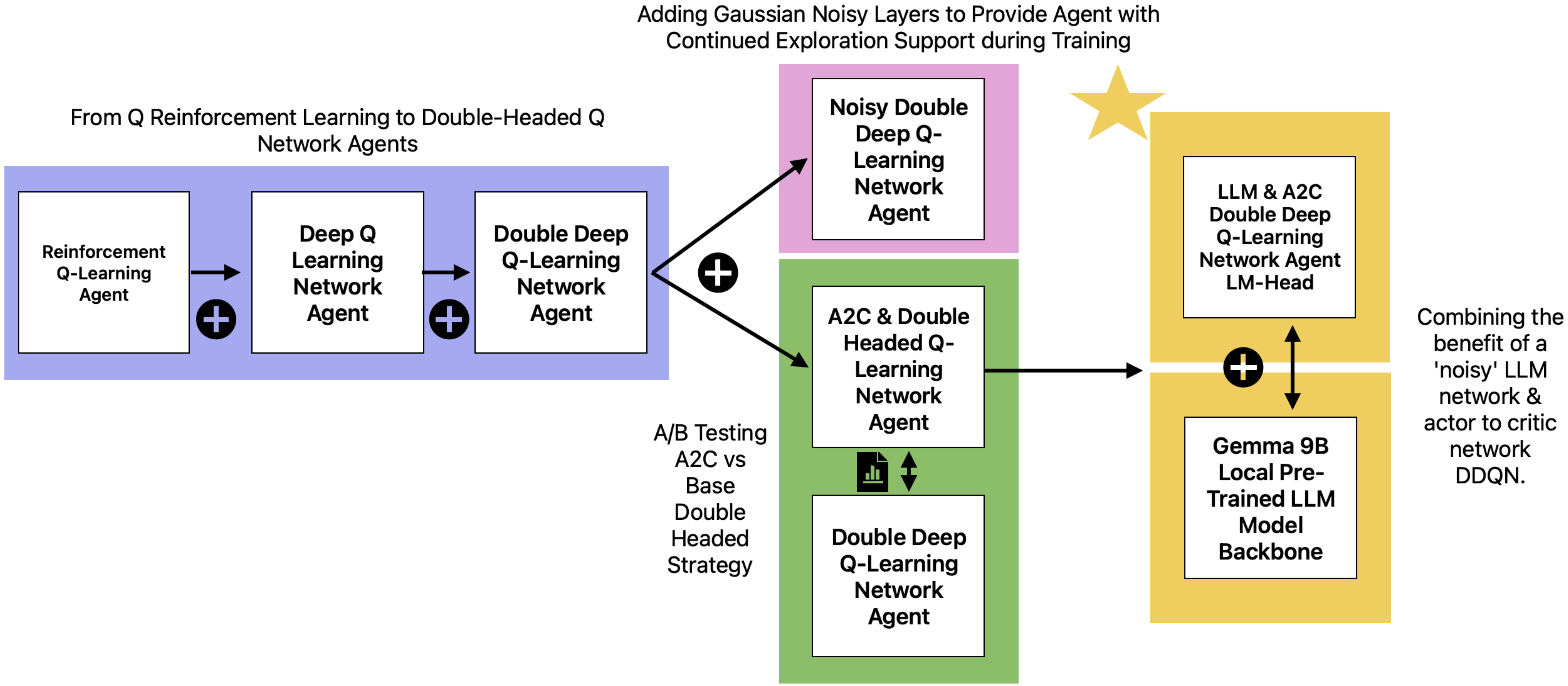
Path to Training a LLM-Based Deep Actor to Critic (A2C) & Deep Double Headed Q-Learning (DDQN) Neural Network Agent to Play Blackjack!

A project presentation by
Alexis Ambriz, Sri Vinay Appari,
Suresh Ravuri, and
Harshavardhan Valmiki

The Blackjack Agent Neural Network Architectures

As we embarked on the A.I Blackjack training journey, we had to remember that the key is to enjoy the game! The proposed A.I should learn the right mix of strategy and a "fun-loving" attitude with experimental playstyles.



DDQN vs Noisy DDQN Implementations

DDQN Agent

```python
class ImprovedDQNAgent:
    def __init__(self, input_dim=3, learning_rate=5e-4, gamma=0.99, epsilon=1.0):
        self.device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
        self.fitness = [] # used to store a series of 'avg reward' during evaluation

        # Larger network
        self.policy_net = nn.Sequential(
            nn.Linear(input_dim, 128),
            nn.ReLU(),
            nn.Linear(128, 128),
            nn.ReLU(),
            nn.Linear(128, 64),
            nn.ReLU(),
            nn.Linear(64, 2)
        ).to(self.device)

        self.target_net = nn.Sequential(
            nn.Linear(input_dim, 128),
            nn.ReLU(),
            nn.Linear(128, 128),
            nn.ReLU(),
            nn.Linear(128, 64),
            nn.ReLU(),
            nn.Linear(64, 2)
        ).to(self.device)

        self.target_net.load_state_dict(self.policy_net.state_dict())

        self.optimizer = optim.Adam(self.policy_net.parameters(), lr=learning_rate)
        self.memory = ReplayMemory(capacity=20000)
```

Predicts w/ Policy Net

```python
    def select_action(self, state):
        """Select action using epsilon-greedy policy"""
        if random.random() < self.epsilon:
            return random.randint(0, 1)

        with torch.no_grad():
            state = torch.FloatTensor(state).unsqueeze(0).to(self.device)
            q_values = self.policy_net(state)
            return q_values.argmax().item()
```

Learns Q w/ Target Net

```python
# Double DQN implementation
with torch.no_grad():
    next_actions = self.policy_net(next_states).argmax(1)
    next_q_values = self.target_net(next_states).gather(1, next_actions.unsqueeze(1
    target_q_values = rewards + (1 - dones.float()) * self.gamma * next_q_values

current_q_values = self.policy_net(states).gather(1, actions.unsqueeze(1)).squeeze(

# Huber loss for better stability
loss = nn.SmoothL1Loss()(current_q_values, target_q_values)

self.optimizer.zero_grad()
loss.backward()
torch.nn.utils.clip_grad_norm_(self.policy_net.parameters(), 1.0)
self.optimizer.step()

return loss.item()
```

Noisy DDQN Agent

Noisy DDQN Architecture

```python
import torch.optim as optim

class NoisyDoubleDQNAgent:
    def __init__(self, input_dim=3, learning_rate=5e-4, gamma=0.99, epsilon=1.0):
        self.device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
        self.fitness = [] # used to store a series of 'avg reward' during evaluation

        self.policy_noisy_linear1 = NoisyLinear(128, 64)
        self.policy_noisy_linear2 = NoisyLinear(64, 2)
        self.target_noisy_linear1 = NoisyLinear(128, 64)
        self.target_noisy_linear2 = NoisyLinear(64, 2)

        # Larger network
        self.policy_net = nn.Sequential(
            nn.Linear(input_dim, 128),
            nn.ReLU(),
            nn.Linear(128, 128),
            nn.ReLU(),
            self.policy_noisy_linear1, # the noisy layers w/o ReLU in-between
            self.policy_noisy_linear2,
        ).to(self.device)

        self.target_net = nn.Sequential(
            nn.Linear(input_dim, 128),
            nn.ReLU(),
            nn.Linear(128, 128),
            nn.ReLU(),
            self.target_noisy_linear1,
            self.target_noisy_linear2
        ).to(self.device)
```

Learns Gaussian Noise

```python
class NoisyLinear(nn.Module):
    """Noisy linear module for NoisyNet.

    Attributes:
        in_features (int): input size of linear module
        out_features (int): output size of linear module
        std_init (float): initial std value
        weight_mu (nn.Parameter): mean value weight parameter
        weight_sigma (nn.Parameter): std value weight parameter
        bias_mu (nn.Parameter): mean value bias parameter
        bias_sigma (nn.Parameter): std value bias parameter
```

Noise Added to Both Nets

```python
    def forward(self, x: torch.Tensor) -> torch.Tensor:
        """Forward method implementation.

        We don't use separate statements on train / eval mode.
        It doesn't show remarkable difference of performance.
        """
        return F.linear(
            x,
            self.weight_mu + self.weight_sigma * self.weight_epsilon,
            self.bias_mu + self.bias_sigma * self.bias_epsilon,
```

DDQN vs Actor-Critic (A2C) Implementation

DDQN vs A2C Agents

Similarity

The noisy DDQN / Q-Learning and off-policy agent was similar in architecture to an A2C agent as it also implemented separate neural network 'heads' to separate the target net (for training) from the policy net (for prediction), while also learning to add noise to the predictions of both. Furthermore, both the state and target policy are used to estimate optimal action values. Similarly, the replay buffer is used to train the policy net an episode behind the target network. The target network then copies the policy network during the next episode for prediction of actions.

Difference

The A2C agent is a combo of SARSA and Q-Learning and is on-policy; architecture's key difference to the DDQN is that it uses a third network providing more separation. There is a feature extraction network, an 'actor' or action network, and a "critic" or a state / Q value network. Another difference is that all three networks are used during both training & prediction.

A2C Agent

Actor-Critic Network Architecture

```python
class ActorCritic(nn.Module):
    def __init__(self, input_dim=3):
        super(ActorCritic, self).__init__()

        # Shared features extractor
        self.features = nn.Sequential(
            nn.Linear(input_dim, 128),
            nn.ReLU(),
            nn.Linear(128, 128),
            nn.ReLU()
        )

        # Actor head (policy network)
        self.actor = nn.Sequential(
            nn.Linear(128, 64),
            nn.ReLU(),
            nn.Linear(64, 2),  # 2 actions: hit or stand
            nn.Softmax(dim=-1)  # Output probabilities for each action
        )

        # Critic head (value network)
        self.critic = nn.Sequential(
            nn.Linear(128, 64),
            nn.ReLU(),
            nn.Linear(64, 1)  # Single value output
        )

    def forward(self, x):
        features = self.features(x)
        action_probs = self.actor(features)
        state_value = self.critic(features)
        return action_probs, state_value
```

```python
class A2CAgent:
    def __init__(self, model, optimizer, gamma=0.99):
        self.model = model
        self.optimizer = optimizer
        self.gamma = gamma
        self.rewards = []
        self.log_probs = []
        self.state_values = []
        self.entropies = []
        self.device = torch.device("cuda" if torch.cuda.is_availab

    def select_action(self, state):
        """Select action using the policy network"""
        state = torch.FloatTensor(state).to(self.device)
        state.requires_grad = True

        # Get action probabilities and state value
        action_probs, state_value = self.model(state.unsqueeze(0))
        action_probs = action_probs.squeeze()

        # Create categorical distribution
        dist = torch.distributions.Categorical(action_probs)
        action = dist.sample()

        # Calculate log probability and entropy
        log_prob = dist.log_prob(action)
        entropy = dist.entropy()

        # Store for training
        self.log_probs.append(log_prob)
        self.state_values.append(state_value)
        self.entropies.append(entropy)
```

Evaluation

A2C Agent vs the DDQN at Blackjack Strategy

When compared to the previous DDQN agent, the A2C agent demonstrated better performance. Across 200 test rounds:

- A2C Win Rate: 35.0%
- DQN Win Rate: 18.0%

During training they also differed:

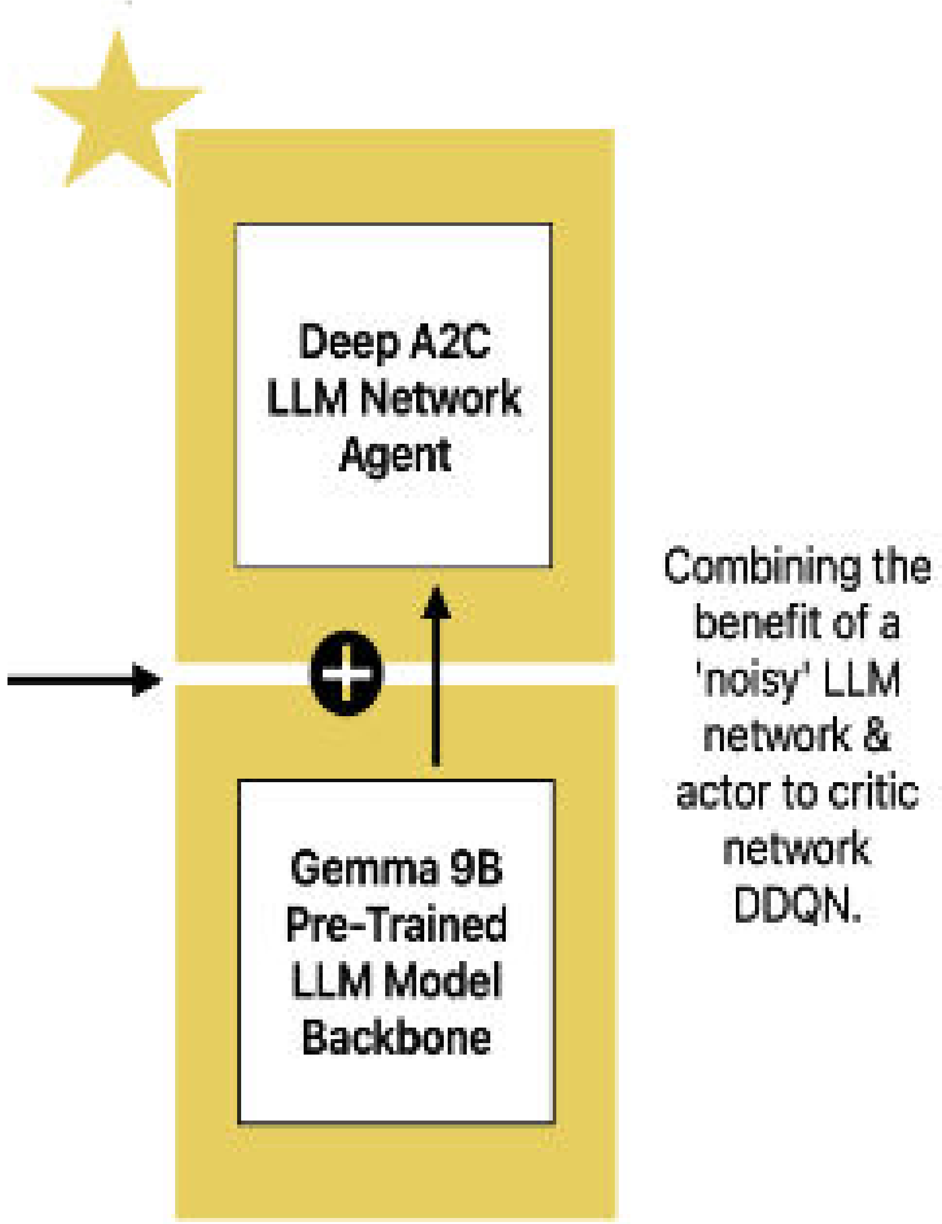
- Average A2C Reward: -0.250
- Average DDQN Reward: -0.591
- Decision Disagreement Rate: 48.0%

Future Work

A2C Agent LM-Head with Gemma 9B LLM Backbone

A2C LLM Agent

After evaluating the A2C reinforcement learning neural network agent and finding that it outperformed our previous DQN, future work could extend an A2C agent by leveraging the large amount of knowledge stored within popular foundation language models such as Gemma 9B or other larger models. Combining the A2C agent as the output 'head' with the language backbone could help diversify the agents learning strategies to potentially boost performance.



Thanks!