

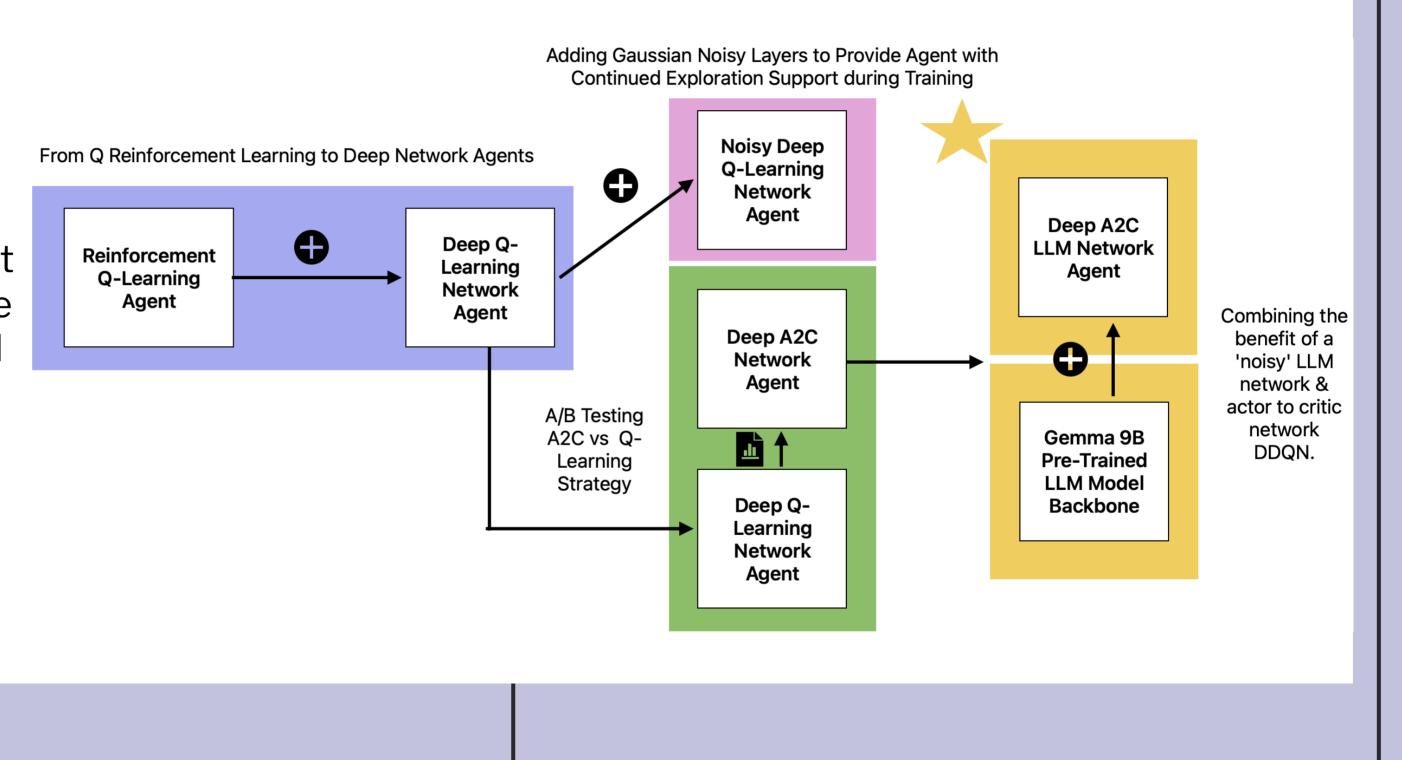
Training a LLM-Based A2C & Deep Q-Learning (DQN) Neural Network Agent to Play Blackjack!



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.







Deep Q-Learning (DQN) vs Noisy DQN Implementations

```
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)
```

DQN Agent Predicts w/ Policy Net

```
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()</pre>
```

Learns Q w/ Target Net

```
# 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 Architecture

///////

```
1 import torch.optim as optim
 3 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)
10
11
          self.target_noisy_linear2 = NoisyLinear(64, 2)
12
13
          # Larger network
          self.policy_net = nn.Sequential(
14
15
              nn.Linear(input dim, 128),
16
              nn.ReLU(),
              nn.Linear(128, 128),
17
18
              nn.ReLU(),
19
              self.policy noisy linear1, # the noisy layers w/o ReLU in-between
              self.policy noisy linear2,
20
          ).to(self.device)
21
22
          self.target_net = nn.Sequential(
23
              nn.Linear(input dim, 128),
24
              nn.ReLU(),
25
              nn.Linear(128, 128),
26
27
              nn.ReLU(),
              self.target_noisy_linear1,
28
              self.target_noisy_linear2
29
          ).to(self.device)
30
```

Noisy DQN Agent Learns Gaussian Noise

```
1 class NoisyLinear(nn.Module):
2    """Noisy linear module for NoisyNet.
3
4    Attributes:
5     in_features (int): input size of linear module
6     out_features (int): output size of linear module
7     std_init (float): initial std value
8     weight_mu (nn.Parameter): mean value weight parameter
9     weight_sigma (nn.Parameter): std value weight parameter
10     bias_mu (nn.Parameter): mean value bias parameter
11     bias_sigma (nn.Parameter): std value bias parameter
```

Noise Added to Both Nets

```
59
      def forward(self, x: torch.Tensor) -> torch.Tensor:
60
          """Forward method implementation.
61
62
          We don't use separate statements on train / eval mode.
63
          It doesn't show remarkable difference of performance.
65
          return F.linear(
66
67
              self.weight_mu + self.weight_sigma * self.weight_epsilon,
68
               self.bias_mu + self.bias_sigma * self.bias_epsilon,
```

//////

DQN vs Actor-Critic (A2C) Implementation

DQN vs A2C Agents

Similarity

The noisy DQN / Q-Learning and offpolicy agent was similar in architecture to
a DQN as it also implemented seperate
neural network 'heads' to seperate 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.

Difference

The A2C is a combo of SARSA and Q-Learning and is on-policy; architecture's key difference to the DQN 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.

Actor-Critic Network Architecture

```
1 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),
 9
10
              nn.ReLU()
11
12
13
          # Actor head (policy network)
14
          self.actor = nn.Sequential(
15
              nn.Linear(128, 64),
16
              nn.ReLU(),
17
              nn.Linear(64, 2), # 2 actions: hit or stand
18
              nn.Softmax(dim=-1) # Output probabilities for each action 51
19
20
21
          # Critic head (value network)
          self.critic = nn.Sequential(
22
23
              nn.Linear(128, 64),
24
              nn.ReLU(),
25
              nn.Linear(64, 1) # Single value output
26
27
28
      def forward(self, x):
29
           features = self.features(x)
          action probs = self.actor(features)
30
          state_value = self.critic(features)
31
32
          return action_probs, state_value
```

A2C Agent

```
33 class A2CAgent:
      def __init__(self, model, optimizer, gamma=0.99):
34
35
           self.model = model
           self.optimizer = optimizer
36
           self.gamma = gamma
37
           self.rewards = []
38
39
           self.log probs = []
           self.state values = []
40
41
           self.entropies = []
42
           self.device = torch.device("cuda" if torch.cuda.is availab
43
44
      def select action(self, state):
           """Select action using the policy network"""
45
           state = torch.FloatTensor(state).to(self.device)
46
47
           state.requires grad = True
48
49
           # Get action probabilities and state value
           action_probs, state_value = self.model(state.unsqueeze(0))
50
           action probs = action probs.squeeze()
52
53
           # Create categorical distribution
54
           dist = torch.distributions.Categorical(action_probs)
55
           action = dist.sample()
56
57
           # Calculate log probability and entropy
58
           log prob = dist.log prob(action)
59
           entropy = dist.entropy()
60
           # Store for training
61
           self.log_probs.append(log_prob)
62
63
           self.state_values.append(state_value)
64
           self.entropies.append(entropy)
```

Evaluation

A2C Agent vs the DQN at Blackjack Strategy

When compared to the previous Deep Q-Network (DQN) implementation, the Actor-Critic (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 DQN Reward: -0.591
- Decision Disagreement Rate: 48.0%

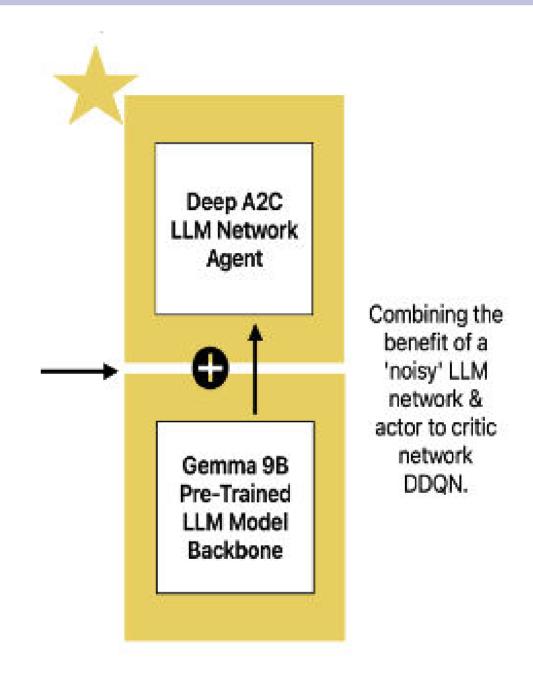
//////

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!