DQN Loma Final Report

Ziyuan Lin

2025.6.10

1 Introduction

Automatic differentiation is a technique to automatically compute the differentials based on functions. Loma is an automatic differentiation language developed in CSE 291. In this final project, I implemented a DQN based on Loma, which uses python as the backbone and Loma as the core.

2 ALE and Data Processing

I have downloaded the Arcade Learning Environment(ALE), which is an environment for game simulation. The frames are captured dynamically through playing. I reduced the input dimensionality to 84×84. The frames byte streams flowing into the loma pipeline. I use four frames as a stack to the loma. The ALE environment is on Linux system. Here are some ALE games samples:

3 Matrix Multiplication and Compile

I have implemented Matrix multiplication as shown below:

```
def matrix_multiply(A : In[Array[float]],
                    B : In[Array[float]],
                    C : Out[Array[float]],
                    M : In[int],
                    N : In[int],
                    K : In[int]):
    i : int = 0
    while (i < M, max_iter := 1000):
        j : int = 0
        while (j < N, max_iter := 1000):
            sum_val : float = 0.0
            k : int = 0
            while (k < K, max_iter := 1000):
                a_idx : int = i * K + k
                b_idx : int = k * N + j
```

```
                sum_val = sum_val + A[a_idx] * B[b_idx]
                k = k + 1

            c_idx : int = i * N + j
            C[c_idx] = sum_val
            j = j + 1
        i = i + 1
```

This is code for matrix multiplication $C = AB$, where $A \in \mathbb{R}^{m\times n}, B \in \mathbb{R}^{n\times k}$. I have set the loop limit to 1000. I also implement the versions for matrix vector and vector matrix multiplication. Then I implemented a SIMD version, but now I cannot use it since my Loma does not support SIMD currently. I have implemented the code to compile the matrix.py.

4 DQN in Loma

In my implementation, I choose my hidden dimension size to be 100. I use Relu as my non-linear function. There are 3 hidden layers and the final output is connected a loss function. For loss functions, I implemented both MSE and Huber loss. Here is a sample code snippets for the I trained the network with 50 episodes and update the DQN with every 5 episodes since I need replay buffer for original DQN update procedure. Here is some sample code:

```
def network_forward(inputs : In[Array[float]],
                    weights : In[Array[float]],
                    outputs : Out[Array[float]],
                    input_size : In[int],
                    output_size : In[int]):

    layer1 : Array[float] = Array[float](100)
    layer2 : Array[float] = Array[float](100)
    layer3 : Array[float] = Array[float](100)

    weight_idx : int = 0
    bias_idx : int = {total_weights}

    i : int = 0
    while (i < {self.hidden_sizes[0]
    if self.hidden_sizes else self.output_size},
    max_iter := 200):
        sum_val : float = 0.0
        j : int = 0
        while (j < input_size, max_iter := 200):
            w_idx : int = weight_idx + i * input_size + j
            sum_val = sum_val +
            inputs[j] * weights[w_idx]
```

```
        j = j + 1

    b_idx : int = bias_idx + i
    sum_val = sum_val + weights[b_idx]

    if sum_val > 0.0:
        layer1[i] = sum_val
    else:
        layer1[i] = 0.0
    i = i + 1
```

5 Python Wrapper for Training

I have implemented a python wrapper for both training and testing(Half complete). In this wrapper, I train the network for 50 episodes and update the DQN every 5 episodes. The reason I do not update every cycle is because I want to implement the replay buffer in original paper. The learning rate is $1e^{-4}$ for the best convergence speed and performance. Here is some code snippets for my python wrapper implementation.

```
def train_loma_dqn(env_name: str = 'CartPole-v1',
                   episodes: int = 500,
                   max_steps: int = 500):

    env = gym.make(env_name)
    state_size = env.observation_space.shape[0]
    action_size = env.action_space.n

    agent = LomaDQNAgent(state_size, action_size,
    hidden_sizes=[64, 32])

    scores = deque(maxlen=100)

    print(f"Training Loma DQN on {env_name}")
    print(f"State size: {state_size},
    Action size: {action_size}")
    print(f"Network architecture:
    {state_size} -> 64 -> 32 -> {action_size}")

    for episode in range(episodes):
        state, _ = env.reset()
        total_reward = 0

        for step in range(max_steps):
            action = agent.act(state)
```

```
        next_state, reward, terminated,
        truncated, _ = env.step(action)
        done = terminated or truncated

        agent.remember(state, action,
        reward, next_state, done)

        state = next_state
        total_reward += reward

        if done:
            break

    scores.append(total_reward)

    if len(agent.memory) > agent.batch_size:
        loss = agent.replay()

    if episode % 5 == 0:
        agent.update_target_network()

    if episode % 50 == 0:
        avg_score = np.mean(scores)
        print(f"Episode {episode},
        Average Score: {avg_score:.2f}, "
               f"Epsilon: {agent.epsilon:.3f}")
```

6 Conclusion

In this final project, I have used Loma as my core to implement DQN. I preprocessed the Atari 2600 game frames as what the original paper has implemented. I built a python wrapper to train the network.

7 Future Work

Instead of fully connected perceptron, I want to implement the convolution and pooling network. I have only implemented testing part partially. I plan to finish the rest part in next two weeks.

8 Reference

[1]Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., ... & Hassabis, D. (2015). Human-level control through deep reinforcement learning. Nature, 518(7540), 529-533.

9 Github Link