standard cross-entropy loss on hard labels

T = Temperature scaling parameter

Knowledge distillation can be implemented through several other techniques such as feature-based, similarity-based, and discrete output-based methods. Feature-based distillation transfers knowledge by aligning hidden representations between teacher and student models, while similarity-based distillation focuses on preserving the relational structure between data points making it suitable for retrieval, ranking, or clustering tasks. Discrete output-based distillation uses the teacher's hard label predictions as targets, treating them as pseudo-ground truth, but it discards the valuable soft class probability distribution available in the teacher's output.

In contrast, response-based distillation is often more effective because it retains the full soft probability distribution over classes, providing the student with richer guidance. This method is architecture-agnostic and does not require alignment of internal layers or computation of pairwise similarities, making it easier to implement and more versatile. It also offers softer learning signals than discrete label methods, helping the student model generalize better. Despite the strengths of other techniques in specific use cases, response-based distillation offers the best trade-off between information depth, simplicity, and flexibility.

Implementation Details:

Guide us through the key code sections related to your knowledge transfer mechanism and training loop. What were the most significant implementation challenges, and how did you overcome them?

KnowledgeDistillationTrainer Class:

Python

```
    def train_epoch(self, train_loader: DataLoader) -> Tuple[float, float,
float, float]:
        """Train for one epoch (hard + soft labels)."""
        self.student_model.train()
        for batch in train_loader:
            input_ids, attention_mask, targets = [t.to(self.device) for t in
batch]

            # 1) Student forward pass → logits
            student_logits = self.student_model(
                input_ids=input_ids,
                attention_mask=attention_mask
            ).logits

            # 2) Hard-label loss (CE)
            ce_loss = self.entropy_loss(student_logits, targets)

            # 3) Teacher forward pass (no grad) → soft logits
            with torch.no_grad():
                teacher_logits = self.teacher_model(
                    input_ids=input_ids,
                    attention_mask=attention_mask
                ).logits

            # 4) KD loss (KLDiv on temperature-scaled distributions)
            kd_loss = (self.temperature ** 2) * self.kd_loss_fn(
                F.log_softmax(student_logits / self.temperature, dim=-1),
                F.softmax(teacher_logits / self.temperature, dim=-1)
            )

            # 5) Combined loss = α·CE + (1-α)·KD
            loss = self.alpha * ce_loss + (1. - self.alpha) * kd_loss
            loss.backward()
```

In each training step, the student model's predictions are compared to both the hard labels (via cross-entropy loss) and the soft targets from the teacher model (via KL divergence between softened logits). These two losses are combined using a weighted sum controlled by the alpha parameter, allowing the student to benefit from both traditional supervision and the teacher's richer output distribution.

During each training cycle, the student model's predictions are evaluated against both the actual classifications (using cross-entropy loss) and the teacher model's probability distributions (using KL divergence of the smoothed outputs). A weighted sum, governed by the alpha parameter, combines these two errors, enabling the student to learn from direct labels and the teacher's more informative output distribution.

Metrics Tracking

Python

```python
def print_detailed_metrics(self, epoch, total_epochs, train_loss,
                           train_kd_loss, train_ce_loss, train_accuracy,
                           valid_loss, valid_accuracy):
    print(f"""
    Epoch {epoch + 1}/{total_epochs}:
    ├─ Combined Loss: {train_loss:.4f}
    ├─ KD Loss: {train_kd_loss:.4f}
    ├─ CE Loss: {train_ce_loss:.4f}
    ├─ KD/CE Ratio: {train_kd_loss/train_ce_loss:.2f}  # Critical monitoring
    ├─ Train Accuracy: {train_accuracy:.4f}
    └─ Valid Accuracy: {valid_accuracy:.4f}
    """)


Epoch 1/3: KD/CE Ratio: 3.24  # High ratio indicates teacher guidance dominance
Epoch 2/3: KD/CE Ratio: 1.43  # Decreasing ratio shows convergence
Epoch 3/3: KD/CE Ratio: 1.16  # Balanced learning achieved
```

The above block ensures that the teacher model's forward pass is executed without tracking gradients, which optimizes memory usage and prevents unnecessary computation. This improves computational efficiency by reducing memory overhead, while also ensuring the teacher **model remains frozen** and does not update during student training.

Learning Rate Scheduling

```python
def setup_optimizer_scheduler(self, train_loader: DataLoader):

   """Setup optimizer and learning rate scheduler."""

       self.optimizer = optim.AdamW(self.student_model.parameters(),
       lr=self.learning_rate)
           self.scheduler = OneCycleLR(
               self.optimizer,
               max_lr=self.learning_rate,
               total_steps=len(train_loader) * self.epochs,
               pct_start=0.1,  # Quick warm-up for distillation
               anneal_strategy='cos'  # Smooth convergence
           )
```

The **OneCycleLR** scheduler dynamically adjusts the learning rate, starting with a quick warm-up (pct_start=0.1) and gradually reducing it using a cosine annealing strategy, which helps prevent overfitting and improves generalization. By computing total_steps from the training data and epochs, the learning rate schedule is precisely aligned with the training duration, ensuring consistent optimization behavior across different runs.

Gradient Clipping

```python
# Add gradient clipping for stability
torch.nn.utils.clip_grad_norm_(self.student_model.parameters(),
1.0)

#Shows stable training (please see the cell output to verify)
Epoch 1: Train Accuracy: 0.8985 → Valid Accuracy: 0.9369
Epoch 2: Train Accuracy: 0.9537 → Valid Accuracy: 0.9463
Epoch 3: Train Accuracy: 0.9647 → Valid Accuracy: 0.9480
```

Gradient clipping is applied here to prevent exploding gradients, which can destabilize training—especially important when distilling from large teacher models. By capping the total gradient norm to 1.0, this line ensures smoother and more controlled updates.

Model Tuning

Discuss the key hyperparameters relevant to your chosen method and how you selected their values (including standard ones like learning rate, epochs).

For training the student model the following hyperparameters were selected.

Hyperparameter	Value
epochs	3
learning_rate	2e-5
temperature	3
alpha	0.7
Gradient clipping	Max_norm = 1

Temperature = 3

The temperature parameter in KD provides a way to control the amount of information the student model learns from the teacher model. It influences the shape of the teacher model's probability distribution, affecting how knowledge is transferred to the student model. The temperature parameter of 3 was chosen as a balanced approach between information extraction and training stability for 3 epochs. This moderate temperature value provides sufficient softening of the teacher's probability distributions to reveal inter-class relationships while avoiding the numerical instability.

Alpha = 0.7

The parameter alpha is a weighting factor that balances the contribution of two loss components and determines how much emphasis is placed on the hard labels (cross-entropy with ground truth). So in this case 70% weight is allocated to cross entropy loss and 30% weight is assigned to knowledge distillation loss (KL divergence). If the alpha value is too large then there is reduced benefit from the teacher's knowledge. The student will ignore the soft target distributions which contain cues about class relationships. Also the impact of KD becomes negligible. If it's too low then the student model does not learn from ground truth labels and solely relies on teachers knowledge which can be risky.

$$L_{total} = \alpha * L_{CE} + (1 - \alpha) * L_{KD}(T)$$

Results Interpretation

Analyze your comparison table. How successful was your strategy in improving the student model beyond its baseline? How did it compare to the teacher? Discuss the resulting trade-offs between performance and model efficiency.

Yellow highlighted shows the metrics and classes where **Student (KD)** outperformed both base line and the teacher model

Baseline - FineTuned - Supervised Learning

Class	Precision	Recall	F1
World	0.9617	0.9516	0.9566
Sports	0.9874	0.9863	0.9868
Business	0.9212	0.9111	0.9161
Sci/Tech	0.9094	**0.9300**	0.9196

Teacher Model

Class	Precision	Recall	F1
World	0.9471	0.9526	0.9499
Sports	0.9736	**0.9916**	0.9825
Business	0.9091	0.9000	0.9045
Sci/Tech	**0.9188**	0.9058	0.9123

Student Model Distilled (Alpha = 0.7, Temp = 3)

Class	Precision	Recall	F1
World	**0.9644**	**0.9547**	**0.9595**
Sports	0.9874	0.9884	**0.9879**
Business	**0.9216**	**0.9153**	**0.9184**
Sci/Tech	0.9150	0.9295	**0.9222**

F-1 Score Comparison Across all Models

Another way to look at the results is to see the F-1 Score across all models

Class	Baseline	Teacher	Student (Distilled)
World	0.9566	0.9499	**0.9595**
Sports	0.9868	0.9825	**0.9879**
Business	0.9161	0.9045	**0.9184**
Sci/Tech	0.9196	0.9123	**0.9222**

Student (distilled) vs Baseline

The Student Model outperforms both Baseline and Teacher for all 4 classes in F1-score. Since, The baseline model was just trained with a standard supervised learning approach, on the other hand the Student (distilled) model gained richer experience from the ground truth labels and teacher model's knowledge. Improvements can be seen in all classes with noticeable increase in Business and Sci/Tech categories. This shows that the distillation process with temperature scaling helped achieve generalization.

Student(distilled) vs Teacher

Compared to the teacher model the student (distilled) also shows better performance in all categories. This is due to dual supervision, the training loop shows that the student model receives two learning signals. One from the hard labels in the cross entropy loss and the other through the soft labels in KL divergence loss. The loss function is combination of hard label and soft label supervision as shown below

```python
# Hard label supervision
ce_loss = entropy_loss(student_logits, target_tensors)

# Soft label supervision
kd_loss = temperature ** 2 * criterion(
    F.log_softmax(student_logits / temperature, dim=-1),
    F.softmax(teacher_logits / temperature, dim=-1)
)
# Combined:
loss = alpha * ce_loss + (1. - alpha) * kd_loss
```

Pipeline & Code Quality:

Discuss the structure, clarity, and potential reusability of your training pipeline code.

The code is organized keeping in mind the modular design It has the following components

TextDataset

```python
class TextDataset(Dataset):
    """Custom dataset class for text classification with tokenization."""

    def __init__(self, data: Any, tokenizer: Any, max_length: int = 150):
        """
        Initialize the dataset.

        Args:
            data: Dataset containing 'label' and 'text' fields
            tokenizer: Tokenizer for text processing
            max_length: Maximum sequence length for tokenization
        """
        self.targets = torch.tensor(data['label'])
        texts = data['text']

        tokens = tokenizer(
            texts,
            return_tensors='pt',
            truncation=True,
            padding='max_length',
            max_length=max_length
        )
....
```

The **TextDataset** class is designed to work with datasets that include text and label fields, which aligns perfectly with AG News. It handles tokenization, padding, and attention masks, making it suitable for any similar classification dataset. We can reuse it with minimal changes for other HuggingFace datasets like Yelp or DBpedia.

DataManager Class

```python
class DataManager:
    """Manages data loading and preprocessing for knowledge distillation."""

  def __init__(self, dataset_name: str, tokenizer: Any, test_size: float = 0.2,
                 max_length: int = 150, batch_size: int = 32):
        """
        Initialize data manager.

        Args:
            dataset_name: Name of the dataset to load
            tokenizer: Tokenizer for text processing
            test_size: Fraction of data to use for validation
            max_length: Maximum sequence length
            batch_size: Batch size for data loaders
        """
        self.dataset_name = dataset_name
        self.tokenizer = tokenizer
......
```

DataManager encapsulates loading, preprocessing, splitting, and batching the AG News dataset using HuggingFace's load_dataset function. Its flexible design allows you to switch to other datasets by simply changing the dataset name. This makes it reusable for a wide range of classification tasks with similar structures.

Device Configuration

```python
if torch.backends.mps.is_available():
    device = torch.device("mps")  # Use Apple GPU via Metal
elif torch.cuda.is_available():
    device = torch.device("cuda")
else:
    device = torch.device("cpu")

print(f"Using device: {device}")
```

The device selection logic automatically detects MPS, CUDA, or CPU, ensuring your code runs optimally across hardware platforms. This is universally applicable to any PyTorch project, including AG News and beyond. It's a drop-in snippet that eliminates the need for manual hardware configuration.

Model Manager Class

```python
class ModelManager:
    """Manages teacher and student models for knowledge distillation."""

    def __init__(self, teacher_model_name: str, student_model_name: str,
                 num_labels: int, device: torch.device, dropout_rate: float =
0.2):
        """
        Initialize model manager.

        Args:
            teacher_model_name: Name/path of teacher model
            student_model_name: Name/path of student model
            num_labels: Number of classification labels
            device: Device to load models on
            dropout_rate: Dropout rate for student model
        """
        self.teacher_model_name = teacher_model_name
        self.student_model_name = student_model_name
.......

    def load_models(self) -> Tuple[nn.Module, nn.Module]:
        """
        Load teacher and student models.

        Returns:
            Tuple of (student_model, teacher_model)
        """
        # Load student model
        self.student_model= AutoModelForSequenceClassification.from_pretrained(
            self.student_model_name,
            num_labels=self.num_labels
        ).to(self.device)
        self.student_model.dropout = nn.Dropout(self.dropout_rate)
.........
```

The ModelManager class is a modular component that encapsulates the loading and configuration of both teacher and student models for knowledge distillation tasks. It accepts model names, the number of output labels, and the target device, making it flexible for various classification problems like AG News. By centralizing model initialization and dropout adjustment, it improves reusability and simplifies switching between architectures (e.g., BERT → DistilBERT).

KnowledgeDistillationTrainer Class

Python

```
class KnowledgeDistillationTrainer:
    """Main trainer class for knowledge distillation."""

    def __init__(self, student_model: nn.Module, teacher_model: nn.Module,
                 device: torch.device, temperature: float = 3.0, alpha: float =
0.7,
                 learning_rate: float = 2e-5, epochs: int = 3):
        """
        ......
        """
        self.student_model = student_model
        self.teacher_model = teacher_model
        self.device = device
        ........
        # Loss functions
        self.entropy_loss = nn.CrossEntropyLoss()
        self.kd_loss_fn = nn.KLDivLoss(reduction='batchmean')

        # Metrics tracker
        self.metrics_tracker = MetricsTracker()

    def accuracy_score(self, batch: Tuple[torch.Tensor,
torch.Tensor,torch.Tensor],
                       model: nn.Module) -> float:
              """Calculate accuracy for a batch."""
              with torch.no_grad():
                  outputs = model(batch[0].to(self.device),
      batch[1].to(self.device))
                  logits = outputs.logits

   def setup_optimizer_scheduler(self, train_loader: DataLoader):
        """Setup optimizer and learning rate scheduler."""
        self.optimizer = optim.AdamW(self.student_model.parameters(),
lr=self.learning_rate)
        self.scheduler = OneCycleLR(
            self.optimizer,
            max_lr=self.learning_rate,

.....
```

```
   def train_epoch(self, train_loader: DataLoader) -> Tuple[float, float,
float, float]:
        """Train for one epoch."""
        self.student_model.train()
        train_loss = 0.0
        .....

        for batch in train_loader:
            self.optimizer.zero_grad()
            input_ids = batch[0].to(self.device)
            attention_mask = batch[1].to(self.device)
            target_tensors = batch[2].to(self.device)

            # Student model predictions
            student_logits = self.student_model(input_ids=input_ids,
attention_mask=attention_mask).logits

       .........
```

This class encapsulates the full training logic for distilling knowledge from a teacher model into a student model. It integrates temperature scaling, cross-entropy loss, and KL divergence loss into a unified training loop, allowing a smooth balance between hard (ground-truth) and soft (teacher) labels using the alpha parameter. This class is reusable for any transformer-based text classification task (like AG News) and can easily be adapted to different model pairs, temperatures, or loss-weighting strategies.

KD Pipeline

The full pipeline has the following components

- Prepare Data
- Load Models
- Initialize Trainer
    - Metrics Tracking
- Train Model
- Save Model

The pipeline first prepares data by downloading, splitting, tokenizing, and batching via `DataManager`, making it easy to swap in any text dataset. It then uses `ModelManager` to load teacher and student transformer models onto the chosen device. Next, `KnowledgeDistillationTrainer` is initialized with hyperparameters, loss functions, and built-in accuracy tracking to coordinate the distillation process. Calling its `train()` method runs

the combined hard-label and soft-label training loop across epochs, after which `save_model()` exports the distilled student for seamless reuse.

Conceptual Alternatives

What other approaches exist for transferring knowledge between models or improving model efficiency? What did you learn from selecting and implementing your strategy? What are its limitations? What further experiments would you run?

Alternative Approaches

There are a couple of other approaches that exist such as

Feature Based Transfer

So instead of only aligning the output logits this uses hidden representations (attention or intermediate embeddings) between student and teacher. This can lead to deeper supervision but will increase the complexity and memory usage

Multi-Teacher Distillation

Rather than a single teacher, multiple teacher models (e.g., fine-tuned on different tasks or domains) can guide the student, combining diverse forms of knowledge.

Learnings

From implementing logit-based knowledge distillation with DistilBERT as the student and BERT as the teacher, key takeaways include:

- Combining cross-entropy loss (with hard labels) and KL divergence (soft targets) using α = 0.7 ensured the student learned both class boundaries and inter-class similarities.
- Softening logits with temperature = 3 enabled smoother gradients and richer learning signals from the teacher.
- Using torch.no_grad() and gradient clipping helped prevent instability from softened logits and high-temperature values.

Limitations

- Student performance is capped by the teacher's quality. If the teacher model is suboptimal, student gains are limited.

- Since only the final logits were distilled, the student may miss out on richer internal representations unless intermediate supervision is added.

- KD performance is sensitive to α and temperature. Suboptimal choices can lead to underfitting or over-regularization.

Future Work

- Hyperparameter Grid Search (α & Temperature)
- Instead of relying solely on final logits, align hidden representations or attention maps between teacher and student.
- Leverage two or more teacher models (e.g., BERT and RoBERTa fine-tuned on related domains) to provide diverse soft targets. Combining their outputs either via ensemble logits or sequential distillation could enrich the student's learning signal and improve robustness
- Evaluate distillation onto a variety of lightweight architectures (TinyBERT, MobileBERT, DistilRoBERTa) to compare parameter-efficiency vs. performance. This helps determine which student backbone yields the best speed/accuracy trade-off for deployment.