Introduction to the problem and its significance

Our project aims to train a reinforcement learning (RL) agent to play the Atari game Pong, leveraging Reinforcement Learning from Human Feedback (RLHF) to enhance its learning process. Pong serves as a simplified testbed to understand how RL techniques, when combined with human feedback, can effectively learn a reward model that approximates human preferences. The simplistic yet structured rewards make Pong an ideal environment for evaluating RL algorithms and reward modeling techniques.

The significance of this project lies in its potential application to more complex tasks where predefined rewards are not as straightforward to define or align with human intentions. By using RLHF, the project investigates how human (or synthetic) feedback can be used to train a reward model that guides the RL agent's behavior. This step is crucial in bridging the gap between human preferences and machine-learned behaviors in broader and more intricate domains, such as autonomous driving, robotic manipulation, or personalized recommendation systems.

Through this pipeline, our project explores the capability of RLHF to learn nuanced reward functions, offering insights into aligning AI systems with human values and preferences in both gaming environments and real-world applications.

Data collection and preprocessing methods

The data for the RL agent came from the ALE/Pong-v5 gymnasium environment which handles the states, actions, reward, and the state transition model. To improve training, various preprocessing methods were put on it such as NoopResetEnv which added Noop commands to the beginning of training episodes to randomize the starting state and lead to better generalizability.

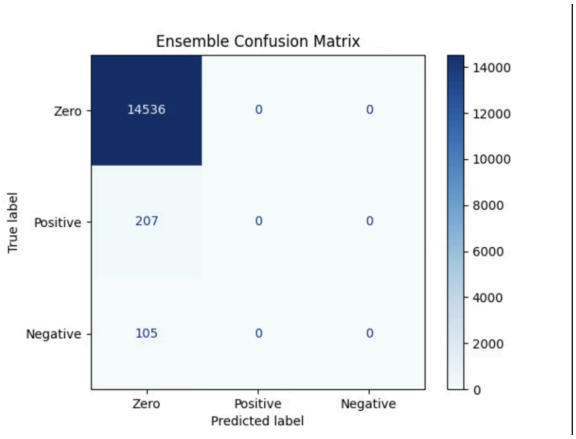
```
def make_env(render_mode=None):
    env = gym.make(
        "ALE/Pong-v5",
        repeat_action_probability=0.0,
        full_action_space=False,
        render_mode=render_mode,
    )
    env = NoopResetEnv(env, noop_max=30)
    env = MaxAndSkipEnv(env, skip=4)
    env = FireResetEnv(env)
    env = WarpFrame(env)
    env = ClipRewardEnv(env)
    env = EpisodicLifeEnv(env)
    return Monitor(env)
```

Pong, as a game, involves an agent controlling a paddle that can move vertically. The agent earns a reward of +1 when it successfully causes the ball to bypass the opponent's paddle and a reward of -1 when the ball bypasses its own paddle. All other interactions in the game state provide a reward of 0. These rewards were used as labels for state actions pairs from roll outs

of the RL environment and this labelled data was fed to our reward model. In our case, we generally rolled out 100 episodes.

```python
def collect_data(model, env, n_episodes=100, down_sample=False):
    states, actions, rewards = [], [], []

    for ep in tqdm(range(n_episodes)):
        obs = env.reset()
        done = False
        while not done:
            action, _ = model.predict(obs, deterministic=True)
            next_obs, reward, done, _ = env.step(action)
            states.append(obs)
            actions.append(action)
            rewards.append(reward)
            obs = next_obs
    if down_sample:
        states, actions, rewards = downsample_rewards(states, actions, rewards)
    return np.array(states), np.array(actions), np.array(rewards)
```

Since the default pong reward means most states receive a reward of 0, we used downsampling to make the amount of data in each class uniform. Prior to downsampling, the high amount of zeros meant that the reward model would always predict 0 even when the ground truth reward was not 0.



Model selection and implementation

The overall architecture involved training an RL agent and rolling out the RL agents and labelling its state action pairs with rewards from the environment which were then fed into an ensemble to CNN's to predict the labelled reward. The ground truth reward labelling in by the RL environment acts as a synthetic human labeller.

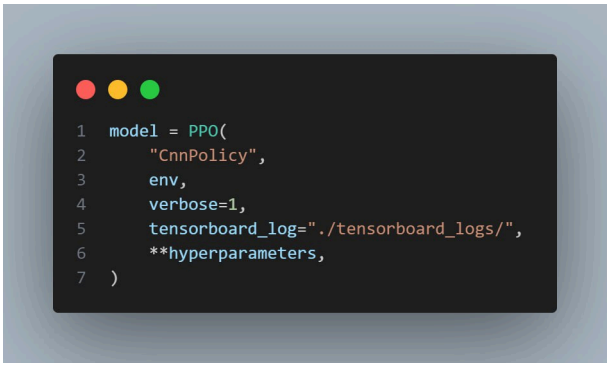
1. **Reinforcement Learning**

   For reinforcement learning, we trained our agents in the OpenAI gymnasium environment which is the industry standard RL simulation environment as it handles the

states, actions and rewards. We used a pre-built environment called ALE/Pong-v5 which contained everything we needed for the pong game.

```
env = gym.make(
    "ALE/Pong-v5",
    repeat_action_probability=0.0,
    full_action_space=False,
    render_mode=render_mode,
)
```

For the actual reward optimization, we used an algorithm called PPO which uses gradients ascent over the reward to update a policy. We used the stable baselines 3 implementation of PPO. Specifically, the CNN policy was used since the state in pong is given as an image of the game screen.



```
model = PPO(
    "CnnPolicy",
    env,
    verbose=1,
    tensorboard_log="./tensorboard_logs/",
    **hyperparameters,
)
```

2. CNN

In order to compare the effectiveness of reinforcement learning, we needed to compare it with a different model as a base case. RewardCNN is a CNN that we use to serve as this base model. The CNN contains 2 convolution layers and 2 linear layers, and then it outputs 3 possible classes: -1, 0, and 1. These classes serve to be our predicted reward for a state and action.

```python
class RewardCNN(nn.Module):
    def __init__(self, state_shape, action_dim):
        super(RewardCNN, self).__init__()
        self.conv1 = nn.Conv2d(6, 16, kernel_size=8, stride=4)
        self.conv2 = nn.Conv2d(16, 32, kernel_size=4, stride=2)

        self.flattened_size = 2592
        self.fc1 = nn.Linear(self.flattened_size, 256)
        self.fc2 = nn.Linear(257, 3)  # Output 3 logits for classification

    def forward(self, state, action):
        cnn_out = F.relu(self.conv1(state))
        cnn_out = F.relu(self.conv2(cnn_out))
        cnn_out = cnn_out.view(state.size(0), -1)
        cnn_out = F.relu(self.fc1(cnn_out))
        action = action.view(action.size(0), -1)
        combined_input = torch.cat((cnn_out, action), dim=1)
        logits = self.fc2(combined_input)
        return logits
```

To create a trained CNN model for the base case, we needed the model to learn on the possible states and actions or otherwise known as the training data. We previously created training data by running the Pong Environment through a process called a rollout. Then we utilized a basic training loop by going through every state, action, and label. We then predict the reward and back propagate the loss while keeping track of the overall training loss.

```python
for epoch in range(num_epochs):
        reward_model.train()
        train_loss = 0.0
        train_accuracy = 0.0

        for state, action, label in train_loader:
            state = state.to(device, dtype=torch.float32)
            action = action.to(device, dtype=torch.float32)
            label = label.to(device, dtype=torch.long)

            optimizer.zero_grad()
            logits = reward_model(state, action)
            loss = criterion(logits, label)
            loss.backward()
            optimizer.step()

            train_loss += loss.item()
            if not tuning:
                train_accuracy += calculate_accuracy(logits, label)

        scheduler.step()
```

3. Ensembles

To improve the performance of our base CNN models, an ensemble to CNN's was used to predict the reward. The cnn.py file would generate a new rollout if the data files did not

exist. Since for ensemble learning we want each member of the ensemble trained on a unique subset of the data, we wrote a bash script which would repeatedly call cnn.py, save the weights, and then delete the old data file which means the next run will be on new data and generate new weights.

```bash
#!/bin/bash

if ["$#" -ne 1]; then
    echo "Usage: $0 <number of ensembles>"
    exit 1
fi

rm -r reward_model
mkdir reward_model
rm -r figures
mkdir figures

for i in $(seq 1 $1); do
    rm cnn_pong_data.npz
    python cnn.py --hyperparams cnn_hyperparameters.json
    mv best_reward_model.pth reward_model/reward_model_$i.pth
    mv loss.png figures/loss$i.png
    mv accuracy.png figures/accuracy$i.png
done

python evaluate_ensemble.py
```

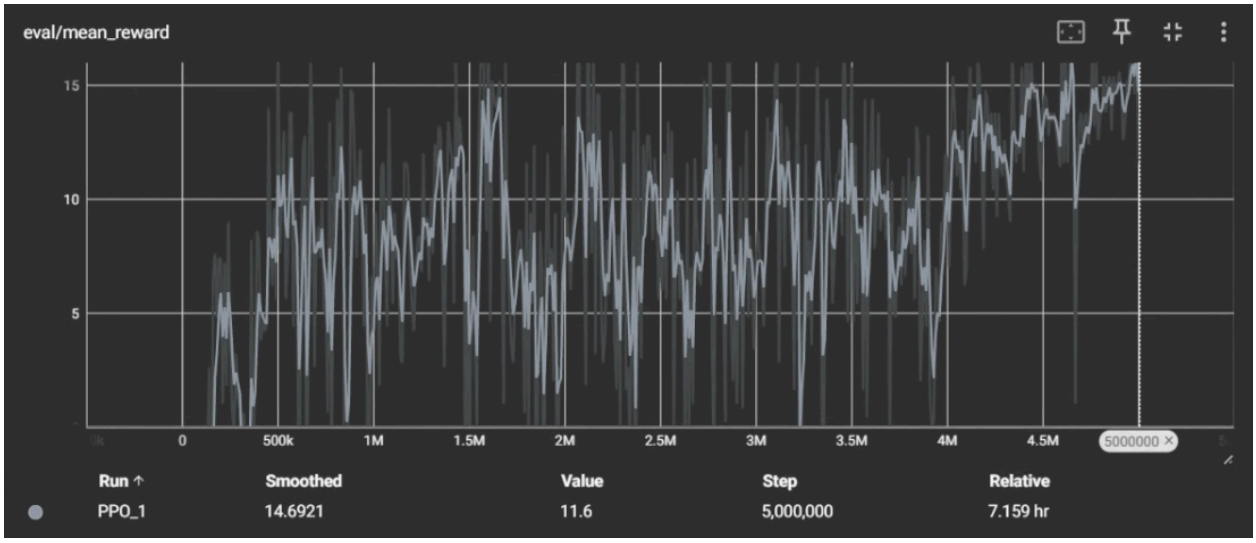
Using these stored weights our ensembles can be evaluated against a test set, or a new fresh roll out not used in the training data. We would vote on the positive, negative, or zero and class and the highest voted class would be our ensemble prediction which can be compared against the ground truth reward to produce accuracy scores and confusion matrices.

```python
def roll_out_and_evaluate(env, model, ensemble, n_episodes=100):
    print("Rolling out trajectories...")
    states, actions, true_rewards = collect_data(model, env, n_episodes, True)

    print("Evaluating...")
    states = torch.tensor(np.array(states), dtype=torch.float32).to(device)
    actions = torch.tensor(np.array(actions), dtype=torch.float32).to(device)
    true_rewards = torch.tensor(np.array(true_rewards), dtype=torch.float32).to(device)
    test_dataset = StateActionRewardDataset(states, actions, true_rewards)
    test_loader = DataLoader(test_dataset, batch_size=16, shuffle=True, drop_last=True)

    all_true_labels = []
    all_ensemble_predictions = []

    for state, action, label in test_loader:
        state = state.to(device, dtype=torch.float32)
        action = action.to(device, dtype=torch.float32)
        label = label.to(device, dtype=torch.long)
        all_true_labels.append(label)

        batch_ensemble_predictions = []
        for model in ensemble:
            with torch.no_grad():
                logits = model(state, action)
                predicted_classes = torch.argmax(logits, dim=1)
                batch_ensemble_predictions.append(predicted_classes)

        # Aggregate predictions from ensemble (majority vote)
        batch_ensemble_predictions = torch.stack(batch_ensemble_predictions, dim=1)
        batch_final_predictions, _ = torch.mode(batch_ensemble_predictions, dim=1)

        # Store final predictions for this batch
        all_ensemble_predictions.append(batch_final_predictions)

    # Concatenate all batches into a single tensor
    all_true_labels = torch.cat(all_true_labels, dim=0)
    all_ensemble_predictions = torch.cat(all_ensemble_predictions, dim=0)
    ensemble_accuracy = calculate_accuracy(all_ensemble_predictions, all_true_labels)
    print(f"Ensemble Accuracy: {ensemble_accuracy:.4f}")

    return (
        all_true_labels.cpu().numpy(),
        all_ensemble_predictions.cpu().numpy(),
        ensemble_accuracy,
    )
```

Results, findings, and key insights

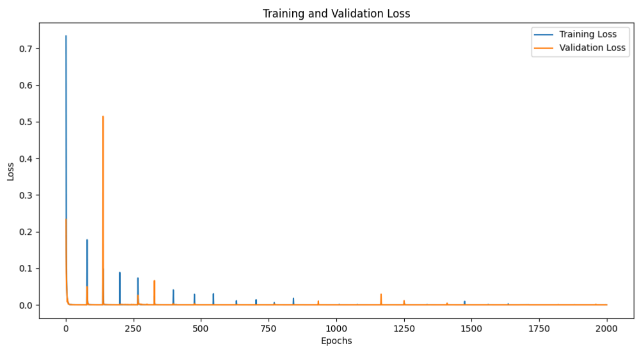
Reinforcement Learning

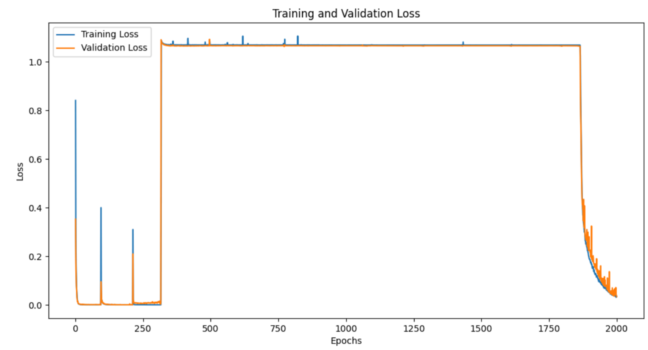
Our reinforcement learning model was trained on the Pong model in a time of 5 million steps. The graph indicates the mean reward given to the reinforcement learning model at each step. The RL model gradually learned over the time steps as shown apparent in the upwards trend of data. It is possible that with even more time steps, the graph could progressively go upwards and achieve a higher overall reward.

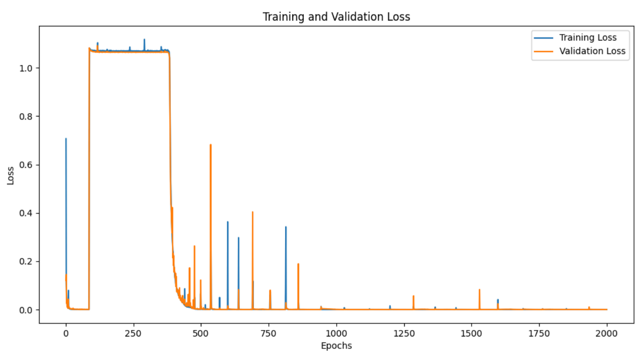


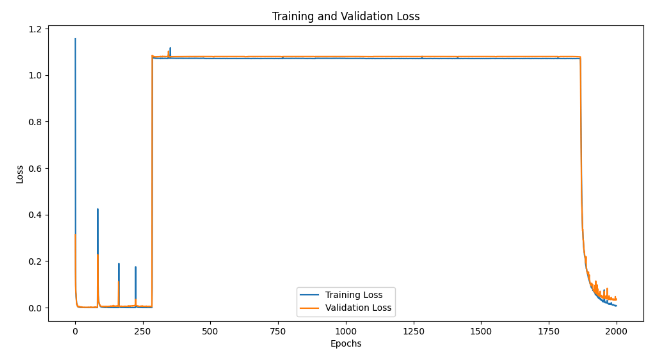
CNN Ensemble

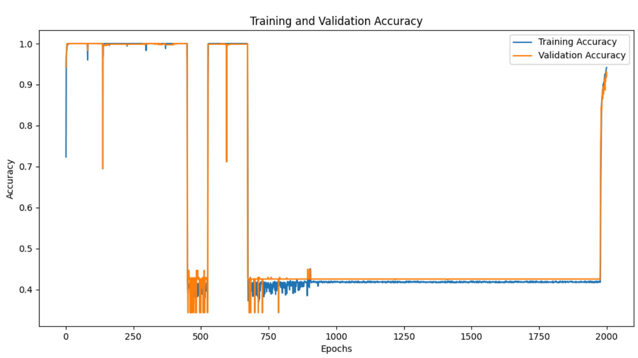
The ensemble of CNNs was created through training a different CNN on newly created training data. Each CNN was trained with 2000 epochs which is on the relatively smaller side. From each ensemble, we produced its respective Loss and Accuracy graph. Some of the ensemble models seemed to exhibit strange behavior of dropping to 40% accuracy. This could be attributed to a phenomenon known as forgetting in neural networks. It is possible that the ensemble forgot about previous tasks or data. This could result in decreased accuracy until it relearns the task or data. The ensemble's return to nearly 100% accuracy could be a representation of this relearning phenomenon. It could also possibly be that it encountered new data that it has not trained out because we serve the data randomly. It could have been possible that the data could have only seen certain states, actions, and labels. Therefore, it could have mispredicted on that unknown dataset. Overall, all of the ensembles seemed to perform extraordinarily well and had nearly 100% accuracy and no loss.


Training and Validation Loss
Training Loss
Validation Loss
Loss
Epochs

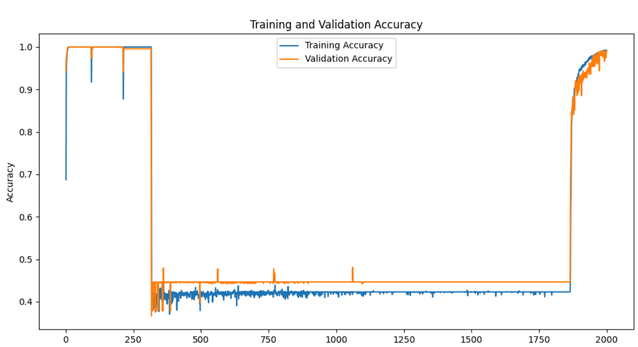

Training and Validation Loss
Training Loss
Validation Loss
Loss
Epochs

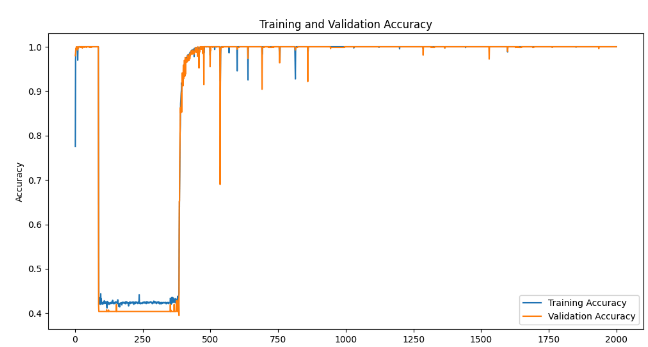

Training and Validation Loss
Training Loss
Validation Loss
Loss
Epochs

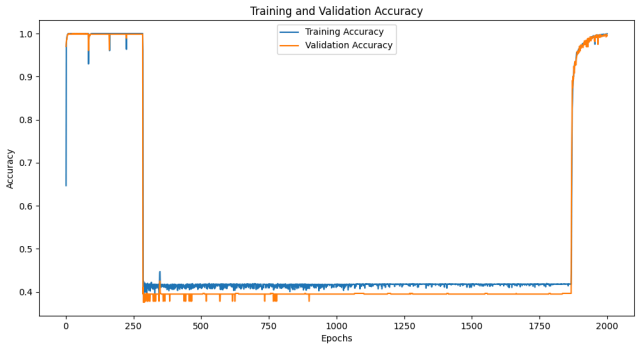

Training and Validation Loss
Training Loss
Validation Loss
Loss
Epochs

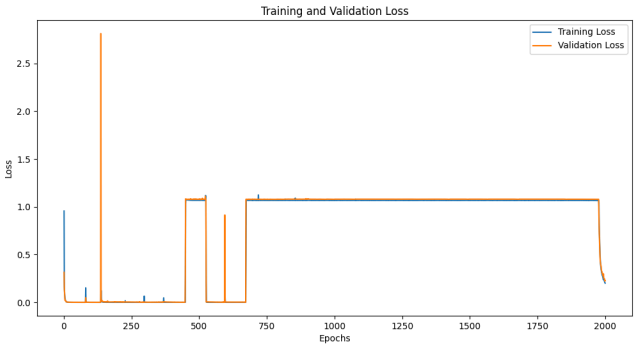

Training and Validation Accuracy
Training Accuracy
Validation Accuracy
Accuracy
Epochs


Training and Validation Accuracy
Training Accuracy
Validation Accuracy
Accuracy
Epochs


Training and Validation Accuracy
Training Accuracy
Validation Accuracy
Accuracy
Epochs

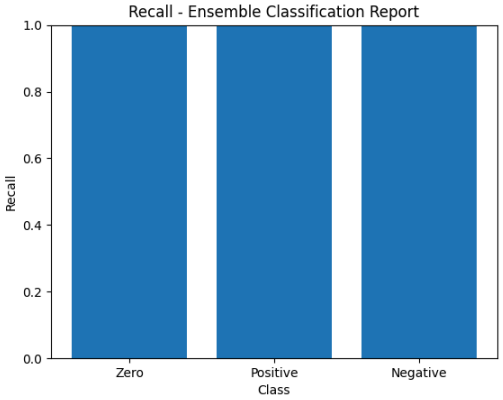

Training and Validation Accuracy
Training Accuracy
Validation Accuracy
Accuracy
Epochs

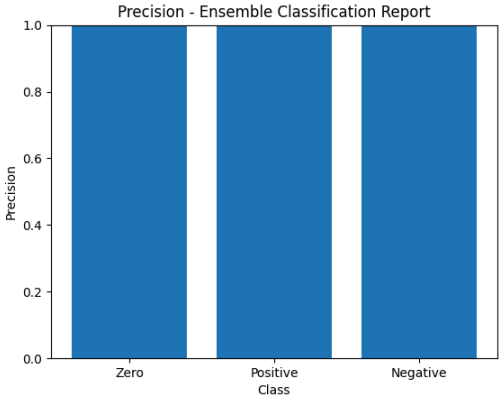




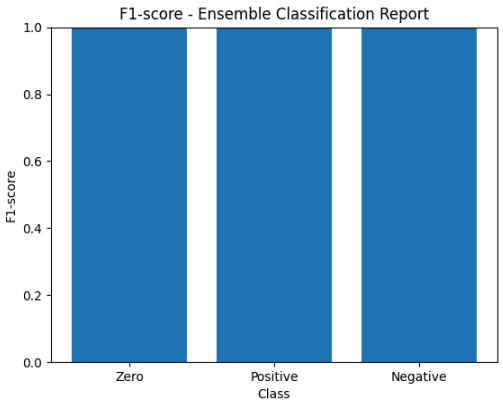
Metrics

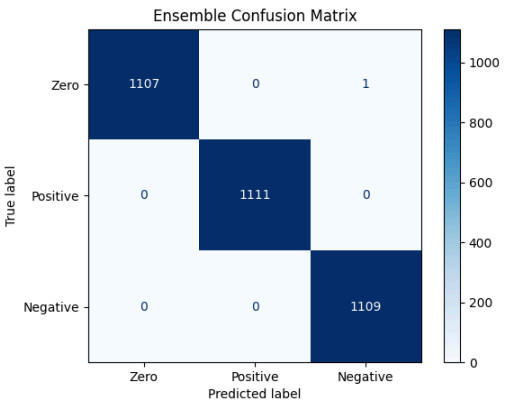
There are 3 metrics that were used to evaluate our CNN ensemble. The metrics are precision, recall, and F1 score. Our ensemble also had nearly 99.97% accuracy. There only existed only one misclassification overall according to the confusion matrix. The ensemble exhibited high precision, recall and F1 score. A high precision indicates that it had a low false positive rate while a high precision indicates a lower false negative rate. The high F1 score indicates our model has an overwhelmingly positive performance.






```
Ensemble Accuracy: 0.9997
```





Challenges faced and how they were overcome

1. **Models not learning**

   One major challenge we had on both the RL and CNN portions was that the models would refuse to learn. In the RL training, we would often have agents which would refuse to explore the state space and get stuck doing a suboptimal action. In the CNN case, our loss would refuse to decrease or decrease slowly. These problems were solved via preprocessing the data using the aforementioned methods and via hyperparameter tuning using Optuna. Optuna would propose values for our various hyperparameters and try to pick the options which maximize the reward in the RL case, or minimize the validation loss in the CNN case.

```python
def objective(trial, train_dataset, val_dataset):
    # Define the search space for Optuna
    hyperparameters = {
        "learning_rate": trial.suggest_float("learning_rate", 1e-5, 1e-2, log=True),
        "weight_decay": trial.suggest_float("weight_decay", 1e-6, 1e-3, log=True),
        "step_size": trial.suggest_int("step_size", 100, 1000),
        "gamma": trial.suggest_float("gamma", 0.01, 0.5),
        "batch_size": trial.suggest_int("batch_size", 1, 64),
    }
    return train(
        train_dataset,
        val_dataset,
        epochs=10000,
        hyperparams=hyperparameters,
        tuning=True,
    )
```

```python
def optimize_ppo(trial, n_envs=1):
    target_rollout_size = 1024 * n_envs
    valid_n_steps = find_valid_n_steps(n_envs, target_rollout_size)
    n_steps = trial.suggest_categorical("n_steps", valid_n_steps)

    return {
        "learning_rate": trial.suggest_float("learning_rate", 1e-7, 1e-2, log=True),
        "ent_coef": trial.suggest_float("ent_coef", 1e-8, 0.1, log=True),
        "clip_range": trial.suggest_float("clip_range", 0.1, 0.4),
        "gamma": trial.suggest_float("gamma", 0.9, 0.999),
        "gae_lambda": trial.suggest_float("gae_lambda", 0.8, 0.95),
        "n_steps": n_steps,
        "max_grad_norm": trial.suggest_float("max_grad_norm", 0.3, 0.9),
    }

```

2. **Not enough memory**

   Because the RL environment generates a lot of state action pairs and each state is an image, our machines very quickly ran out of memory leading to a 4057 error which would kill our process. The resolution for this is using ensembles as each member of the ensemble only needs a small portion of the data which can be generated on the spot instead of loading all of the data into memory all at once.

Real-world applications of your findings

In the real world, we already have an example of RLHF being used very effectively: LLMs. LLMs use RLHF in the alignment step where they are making sure that they are not only giving correct answers, but also answers which align with what humans want (not racist etc.). This demonstrates the use case of RLHF in generating rewards for complicated tasks where the reward cannot be hardcoded due to its complexity.

Our architecture can work in real-world applications with only small tweaks. We need to replace the RL ground truth reward with human supplied labels. This means a human needs to spend

the time scoring each of the state action pairs. Once this is done, this means we have labels which represent the underlying human preference reward function.The rest of the architecture stays the same with the ensemble training to produce a new reward model. We can then use this new reward model to train a new better agent.

In practice, this can be used in autonomous vehicles as goal based training alone will cause agents which successfully reach their goal without crashing but drive very erratically in a way that would negatively impact surrounding human drivers. Using RLHF, we hope that we can use human labelling to reward more human compatible driving patterns and improve the driving performance of the vehicles.