NI 기술 문서

Deep Q-networks를 활용한 OpenStack 환경 내 Auto-scaling 방법

문서 ID	NI-Scaling-01
문서 버전	v2.0
작성일	2020.09.16
작성자	포항공과대학교 이도영

본 연구는 과학기술정보통신부 및 정보통신기술진흥센터의 정보통신·방송 연구개발사업의 일환으로 수행하였음. [IITP-2018-0-00749, 인공지능 기반 가상 네트워크 관리기술 개발]

- 문서 변경 기록

날짜	버전	변 경 내 역 설 명	작성자
2020.08.08	v1.0	초안 작성	이도영
2020.08.24	v1.1	그림 갱신	이도영
2020.08.29	v1.2	오타 수정 및 내용 보완	이도영
2020.09.02	v1.3	오타 수정	이도영
2020.09.16	v2.0	참고문헌 추가	이도영

목 차

요 약 문

오늘 날 5G 네트워크 시대가 도래 하면서, 급변하는 서비스 요구사항을 만족시키기 위해 유연하고 민첩한 네트워크 구축 및 관리가 요구되고 있다. 네트워크 기능 가상화 (NFV, Network Function Virtualization)는 이를 실현하기 위한 기술 중 하나로, 전용 하드웨어를 통해 제공되는 네트워크 기능을 소프트웨어 형태로 구현하여 상용 서버에서 가상 네트워크 기능 (VNF, Virtual Network Function)으로 운영하는 것이다. NFV는 네트워크 기능을 네트워크에 유연하게 적용할 수 있는 장점이 있지만, 한편으로는 수 많은 가상 자원을 생성하기 때문에 네트워크 관리를 복잡하게 만드는 원인이 된다. 복잡한 네트워크를 사람이 직접 관리하는 것은 어려운 일이기 때문에, 최근에는 네트워크 관리에 인공지능 (AI, Artificial Intelligence)을 적용하는 연구가 주목을 받고 있다.

NFV 환경의 Auto-scaling은 VNF 라이프 사이클 (Life-cycle) 관리 기능 중 하나로, VNF 인스턴스 (Instance)에 할당된 자원을 조절하거나 (Scale-up/down), VNF 인스턴스의 개수를 추가/제거하는 것이다 (Scale-in/out). 본 문서에서는 VNF 인스턴스의 개수를 조절하는 Scale-in/out의 경우를 다룬다. Scale-in/out은 동적으로 변하는 네트워크 상황에 대응하여 서비스 요구사항을 만족시킬 수 있는 기술이지만, 네트워크 상황에 대응하여 최적 VNF 인스턴스 개수를 결정하는 것이 요구된다. 또한, NFV 환경에서 운영되는 대부분의 서비스는 네트워크 기능을 적용한 Service Function Chaining (SFC)를 통해 제공된다. 따라서 NFV 환경에서의 Auto-scaling 문제는 단일 종류의 VNF 인스턴스들의 개수만 조절하는 것이 아니라, SFC를 구성하는 다계층 (Multi-tier) VNF 인스턴스들의 개수를 고려해야 한다. 본 문서에서는 NFV 환경에서 다계층 VNF 인스턴스들의 Scale-in/out을 위한 Deep Q-networks (DQN) 기반 Auto-scaling 방법에 대해 서술한다.

1 NFV 환경에서의 Auto-scaling 방법

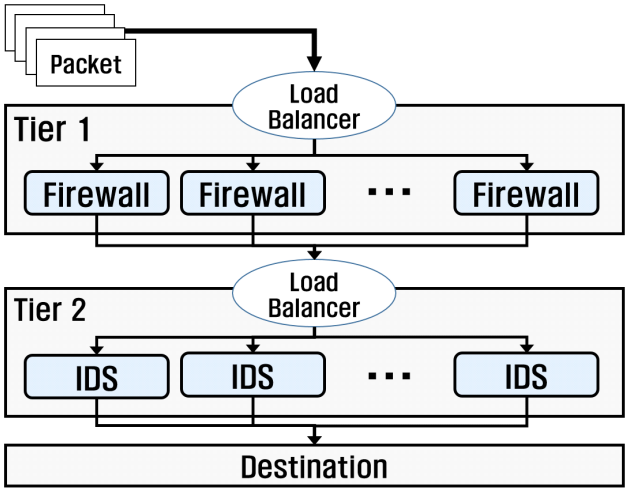
1.1 Auto-scaling 개요

NFV 환경의 Auto-scaling은 네트워크 변화에 대응하여 필요한 양의 자원을 동적으로 추가/제거할 수 있는 기능이다. 여기서 자원은 각 VNF 인스턴스 (Instance)에 할당 된 컴퓨팅 자원 (CPU, 메모리, 디스크 등) 또는 VNF 인스턴스들의 개수로 정의할 수 있다. VNF 인스턴스에 할당 된 컴퓨팅 자원을 조절하는 것은 Scale-up/down, VNF 인스턴스의 개수를 조절하는 것을 Scale-in/out이라 구분한다. 본 문서에서는 Scale-in/out을 위한 Auto-scaling 기능 및 방법에 대해서 서술한다.

Auto-scaling 기능을 쉽게 구현하는 방법은 임계값 기반 (Threshold-based auto-scaling)으로 Scale-in/out을 수행하는 것이다. 임계값 기반 방법은 측정된 성능 지표가 정의된 임계값에 초과 또는 미달하였을 경우에 Scaling을 수행한다. 이러한 방법은 임계값과 성능 지표가 되는 항목을 정의하면 별도의 복잡한 알고리즘의 구현 없이 쉽게 적용할 수 있다는 장점이 있다. 하지만, 최적의 VNF 인스턴스 개수를 유지하기 위한 임계값을 정의하기 위해서는 서비스가 운영되고 있는 네트워크에 대한 이해와 VNF 종류 등에 따른 전문적 지식을 요구한다는 단점이 있다. 또한, NFV 환경에서 제공되는 대부분의 서비스는 Service Function Chaining (SFC)을 통해 일련의 네트워크 기능을 트래픽에 적용하고 있다. 따라서 Auto-scaling을 적용할 때, 단일 종류의 VNF 인스턴스를 고려하는 것이 아니라 다계층 (Multi-tier)으로 구성된 SFC에서 각 계층의 VNF 인스턴스 개수 또한 고려해야 한다. 복잡한 NFV 환경에서 효율적으로 Auto-scaling 기능을 SFC에 적용할 수 있도록, 본 문서에서는 강화학습 (RL, Reinforcement Learning)을 활용한 Auto-scaling 방법에 대해 서술한다.

1.2 Service Function Chaining을 위한 Auto-scaling 방법

NFV 환경의 SFC를 위한 Auto-scaling은 SFC를 구성하는 각 종류의 VNF 인스턴스 개수를 조절하는 기능이다. SFC는 다계층 (Multi-tier) 구조를 가지며 각 계층마다 같은 종류의 VNF 인스턴스들이 존재한다. 같은 계층 (Tier)에 속하는 VNF 인스턴스들은 각 계층마다 존재하는 로드 밸런서 (Load-balancer)에 의해 트래픽을 분배 받는다. (그림 1)은 Firewall과 IDS (Intrusion Detection System), 2-계층으로 구성된 SFC의 예시를 보이고 있다. (그림 1)과 같은 SFC에 Auto-scaling을 적용할 경우, SFC를 구성하는 전체 VNF 인스턴스 개수 뿐 아니라, 각 계층에 존재하는 Firewall 인스턴스의 개수와 IDS 인스턴스의 개수도 고려해야 한다. 예를 들어, 트래픽이 급증하여 SFC의 부하가 심해져 SFC를 통과하는 패킷의 처리 성능이 저하된다면, 모든 계층의 인스턴스 개수를 증가시키는 것이 아니라 병목 현상을 발생시키는 계층에 해당 종류의 인스턴스를 추가해야 한다.



(그림 1) 2-계층 구조를 갖는 SFC 예시

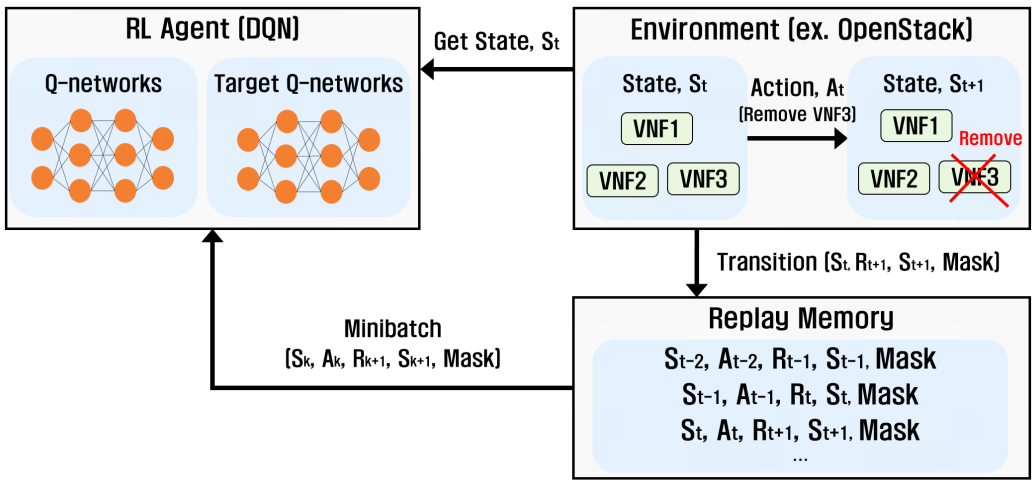
따라서 본 문서에서 서술하는 Auto-scaling 방법은 Scaling이 필요한 상황이 발생할 때, Scaling이 필요한 계층을 판단하고 해당 계층의 인스턴스를 어떤 물리 서버에서 추가/제거할지 결정한다.

2 DQN 기반 Auto-scaling 방법

2.1 Auto-scaling을 위한 강화학습 문제 정의

강화학습은 인공지능 기술 중 하나로, 최대의 누적 보상을 주는 행동 (Action)들을 수행할 수 있도록 시행착오를 거쳐 최적의 정책 (Policy)을 찾는 학습 방법이다. 일반적으로 강화학습은 에이전트 (Agent)와 환경 (Environment)으로 구성되며, 에이전트는 정책에 따라 현재 상태 (State)에서 특정 행동을 수행하게 된다. 이를 통해 다음 상태로 이동하게 되며, 수행한 행동에 대한 보상을 얻게 된다. 강화학습의 목적은 각 상태에서 다양한 행동을 수행하며 보상을 최대화하는 정책을 찾는 것이기 때문에 상태와 행동, 그에 대한 보상의 정의가 필요하다.

본 문서에서 서술하는 강화학습 기반 Auto-scaling 방법은 임계값에 기반한 Auto-scaling 방법과는 달리, 에이전트가 현재 상태에서 Scaling 수행 여부를 결정한다. 본 문서에서는 VNF 인스턴스 개수를 조절하는 Scale-in/out을 고려하기 때문에 Scaling 수행 여부에 따라 SFC를 구성하는 전체 VNF 인스턴스 개수가 변화한다. 따라서 VNF 인스턴스를 추가 (Add) 또는 제거 (Remove)하거나, 현재 개수를 유지 (Maintain)하는 것을 강화학습의 행동으로 정의할 수 있다. 본 문서에서 다루는 SFC 대상 Auto-scaling은 상태를 정의하는 데 고려할 수 있는 요소들이 다양하기 때문에, 무수히 많은 상태가 존재할 수 있다. 또한, 존재할 수 있는 상태가 많다는 것은 강화학습 문제를 해결할 때 각 상태를 표현하는 데 어려움이 있다는 것을 의미한다. 본 문서에서는 무수히 많은 상태를 표현하고 활용할 수 있도록, 강화학습 알고리즘 중 하나인 Deep Q-networks (DQN)을 사용하여 Auto-scaling 문제를 정의한다.

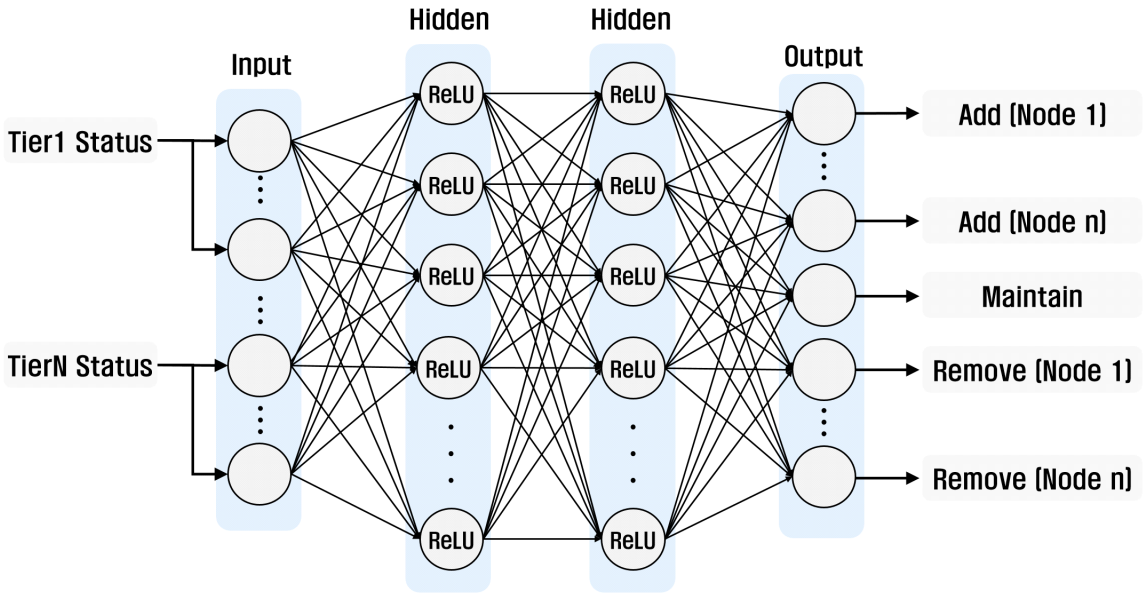


(그림 2) Deep Q-networks (DQN) 기반 Auto-scaling 문제 정의

DQN은 Q-learning과 마찬가지로, 특정 상태에서 행동을 수행할 때 얻을 수 있는 보상을 예측하는 지표인 Q-value를 반복적으로 학습하는 방법이다. 학습된 Q-value는 특정 상태에서

어떤 행동을 수행할지 결정하는 정책으로 사용한다 (예를 들어, 특정 상태에서 Q-value가 가장 큰 행동을 선택). 다만, DQN에서는 Q-value를 학습하는 방법이 Q-learning과는 차이가 존재한다. Q-learning에서는 각 상태의 행동에 따른 Q-value를 표현하기 위해 일반적으로 표형식 표현 (tabular representation)을 사용하지만, 이는 복잡한 Auto-scaling 문제의 많은 상태를 나타내는 데 적합하지 않다. 따라서 DQN은 표형식 표현 대신 심층 네트워크를 사용하여 Q-value를 표현하고 갱신한다. 본 문서의 DQN은 (그림 2)처럼 에이전트에서 심층 네트워크를 사용한 DQN으로 Q-value를 학습하며, OpenStack 환경에서 Auto-scaling을 수행해 VNF 인스턴스 개수를 조절한다. 또한, 에이전트는 Replay Memory로부터 데이터를 Mini-batch 방식으로 받아 학습을 한다.

에이전트에서는 안정적인 학습을 위해 두 개의 심층 네트워크인 Q-network와 Target Q-network를 사용한다. 이는 Q-value를 학습하는 과정에서 최적 값으로 수렴하지 않고 발산하는 것을 방지한다. 또한, 특정 상태에서 Scaling 행동을 수행했을 때 받게 되는 보상과 다음 상태, 그리고 Scaling이 성공했는지 여부를 Replay Memory에 $(S_t, A_t, R_{t+1}, S_{t+1}, Mask)$ 형태로 저장한다. S, A, R은 각각 상태, 행동, 보상이며, $Mask$는 선택 된 Scaling이 정상적으로 수행되었는지를 나타내는 값이다. 예를 들어, 물리 서버에 가용 자원이 없어 VNF 인스턴스를 추가할 수 없는 경우는 Scaling에 실패하여 $Mask$ 값이 0으로 할당된다. Replay Memory에 저장된 데이터들을 Mini-batch 방식으로 학습을 하면, 데이터 간 상관관계로 인해 잘못된 네트워크 파라미터를 학습하는 것을 방지할 수 있다.



(그림 3) Auto-scaling을 위한 DQN 모델

본 문서의 Auto-scaling을 위한 DQN은 (그림 3)처럼 각 계층의 Status를 입력으로 받아 수행해야 하는 Scaling 행동을 출력한다. 각 계층의 Status는 5개의 데이터로 구성이 되며, 데이터 종류는 계층의 평균 CPU 사용량, 평균 메모리 사용량, 디스크 작업 수행 횟수, VNF 인스턴

스 개수, VNF 인스턴스의 분포도이다. 이 중 평균 CPU 사용량, 평균 메모리 사용량, 디스크 작업 횟수는 상태를 정의하는 시점부터 10초 전까지의 관측 데이터의 평균으로 계산한다. 계층 Status에서 CPU와 메모리를 고려하는 이유는 해당 자원들이 충분치 않으면 패킷 처리가 지연 (Delay)되거나 이로 인한 패킷 손실 (Packet loss)을 발생 시키는 등, 패킷 처리 성능에 영향을 크게 미치는 요소들이기 때문이다 [3, 4]. 또한, 디스크 작업 수행 횟수는 디스크를 읽거나 쓰는 작업 횟수를 의미하는데, 메모리 자원이 과도하게 사용될 경우 Swap 작업이 발생하여 디스크 작업 횟수가 높게 측정될 수 있다. Swap 작업은 메모리에 저장할 데이터 일부를 디스크에 저장하는 것인데, 메모리 작업에 비해 디스크 작업은 속도가 느리기 때문에 병목 현상을 발생시켜 간접적으로 패킷 처리 성능에 영향을 미친다. 그 외에는 각 계층에 속한 VNF 인스턴스 개수와 VNF 인스턴스 분포도를 계층 Status로 고려한다. VNF 인스턴스 분포도는 OpenStack 환경 내에서 VNF 인스턴스를 생성할 수 있는 총 물리 서버 개수 대비 실제 VNF 인스턴스가 배치된 물리 서버 개수로 계산된 값이다. 예를 들어 10개의 가용 서버가 있는데, 그 중 3개의 서버에 현재 계층의 VNF 인스턴스가 배치되어 있을 경우, 분포도 값은 0.3이 된다.

(그림 3)처럼 2개의 은닉 층 (Hidden layer)을 가진 DQN에 각 계층의 Status를 입력하면, Scale-out의 VNF 인스턴스 추가 (Add), Scale-in의 VNF 인스턴스 제거 (Remove), 현재 배치되어 있는 VNF 인스턴스를 그대로 유지 (Maintain)하는 행동을 출력한다. 이 때, 본 문서의 Auto-scaling 방법은 단순히 VNF 인스턴스 개수를 조절하는 것이 아니라, 어떤 물리 서버에서 Scaling을 수행할 지도 결정한다.

DQN 모델을 통해 상태를 입력 값으로 넣고 출력으로 Scaling 행동을 수행하면, 현재 상태에서 수행한 행동의 보상 값을 계산해야 한다. DQN 기반 Auto-scaling 방법은 (수식 1)을 통해 보상 값을 계산한다.

$$R_{t+1} = -\frac{resTime}{SLO} + \alpha Dist._{VNF} \times e^{-\beta Node_{Util.}} \qquad \text{(수식 1)}$$

(수식 1)에서 $resTime$은 Scaling을 수행한 SFC를 통해 패킷을 전송하고, 응답 패킷을 받을 때까지 소요되는 응답 시간 (Response time)을 의미한다. 또한, DQN 기반 Auto-scaling 방법에서는 SLO (Service Level Objectives)로 응답 시간을 활용한다. SFC를 경로를 통해 측정되는 응답 시간은 가변성이 크기 때문에, 측정된 결과를 그대로 활용하면 보상 값에 큰 영향을 미치게 된다. 따라서 본 문서에서는 측정된 응답 시간인 $resTime$를 그대로 사용하는 것이 아니라, 미리 정의된 SLO 대비 응답 시간이 얼마나 되는 지를 비율로 환산하여 보상 값에 반영한다. 그 외 (수식 1)에서의 $Node_{Util.}$와 $Dist._{VNF}$는 (수식 2)로 계산되는 값들이다.

$$Node_{Util.} = \frac{Node_{used}}{Node_{total}}, \quad Dist._{VNF} = \prod_{i=1}^{n} \frac{VNF_{node_i}}{VNF_{total}} \qquad \text{(수식 2)}$$

$Node_{Util.}$는 OpenStack 환경에서 가용할 수 있는 총 물리 서버 개수 ($Node_{total}$) 대비 SFC를 구성하는 VNF 인스턴스가 배치되어 있는 물리 서버 개수 ($Node_{used}$)의 비율을 의미한다. 이

는 SFC를 구성하는 데 있어서 많은 수의 물리 서버를 활용하지 않고, 적은 수의 물리 서버를 활용했는지 여부를 반영한다. 반면, $Dist._{VNF}$는 SFC를 구성하는 VNF 인스턴스들의 분포도를 나타내는 값이다. SFC를 구성하는 전체 VNF 인스턴스 개수 (VNF_{total}) 대비 VNF 인스턴스가 배치된 각 물리 서버에서 실행되는 VNF 인스턴스 개수의 비율을 곱하여 계산한다. $Dist._{VNF}$는 VNF 인스턴스들이 적은 수의 물리 서버에 밀집되어 배치되면 높은 값을 가지게 되고, 많은 서버들에 분산 배치되어 있을 경우 작은 값을 가지게 된다. SFC의 각 계층 내 로드 밸런서는 VNF 인스턴스들에게 트래픽을 분배하기 때문에, SFC를 구성하는 VNF 인스턴스들이 많은 물리 서버에 분산 배치되어 있을 경우 트래픽 또한 해당 물리 서버들로 전달되어야 한다. 결국, 각 계층에 속한 VNF 인스턴스들이 크게 분산되어 있다면, 트래픽의 패킷 전달 시간과 응답 시간이 가변적일 수 있다.

(수식 2)에서 계산된 값들은 (수식 1)에서 활용되어, 최종적으로 Scaling 행동의 보상 값으로 환산된다. (수식 2)의 값들은 지수 함수 e에서 사용되는데, $Dist._{VNF}$와 $Node_{Util.}$의 값은 가중치 α와 β로 보정된다. 따라서 (수식 1)로 계산되는 보상은 Scaling 행동으로 인해 SFC를 통과하는 패킷의 응답 시간이 짧고, SFC를 구성하기 위해 적은 물리 서버를 활용하며 VNF 인스턴스들이 밀집하여 배치되었을 경우 높은 값을 갖게 된다.

본 문서에서 서술하는 Auto-scaling 방법은 다계층으로 이루어 진 SFC를 대상으로 하기 때문에 어떤 계층에 Scaling을 적용할지 결정하는 과정이 요구된다. 특정 상태에서 Scaling이 필요하다고 에이전트가 판단했을 경우, (수식 3)에 의해 Scaling을 적용 할 계층을 선택한다. (수식 3)은 각 계층마다 점수 (Score)를 계산하여, 가장 높은 점수를 가지는 계층에 Scaling을 적용한다. 각 계층의 점수는 $mask$와 함수 $f(Tier_i)$ 값의 곱으로 정의된다. 이 중, $mask$는 해당 계층 내에서 Scaling이 불가능한 경우에 0, 가능한 경우에는 1을 할당하여 점수를 보정한다.

$$score = mask \times f(Tier_i) \qquad \text{(수식 3)}$$

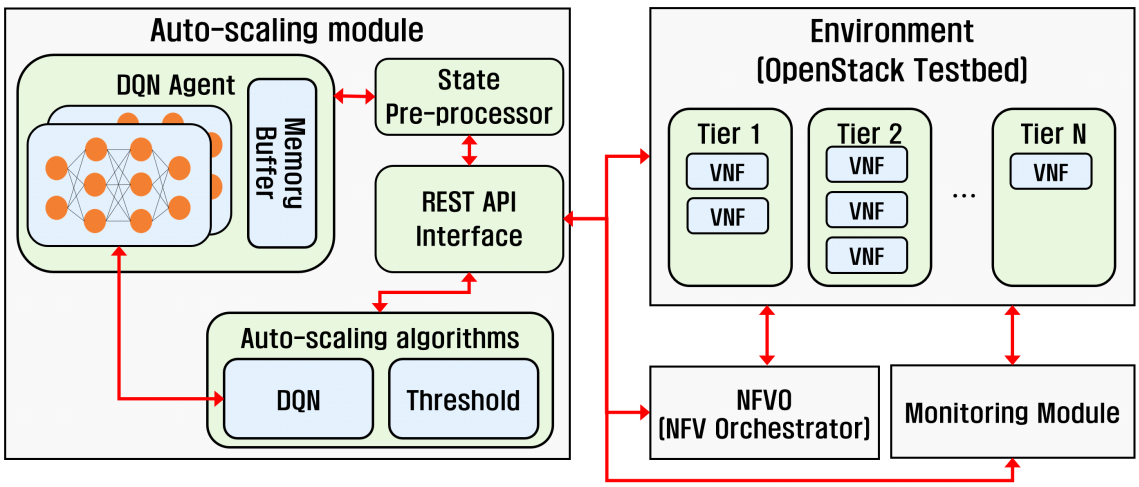
$f(Tier_i)$는 (수식 4)와 (수식 5)에 의해 정의되며, 각 계층의 함수가 Scale-in/out에 얼마나 적합한지를 보여주는 함수이다. Scaling을 적용할 계층은 각 계층의 CPU 사용량과 메모리 사용량을 기반으로 한다. 이는 (수식 4)처럼 $resUtil$로 정의하며, 각각 가중치 α와 β로 보정한다. (수식 5)는 지수 함수의 입력 값으로 $resUtil$을 활용하여 각 계층이 Scaling 함수에 적합한지를 점수로 환산한다. 지수 함수의 값에는 해당 계층의 VNF 인스턴스 분포도를 곱한다. Scale-in의 경우에는 SFC를 구성하는 VNF 인스턴스들이 배치되어 있는 물리 서버 대비 해당 계층에 속하는 VNF 인스턴스들이 배치되어 있는 물리 서버의 개수를 곱한다. Scale-out의 경우에는 해당 값의 역수를 취하여 곱해준다. 즉, Scale-in의 경우에는 자원 사용량이 낮고, VNF 인스턴스들이 여러 물리 서버에 분산되어 있는 계층에 큰 점수를 준다. 반면, Scale-out의 경우에는 자원 사용량이 높고, VNF 인스턴스들이 소수 물리 서버에 밀집해 있는 계층에 큰 점수를 준다. 본 문서의 DQN 기반 Auto-scaling 방법은 참고문헌 [1]에 자세히 서술되어 있다.

$$resUtil = \alpha CPU_{Util.} + \beta MEM_{Util.} \qquad \text{(수식 4)}$$

$$f(Tier_i) = \begin{cases} \frac{Node_{tier}}{Node_{used}} \times e^{-resUtil}, & \text{if scale-in} \\ \frac{Node_{used}}{Node_{tier}} \times e^{resUtil}, & \text{else if scale-out} \end{cases} \qquad \text{(수식 5)}$$

2.2 DQN 기반 Auto-scaling 모듈 구현

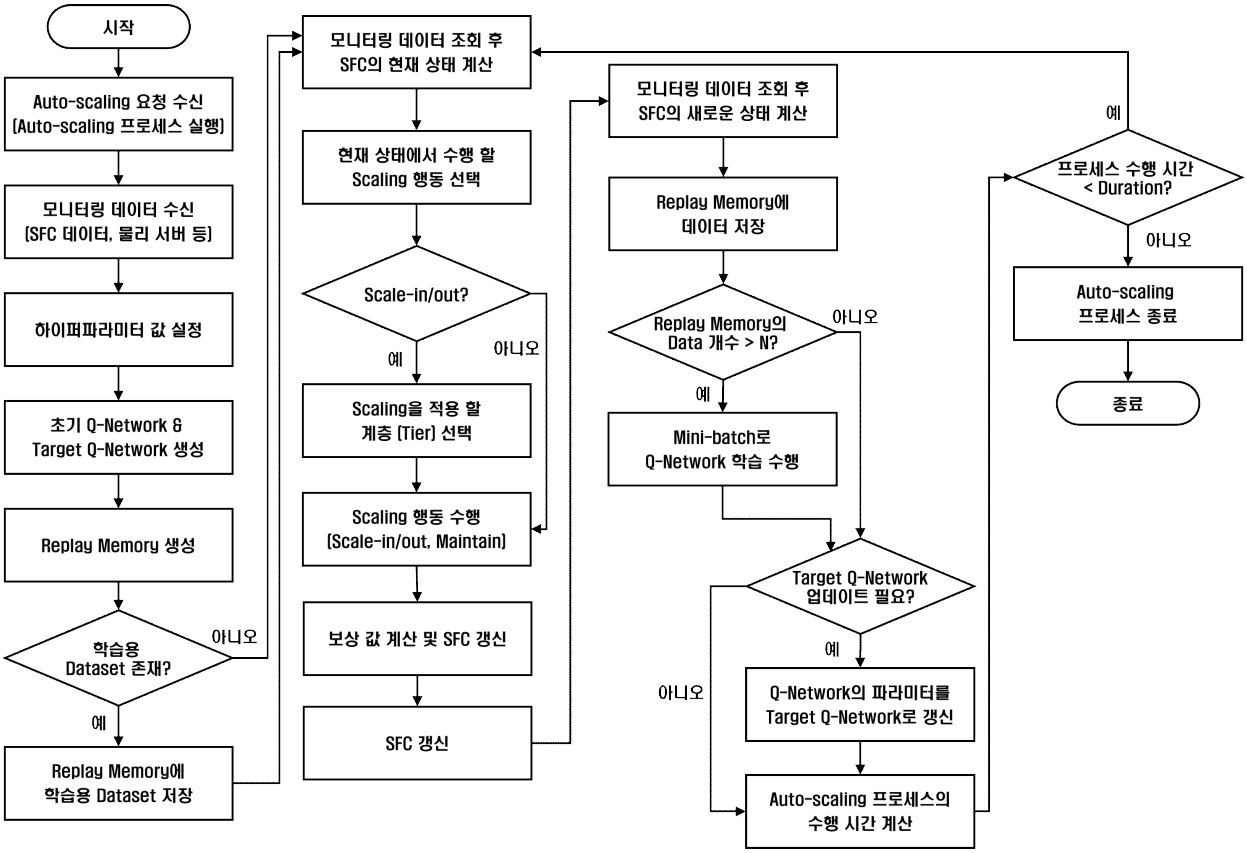
본 문서의 DQN 기반 Auto-scaling 기능은 OpenStack 환경에서 실제 동작할 수 있는 모듈(Module) 형태로 구현되었다. (그림 4)는 DQN 기반 Auto-scaling 모듈 (이하 Auto-scaling 모듈)의 구조도를 보인다. OpenStack 환경에는 다수의 VNF 인스턴스들이 설치될 수 있고, Networking-SFC 플러그인 [2]를 통해 이들을 연결시켜 SFC를 생성할 수 있다. Auto-scaling 모듈은 Scaling을 적용할 SFC를 식별한 후, 지속적으로 해당 SFC에 Scaling 행동을 적용한다. NFVO (NFV Orchestrator)는 Auto-scaling 모듈 및 OpenStack 환경과 상호작용하며 SFC에 VNF 인스턴스를 추가하거나 제거하는 기능을 제공한다. SFC에 적용하기 위한 Auto-scaling 요청이 NFVO에 도착하면, NFVO는 해당 요청 메시지를 REST API를 통해 Auto-scaling 모듈로 전달한다. 모니터링 모듈 (Monitoring Module)은 Auto-scaling 기능 수행을 위해 Auto-scaling 모듈에서 요구하는 데이터를 제공한다. Auto-scaling 모듈은 임계값 (Threshold) 기반 Scaling 알고리즘과 DQN 기반 알고리즘을 제공한다.



(그림 4) Auto-scaling 모듈 구조도

임계값 기반으로 Auto-scaling을 적용할 경우, Scale-in을 위한 임계값과 Scale-out을 위한 임계값을 각각 정의한다. 본 문서에서는 SFC를 통해 측정되는 응답 시간 (Response time)을 임계값의 성능 지표로 사용한다. 측정된 응답시간이 Scale-in을 위한 임계값보다 작으면

Scale-in을 수행하며, Scale-out을 위한 임계값보다 크면 Scale-out을 수행한다. 또한, SFC는 다계층으로 구성되기 때문에 Scaling을 적용할 계층을 선택해야 한다. 임계값 기반으로 Scaling 여부를 결정했을 때, 어떤 계층에서 Scaling을 수행할지 또한 결정된다.



(그림 5) Auto-scaling 모듈의 DQN 기반 Auto-scaling 기능 수행 순서도

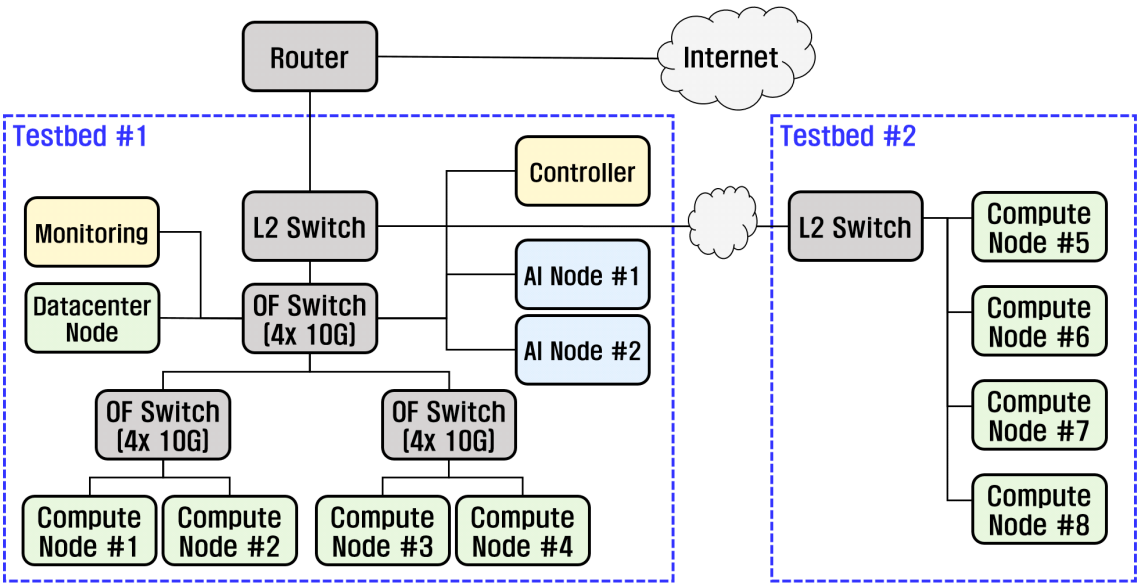
임계값 기반과는 달리, DQN 기반으로 Auto-scaling을 적용할 경우 학습을 위한 과정이 필요하다. (그림 5)는 DQN 기반 Auto-scaling을 요청했을 때, Auto-scaling 기능을 수행하는 순서를 보인다. Auto-scaling 모듈이 Auto-scaling을 적용할 SFC의 이름과 Auto-scaling을 수행하는데 필요한 파라미터 (Parameter)가 포함 된 요청 메시지를 수신하면 Auto-scaling 프로세스를 실행한다. 이 때, Auto-scaling 프로세스는 쓰레드 (Thread)로 동작한다. 프로세스가 실행되면 Auto-scaling을 적용할 SFC의 데이터와 물리 서버 정보 등, Auto-scaling에서 필요한 데이터를 모니터링 모듈로 요청하여 받아온다. 이 후, DQN의 하이퍼파라미터 (Hyperparameter) 값을 설정하고, Q-Network와 Target Q-Network를 생성한다. 다음으로는 Replay Memory를 생성하는데, 만약 Replay Memory로 불러올 학습용 데이터 (Dataset)이 미리 파일 형태로 존재한다면, 해당 데이터를 읽어서 Replay Memory에 저장한다. Replay Memory 생성까지 완료된 후에는 본격적인 Auto-scaling을 수행한다.

먼저, Auto-scaling 대상이 되는 SFC의 Status를 가져온 후, 현재 상태를 계산한다. 계산된 상태는 DQN에 입력되고, 어떤 물리 서버에서 어떤 Scaling을 수행할 것인지를 나타내는 행동이 출력된다. 출력된 결과가 Scale-in/out 일 경우, Scaling을 적용할 계층 (Tier)를 선택한다.

계층까지 결정된 후에는 DQN의 결과로 선택된 Scaling 행동을 수행한다. 행동을 수행하고 나서, 에이전트는 Scaling으로 인한 보상 값을 계산하고, 새롭게 추가 또는 제거된 VNF 인스턴스를 반영하여 SFC를 갱신한다. SFC 갱신을 완료한 후에는 갱신한 SFC의 Status를 가져 온 후, 새로운 상태를 계산한다. 새로운 상태까지 계산한 후에는 (현재 상태, 행동, 보상, 새로운 상태, 행동 성공 여부)를 Replay Memory에 저장한다. Replay Memory에 최소 개수 N개 이상의 데이터가 쌓이면 이를 Mini-batch로 샘플링 (Sampling)하여 Q-Network를 학습한다. 본 문서의 DQN은 Scaling 행동을 일정 횟수 반복할 때마다 Q-Network의 학습 된 네트워크 파라미터를 Target Q-Network로 복사 및 갱신한다. 매 Scaling 행동을 수행할 때마다 Auto-scaling 모듈은 Auto-scaling 프로세스가 실행되고 현재까지 걸린 시간을 계산한다. 미리 정의된 수행 시간 한도 (Duration)보다 실제 수행된 시간이 길면 Auto-scaling 프로세스를 종료한다. 반면, Auto-scaling 프로세스의 만료까지 시간이 남아있을 경우, 다시 현재 SFC의 상태를 계산한 후 Scaling 과정을 반복한다. 본 문서에서 구현한 Auto-scaling 모듈은 Github를 통해 공개되어 있다 [5].

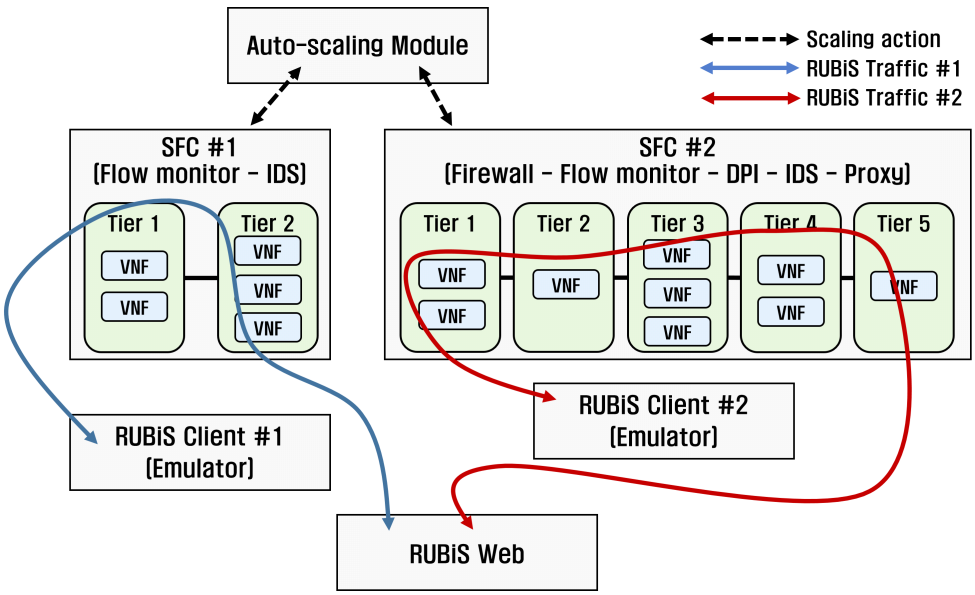
2.3 성능 검증

DQN 기반 Auto-scaling 방법의 성능 검증을 위해 (그림 6)과 같이 OpenStack 기반으로 테스트베드를 구축하였다. 테스트베드는 OpenStack 환경을 관리하는 컨트롤러 노드 (Controller Node)와 VNF 인스턴스가 설치 될 수 있는 8개의 컴퓨트 노드 (Compute Node)로 구성된다. 테스트베드 내에 VNF 인스턴스를 설치할 때에는 임의의 컴퓨트 노드를 선택하여 VNF가 설치된 VM을 생성하며, 각 VM에는 하나의 VNF만 동작한다. 본 문서의 OpenStack 테스트베드 구축을 위해 사용한 OpenStack 설치 스크립트 파일은 Github를 통해 공개되어 있다 [6].



(그림 6) OpenStack 기반 테스트베드

성능 검증 시나리오는 (그림 7)처럼 성능 검증을 위해 각각 2-계층 SFC와 5-계층 SFC를 구성하였다. 2-계층 SFC는 Flow monitor와 IDS로 구성되며, 5-계층 SFC는 Firewall, Flow monitor, DPI, IDS, Proxy로 구성된다. 이 때, Firewall VNF는 Iptables [7], Flow monitor는 ntopng [8], DPI는 nDPI [9], IDS는 Suricata [10], 그리고 Proxy 기능을 제공하는 VNF는 HAProxy [11]을 사용한다. 본 문서의 Auto-scaling은 Scaling 대상이 되는 SFC에서 측정 된 응답 시간을 임계값의 지표와 DQN에서 보상을 계산할 때 요소로 사용한다. 또한, Auto-scaling의 성능 검증을 위해서 동적으로 변화하는 트래픽을 생성할 필요가 있기 때문에, 본 성능 검증 시나리오에서는 RUBiS [12]를 활용한다. RUBiS는 eBay와 같은 경매 사이트 환경을 만들고 성능 시험을 수행할 수 있는 오픈소스 도구이다. RUBiS는 경매 사이트에 해당하는 RUBiS Web과 경매 사이트에 접근하는 사용자 요청 트래픽을 동적으로 생성하는 RUBiS Client를 에뮬레이터 (Emulator)로 제공한다. 각 VNF 인스턴스에 할당 된 자원은 vCPU 1개, RAM 1024MB, 디스크 10GB이다. 본 문서의 DQN 기반 Auto-scaling의 성능 검증을 위해, 2-계층 SFC와 5-계층 SFC에 임계값 기반 Auto-scaling 방법과 DQN 기반 Auto-scaling 방법을 적용한 경우를 비교하였다.



(그림 7) 성능 검증 시나리오를 위한 SFC 구성

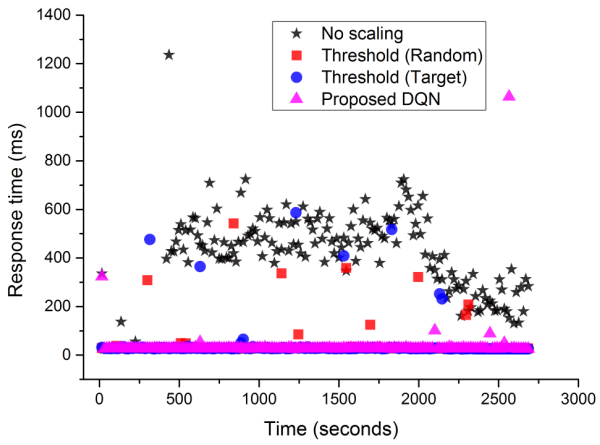
DQN 학습에 사용한 초기 변수 값은 [표 1]과 같다. 학습률 η의 초기 값은 0.01로 고정하여 사용하였으며, 미래의 보상을 얼마나 가중치를 두고 환산할지 결정하는 할인율 γ 값은 0.98로 설정하였다. 또한, ε-greedy 알고리즘의 초기 ϵ값은 0.08로 설정하였다. 마지막으로, Replay Memory 크기는 5000으로 설정하여, 최대 5000개의 데이터를 가질 수 있도록 설정하였으며 Batch 크기는 32로 설정해 학습을 위한 샘플링 과정에서 32개의 데이터를 Replay Memory서 가져온다.

표 1. DQN 초기 변수 정보

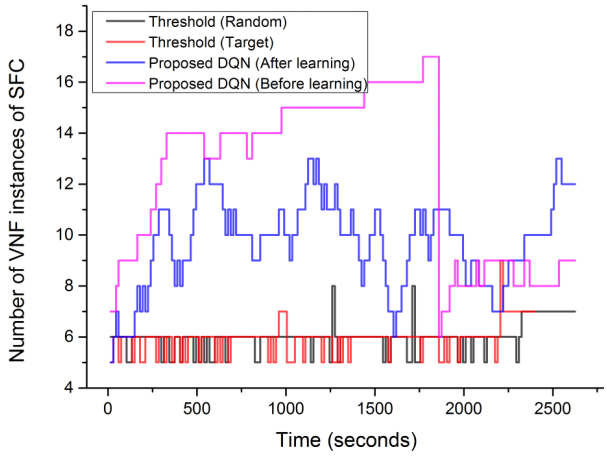
변수 종류	값
η (학습률)	0.01
γ (할인율)	0.98
ϵ (탐험 확률)	0.08
Replay Memory 크기	5000
Batch 크기 (Sampling 단위)	32

모든 설정을 마치고 나서, 서로 다른 방법으로 세 가지의 Auto-scaling 방법을 각 2-계층 SFC와 5-계층 SFC에 적용하였다. 각 Auto-scaling 방법은 다음과 같다.

1. **Threshold (Random)**: 임계값 기반 Auto-scaling 적용하며 임의 Scaling 적용 계층 선택
2. **Threshold (Target)**: 임계값 기반 Auto-scaling 적용하며 특정 Scaling 적용 계층 선택
3. **Proposed DQN**: DQN 기반 Auto-scaling 적용하며 특정 Scaling 적용 계층 선택



(그림 8) 5-계층 SFC 내 응답시간 측정



(그림 9) 5-계층 SFC 내 VNF 인스턴스 개수

(그림 7)과 같은 성능 검증 시나리오에서 각 RUBiS Client를 통해 트래픽을 동적으로 발생시키면, SFC를 통해 RUBiS Web으로 약 45분 (2700초) 동안 트래픽을 생성한다. 5-계층 SFC에 트래픽을 발생시키면서 서로 다른 Scaling 방법을 적용했을 경우에 측정된 응답 시간과 VNF 인스턴스 개수의 변화 결과는 (그림 8, 9)와 같다. (그림 8)에서 보이는 것처럼 Scaling을 적용하지 않았을 경우에는 트래픽으로 인해 SFC에 큰 부하가 발생해 패킷 처리 성능이 저하되기 때문에 대부분의 시간 동안 높은 응답 시간을 가진다. 반면, 임계값 기반 또는 DQN 기반의 Auto-scaling을 적용했을 때는 전체적으로 짧은 응답 시간이 측정되며, 제안하는 방법 (Proposed DQN)의 경우에 전반적으로 제일 짧은 응답 시간이 측정된다.

(그림 9)는 Auto-scaling을 수행하는 동안 5-계층 SFC를 구성하는 VNF 인스턴스 개수의 변화를 보이는 그래프이다. 임계값 기반으로 Auto-scaling을 수행했을 경우에는 임계값 대비 응답시간이 미달하거나 초과했을 경우에만 발생하기 때문에 큰 개수의 변화는 없다. 반면, Proposed DQN에서는 현재 상태에 대응하여 동적으로 VNF 인스턴스 개수를 조절한다. 특히,

DQN은 충분히 학습을 했는지 여부에 따라 정책이 다르기 때문에 현재 상태에서 선택할 수 있는 Scaling 행동이 다를 수 있다. 이를 비교하기 위해 (그림 9)의 그래프에서는 학습이 충분히 되기 전과 후의 Scaling으로 인한 SFC의 VNF 인스턴스 개수의 증감을 함께 보이고 있다. 구체적으로는, 2000개의 데이터를 미리 Replay Memory에 저장해놓고 학습을 200번 반복한 Scaling 정책과 학습을 수행하기 전 초기 Scaling 정책으로 DQN 기반 Auto-scaling을 적용하며 인스턴스 개수의 증감 상태를 관찰하였다. 이를 통해, 학습이 되기 전 DQN은 이미 결정된 VNF 인스턴스의 개수를 유지하려는 경향을 보였으며, 학습이 된 후에는 동적인 트래픽 변화에 맞춰 인스턴스 개수를 유연하게 조절하는 것을 확인할 수 있었다. 즉, (그림 8, 9)를 통해 제안하는 DQN 기반 Auto-scaling 방법은 학습을 충분히 한 후에 안정적으로 트래픽 증감에 맞춰 VNF 인스턴스 개수를 조절할 수 있다는 것을 확인하였다.

표 2. Auto-scaling 성능 검증 결과

		평균 응답 시간	표준 편차	SLO 위반 비율
2-계층 SFC	Threshold (Random)	38.92ms	74.36	5.2%
	Threshold (Target)	35.26ms	55.52	4.2%
	Proposed DQN	26.38ms	10.71	5.3%
5-계층 SFC	Threshold (Random)	42.57ms	65.53	7.1%
	Threshold (Target)	45.36ms	82.04	5.7%
	Proposed DQN	35.91ms	81.38	4.6%

[표 2]는 2-계층 SFC와 5-계층 SFC에 서로 다른 방법의 Auto-scaling 기능을 적용했을 때 측정된 응답 시간의 평균, 표준 편차, SLO 위반 비율의 결과이다. 이 때, Proposed DQN은 학습을 충분히 한 상태에서 각 성능 지표를 측정하였다. 2-계층 SFC와 5-계층 SFC에 각 Auto-scaling 방법을 적용했을 때, Proposed DQN가 가장 짧은 평균 응답 시간을 보였다. 또한, 표준 편차에서도 비교적 작은 값을 보였다. 다만, DQN 기반 Auto-scaling에서는 가끔 응답 시간이 다른 Auto-scaling 방법에 비해 지나치게 큰 값이 측정될 때가 있어서 표준 편차와 SLO 위반 비율에 영향을 미쳤다. 이는 강화학습 알고리즘의 특성 상, 가끔 임의의 행동을 수행하는 것과 최적화가 완전히 이루어지지 않은 정책으로 인해 잘못된 행동을 수행하기 때문일 것으로 추정된다. 하지만, 학습이 충분히 수행된 DQN을 통해 Auto-scaling을 적용 했을 때, 임계값 기반으로 Auto-scaling을 수행했을 때보다 전반적으로 낮은 SLO 위반 비율을 보이는 것을 확인하였다. 이 때, SLO 위반은 매 주기마다 측정된 응답 시간이 SLO 기준 값을 초과했을 때로 판정하였다. SLO 위반 비율을 위한 기준 값은 45ms로 정의했는데, 해당 값은 큰 부하가 발생하지 않았을 때, 본 성능 검증 환경에서 SFC를 통해 RUBiS Web을 조회하는 메시지의 응답 시간들을 측정하여 결정한 값이다. 결과적으로, 제안하는 DQN 기반 Auto-scaling 방법은 응답시간 및 SLO 위반 비율 측면에서 효과적이었으며, 특히 5-계층 SFC에 적용했을 때 SLO 위반 비율은 4.6%로 관찰되었다.

3 DQN 기반 Auto-scaling 모듈 사용법

3.1 환경 구축

본 문서에서 다루는 DQN 기반 Auto-scaling 방법은 OpenStack 환경에서 동작할 수 있는 형태로 구현되었다. 앞서 서술한 것처럼, 구현된 Auto-scaling 모듈은 임계값 기반 또는 DQN 기반의 Auto-scaling 프로세스를 생성하기 위해 OpenStack 테스트베드 내 존재하는 각 VNF 인스턴스들의 데이터를 활용하고, VNF 인스턴스를 추가 또는 제거할 때마다 SFC를 OpenStack 내 갱신해야 한다. 이를 위해 OpenStack 기반 환경 뿐 아니라, Auto-scaling 모듈과 상호 연동되는 NFVO와 모니터링 모듈의 설치가 선행되어야 한다. OpenStack 환경 구축을 위한 스크립트 파일 및 설명서 [6]과 NFVO 및 모니터링 모듈 설치를 위한 코드, 설명서 [13]은 Github를 통해 공개되어 있기 때문에 본 문서에서 해당 요소들의 설치 과정은 생략한다. OpenStack 환경이 구축되고, NFVO와 모니터링 모듈이 설치한 후에는 Auto-scaling 모듈을 설치해서 실행할 수 있다. 본 문서에서 모듈 설치와 실행은 Ubuntu 16.04에서 수행하였다.

Auto-scaling 모듈은 Python으로 작성되었으며, Python v3.5.2 이상에서 실행하는 것을 권장한다. Auto-scaling 모듈을 설치하고 실행시키기 위해서는 관련 패키지들을 미리 포함하고 설치해야 한다. Python 환경에서는 패키지의 버전 문제에 따른 호환 문제가 쉽게 발생할 수 있으므로, 다른 기능들과 충돌을 막기 위해 가상 환경에서 Auto-scaling 모듈을 설치한다. 가상 환경 생성은 virtualenv를 사용한다. 아래 명령어들을 통해 Python3 패키지 설치 도구인 pip3를 먼저 설치한 후, 이를 통해 virtualenv까지 설치할 수 있다. virtualenv를 설치한 후에는 임의의 가상 환경 명을 갖는 독립된 환경을 구축할 수 있다. 가상 환경에서 포함되고 설치되는 Python 관련 패키지들은 다른 환경과는 서로 간섭이 되지 않아 Auto-scaling 모듈을 위한 패키지를 안정적으로 불러오고 설치할 수 있다.

```
# Python 패키지 설치 도구 pip3 설치 후 가상환경 구성을 위한 virtualenv 설치
sudo apt-get update
sudo apt-get install python3-pip
sudo pip3 install virtualenv

# virtualenv를 통한 가상 환경 생성 후 활성화
# test 대신 원하는 환경 명 지정 가능
virtualenv test
source test/bin/activate

# 가상 환경 비활성화
deactivate
```

virtualenv 명령어를 통해 가상 환경을 생성한 후에 활성화하면 CLI 계정 명 왼쪽에 현재 활성화된 가상 환경 명이 표시된다. (그림 10)은 test라는 가상 환경을 생성하고 활성화 한 후에, 비활성화까지 진행한 화면을 보이고 있다.

```
ubuntu@dy-test-1:~$ virtualenv test
created virtual environment CPython3.5.2.final.0-64 in 285ms
  creator CPython3Posix(dest=/home/ubuntu/test, clear=False, global=False)
  seeder FromAppData(download=False, pip=bundle, setuptools=bundle, wheel=bundle, via=copy, app_data_dir=/home/ubuntu/.local/sha
re/virtualenv)
    added seed packages: pip==20.1.1, setuptools==49.2.0, wheel==0.34.2
  activators PowerShellActivator,BashActivator,CShellActivator,PythonActivator,FishActivator,XonshActivator
ubuntu@dy-test-1:~$
ubuntu@dy-test-1:~$ ls
test
ubuntu@dy-test-1:~$
ubuntu@dy-test-1:~$ source test/bin/activate
(test) ubuntu@dy-test-1:~$
(test) ubuntu@dy-test-1:~$ deactivate
ubuntu@dy-test-1:~$
```

(그림 10) virtualenv를 통한 가상 환경 생성

3.2 Auto-scaling 모듈 설치 및 실행

앞선 과정에서 pip3 설치까지 완료하고 가상 환경까지 활성화한 후에 Auto-scaling 모듈 설치를 진행한다. Github [5]에 공개 되어있는 Auto-scaling 모듈을 내려 받고, 실행에 필요한 패키지를 설치한다. Auto-scaling 모듈을 위한 패키지는 requirements.txt 파일에 정리되어 있으며, 이를 사용하여 패키지를 설치한다. 패키지를 설치한 후에는 Python 명령어를 통해 Auto-scaling 모듈을 실행한다. 아래 명령어를 순서대로 입력하면 Auto-scaling 모듈 내려 받기, 필수 패키지 설치, 모듈 실행까지 수행할 수 있다. 참고로, Auto-scaling 모듈은 pytorch를 기반으로 DQN을 구현하였기 때문에 torch 패키지를 설치해야 한다. 하지만, torch 패키지 설치 명령어는 사용하는 운영체제, GPU 종류 등에 따라 가변적이기 때문에 torch 설치는 pytorch 공식 홈페이지 (https://pytorch.org/)의 설명을 바탕으로 별도로 설치할 필요가 있다.

```
# Auto-scaling 모듈 다운로드 후 필수 패키지 설치
git clone https://github.com/dpnm-ni/ni-auto-scaling-module-public
cd ni-auto-scaling-module-public
sudo pip3 install -r requirements.txt

# Auto-scaling 모듈 실행
python3 -m server
```

Auto-scaling 모듈 실행 명령어를 입력하면 Flask 기반으로 웹 서버가 동작한다. Auto-scaling 모듈은 Auto-scaling 생성 요청 메시지를 받으면 임계값 기반 또는 DQN 기반 Auto-scaling 프로세스를 생성한다. 또한, Auto-scaling 프로세스의 정보 조회 및 삭제 등의 기능을 제공하기 때문에 웹 서버로 동작하면서 메시지를 처리한다. Auto-scaling 모듈에서 설정되어 있는 기본 웹 서버 포트는 8004이며, 포트 변경을 원할 경우에는 내려 받은

Auto-scaling 모듈 폴더의 server/__main__.py 파일에서 (그림 11)과 같이 port라고 명시된 부분을 수정한다.

```
def main():
    app = connexion.App(__name__, specification_dir='./swagger/')
    app.app.json_encoder = encoder.JSONEncoder
    app.add_api('swagger.yaml', arguments={'title': 'NI Auto-Scaling Service'})
    app.run(port=8004)
```

(그림 11) Auto-scaling 모듈 server/__main__.py 파일 - Port 설정 가능

Auto-scaling 모듈은 Auto-scaling 과정에서 VNF 인스턴스의 생성 및 제거, SFC의 갱신을 위해 NFVO 및 모니터링 모듈과 상호작용한다. 이를 위해 설정 파일에 각 모듈이 동작하는 Host의 IP 주소와 포트 번호를 미리 입력해야 한다. 또한, Auto-scaling 과정에서 필요로 하는 파라미터 (Parameter)를 미리 정의할 필요가 있다. 내려 받은 Auto-scaling 모듈 폴더 내에서 config/config.yaml을 열면, 모니터링 모듈에 해당하는 ni_mon, NFVO에 해당하는 ni_nfvo의 host 정보를 각각 수정하고 저장한다. 또한, 그 외에 Github 페이지 [5]에 있는 문서를 참조하여 (그림 12)의 파라미터들을 설정한다.

```
ni_mon:
  host: http://<ni_mon_ip>:<ni_mon_port>    # Configure here to interact with a monitoring module
ni_nfvo:
  host: http://<ni_nfvo_ip>:<ni_nfvo_port>  # Configure here to interact with an NFVO module
instance:                                   # Information of new instance created by a scale-out action
  id: <instance_ssh_id>                     # SSH ID of new VNF instance
  password: <instance_ssh_pw>               # SSH ID of new VNF instance
  prefix_splitter: '-'                      # Prefix to classify VNF instance name
  max_number: 5                             # Maximum number of VNF instances allowed in each tier
  min_number: 1                             # Minimum number of VNF instances allowed in each tier
sla_monitoring:                             # To access traffic generator and create traffic
  src: <IP of traffic generator>            # IP of traffic generator (can be an instance in OpenStack)
  id: <ID of traffic generator>             # ID of traffic generator (instance in OpenStack)
  ssh_id: <ssh_id of the traffic generator> # SSH ID of traffic generator
  ssh_pw: <ssh_pw of the traffic generator> # SSH PW of traffic generator
  num_requests: 100                         # Number of messages (This module generates HTTP messages)
  dst: <IP of destination>                  # Destination of traffic
image:                                      # Image IDs used by OpenStack
  firewall: <OpenStack Image ID>
  flowmonitor: <OpenStack Image ID>
  dpi: <OpenStack Image ID>
  ids: <OpenStack Image ID>
  proxy: <OpenStack Image ID>
flavor:                                     # Flavor ID used by OpenStack
  default: <OpenStack Flavor ID>
```

(그림 12) Auto-scaling 모듈 config/config.yaml 파일 설정

2장에서 설명한 것처럼 DQN을 수행하기 위해서는 사용자에 의해 하이퍼파라미터 값들이 설정되어야 한다. 또한, OpenStack 환경에서는 SFC를 구성하는 각 VNF 인스턴스들이 1개 이상의 네트워크 포트를 가질 수 있기 때문에 어떤 포트를 활용하여 SFC를 갱신 할 것인지 미리 명시해야 한다. DQN을 위한 하이퍼파라미터 값과 SFC 갱신에 사용할 네트워크 포트 식별을 위한 OpenStack Network ID는 내려 받은 Auto-scaling 모듈 폴더 내에서

auto_scaling.py 파일을 열어 수정한다. OpenStack Network ID는 OpenStack Dashboard 또는 OpenStack CLI를 통해 확인 가능하며, 해당 네트워크에 속한 포트들을 식별하여 Auto-scaling으로 인해 SFC를 갱신할 때 활용한다.

```
# Parameters
# OpenStack Parameters
openstack_network_id = "" # Insert OpenStack Network ID to be used for creating SFC

# <Important!!!!> parameters for Reinforcement Learning (DQN in this codes)
learning_rate = 0.01            # Learning rate
gamma         = 0.98            # Discount factor
buffer_limit  = 5000            # Maximum Buffer size
batch_size    = 32              # Batch size for mini-batch sampling
num_neurons = 128               # Number of neurons in each hidden layer
epsilon = 0.08                  # epsilon value of e-greedy algorithm
required_mem_size = 200         # Minimum number triggering sampling
print_interval = 20             # Number of iteration to print result during DQN
```

(그림 13) DQN의 Parameter 및 OpenStack Network ID 설정

설정을 마무리하고 실행 명령어를 입력하여, 정상적으로 Auto-scaling 모듈이 실행되면 (그림 14)과 같은 출력 화면을 볼 수 있다. 본 문서에서는 기본 포트 8004을 변경 없이 사용한다.

```
(scaling) dpnm@NI-AI-Node2:~/doyoung/running_modules/ni-auto-scaling-module-public$ python3 -m server
 * Serving Flask app "__main__" (lazy loading)
 * Environment: production
   WARNING: This is a development server. Do not use it in a production deployment.
   Use a production WSGI server instead.
 * Debug mode: off
 * Running on http://0.0.0.0:8004/ (Press CTRL+C to quit)
```

(그림 14) Auto-scaling 모듈 실행 화면

3.3 Auto-scaling 모듈 사용법

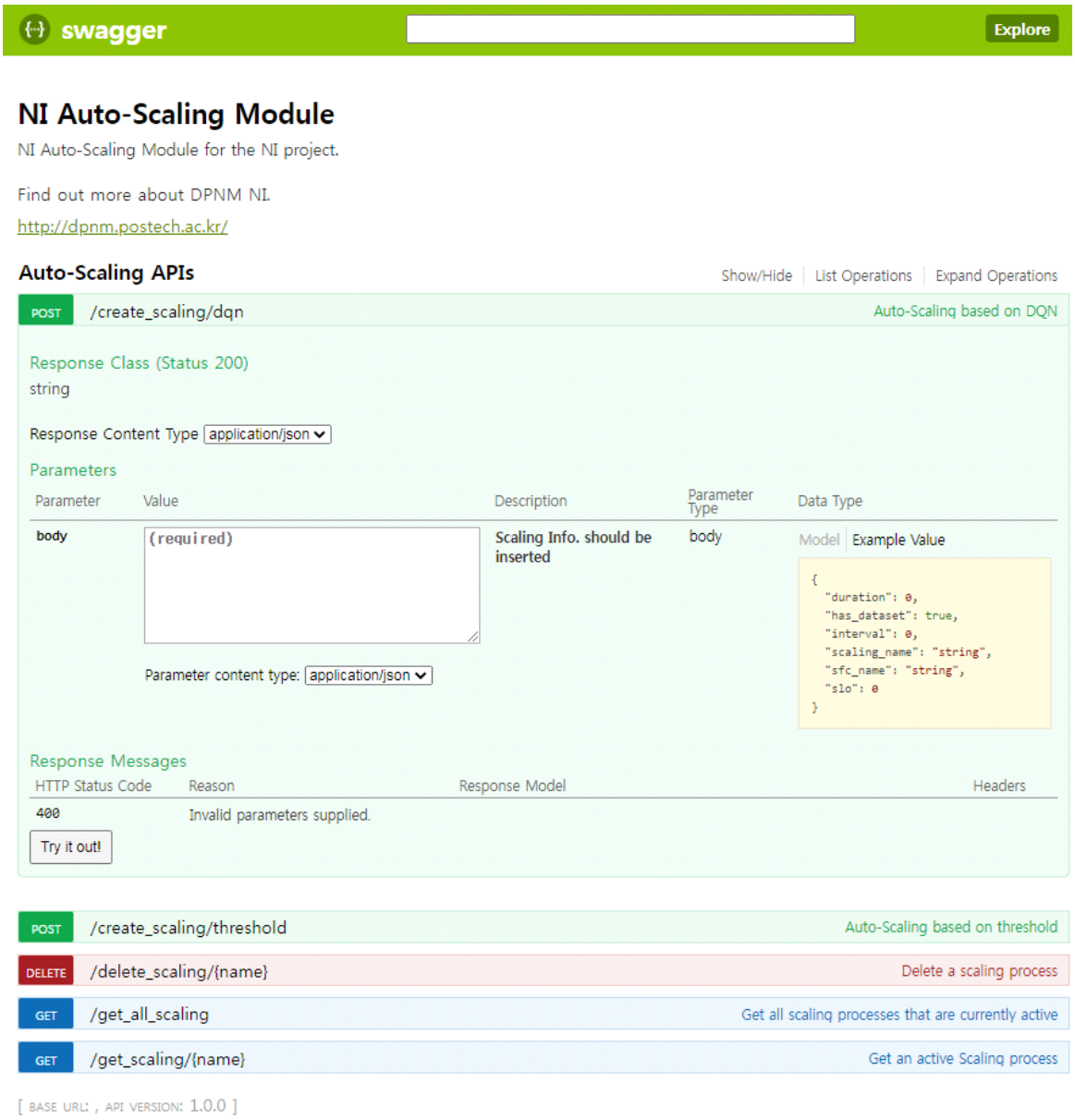
Auto-scaling 모듈이 정상적으로 실행되고, 설정에 오류가 없을 경우 Auto-scaling 모듈은 (그림 15)처럼 제공되는 Web UI를 아래 경로를 통해 확인할 수 있다.

http://<Auto-scaling 모듈이 실행되는 Host IP>:<모듈 포트 번호>/ui

Auto-scaling 모듈은 Auto-scaling 프로세스 생성/조회/삭제 요청 메시지를 각각 HTTP POST/GET/DELETE 메시지로 받는다. Auto-scaling 프로세스 요청 메시지는 Web UI를 통해 직접 Auto-scaling 모듈로 입력하거나, HTTP 메시지를 생성하여 Auto-scaling 모듈로 전달하는 방법 두 가지가 존재한다. 본 문서에서는 Auto-scaling 모듈의 Web UI를 통해 Auto-scaling 프로세스 요청 메시지를 전달하는 방법에 대해 서술한다. Auto-scaling 모듈은

HTTP POST/GET/DELETE로 요청 메시지를 받기 때문에 해당 메시지를 수신 할 경로를 다음과 같이 제공한다.

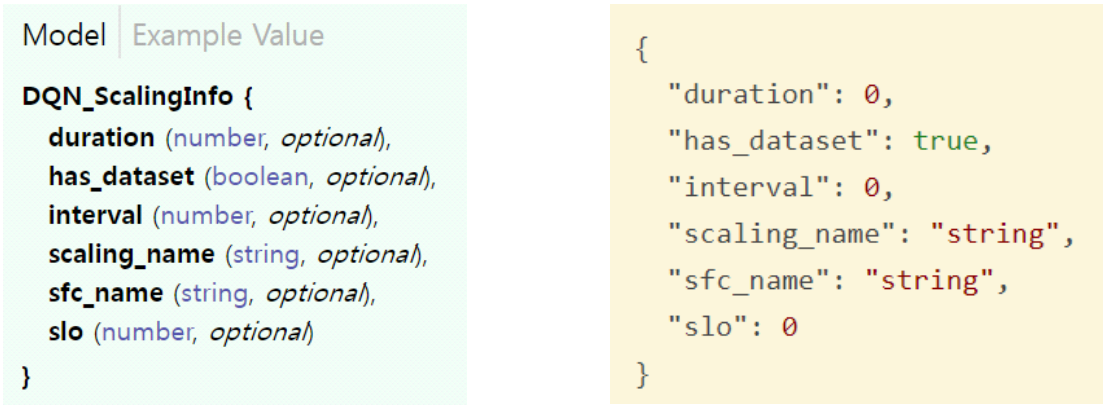
1) **(POST)** DQN 기반 Auto-scaling 생성: http://<모듈 Host>:<포트>/create_scaling/dqn
2) **(POST)** 임계값 기반 Auto-scaling 생성: http://<모듈 Host>:<포트>/create_scaling/threshold
3) **(DELETE)** 특정 프로세스 종료: http://<모듈 Host>:<포트>/delete_scaling/{name}
4) **(GET)** 실행 중인 모든 프로세스 조회: http://<모듈 Host>:<포트>/get_all_scaling
5) **(GET)** 실행 중인 특정 프로세스 조회: http://<모듈 Host>:<포트>/get_scaling/{name}



(그림 15) Auto-scaling 모듈 실행 화면

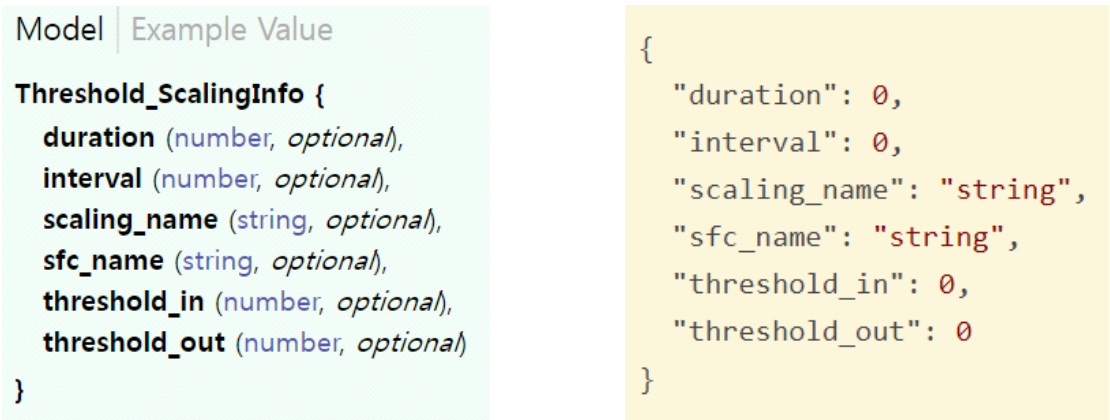
HTTP POST 메시지를 통해 Auto-scaling 프로세스를 생성할 때는 메시지의 Body에 명시 된 데이터를 활용한다. DQN 기반 Auto-scaling 프로세스를 생성할 때는 (그림 16)처럼 요청 메

시지의 Body에 DQN_ScalingInfo라고 정의 된 모델이 JSON으로 표현된다. DQN_ScalingInfo 모델은 number, boolean, string 형태의 데이터로 구성된다. (그림 16)에서 왼 쪽은 DQN_ScalingInfo 모델의 정의를 보이며, 오른 쪽은 Auto-scaling 프로세스 생성 요청 메시지에 각 데이터들이 포함될 때 어떤 형식으로 기입되어야 하는지 보이는 예시이다. 요청 메시지에 명시할 때는 number, boolean, string 형태에 따라 알맞은 형태로 입력한다.



(그림 16) DQN 기반 Auto-scaling 생성 요청 메시지 Body 모델 및 예

임계값 기반 Auto-scaling 프로세스를 생성할 때도 요청 메시지의 Body에 모델이 JSON으로 표현되며, 해당 모델인 Threshold_ScalingInfo는 (그림 17)과 같이 정의된다. Threshold_ScalingInfo 모델은 number, string 형태의 데이터로 구성된다. (그림 17)에서 왼 쪽은 Threshold_ScalingInfo 모델의 정의이며, 오른 쪽은 Auto-scaling 프로세스 생성 요청 메시지에 각 데이터들이 포함될 때 어떤 형식으로 기입되어야 하는지 보이는 예시이다. 요청 메시지에 명시할 때는 number, string 형태에 따라 알맞은 형태로 입력한다.



(그림 17) 임계값 기반 Auto-scaling 생성 요청 메시지 Body 모델 및 예

DQN_ScalingInfo과 Threshold_ScalingInfo에서 공통으로 요구되는 데이터는 4개가 존재하며, 각 의미는 다음과 같다.

- scaling_name: Auto-scaling 프로세스의 이름
- sfc_name: Auto-scaling을 적용할 대상이 되는 SFC 이름
- duration: Auto-scaling 프로세스 수행 시간 [unit: millisecond] (duration 동안 프로세스 실행)
- interval: Auto-scaling 수행 주기 [unit: second] (Interval마다 한번 씩 Scaling 수행)

또한, DQN_ScalingInfo에서는 추가로 다음과 같은 데이터가 요구된다. 이 중, has_dataset이 True일 경우, scaling_name과 같은 이름을 가진 파일에서 데이터를 읽어 Replay Memory에 저장한다.

- has_dataset: 학습용 데이터 존재 여부
- slo: SFC의 정의 된 SLO 값 [unit: millisecond]

마지막으로, Threshold_ScalingInfo에서는 다음과 같은 데이터가 추가로 요구된다.

- threshold_in: Threshold-in을 발생시키는 임계값 [unit: millisecond]
- threshold_out: Threshold-out을 발생시키는 임계값 [unit: millisecond]

그 외에 Auto-scaling 프로세스 조회 및 삭제를 위한 HTTP GET/DELETE 메시지는 Body에 별도의 데이터를 필요로 하지 않는다. 대신, 특정 Auto-scaling 프로세스를 조회하거나 삭제할 때는 해당 Auto-scaling 프로세스를 특정 지을 수 있도록 scaling_name을 URL의 파라미터 항목 {name} 기입한다.

Response Body


```
{
  "active_flag": true,
  "createdTime": "2020-08-29T19:55:28.202457Z",
  "duration": 3600,
  "interval": 30,
  "scaling_name": "dy-scaling",
  "sfc_name": "dy-sfc",
  "threshold_in": 10,
  "threshold_out": 30,
  "type": "threshold"
}
```


Response Code

```
200
```

Response Headers


```
{
  "content-length": "244",
  "content-type": "application/json",
  "date": "Sat, 29 Aug 2020 10:55:28 GMT",
  "server": "Werkzeug/0.16.1 Python/3.6.9"
}
```


(그림 18) Auto-scaling 모듈의 임계값 기반 프로세스 생성 요청 메시지에 대한 응답 메시지

(그림 18)은 임계값 기반 Auto-scaling 프로세스 생성을 요청했을 때, Auto-scaling 모듈이 출력하는 응답 메시지이다. 문제없이 Auto-scaling 프로세스가 생성되었을 경우, 정상적인 응답 메시지에 기입되는 응답 코드 (Response code) 200과 함께, 응답 메시지 Body에 생성된 Auto-scaling과 관련된 정보를 받을 수 있다. 관련 정보로는 현재 프로세스가 동작 중인지 여부를 보여주는 active_flag, 생성된 시간 createdTime, DQN 기반 프로세스인지 임계값 기반 프로세스인지를 보여주는 type과 함께 Auto-scaling을 생성할 때 요청했던 Body에 기입한 내용들이 같이 포함된다. Auto-scaling 프로세스 조회 및 삭제 메시지를 보냈을 때도 같은 정보들이 응답 메시지로 회신된다 (삭제의 경우에는 삭제 된 프로세스의 정보 회신). 만약, 응답 메시지로 오류를 받았을 경우에는 NFVO 및 모니터링 모듈에 문제가 발생해서 정상적으로 OpenStack 환경에 SFC를 갱신하지 못했거나, Auto-scaling 생성 요청 메시지에 잘못된 형태로 값이 입력 된 경우가 대부분이니 각 부분을 확인할 필요가 있다.

4 결론

본 문서에서는 NFV 환경의 SFC에 적용할 수 있는 DQN 기반 Auto-scaling 방법에 대해 서술하였다. DQN 기반 Auto-scaling 방법은 SFC를 구성하는 각 계층의 자원 활용률, Disk 작업 횟수, 인스턴스 개수, 인스턴스 분포도를 상태로 정의하고, Auto-scaling이 필요할 경우, 어떤 물리 서버에서 인스턴스를 추가/제거할지 결정한다. 본 문서의 Auto-scaling 방법은 OpenStack 환경에서 동작할 수 있는 모듈 형태로 구현되어 성능 검증을 수행하였다. 평가 결과, DQN 기반 Auto-scaling 방법은 임계값에 기반한 Auto-scaling 방법보다 더 효율적으로 인스턴스 개수를 조절하면서 SLO 위반을 최소화하였다.

향후 연구로는 GNN (Graph Neural Network)을 활용해 네트워크 토폴로지 정보를 최적 Auto-scaling 정책에 적용할 예정이다. GNN은 네트워크 토폴로지를 머신러닝 알고리즘에 적용되는 기능으로 표현할 수 있기 때문에, GNN을 활용하여 상태를 정의한다면 효과적으로 Auto-scaling을 위한 강화학습 문제에 적용할 수 있을 것으로 기대된다. 또한, 본 문서에서 서술한 DQN 기반 Auto-scaling 방법을 개선하여 VNF 인스턴스 수를 최적화하는 데 필요한 학습 시간을 줄이고, 다른 Auto-scaling 방법과의 비교를 포함하는 다양한 시나리오에서 성능 검증을 수행할 계획이다.

5 참고문헌

[1] Doyoung Lee, Jae-Hyoung Yoo, James Won-Ki Hong, "Deep Q-Networks based Auto-scaling for Service Function Chaining", 16th International Conference on Network and Service Management (CNSM 2020), Izmir, Turkey, Nov. 2-6, 2020

[2] OpenStack networking-sfc, [web] https://docs.openstack.org/networking-sfc/ocata/

[3] Gallenmüller, S., Emmerich, P., Wohlfart, F., Raumer, D., & Carle, G. (2015, May). Comparison of frameworks for high-performance packet IO. In 2015 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS) (pp. 29-38). IEEE.

[4] Dobrescu, Mihai, Norbert Egi, Katerina Argyraki, Byung-Gon Chun, Kevin Fall, Gianluca Iannaccone, Allan Knies, Maziar Manesh, and Sylvia Ratnasamy. "RouteBricks: exploiting parallelism to scale software routers." In Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles, pp. 15-28. 2009.

[5] DPNM NI Project Github, [web] https://github.com/dpnm-ni/ni-auto-scaling-module-public

[6] DPNM NI Project Github, [web] https://github.com/dpnm-ni/ni-testbed-public

[7] D. Coulson, “Network security iptables,” 2003.

[8] L. Deri, M. Martinelli, and A. Cardigliano, “Realtime high-speed network traffic monitoring using ntopng,” in 28th Large Installation System Administration Conference (LISA14), 2014, pp. 78–88.

[9] L. Deri, M. Martinelli, T. Bujlow, and A. Cardigliano, “ndpi: Open-source high-speed deep packet inspection,” in 2014 International Wireless Communications and Mobile Computing Conference (IWCMC). IEEE, 2014, pp. 617–622.

[10] Suricata: Open Source IDS/IPS/NSM engine. [Online]. Available: https://suricata-ids.org/

[11] W. Tarreau et al., “Haproxy-the reliable, high-performance tcp/http load balancer,” 2012.

[12] C. Amza, E. Cecchet, A. Chanda, A. L. Cox, S. Elnikety, R. Gil, J. Marguerite, K. Rajamani, and W. Zwaenepoel, “Specification and implementation of dynamic web site benchmarks,” in 5th Workshop on Workload Characterization, no. CONF, 2002.

[13] DPNM NI Project Github, [web] https://github.com/dpnm-ni/ni-mano