Reinforcement Learning for Pseudo-Labeling

Capstone Project Presentation

Group-11

Students:

Guided by: Prof. Amir Jafari, Prof. Tyler Wallet

George Washington University

Outline

Introduction

- **Problem:** Labeled data is often scarce, and manual annotation is expensive and time-intensive.
- **Challenge:** Conventional pseudo-labeling methods risk error propagation, degrading model performance.
- **Idea:** Employ Reinforcement Learning (RL) to develop an adaptive pseudo-labeling strategy that optimizes sample selection and enhances model accuracy.

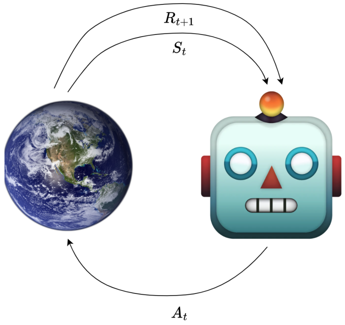
Background and Literature Review

- **Reinforcement Learning (RL)** offers a dynamic alternative to traditional heuristic-based **pseudo-labeling** methods.
- The team began with introductory RL implementations using examples from the **Gymnasium** library to build foundational understanding.
- Conducted an initial review and implementation of five RL-based pseudo-labeling studies to establish a research baseline.

Code Files

- **main.py**: Entry point for training process.
- **env.py**: Custom RL environment handling pseudo-labeling episodes.
- **model.py**: Defines feature extractor and classifier.
- **utils.py**: Helper functions for data loading, metrics, etc.

- **Reinforcement Learning:** Agent learns through interaction and feedback from the environment.
- **Sequential Decision Making:** Each action affects future states and rewards.
- **Markov Decision Process (MDP):** Framework for modelling decision making.



Markov Decision Process

MDP Formulation

State

Image features, model predictions, entropy (uncertainty).

$$\mathcal{S} = \{\mathbf{X};\ \text{softmax}(\mathbf{y});\ \mathcal{L}\}$$

Action

Assign pseudo-label or skip.

$$\mathcal{A} = \{0, \ldots, 9, ;\ \text{skip}\}$$

Reward

Positive for correct pseudo-labels, penalty for wrong ones.

Workflow

1. **Load** labeled and unlabeled MNIST data.
2. **Extract features** with CNN.
3. RL agent selects **pseudo-label** actions.
4. **Retrain** classifier with newly labeled data.

env.reset()

```python
def reset(self, *, seed=None, options=None):
    super().reset(seed=seed)
    self.current_idx = 0
    self.newly_pseudo_labeled_data = []
    self.episode_reward = 0.0

    perm = torch.randperm(self.unlabeled_x.size(0))
    self.unlabeled_x = self.unlabeled_x[perm]
    self.unlabeled_y_true = self.unlabeled_y_true[perm]

    self.baseline_accuracy = self.downstream_model.evaluate_model(self.val_x, self.val_y)

    return self._get_state(), {}
```

- **Input:** random seed and options
- **Action performed:** resets the environment by
    - resetting counters and rewards
    - shuffling unlabeled dataset
    - recalculating model's baseline accuracy on validation data
- **Output:** initial environment state and empty info dictionary

model.policy()

```
def policy(self, state):
```

```
    with torch.no_grad():
        state_tensor = torch.FloatTensor(state).unsqueeze(0).to(self.device)
        logits = self.pi(state_tensor)
        action_dist = torch.distributions.Categorical(logits=logits)
        action = action_dist.sample()
        return action.item()
```

- **Input:** current state
- **Action performed:**
    - Converts the state to a tensor
    - passes it through the policy network to obtain logits
    - creates a categorical probability distribution, and samples one action from it.
- **Output:** selected action as integer

env.step()

- **Input:**
  - Current image index and the action selected by the agent.
- **Action Performed:**
  - Computes reward based on prediction and action
  - Updates pseudo-labeled data
  - Trains downstream model at episode end
- **Output:**
  - Returns next state, total reward, termination flag, and performance info (accuracy, final reward).

env.step()

```
def step(self, action):
    img_idx = self.current_idx
    reward = 0
    if action < self.num_classes:
        true_label = self.unlabeled_y_true[img_idx]
        preds = self.downstream_model.get_predictions(self.unlabeled_x[img_idx].unsqueeze(0))
        pred_label = torch.argmax(preds).item()
        reward = self.calculate_reward(pred_label, true_label, action, preds)
        self.episode_reward += reward
        self.newly_pseudo_labeled_data.append(
            (self.unlabeled_x[img_idx], torch.tensor(action, device=self.device))
        )
    else:  # Action is skip (action == num_classes)
        reward = self.calculate_reward(None, None, action, None)

    self.current_idx += 1
    terminated = (self.current_idx >= len(self.unlabeled_x))
    info = {}
    if terminated:
        self.downstream_model.train_model((self.labeled_x, self.labeled_y), self.newly_pseudo_labeled_data)
        new_accuracy = self.downstream_model.evaluate_model(self.val_x, self.val_y)
        info['new_accuracy'] = new_accuracy
        accuracy_bonus = new_accuracy - self.baseline_accuracy
        reward += accuracy_bonus
        info['final_reward'] = self.episode_reward + accuracy_bonus

    next_state = self._get_state()
    return next_state, reward, terminated, False, info
```

model.update()

- **Input:** Sampled states, actions, and rewards from agent experience.
- **Action performed:**
  - Computes advantages using rewards-to-go and value estimates.
  - Updates policy network to maximize expected reward.
  - Updates value network to minimize prediction error.
- **Output:** Returns policy loss and value loss for performance tracking.

model.update()

```python
def update(self, sampled_states, sampled_actions, sampled_rewards):
    # Compute advantages
    rewards_to_go, values, adv = self.compute_advantage(sampled_states, sampled_rewards)

    # Convert to tensors
    states_tensor = torch.FloatTensor(np.array(sampled_states)).to(self.device)
    actions_tensor = torch.LongTensor(sampled_actions).to(self.device)

    # Update policy network
    self.optimizer_pi.zero_grad()
    logits = self.pi(states_tensor)
    log_probs = F.log_softmax(logits, dim=1)
    action_log_probs = log_probs.gather(1, actions_tensor.unsqueeze(1)).squeeze(1)
    policy_loss = -torch.mean(action_log_probs * adv.detach())
    policy_loss.backward()
    self.optimizer_pi.step()

    # Update value network
    self.optimizer_v.zero_grad()
    value_preds = self.v(states_tensor).squeeze(1)
    value_loss = torch.mean((rewards_to_go - value_preds) ** 2)
    value_loss.backward()
    self.optimizer_v.step()

    return policy_loss.item(), value_loss.item()
```

main.py (Bringing everything together)

```python
def main(EPISODES, NUM_LABELED, NUM_UNLABELED, NUM_VALIDATION):
    env = PseudoLabelEnv(NUM_LABELED=NUM_LABELED, NUM_UNLABELED=NUM_UNLABELED, NUM_VALIDATION=
      NUM_VALIDATION)
    state_dim = env.observation_space.shape[0]
    action_dim = env.action_space.n
    agent = PolicyGradient(state_dim=state_dim, num_actions=action_dim)
    for episode in tqdm(range(EPISODES), desc="Training Progress"):
        state, info = env.reset()
        terminated = False
        sampled_states, sampled_actions, sampled_rewards = [], [], []
        while not terminated:
            action = agent.policy(state)
            next_state, reward, terminated, _, info = env.step(action)

            # Store the experience
            sampled_states.append(state)
            sampled_actions.append(action)
            sampled_rewards.append(reward)

            state = next_state

        agent.update(np.vstack(sampled_states), sampled_actions, sampled_rewards)

    env.close()
```

Observations

- RL policy learns to prefer high-confidence samples.
- Entropy and softmax probabilities guide state representation.
- Caveat: Reward shaping has big impact on stability.

Model	**Error Rate (%)**
Random	-
Downstream Model	-
RLPseudolabelEnv	-

Table 1.1 Comparison of different models

Conclusion

- Demonstrated feasibility of RL-based pseudo-labeling.
- Compared to naive pseudo-labeling, RL improves selection.
- Future work: extend to larger datasets and test other RL algorithms.

Thank You

Questions?