```
#####################################################################
#                                   #
# CS435 Generative AI: Security, Ethics and Governance      #
#                                   #
# Instructor: Dr. Adnan Masood                  #
# Contact:   adnanmasood@gmail.com                  #
#                                   #
# Notebook is MIT Licensed                  #
#####################################################################
```

Reinforcement Learning from Human Feedback (RLHF)

Table of Contents

Overview

Reinforcement Learning from Human Feedback (RLHF) is a technique where machines learn to perform tasks or generate outputs that humans find more aligned with their intentions, preferences, or values. Instead of learning purely from an automated reward signal, the model integrates human feedback (like preferences, rankings, or explicit reinforcement signals) to adjust its behavior.

Building an Intuitive Understanding

(A middle-school-friendly explanation)

Imagine you have a robot friend who wants to learn how to make the best sandwich for you. You taste each sandwich it makes, and you give the robot a thumbs-up or thumbs-down. Over time, the robot figures out what makes you happy and makes better sandwiches. **That** is Reinforcement Learning from Human Feedback: using people's opinions to teach a machine how to do something **people** like.

(A slightly more detailed explanation)

Now, think of a computer program trying to answer questions in a friendly, helpful way. It might see examples of good answers and bad answers (provided by humans). The program learns patterns of what makes an answer "good" (maybe it's correct, polite, and easy to read). Over time, it figures out how to answer better. In RLHF, the computer updates its approach based on human feedback about whether it's doing the right thing. This feedback becomes a "reward" for the computer.

(Going deeper into the concepts)

In standard machine learning, you often have a fixed dataset of questions and correct answers. However, with **Reinforcement Learning (RL)**, an agent (our model) performs actions and then observes rewards. When we add a human in the loop, we use people's preferences or evaluations as part of those rewards. This method can correct the model in nuanced situations where pure algorithmic rewards might be tricky to design.

For instance, a language model might generate text. Humans then rank these outputs from best to worst. The model uses these rankings to learn a reward function—a function that estimates how good any answer is. The RL algorithm tries to maximize this learned reward function, effectively aligning model outputs with human preferences.

(A more detailed, mathematical discussion but still approachable)

Typically, RLHF has three main steps:

1. **Supervised Fine-Tuning (SFT)**: The model is first trained or fine-tuned on high-quality data so it produces reasonable outputs.
2. **Reward Modeling**: Human labelers rank different outputs (e.g., answers). We use these rankings to train a **reward model** R_θ which assigns higher scores to outputs that humans like.
3. **Policy Optimization (RL)**: We treat the language model's responses as "actions" and feed them into the reward model. We adjust the model's parameters to **maximize** the predicted reward from R_θ. Algorithms like **PPO (Proximal Policy Optimization)** are commonly used.

Mathematically, if π_ϕ is our policy (the language model), we want to solve:

$$\max_\phi \mathbb{E}_{x\sim D, y\sim\pi_\phi(x)}[R_\theta(x,y)].$$

Here,

- x is some input (question or context),
- y is a generated output (answer),
- R_θ is the reward model parameterized by θ,
- π_ϕ is the policy (also the language model) parameterized by ϕ.

In simpler terms, **pick the parameters** ϕ that yield the highest average reward according to the human-trained reward model.

(A deeper dive into theoretical aspects)

In a more formal RL sense, RLHF can be viewed as a structured approach to the credit assignment problem in partially observed environments, where human-generated signals replace or augment traditional numeric reward signals. Because human feedback is often **noisy and subjective**, there's research into Bayesian models of annotation to handle inter-annotator disagreement, as well as active learning strategies to query the most informative samples for human evaluation.

Further complexity arises in ensuring **robustness** and **generalization** of the learned reward function. We must also consider the possibility of **reward hacking**, where the policy exploits the learned reward model's imperfections. Techniques like **KL regularization** (often used in PPO) help keep the updated policy close to the supervised policy to avoid drifting into strange territory.

Mathematically, advanced treatments involve measuring the divergence between the policy distribution and the baseline distribution, using metrics like the Kullback-Leibler (KL) divergence: $$ \mathcal{L}_{\mathrm{RLHF}}(\phi) = \mathbb{E}_{x \sim D, y \sim \pi_\phi} [R_\theta(x,y)] - \beta \; KL\big(\pi_\phi(y|x)\,\|\,\pi_{\mathrm{SFT}}(y|x)\big). $$ Here, β is a hyperparameter controlling how tightly we want to stay near the supervised fine-tuned policy π_{SFT}. This encourages the model not to stray too far from safe, known-good behaviors.

Intuition Behind RLHF

Many tasks require **human judgment**—for instance, deciding if a response is polite, if an image is suitable for children, or if an answer is factually correct. Pure automatic rewards can be insufficient or too simplistic. RLHF effectively **outsources** part of the reward design to humans, who can judge or rank outputs. As the model sees more examples of what is "approved" versus what is "not approved," it aligns its internal policy to produce the sorts of outputs that people prefer.

Brief History and Invention of RLHF

RLHF builds on the foundational ideas of **reinforcement learning** (from the 1970s-1980s) and **preference learning** (where a function is learned from preference comparisons). Over the last decade, it became prominent for large language models to ensure they generate **more human-aligned** responses. The popular usage in tools like **GPT** has shown how training a model with feedback (through advanced RL algorithms) can significantly improve user satisfaction.

Underlying Technology

1. **Supervised Fine-Tuning**: Start with a pretrained model. Fine-tune it on high-quality data.
2. **Reward Model**: Train a separate model to predict human preference scores or rankings.
3. **RL Optimization (e.g., PPO)**: Update the main model's parameters to **maximize** the reward model's score for each output.
4. **Safety and Alignment**: Techniques such as KL-regularization and carefully curated instructions are used to ensure the model doesn't drift into undesired behaviors.

Mathematical Explanation

1. We have a function that measures how good an answer is, based on what humans said. We call that a *reward*. Our model tries different answers. If the reward is high, it will make similar answers next time. If the reward is low, it will avoid those answers.
2. **More Formally**: We have a policy $\pi_\phi(y|x)$$\pi_\phi(y|x)$ that gives the probability of generating answer $y$$y$ given an input $x$$x$. We also have a reward function $R_\theta(x,y)$$R_\theta(x,y)$. We want to choose $\phi$$\phi$ (the parameters of the model) to maximize the expected reward: $$ \max_{\phi} \mathbb{E}_{x \sim D, y \sim \pi_\phi(x)}[R_\theta(x, y)]. $$$$ \max_{\phi} \mathbb{E}_{x \sim D, y \sim \pi_\phi(x)}[R_\theta(x, y)]. $$
3. **Tie to Observations**: Humans provide the ground truth for the reward function by labeling or ranking outputs. That means the reward model is trained on data like: "Given output A and output B for the same question, which one did humans prefer?"

Illustrative Example with Code

In this simplified example, we'll imagine we have a small set of **text replies** to user questions, and humans have labeled which replies they prefer. We'll train:

1. A **Reward Model** that takes a text and outputs a score.
2. A simple **Policy** (which might just produce a short piece of text) that tries to maximize that reward.

Note: This is a toy example to illustrate the concepts. Real RLHF typically uses large language models and advanced reinforcement learning methods like PPO.

Example Calculations

- **Weights (W)** and **Biases (b)** in a neural network are the numbers that get updated during training. They help the model represent **how** to convert an input (like a piece of text) into an output (like a reward score).
- **Forward Pass**: We pass an input through the network layers, multiply by weights, add biases, and apply activation functions to get predictions.
- **Backward Pass** (a.k.a. **Backpropagation**): We calculate how "off" our prediction was from the target (the feedback). We then adjust the weights and biases to minimize that error (or maximize reward).
- **Loss Function**: In reward modeling, this might be a *ranking loss*, or in RL, an RL objective. The point is to define how we measure "good" or "bad" predictions so we can update the model.

For instance, if a single input sample says: "For question 'Q', between answer A and answer B, humans preferred B." We might want the network to produce `score(B) > score(A)`. If `score(A)` is 0.8 and `score(B)` is 0.5, we have an error. We'll adjust parameters to push `score(B)` higher next time.

Step-by-Step Example with a Public Dataset

For simplicity, let's make a tiny artificial dataset of possible answers to a single question and which ones humans prefer. Our question might be: "What is the capital of France?"

Toy Data

We have the following answers:

1. **"Paris"** - Humans love this answer (correct, concise).
2. **"paris"** - Humans like this but slightly less due to style.
3. **"The capital of France is Paris."** - Also good, maybe even better style.
4. **"I think it might be Berlin?"** - Humans dislike this (incorrect).
5. **"Paris, obviously!"** - Good, but maybe too informal.

Humans rank them as something like:

```
Ranking: 3 > 1 > 2 > 5 >>> 4
```

Meaning #3 is best, #1 is second best, #2 is third best, #5 is fourth, and #4 is last. We'll use these to train a tiny reward model. Then we'll have a policy that, given the question, tries to produce the best answer (highest reward).

```
import torch
import torch.nn as nn
import torch.optim as optim

# For reproducibility
```

```
torch.manual_seed(42)

# Step 1: Create a toy dataset
answers = [
    "Paris",
    "paris",
    "The capital of France is Paris.",
    "I think it might be Berlin?",
    "Paris, obviously!"
]

# Humans rank them: 3 > 1 > 2 > 5 >>> 4
# We'll convert that into a list of pairs (A_i, A_j) where A_i should have higher reward than A_j.
# 3 > 1, 3 > 2, 3 > 5, 3 > 4, 1 > 2, 1 > 5, 1 > 4, 2 > 5, 2 > 4, 5 > 4

ranking_pairs = [
    (2, 0),  # index 2 better than index 0
    (2, 1),
    (2, 4),
    (2, 3),
    (0, 1),
    (0, 4),
    (0, 3),
    (1, 4),
    (1, 3),
    (4, 3)
]

# Step 2: Create a simple reward model
# We'll represent each answer with an embedding (just random for now),
# and use a small linear layer to produce a scalar reward.

class RewardModel(nn.Module):
    def __init__(self, vocab_size=50, embedding_dim=8):
        super().__init__()
        # We'll pretend each answer is an index and we have an embedding for each.
        # This is just to illustrate.
        self.embeddings = nn.Embedding(vocab_size, embedding_dim)
        self.linear = nn.Linear(embedding_dim, 1)

    def forward(self, x):
        # x is an integer index of the answer
        emb = self.embeddings(x)
        # shape of emb is (batch_size, embedding_dim)
        score = self.linear(emb)
        # shape of score is (batch_size, 1)
        return score

# Let's map each answer to an integer ID in this toy setting
answer_to_id = {
    0: 10, # just arbitrary IDs, but unique for each answer
    1: 11,
    2: 12,
    3: 13,
    4: 14
}

reward_model = RewardModel(vocab_size=100, embedding_dim=8)
optimizer = optim.Adam(reward_model.parameters(), lr=0.01)

# Step 3: Define a ranking loss.
# We'll use a simple margin ranking loss: we want R(A_i) > R(A_j) by some margin.

margin = 0.1
ranking_loss_fn = nn.MarginRankingLoss(margin=margin)

# Step 4: Train the reward model

epochs = 200
for epoch in range(epochs):
    epoch_loss = 0.0
    for better_idx, worse_idx in ranking_pairs:
        # Prepare the input
        better_id = torch.LongTensor([answer_to_id[better_idx]])
        worse_id = torch.LongTensor([answer_to_id[worse_idx]])

        # Forward pass
        better_score = reward_model(better_id)
        worse_score = reward_model(worse_id)

        # For margin ranking loss: we want better_score to be at least margin above worse_score.
        # target = +1 indicates better_score should be > worse_score.
```

```
        target = torch.ones(better_score.shape) # Change this line to match the shape of better_score and worse_score
        loss = ranking_loss_fn(better_score, worse_score, target)

        # Backward
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        epoch_loss += loss.item()

    if (epoch+1) % 50 == 0:
        print(f"Epoch {epoch+1}/{epochs}, Loss: {epoch_loss:.4f}")


# Now we can see what the model scores each answer
print("\nFinal learned scores for each answer:")
for i, ans in enumerate(answers):
    ans_id = torch.LongTensor([answer_to_id[i]])
    score = reward_model(ans_id).item()
    print(f"Answer {i}: '{ans}' -> Score: {score:.3f}")
```

```
Epoch 50/200, Loss: 0.0000
Epoch 100/200, Loss: 0.0000
Epoch 150/200, Loss: 0.0000
Epoch 200/200, Loss: 0.0000

Final learned scores for each answer:
Answer 0: 'Paris' -> Score: 0.806
Answer 1: 'paris' -> Score: 0.700
Answer 2: 'The capital of France is Paris.' -> Score: 0.951
Answer 3: 'I think it might be Berlin?' -> Score: 0.187
Answer 4: 'Paris, obviously!' -> Score: 0.349
```

Interpretation

Ideally, the reward model will score the best answer (index 2) highest, then index 0, then index 1, etc., following our ranking. This is a toy approach, so it might not be perfect, but it should hopefully reflect the ranking we taught it.

Next, let's imagine we have a **policy** that tries to pick which answer to give. We'll do something extremely simplified: the policy is just a set of learnable parameters that produce a **probability** of picking each answer. We'll then try to **maximize the expected reward** according to our reward model.

```
import torch.nn.functional as F

class SimplePolicy(nn.Module):
    def __init__(self, num_answers=5):
        super().__init__()
        # We'll keep a raw score for each answer
        self.logits = nn.Parameter(torch.zeros(num_answers))

    def forward(self):
        # Convert logits to probabilities
        probs = F.softmax(self.logits, dim=-1)
        return probs

policy = SimplePolicy()
policy_optimizer = optim.Adam(policy.parameters(), lr=0.01)

epochs = 100
for epoch in range(epochs):
    policy_optimizer.zero_grad()

    # The policy picks an answer i with probability p_i
    # The expected reward = sum_i p_i * R(answer_i)
    probs = policy()
    # Compute the reward for each answer from the reward model
    rewards = []
    for i in range(len(answers)):
        ans_id = torch.LongTensor([answer_to_id[i]])
        r = reward_model(ans_id)
        rewards.append(r)
    # Stack them
    rewards = torch.stack(rewards).squeeze()
    # Expected reward
    expected_reward = torch.sum(probs * rewards)

    # We want to MAXIMIZE the reward, but PyTorch by default MINIMIZES.
    # So we take negative.
    loss = -expected_reward
```

```
    loss.backward()
    policy_optimizer.step()

# Now see what probabilities the policy assigns:
print("\nPolicy probabilities:")
final_probs = policy().detach().numpy()
for i, ans in enumerate(answers):
    print(f"Answer {i}: '{ans}' -> P = {final_probs[i]:.3f}")
```

```
Policy probabilities:
Answer 0: 'Paris' -> P = 0.329
Answer 1: 'paris' -> P = 0.156
Answer 2: 'The capital of France is Paris.' -> P = 0.393
Answer 3: 'I think it might be Berlin?' -> P = 0.062
Answer 4: 'Paris, obviously!' -> P = 0.060
```

After training, the policy should (in theory) put most of its probability on the answers that the **reward model** scored highest. In a real RLHF setting with a large language model, we'd generate a wide variety of possible responses and use an algorithm like PPO to update the model parameters to favor those that get higher reward.

What Illustrative Problem Does RLHF Solve?

It solves the challenge of capturing **human preferences** in tasks where it's difficult to encode a purely "automatic" reward function. By systematically using human judgments, we can guide models to produce outputs more aligned with what humans want—like polite chatbot responses, accurate question answering, or creative writing with a certain style.

What Real-World Problem Does RLHF Solve?

In many real applications, we want AI to produce not just **technically correct** answers, but answers that consider context, ethics, politeness, and user experience. For example, a **virtual assistant** might provide correct answers but in an unfriendly tone. By applying RLHF, we can fine-tune the assistant's style and content so it becomes not only **correct** but also **helpful and pleasant** to interact with.

How to Solve a Real-World Problem Using RLHF

1. **Identify the Task**: e.g., building a helpful customer service chatbot.
2. **Gather Human Feedback**: Let real users chat with a prototype, and label good vs. bad responses.
3. **Train a Reward Model**: Use these labels or preference rankings to train a model that scores the chatbot's responses.
4. **Use RL to Optimize**: Use an RL algorithm (like PPO) to adjust the chatbot's parameters, so it consistently generates high-scoring (preferred) responses.
5. **Validate and Iterate**: Keep collecting more feedback to refine the reward model and further align the chatbot.

This ensures your solution is grounded in actual human preferences, which are often more complex and subtle than simple correctness.

Questions to Ponder & Their Answers (with code examples)

1. **How do we handle conflicting human labels?**
    - We can treat them as noisy labels and use techniques like majority voting, or we can model labelers individually using Bayesian methods. Essentially, we allow for some disagreement and try to find a consensus or distribution of preferences.
2. **How do we make sure the model doesn't exploit the reward model's weaknesses?**
    - Techniques like **reward model regularization**, **adversarial training**, or restricting how far the policy can move from its initial state (like KL-regularization in PPO) help mitigate reward hacking.
3. **Could the model overfit to the training data of preferences?**
    - Yes, overfitting is possible. We need validation sets of preferences and continuous user feedback. Cross-validation or further user studies can help measure real-world performance.
4. **What if the human feedback is biased or unrepresentative?**
    - This can cause the model to learn skewed behaviors. Collecting feedback from diverse, carefully screened groups can help mitigate bias.
5. **Can RLHF be used in other domains beyond NLP (Natural Language Processing)?**

- Absolutely. RLHF can be used for robotics (humans rank certain movements or decisions) or recommendation systems (humans provide direct feedback on recommended items).

A Sample Exercise

Below is a **simplified** code snippet where you need to complete the **TODO** items. This code demonstrates a basic ranking-based training loop. You can adapt it for your own RLHF experiments.

Instructions

1. Read through the code.
2. Complete the **TODO** sections.
3. Run the code to see how the model's parameters update.

```
#  Code Sample for Students to Complete

import torch
import torch.nn as nn
import torch.optim as optim

################################################
# TODO 1: Create your own simple dataset
# Hint: Make a list of pairs (better_answer, worse_answer)
# for instance, dataset = [(0, 1), (0, 2)] etc.
# indicating answer 0 is better than answer 1, etc.
################################################
dataset = [ # Replace with your own pairs
   # (better_idx, worse_idx)
   (0, 1),
   (1, 2),
   (0, 2)
]

################################################
# TODO 2: Create a model that has an embedding layer and a linear layer
# to predict a scalar reward.
# Name it MyRewardModel.
################################################
class MyRewardModel(nn.Module):
   def __init__(self, num_answers=3, embed_dim=5):
      super().__init__()
      # TODO: define your embedding and linear layers
      self.embedding = nn.Embedding(num_answers, embed_dim)
      self.linear = nn.Linear(embed_dim, 1)

   def forward(self, x):
      # TODO: pass the input through embedding and linear
      emb = self.embedding(x)
      score = self.linear(emb)
      return score

# We'll pretend we have 3 possible answers. Indices 0, 1, 2.
model = MyRewardModel(num_answers=3, embed_dim=5)
optimizer = optim.Adam(model.parameters(), lr=0.01)

# We'll use a margin ranking loss.
ranking_loss = nn.MarginRankingLoss(margin=0.5)

# Change this line to match the shape of better_score and worse_score
# target = torch.tensor([1.0])  # Original line - Incorrect shape
target = torch.tensor([1.0]).view_as(better_score)


# We'll do a few epochs of training
num_epochs = 50

for epoch in range(num_epochs):
   total_loss = 0.0
   for (better_idx, worse_idx) in dataset:
      better_tensor = torch.LongTensor([better_idx])
      worse_tensor = torch.LongTensor([worse_idx])

      better_score = model(better_tensor)
      worse_score = model(worse_tensor)

      loss = ranking_loss(better_score, worse_score, target)
      optimizer.zero_grad()
```

```
        loss.backward()
        optimizer.step()
        total_loss += loss.item()

    if (epoch+1) % 10 == 0:
        print(f"Epoch {epoch+1}, Loss: {total_loss:.4f}")

print("\nLearned reward scores:")
for i in range(3):
    score = model(torch.LongTensor([i])).item()
    print(f"Answer {i} -> Score: {score:.3f}")
```

```
Epoch 10, Loss: 0.5033
Epoch 20, Loss: 0.0077
Epoch 30, Loss: 0.0000
Epoch 40, Loss: 0.0000
Epoch 50, Loss: 0.0000

Learned reward scores:
Answer 0 -> Score: 0.890
Answer 1 -> Score: 0.174
Answer 2 -> Score: -0.497
```

Glossary

- **RL (Reinforcement Learning)**: A type of machine learning where an agent learns to take actions to maximize some reward.
- **Human Feedback**: In RLHF, this is typically preference data (like user rankings) or explicit thumbs-up/thumbs-down signals.
- **Reward Model**: A neural network that assigns a "goodness" score to outputs, trained on human feedback.
- **Policy**: The model or agent that generates actions (like text responses) and is optimized to get high reward.
- **Supervised Fine-Tuning (SFT)**: Training a model on a supervised dataset before applying RL.
- **PPO (Proximal Policy Optimization)**: A popular RL algorithm often used in RLHF for stable training.
- **KL Divergence**: A measure of how one probability distribution differs from another.
- **Margin Ranking Loss**: A loss function that ensures a "better" item is ranked higher than a "worse" one by at least some margin.
- **Overfitting**: When a model learns details and noise in the training data to the extent that it negatively impacts its performance on new data.
- **Reward Hacking**: When an agent finds a way to maximize the reward function in unintended ways, exploiting loopholes.

End of Notebook

You now have a comprehensive view of **RLHF**—from a simple sandwich analogy to the advanced mathematics and code. Feel free to experiment with these code snippets and explore the exciting world of **human-aligned AI**!

```
import os, sys, platform, datetime, uuid, socket

def signoff(name="Anonymous"):
    colab_check = "Yes" if 'google.colab' in sys.modules else "No"
    mac_addr = ':'.join(format((uuid.getnode() >> i) & 0xff, '02x') for i in reversed(range(0, 48, 8)))
    print("+++ Acknowledgement +++")
    print(f"Executed on: {datetime.datetime.now()}")
    print(f"In Google Colab: {colab_check}")
    print(f"System info: {platform.system()} {platform.release()}")
    print(f"Node name: {platform.node()}")
    print(f"MAC address: {mac_addr}")
    try:
        print(f"IP address: {socket.gethostbyname(socket.gethostname())}")
    except:
        print("IP address: Unknown")
    print(f"Signing off, {name}")

signoff("Ali Muhammad Asad")
```

```
+++ Acknowledgement +++
Executed on: 2025-01-29 01:45:43.042371
In Google Colab: No
System info: Linux 6.8.0-51-generic
Node name: alimuhammad-Inspiron-7559
MAC address: 20:47:47:74:94:05
IP address: 127.0.1.1
Signing off, Ali Muhammad Asad
```

Start coding or generate with AI.