2. Pairwise Reward Model Training

RL_ppo_train_with_reward.py implements training of a **pairwise reward model** using a dataset of human preference comparisons. The core idea is to teach a model to assign higher scores to "preferred" text outputs over "rejected" ones.

The **pairwise loss** used is:

```
L = -log σ(r_chosen - r_rejected)
```

where `r_chosen` is the predicted reward for the preferred output, and `r_rejected` is for the rejected output.

Key Parts of the Implementation

1. Configuration

The script defines model, dataset, and training parameters at the top:

```
RM_MODEL_NAME = "bert-base-uncased"
DATA_PATH = "output/pairwise_prefs_part_1_test.jsonl"
OUTPUT_DIR = "output/reward_model_pairwise"
DEVICE = "cuda" if torch.cuda.is_available() else "cpu"
BATCH_SIZE = 8
EPOCHS = 5
LR = 5e-5
MAX_LENGTH = 256
VAL_SPLIT = 0.1
CSV_LOG = os.path.join(OUTPUT_DIR, "training_log.csv")
```

This allows easy modification of model type, dataset path, batch size, and training hyperparameters.

---

2. Dataset Class

A **custom PyTorch Dataset** loads the JSONL file containing pairs of preferred and rejected texts. Each item is tokenized and returned as input tensors:

```
class PairwiseDataset(Dataset):
    def __init__(self, path, tokenizer, max_length=256):
        self.data = []
        self.tokenizer = tokenizer
        self.max_length = max_length
        with open(path, "r", encoding="utf-8") as f:
            for line in f:
                self.data.append(json.loads(line))

    def __len__(self):
        return len(self.data)

    def __getitem__(self, idx):
        item = self.data[idx]
        chosen_enc = self.tokenizer(item["chosen"], truncation=True,
max_length=self.max_length, padding="max_length", return_tensors="pt")
        rejected_enc = self.tokenizer(item["rejected"],
truncation=True, max_length=self.max_length, padding="max_length",
return_tensors="pt")
        return {
            "chosen_input_ids": chosen_enc["input_ids"].squeeze(0),
            "chosen_attention_mask":
chosen_enc["attention_mask"].squeeze(0),
            "rejected_input_ids":
rejected_enc["input_ids"].squeeze(0),
            "rejected_attention_mask":
rejected_enc["attention_mask"].squeeze(0),
        }
```

- `chosen` is the preferred text.
- `rejected` is the text that was not preferred.
- Both are tokenized with padding/truncation to `MAX_LENGTH`.

---

3. Model and Tokenizer

The script loads a **pretrained transformer model** for sequence classification with a single output (reward score):

```
tokenizer = AutoTokenizer.from_pretrained(RM_MODEL_NAME)
rm_model =
AutoModelForSequenceClassification.from_pretrained(RM_MODEL_NAME,
num_labels=1)
rm_model.to(DEVICE)
```

- The model outputs a single scalar per input.
- Device is automatically set to GPU if available.

---

4. DataLoader with Validation Split

The dataset is split into training and validation sets:

```
val_size = int(len(dataset) * VAL_SPLIT)
train_size = len(dataset) - val_size
train_dataset, val_dataset = random_split(dataset, [train_size,
val_size])
train_loader = DataLoader(train_dataset, batch_size=BATCH_SIZE,
shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=BATCH_SIZE,
shuffle=False)
```

- `VAL_SPLIT` determines the fraction of data used for validation.
- Training loader shuffles data; validation loader does not.

---

5. Optimizer

AdamW optimizer is used:

```
optimizer = AdamW(rm_model.parameters(), lr=LR)
```

- Standard choice for fine-tuning transformer models.

---

6. Training Loop

The main training loop runs for a specified number of epochs:

```
for batch in train_loader:
    optimizer.zero_grad()

    chosen_ids = batch["chosen_input_ids"].to(DEVICE)
    chosen_mask = batch["chosen_attention_mask"].to(DEVICE)
    rejected_ids = batch["rejected_input_ids"].to(DEVICE)
    rejected_mask = batch["rejected_attention_mask"].to(DEVICE)

    chosen_scores = rm_model(input_ids=chosen_ids,
attention_mask=chosen_mask).logits.squeeze(-1)
    rejected_scores = rm_model(input_ids=rejected_ids,
attention_mask=rejected_mask).logits.squeeze(-1)

    loss = -torch.nn.functional.logsigmoid(chosen_scores -
rejected_scores).mean()
    loss.backward()
    optimizer.step()
```

- The **pairwise loss** encourages the model to score preferred text higher than rejected text.
- `logsigmoid` ensures numerical stability and smooth gradients.

---

7. Validation

After each epoch, the model evaluates accuracy on the validation set:

```
with torch.no_grad():
    for batch in val_loader:
        chosen_scores = 
rm_model(input_ids=batch["chosen_input_ids"].to(DEVICE), 
attention_mask=batch["chosen_attention_mask"].to(DEVICE)).logits.squee
ze(-1)
        rejected_scores = 
rm_model(input_ids=batch["rejected_input_ids"].to(DEVICE), 
attention_mask=batch["rejected_attention_mask"].to(DEVICE)).logits.squ
eeze(-1)
        correct += (chosen_scores > rejected_scores).sum().item()
val_accuracy = correct / total
```

- Computes fraction of cases where the model correctly scores `chosen > rejected`.

---

8. Checkpointing and Logging

- Saves model and tokenizer each epoch:

```
checkpoint_dir = os.path.join(OUTPUT_DIR, 
f"checkpoint-epoch{epoch+1}")
rm_model.save_pretrained(checkpoint_dir)
tokenizer.save_pretrained(checkpoint_dir)
```

- Logs training loss and validation accuracy to CSV:

```
pd.DataFrame(log_data).to_csv(CSV_LOG, index=False)
```

- Final model is saved at the end of training.

---

9. Summary

- The script implements a **self-contained pipeline** for training a reward model on human preference data.
- **Highlights:**
  - Pairwise loss function
  - Dataset tokenization with attention masks
  - Training and validation loops
  - Model checkpointing and logging
- Designed to be flexible: can swap in different base models, adjust batch size, epochs, and validation split easily.