<PROJECT 최종보고서>

On-device 용
Neural Network Trainer 개발

송우경 (2018-18568)

장민혁 (2019-15714)

이현우 (2020-12907)

엄현상 교수님

손기성 담당자님 (Samsung Electronics Co. Samsung Research)

Table of Contents

A. Abstract

NNTrainer란 on device 학습을 위해 설계된 경량화된 딥러닝 프레임워크로, 메모리와 연산 자원이 제한된 환경에서도 신경망 모델의 학습과 추론을 효율적으로 수행할 수 있도록 최적화된 라이브러리이다. 본 프로젝트는 이러한 NNTrainer 라이브러리에 Conv2DTranspose 레이어, CosineAnnealingLR 및 LinearLR 학습률 스케줄러, AdamW 및 RMSProp 옵티마이저를 추가하여 open source contribution을 수행하는 것을 목표로 한다. 해당 모듈(이하 OP 혹은 모듈이라 통칭)들은 각기 학습 과정에서 중요한 역할을 수행하며, 서로 유기적으로 상호작용하여 효율적인 모델 학습을 지원한다. Conv2DTranspose 레이어는 주로 이미지 생성 모델에 필요한 연산을 수행하며, 스케줄러는 학습 단계에 따라 학습률을 동적으로 조정하여 모델의 수렴을 돕고, 옵티마이저는 각 레이어의 가중치를 최적화하여 모델이 안정적으로 학습할 수 있도록 도와준다. 총 5개의 OP를 추가함으로써, NNTrainer를 통해 on device에서 diffusion model과 같은 모델을 지원할 수 있게 한다. 본 프로젝트에서는 해당 OP들을 추가한 후에, demo로써 VAE 모델 학습 및 테스트를 수행하여 on-device 에서의 효율적 학습 성능을 확인한다.

B. Introduction

본 프로젝트는 삼성전자 주식회사와 함께 한다. 삼성전자 주식회사는 consumer electronics 에 주력하는 기업인데, 그렇기 때문에 요사이의 트렌드인 인공 신경망 (artificial neural networks) 을 바탕에 두는 인공지능 기술을 자사 제품에 도입하는 데 아마도 다음과 같은 특수한 난관을 겪을 터이다:

1. 인공지능 서비스를 제공하는 매체 내지는 플랫폼 (이 경우 주로 스마트폰이 되겠다) 을 만드는 데 머물자니 기회비용이 있다.
2. ChatGPT 로 대표되는 일반적인[1] 인공지능 서비스를 직접 제공하자니 경쟁력이 부족하다.
3. 경쟁력을 위하여 개인화된 인공지능 서비스를 제공하자니 학습에 필요한 방대한 양의 개인 정보가 없거나, 있다 한들 그 소유가 특히 개인 정보 보호 (privacy) 와 관련하여서 문제시된다.

곧 삼성전자와 같이 (1) consumer electronics 에 주력하면서 (2) 최신 인공지능 기술의 효용이 자사 제품에 고스란히 미치기를 바라는 기업으로서 당장에 할 수 있는 일은 고객으로부터 방대한 양의 데이터를 수집하지 않거나, 수집하더라도 개인 정보가 보호되게끔 하면서 개인화된 인공지능 서비스를 제공하는 방법—곧 구체적으로는 인공 신경망을 학습시키고 추론시키는 방법을 찾는 일 말고는 없을 것이다. 그리고 글쓴이가 따져보건대 그러는 방법은 큰 틀에서 다음 두 가지 뿐이다:

1. Privacy-Enhancing Technologies (PETs)[2]

---

[1] 개인화 (personalized) 되지 않았다는 의미에서 "일반적"이라는 표현을 사용하였다.

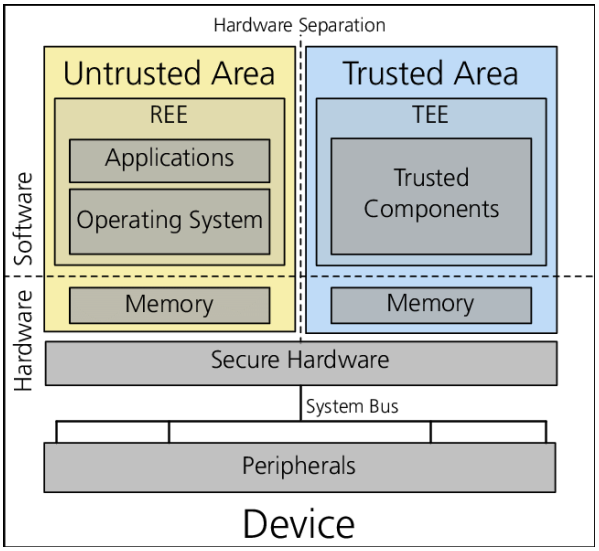
[2] Liv d'Aliberti, Evan Gronberg, Joseph Kovba, "Privacy-Enhancing Technologies for Artificial

2. "온디바이스 AI"[3]

이 둘 가운데 삼성전자는 후자의 접근을 택하여, NNStreamer 라는 인공 신경망 pipelining 프레임워크를 개발하고, 그 일환으로 NNTrainer 라는 온디바이스 인공 신경망 훈련에 최적화된 라이브러리를 만들다가, 지난 2020년 3월에 LF AI & Data Foundation 에 기여를 한 것으로 보인다. 본 프로젝트는 여전히 incubation-stage 에 있는 NNStreamer 프로젝트 중에서 특히 NNTrainer 라이브러리에 몇몇 중요한 기여를 하는 것을 목표로 한다.

C. Background Study

a. 관련 접근방법/기술 장단점 분석



온디바이스 인공 신경망 훈련에 최적화된 라이브러리를 개발하는 데 주의해야 할 점을 손쉽게 파악하기 위해서는 잠시 앞서 언급한 1. PETs 대 2. 온디바이스 AI 를 비교해보는 것이 좋을 것이다.

두 접근의 본질적 차이는 개인 정보와 인공 신경망이 어디에서 만나는가 하는 데 있다. PETs 를 기반으로 하는 방법에서는 인공 신경망은 서버 컴퓨터에 가만히 있고 움직이는 것은 개인 정보가 된다. 그리하여 둘은 (on-premises 든 클라우드든) 서버 컴퓨터 안에서—곧 사용자의 디바이스 밖에서—만나게 된다. 이러한 접근 하에서는 학습과 추론을 비롯한 대부분의 processing 은 여전히 서버에서 이루어질 것이다. 곧, ChatGPT 와 같은 이미 나온 인공지능 서비스의 구조를 거의 그대로 따르는 한편, 개인 정보 보호를 위해서

1. 도착한 고객 데이터에 대해서는 정해진 하드웨어에서 정해진 소프트웨어만을 가지고 처리를 하는 방법부터 (*trusted execution environments* or *TEEs*)[4]
2. 클라이언트 (디바이스) 에서 개인 정보를 보낼 때 서버 (인공 신경망) 가 연산은 할 수 있지만

Intelligence-Enabled Systems" (arXiv:2404.03509v1 **[cs.CR]**)

[3] 삼성을 비롯한 industry 에서 발명한 표현으로 추정되어 큰따옴표에 넣었다. See https://semiconductor.samsung.com/kr/technologies/processor/on-device-ai/, https://www.technologyreview.com/hub/ubiquitous-on-device-ai/.

[4] 그림 출처: Operating System Support for Run-Time Security with a Trusted Execution Environment - Scientific Figure on ResearchGate. Available from: https://www.researchgate.net/figure/Trusted-Execution-Environment-TEE-definition-In-order-to-support-a-TEE-a-device-needs_fig1_297732884 [accessed 6 Nov 2024]

그 내용은 볼 수 없는 특수한 encryption scheme 을 사용하는 방법까지 (*homomorphic encryption*)

다양한 (신)기술을 이용하여 사용자의 개인 정보가 서버에 보내져도 괜찮은 환경을 마련한다는 점에서 다르다.

반대로, 2. 온디바이스 AI 식 접근은 privacy 의 문제를 서버가 사용자의 데이터를 아예 취급하지 않게 함으로써 해결한다. 곧, 이 경우 사용자 데이터는 디바이스를 떠나지조차 않고, 굳이 따지자면 인공신경망이 서버로부터 보내져 "움직인다".

원래는 냉장고나 스마트폰보다는 조금 더 강력한 컴퓨터 시스템에서 이루어지던 학습과 추론 中 얼마를 온디바이스로 delegate 할지는 다른 여러 사항에 달렸을 것이다. 현재로서는 냉장고의 임베디드 시스템은 물론 스마트폰에서도 유용할 인공 신경망을 pre-training 할 수는 없음을 인지한 바, NNTrainer 의 목표도 (아쉬운 대로) 공개 데이터로 pre-train 된 모델을 디바이스에서 개인 데이터를 이용하여 fine-tuning 하는 것을 잘 하는 데 잡은 것으로 보인다.

두 접근의 장단점은 그리하여 명백하다: PETs 를 사용하면 훨씬 풍부한 컴퓨팅 자원을 가지고 아마도 더 좋은 인공지능 서비스를 제공할 수 있겠지만, 문제는 PETs 를 들이는데 드는 비용, 혹은 들이고자 하는 PET 의 성숙도 그 자체가 되겠다.

반면에 온디바이스 AI 는 그 immediateness 가 장점이다: 기술적으로 당장에 할 수 없는 것이 단 하나도 없다. 하지만 시간 · 공간 · 전력 따위가 경우에 따라서는 상당히 제한된 환경에서도 사용자 데이터를 가지고 충분히 효용 있는 인공 신경망을 fine-tuning 해야 하는 공학적인 문제를 풀어야 하며, 어쩌면 이런 맥락에서 NNTrainer 라이브러리가 탄생한 것일 것이다 (실제로 NNTrainer 논문은 memory efficiency 를 중심으로 NNTrainer 를 소개한다.[5])

b. 프로젝트 개발환경

NNTrainer 는 새로 시작하는 프로젝트가 아니다 보니 "표준" 개발 환경이라 할 수 있는 것이 그 GitHub repository 에 꽤나 상세히 기술돼 있다. 하지만 그럼에도 "아는 사람만 아는" 정보가 있다. 이러한 사항까지 포함한 Dockerfile 을 Appendix 로 첨부하였다.

넓은 의미의 "개발 환경"에 관해서 논하자면,

---

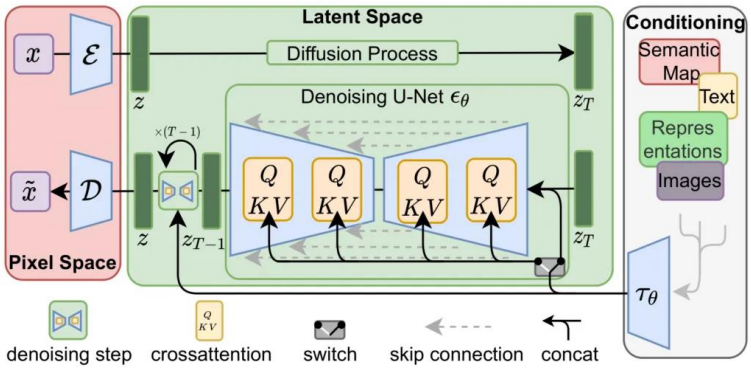
[5] Jijoong Moon, Hyeonseok Lee, Jiho Chu, Donghak Park, Seungbaek Hong, Hyungjun Seo, Donghyeon Jeong, Sungsik Kong, and Myungjoo Ham. 2024. A New Frontier of AI: On-Device AI Training and Personalization. In Proceedings of the 46th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP '24). Association for Computing Machinery, New York, NY, USA, 323–333. https://doi.org/10.1145/3639477.3639716

- NNTrainer 는 C 와 C++ API 를 제공하는 라이브러리인데,[6] 내부에서는 주로 C++ 를 사용한다. 따라서 NNTrainer 에 기여하려면 웬만해서는 C++ 프로그래밍을 해야 한다.
- NNTrainer 는 LF AI & Data Foundation incubation-stage project 이니만큼 오픈 소스 프로젝트이고, GitHub 에 host 되어있다. 따라서 NNTrainer 에 기여하려면 Git 과 GitHub 을 사용해야 한다.
- NNTrainer 를 build 및 install 해보고 싶을 수 있다. 이 경우 repository root 에서 `meson build/ && ninja -C build/ -j 1 install` 을 실행하면 된다.

D. Goal/Problem & Requirements

앞서 말했듯이 본 프로젝트는 NNTrainer 라이브러리에 몇몇 중요한 기여를 하는 것을 목표로 한다. 어떤 기여를 할지는 정하기 나름이었는데, 팀원들끼리, 그리고 회사 담당자와의 긴밀한 논의 끝에 요사이 또 하나의 트렌드인 multimodal large language model (LLM) 을 온디바이스에서도 돌릴 수 있도록 (곧 학습시키고 추론시킬 수 있도록) NNTrainer 에 필요한 기능을 추가하는 것이 좋겠다는 결론에 도달하였다.

하지만, NNTrainer 로 multimodal LLM 을 구동할 수 있으려면 우선 diffusion model 을 구동할 수 있어야 한다. diffusion model 내부에는 U-net이 있고 여기에 NNTrainer에 아직 구현되지 않은 Conv2dTranspose layer가 사용된다.



E. Approach

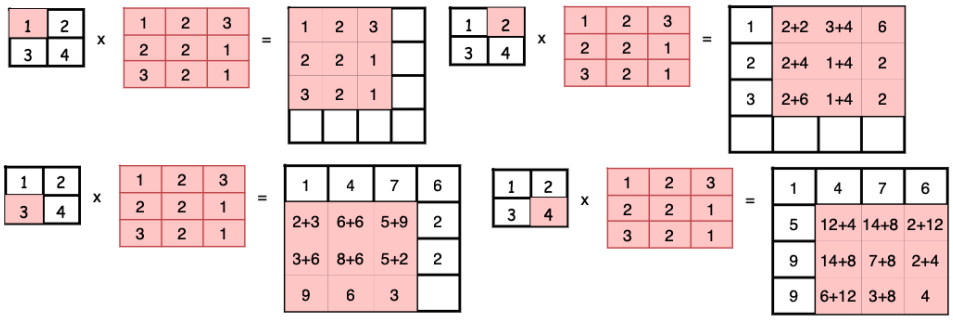
본 프로젝트는 on-device LLM으로의 변화에 발맞춰 NNTrainer가 multimodal LLM을 지원할 수 있게 각종 Layer, Optimizer, LR scheduler를 구현하고, 이들을 이용해 실제 VAE model을 만들고 학습시켜 우리의 구현을 테스트하는 것을 목표로 한다.

이를 위해 우선 NNTrainer GitHub repo에서 코드를 다운 받아 각자 local machine ubuntu에서 main

[6] https://github.com/nnstreamer/nntrainer#apis

branch를 build한다. 여러 패키지 이슈를 해결하고 Unit Test를 거친 후 성공적으로 개발 환경을 만들고 각자 할당된 operation과 유사한 operation들이 어떻게 구현되어 있는지 확인하고 새로 코드를 짜면서 필요한 부분을 추가한다.

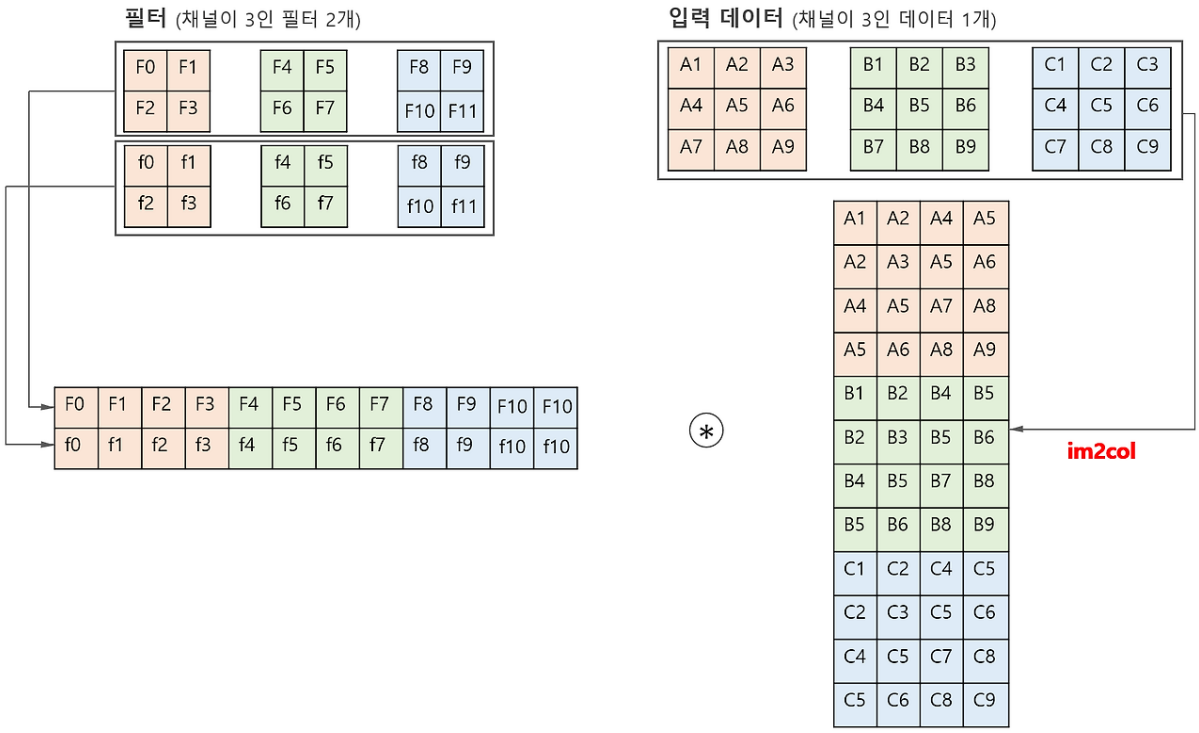
Conv2dTranspose layer는 일반 convolution의 input과 output이 바뀐 형태라고 볼 수 있다. Conv2dTranspose의 output에 filter를 Convolution했을 때 input의 shape이 나와야하고, 실제 Conv2dTranspose 계산에서는 output에서 filter를 convolution해서 나올 수 있는 input의 특정한 위치의 값에 대하여 filter의 값들을 곱하여 해당하는 output 위치에 더해준다.



구현 시 주의할 점은 Conv2dTranspose를 naive하게 각 인풋에 대하여 필터를 직접 적용해 계산하면 modern pipelined CPU 또는 GPU에서의 동작을 목표로 하는 NNTrainer의 특성상 branch diversion 때문에 엄청난 비효율을 낳게된다. 이를 없애기 위해 원래 repo의 Conv2d layer에서도 적용된 im2col 기술을 Conv2dTranspose layer에 맞게 구현하여 수백배 이상의 성능 향상을 얻는 것이 Conv2dTranspose layer를 구현하는 과정에서 핵심이라고 할 수 있다.

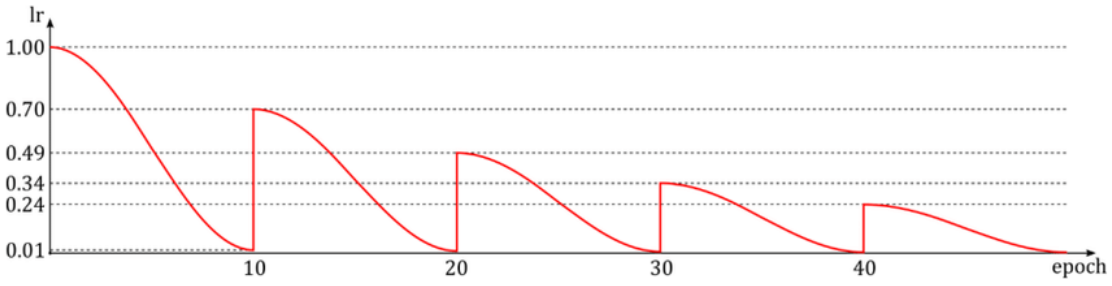
im2col은 convolution연산을 하나의 행렬곱으로 바꿔주는 기술로, pipelined CPU에서 branch instruction으로 인한 stall을 줄이고 GPU에서는 branch diversion을 줄여 연산장치들의 성능을 극대화한다. 사실 im2col 과정에서 naive Conv2dTranspose에서 거치는 for문은 그대로 돌면서 행렬을 만들기에(Appendix 참조) branch 개수 자체는 큰 변화가 없지만, im2col에서는 각 branch 내에서 floating point multiplication이 빠지고 데이터를 이동하기만 하기에 훨씬 더 빨리 for문을 끝낼 수 있다.

convolution 연산의 input과 filter는 각각 (batch size, in_channel, in_height, in_width)과 (out_channel, in_channel, kernel_height, kernel_width)의 shape을 가지고 있다. 이때 아래 그림처럼 filter는 reshape을 통해 (out_channel, in_channel*kernel_height*kernel_width)의 shape을 가진 2차원 행렬이 되고, input은 각 필터 위치에 맞는 value들을 재배치하여 (in_channel*kernel_height*kernel_width, out_height*out_width)의 shape을 가진 2차원 행렬이 된다.



NNTrainer의 Conv2d에는 im2col이 구현되어 있지만 Conv2dTranspose는 각 필터의 역할이 완전히 달라지기에 새로운 함수를 짜야 한다. 또한 layer를 학습시키려면 각 layer의 parameter들의 gradient를 계산하고 이전 layer로 input의 gradient를 역전파해주는 함수(각각 calcGradient(), calcDerivative()로 불림)도 구현해야 한다. 여기에는 행렬로 변환된 gradient를 다시 원래 input의 shape으로 바꿔주는 col2im이 필요하다.

CosineAnnealingLR은 모델이 local minima에서 효과적으로 벗어나 global minimum을 향해 갈 수 있도록 도와주는 LR scheduler로 현대의 수많은 모델을 학습할 때 쓰이고 있다. 아래 그래프에서 볼 수 있듯이 cosine 함수를 따라 주기적으로 Learning Rate을 높여주어 local minima에서 벗어날 수 있게 한다.



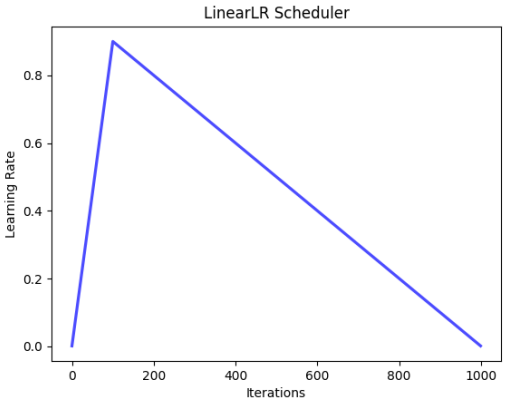
CosineAnnealingLR은 NNTrainer의 이미 구현된 다른 scheduler를 보고 다음 Learning Rate을 결정하는 부분을 해당 함수의 그래프에 맞게 바꿔주어 구현하면 된다. 이때 CosineAnnealingLR에 필요한 hyperparameter가 있으면 새로 추가해준다.

AdamW 역시 현대 딥러닝 모델 학습에서 자주 쓰이는 optimizer이다. 아래 식을 이용하여 Adam

optimizer에 weight decay를 추가하여 구현할 수 있다.

$$
\begin{aligned}
&\textbf{for } t = 1 \textbf{ to } \ldots \textbf{ do} \\
&\quad \textbf{if } maximize: \\
&\quad\quad g_t \leftarrow -\nabla_\theta f_t(\theta_{t-1}) \\
&\quad \textbf{else} \\
&\quad\quad g_t \leftarrow \nabla_\theta f_t(\theta_{t-1}) \\
&\quad \theta_t \leftarrow \theta_{t-1} - \gamma\lambda\theta_{t-1} \\
&\quad m_t \leftarrow \beta_1 m_{t-1} + (1-\beta_1) g_t \\
&\quad v_t \leftarrow \beta_2 v_{t-1} + (1-\beta_2) g_t^2 \\
&\quad \widehat{m_t} \leftarrow m_t / \left(1-\beta_1^t\right) \\
&\quad \widehat{v_t} \leftarrow v_t / \left(1-\beta_2^t\right) \\
&\quad \textbf{if } amsgrad \\
&\quad\quad \widehat{v_t}^{max} \leftarrow \max(\widehat{v_t}^{max}, \widehat{v_t}) \\
&\quad\quad \theta_t \leftarrow \theta_t - \gamma \widehat{m_t} / \left(\sqrt{\widehat{v_t}^{max}} + \epsilon\right) \\
&\quad \textbf{else} \\
&\quad\quad \theta_t \leftarrow \theta_t - \gamma \widehat{m_t} / \left(\sqrt{\widehat{v_t}} + \epsilon\right)
\end{aligned}
$$

LinearLR은 기본적인 LR scheduler 중의 하나로 일정하게 learning rate을 감소시켜 모델을 최적화시킨다. 많은 hyperparamer가 필요하지 않아 가볍게 모델을 테스트해보기에 좋고, diffusion model paper에서 학습할 때 쓰여 이것도 NNTrainer에 구현하기로 하였다.



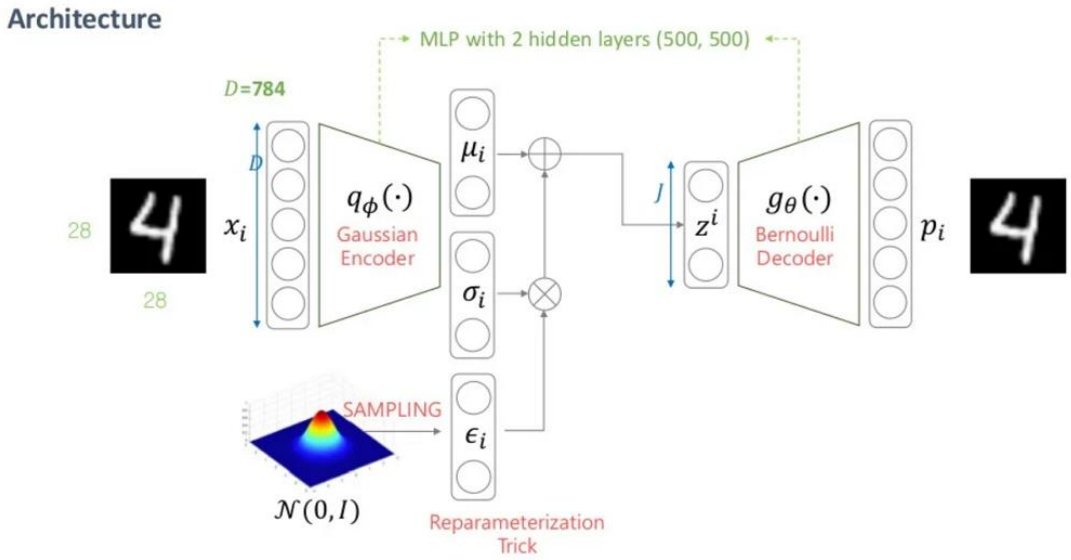
RMSProp 또한 기본적인 optimizer 중의 하나로 아래 식과 같이 다음 weight이 결정된다. Adam과 다르게 exponential average of gradients를 저장할 필요가 없어 메모리를 아낄 수 있다. 이것도 diffusion model paper 학습할 때 쓰여 구현하기로 하였다.

$$
\begin{aligned}
&For\ each\ Parameter\ w^j \\
&(j\ subscript\ dropped\ for\ clarity) \\
&\nu_t = \rho\nu_{t-1} + (1-\rho) * g_t^2 \\
&\Delta\omega_t = -\frac{\eta}{\sqrt{\nu_t + \epsilon}} * g_t \\
&\omega_{t+1} = \omega_t + \Delta\omega_t
\end{aligned}
$$

$\eta : Initial\ Learning\ rate$
$\nu_t : Exponential\ Average\ of\ squares\ of\ gradients$
$g_t : Gradient\ at\ time\ t\ along\ \omega^j$

최종적으로 우리가 만든 operations들을 이용해 U-net을 만들어 아래 그림과 같은 VAE 모델을

구성하고, MNIST dataset을 이용해 이를 학습시키고 테스트한다.



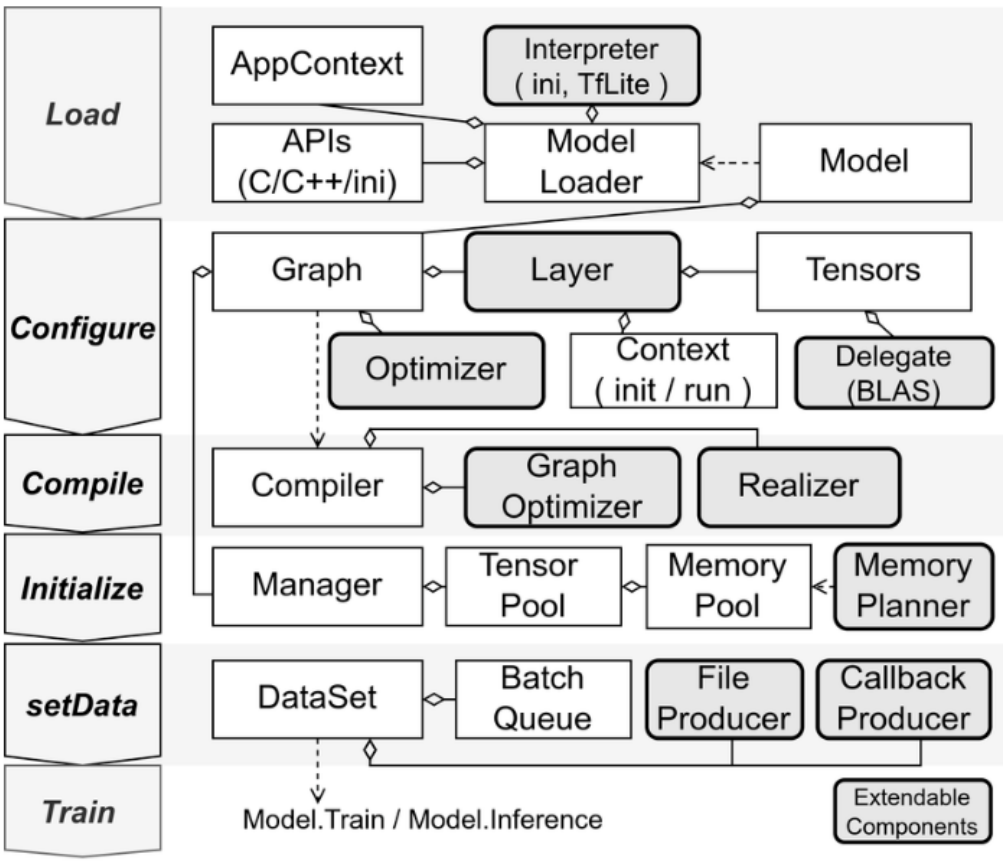
VAE(Variational AutoEncoder)는 Kingma 의 논문에 최초로 등장한 generative model의 일종으로 input과 output 사이의 latent space를 정규 분포로 가정하고 같은 이미지의 input과 output으로 이를 학습시켜 정규 분포 parameter를 얻어내는 모델이다. Reparametrization trick 을 이용하여 random sampling 에도 불구하고 sample 의 분포를 규정하는 parameter 에 대한 backpropagation 은 이루어질 수 있게 만든 것이 특징이다.

MNIST train dataset의 각 이미지를 NNTrainer로 만들어진 VAE 모델에 input=output으로 넣어 학습시킬 수 있고, test dataset의 이미지를 input으로 넣고 latent space에 노이즈를 추가하여 인풋에 대하여 비슷한 이미지를 생성할 수 있음을 보인다.

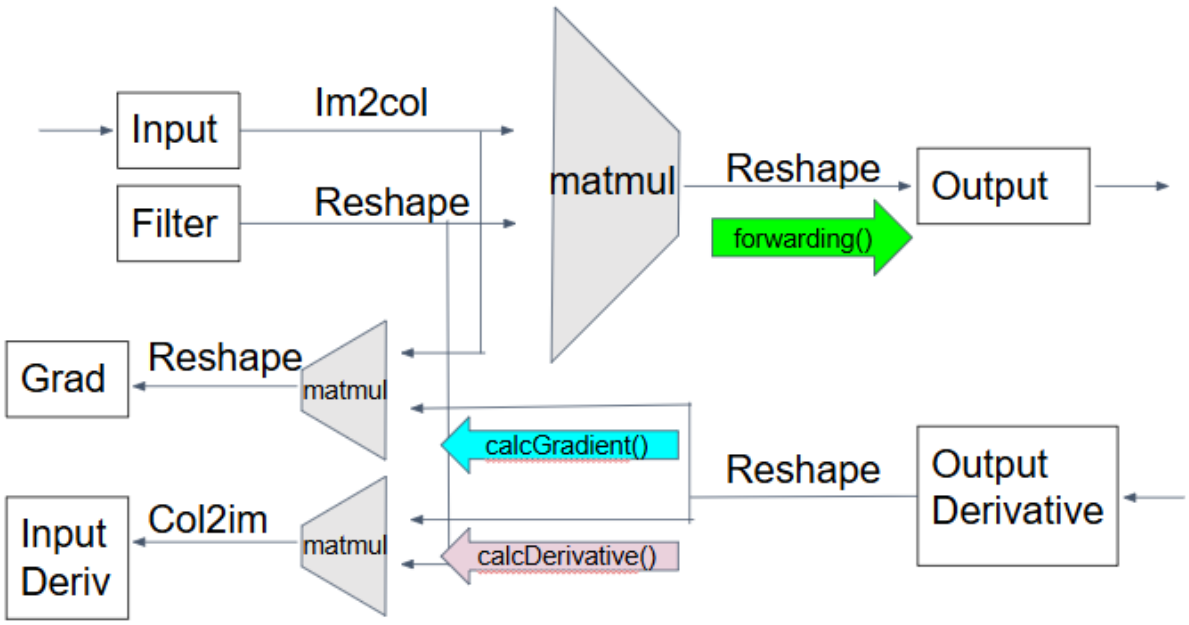
F. Project Architecture

a. Architecture Diagram

아래 그림은 NNTrainer architecture diagram이다.



아래 그림은 NNTrainer 내에 구현할 Conv2dTranspose layer 세 가지 핵심함수의 pipeline이다.



b. Architecture Description

NNTrainer는 on-device 환경에서 딥러닝 모델 학습을 최적으로 하는 데 목적을 두고 있고, architecture도 이에 따라 구성되어 있다. on-device 환경에서는 메모리가 상당히 제한적이기에 peak

memory consumption을 낮추는 것이 모델 학습을 시키는데 있어 중요하다. NNTrainer에서는 이를 위해 proactive swap을 사용한다. 일반적으로 swapping은 메모리가 부족할 때 당장 필요 없는 데이터를 disk로 옮길 때 일어난다. 디스크에 데이터를 저장하거나 디스크에서 데이터를 불러오는 동작은 엄청나게 큰 latency를 발생시키고 모델 학습을 느려지게 만든다. NNtrainer에서는 모델 학습에서 memory peak가 될 시점을 예측하여 데이터를 memory에서 disk로 미리 swap시킨다. 따라서 메모리에서 디스크로 데이터가 이동하는 latency가 모델 학습에서 일어나는 연산들에 hide된다. 이로써 on-device에서 학습을 빠르게 만드는 것이다. architecture에서 볼 수 있는 memory planner가 이러한 역할을 담당한다.

또한 graph optimizer가 peak memory consumption이 낮아지도록 연산들의 순서를 reorder한다. 각 operation을 작은 연산의 단위인 procedure로 쪼개 memory가 많이 필요한 procedure와 적게 필요한 procedure로 나눈다. memory가 많이 필요한 procedure들만 연산 queue에 들어가는 것을 막고 이들과 memory가 적게 필요한 procedure도 같이 queue에 들어가게 한다.

Layer, Optimizer 부분이 우리가 본격적으로 작업할 모듈들이다. Layer는 fully connected layer, convolution layer, attention layer 등 딥러닝 모델의 building block을 모아놓은 모듈이다. Optimizer는 LR scheduler와 weight optimizer를 모아놓은 모듈로 학습시 loss가 잘 떨어질 수 있게 weight update를 다르게 하는 다양한 테크닉의 집합이라고 볼 수 있다.

NNTrainer가 on-device 환경에서 최적화된 framework를 지향하는 만큼 우리가 작성한 코드들이 이를 타겟으로 하여 최적화시켜야 하는 것이 아닌가 질문할 수 있다. 그러나 NNTrainer는 architecture를 봐도 알 수 있듯이 모델 학습 시 system적인 부분(메모리, 연산의 할당과 스케줄링)의 최적화를 강점으로 하고 있고 오히려 새로운 operation을 만들 때는 이러한 부분과의 충돌에 대한 고려 없이 일반적으로도 가장 최적화된 알고리즘을 그대로 가져다 써도 되도록 설계되었다. 따라서 우리도 on-device 환경에 대한 고려보다는 각종 operation의 최적화된 implementation을 NNTrainer에 이식시키는 데 프로젝트의 주안점을 둔다.

Architecture Diagram의 두 번째 그림은 우리가 구현한 Conv2dTranspose의 pipeline으로 핵심 함수인 forwarding, calcGradient, calcDerivative가 각각 어떤 input을 받고 어떤 과정을 거쳐 output을 내는지 보여준다.

forwarding()은 layer에서 input과 filter에 대하여 Conv2dTranspose 연산을 수행해 output을 내는 과정이다. im2col을 이용했기에 구체적으로는 input과 filter를 받아 각각 적절한 shape의 행렬로 변환해 matmul을 수행하고 다시 원래 output shape에 맞게 결과를 reshape한다.

calcGradient()는 input과 다음 layer로부터 역전파되어온 derivative를 이용하여 filter의 gradient를 계산하는 과정이다. 결국 gradient 계산도 matmul로 환원되기에 im2col된 input을 저장하고 있다면 output derivative만 reshape 시켜 matmul하면 바로 구할 수 있다.

calcDerivative()는 filter와 다음 layer로부터 역전파되어온 derivative를 이용해 이전 layer로 보낼 input derivative를 계산하는 과정이다. 이 또한 matmul로 환원되기에 각각 reshape 시켜서 matmul하면 im2col된 input derivative를 구할 수 있다. 여기서는 해당 행렬을 다시 4차원 tensor로 바꿔주는 col2im의 구현이 필요하다.

G. Implementation Spec

a. Input/Output Interface

이번 프로젝트에서는, CosineAnnealingLR scheduler, LinearLR scheduler, AdamW optimizer, RMSProp optimizer, Conv2DTranspose Layer로, 총 2개의 scheduler와 2개의 optimizer, 1개의 layer를 구현하여 NNTrainer에 contribute하게 된다. 각 5 종류의 OP 각각의 input/output interface는 다음과 같다.

1-a. CosineAnnealingLR scheduler input

CosineAnnealingLR scheduler를 구성하기 위해 먼저 사용되는 주요 input은, 학습률을 조정하는데 필요한 MaxLearningRate, MinLearningRate, DecaySteps 파라미터들이다. MaxLearningRate는 학습률 조정의 시작값으로 설정되며, 스케줄러가 학습 초기 단계에서 사용할 최대 학습률을 나타낸다. MinLearningRate는 학습률이 점차 감소하면서 도달할 최저값을 지정하는 파라미터로, 학습 후반부에 학습률이 안정된 작은값으로 수렴할 수 있도록 한다. DecaySteps는 학습률이 cosine annealing 방식을 따라 MaxLearningRate에서 MinLearningRate로 변하는 주기를 설정하는 역할을 한다. 이 값을 통해 스케줄러가 학습률을 조정하는 주기성을 설정할 수 있다. 사용자는 이 파라미터들을 setProperty() 함수를 통해 설정할 수 있으며, 설정된 파라미터들은 lr_props라는 멤버 변수에 저장되어 각 epoch마다 학습률을 조정하는데 사용된다. 그리고, 사용자는 getLearningRate() 함수에 size_t integration 값을 통해 현재 단계에서의 학습률을 얻어낼 수 있다.

1-b CosineAnnealingLR scheduler output

cosineAnnealingLR scheduler의 주된 output은 현재 epoch에서 사용될 학습률(Learning Rate)이다. 매 epoch마다 getLearningRate() 함수가 호출되어 MaxLearningRate에서 MinLearningRate 사이의 값을 cosine 함수를 기반으로 계산한다. 학습률은 다음의 cosine annealing 공식에 따라 설정된다.

$$learning\_rate = MinLearningRate + (MaxLearningRate - MinLearningRate) \times 0.5 \times \left(1 + cos\left(\frac{\pi \times (iteration mod DecaySteps)}{DecaySteps}\right)\right)$$

위 공식을 통해 사용자는 cosine 주기에 따라 DecaySteps에 맞춰 주기적으로 최저값과 최대값 사이에서 조정되는 학습률을 얻을 수 있다.

2-a. LinearLR scheduler input

LinearLR scheduler의 주요 input은 학습률을 선형적으로 조정하는데 필요한 MaxLearningRate, MinLearningRate, Decaysteps 파라미터들이다. MaxLearningRate는 학습 초기에 사용할 최대 학습률을 나타내며, 스케줄러가 학습 초반 단계에서 시작될 학습률의 상한선으로 설정한다. MinLearningRate는 학습이 진행됨에 따라 학습률이 점차 감소해 도달할 최저값을 의미한다. 학습 후반부에 학습률이 안정적인 작은 값으로 수렴할 수 있도록 설정된다. DecaySteps는 학습률이 MaxLearningRate에서 MinLearningRate로 선형적으로 감소하는 데 걸리는 총 단계를 설정하는 값으로, 이 값을 통해 학습률이 얼마나 빠르게 줄어들지 결정한다. 사용자는 위 값들을 setProperty() 함수를 통해 설정할 수 있으며, 설정된 파라미터들은 lr_props라는 멤버 변수에 저장되어 각 epoch마다 학습률을 조정하는데 사용된다. 그리고, 사용자는 getLearningRate() 함수에 size_t integration 값을 통해 현재 단계에서의 학습률을 얻어낼 수 있다.

2-b. LinearLR scheduler output

LinearLR scheduler의 주요 output은 현재 epoch에서 사용될 학습률(Learning Rate)이다. 각 epoch마다 getLearningRate() 함수가 호출되어, 학습률은 MaxLearningRate에서 MinLearningRate로 선형적으로 감소하는 방식으로 계산된다. 학습률은 아래와 같은 공식을 통해 설정된다.

$$learning\_rate = MaxLearningRate - \left(\frac{MaxLearningRate - MinLearningRate}{DecaySteps}\right) \times iteration$$

위 공식을 통해 사용자는 학습이 진행됨에 따라 MaxLearningRate에서 MinLearningRate까지 선형적으로 감소하는 학습률을 얻을 수 있다. DecaySteps 동안 선형적으로 학습률이 감소하게 되어, 학습 후반으로 갈수록 점차 작은 학습률을 적용하여 안정적인 학습을 진행할 수 있다.

3-a. AdamW optimizer input

AdamW optimizer를 구성하는 주요 input들은 학습률과 가중치 업데이트에 필요한 파라미터들로 이루어져 있다. Learning Rate는 각 파라미터를 업데이트할 때 곱해지는 기본 학습률로, AdmaW optimizer의 업데이트 강도를 조절한다. beta1과 beta2는 각각 1차 모멘텀과 2차 모멘텀을 추정하는데 사용되는 지수 감소 비율을 뜻한다. beta1은 1차 모멘텀 계수 조절해 업데이트 방향에 대한 이동 평균을 유지하고, beta2는 2차 모멘텀 계수를 조절하여 그래디언트 제곱에 대한 이동 평균을 유지한다. epsilon은 매우 작은 값으로서, 0으로 나누는 오류를 방지한다. TorchRef는 Pytorch와의 호환성을 위한 플래그로, true인 경우에 pytorch의 학습률 업데이트 방식을 따른다. 이러한 파라미터들은 setProperty() 함수를 통해 설정할 수 있으며, adam_props라는 멤버 변수에 저장되어 각 iteration마다 가중치 업데이트를 수행할 때 사용된다. 사용자는 applyGradient() 함수를 호출하여, 현재 파라미터에 대한 그래디언트 정보를 업데이트하고 최적화된 가중치를 얻을 수 있다.

3-b. AdamW optimizer output

AdamW optimizer의 주요 output은 현재 epoch에서 업데이트된 가중치(Weights)이다. 매 iteration마다 applyGradient() 함수가 호출되며, 주어진 학습률, 모멘텀, 감쇠 비율을 반영하여 현재 파라미터의 그래디언트를 업데이트 한다. AdamW opimizer의 가중치 업데이트는 다음과 같은 절차를 따른다. (1) 1차 모멘텀과 2차 모멘텀 업데이트 : 현재 그래디언트를 기반으로 1차 모멘텀과 2차 모멘텀 변수를 업데이트 하여, beta1과 beta2를 사용해 가중 이동 평균을 계산한다. (2) 편향보정 : beta1과 beta2에 대해 편향 보정값을 적용해, 초기 단계의 편향효과를 줄인다. (3) 가중치 업데이트: 보정된 2차 모멘텀을 기준으로 그래디언트를 정규화하고, 1차 모멘텀과 epsilon을 사용하여 최종적으로 조정된 그래디언트를 계산한다. 위 3개의 과정을 통해 계산된 그래디언트는 가중치에 적용된다.

4-a. RMSProp optimizer input

RMSProp optimizer의 input은 가중치 업데이트와 학습 안정성을 위한 주요 파라미터들로 구성된다. Learning Rate는 각 파라미터에 곱해지는 기본 학습률로, optimizer의 학습 강도를 조절한다. alpha는 2차 모멘텀을 추정하는 지수 감소 비율을 나타낸다. 이 파라미터는 과거의 그래디언트 크기에 얼마나 영향을 받을지 결정한다. epsilon은 매우 작은 값으로, 0으로 나누는 오류를 방지하기 위해 사용된다. Weight decay는 학습 과정에서 가중치의 감쇠율을 설정하여, 과적합을 방지하는데 사용하는 파라미터이다. 이러한 파라미터들은 setProperty() 함수를 통해 사용자가 설정가능하며, 각 파라미터들은 rmsprop_props 라는 멤버 변수에 저장되어 applyGradient() 함수를 호출하여 현재 파라미터에 대한 그래디언트를 업데이트하고, 최적화된 가중치를 얻을 수 있다.

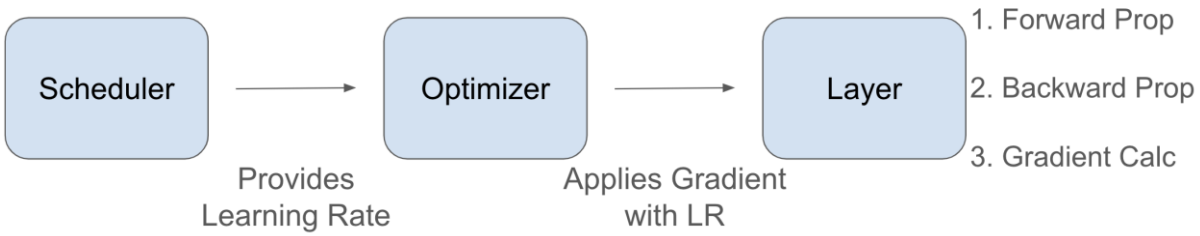
4-b. RMSProp optimizer output

RMSProp optimizer의 주요 output은 현재 epoch에서 최적화된 가중치(Weights)이다. 매 iteration마다 applyGradient() 함수가 호출되어, 각 파라미터는 RMSProp 알고리즘에 따라 업데이트 된다. 해당 가중치 업데이트 과정은 다음과 같은 절차를 따른다. (1) 2차 모멘텀 업데이트 : 현재 그래디언트의 제곱을 기반으로, 2차 모멘텀 변수를 업데이트하며, alpha 계수를 사용해 과거의 그래디언트 제곱에 대한 이동 평균을 계산한다. (2) 정규화 및 그래디언트 조정 : 2차 모멘텀에 epsilon을 더한 값으로 그래디언트를 정규화하여, 큰 변화가 발생하지 않도록 안정적인 학습률을 적용한다. (3) 가중치 업데이트 : 정규화된 그래디언트를 학습률과 함께 적용해 최종적으로 가중치를 업데이트한다. 이때 weight decay의 값이 설정되어 있으면, 가중치에 감쇠를 적용하여 학습이 지나치게 특정 방향으로 수렴하는 것을 방지한다. 위 3개의 과정을 통해 계산된 그래디언트는 가중치에 적용된다.

5-a. Conv2DTranspose Layer input

Conv2DTranspose layer의 주요 input은 input tensor 및 역전파 과정에서 활용되는 필터와 기타 여러 파라미터들로 구성된다. Input Tensor는 Conv2DTranspose layer의 입력 데이터로, 일반적으로 [Batch, Channels, Height, Width] 형식을 따른다. 이 텐서는 Transpose Convolution 과정을 통해 크기를 확장하고 특징 맵을 생성한다. Filter size는 출력 채널 수를 정의하는 필터의 개수로, 각 출력 채널에

대한 필터의 역할을 한다. Kernel Size는 커널의 높이와 너비를 정의해 필터의 크기를 결정한다. Stride는



스트라이드의 크기로, 입력텐서의 특정 위치에서 다음 위치로 이동할 거리를 결정한다. Padding은 입력 텐서 가장자리에 추가되는 패딩의 값으로, 출력 크기를 조정하고 가장자리 정보 손실을 방지한다. Dilation은 커널의 확장 인자를 정의해 커널 요소들 간의 간격을 조절한다. Bias는 필터의 추가되는 바이어스 값이다. 이라한 파라미터들은 setProperty() 함수를 통해 설정되고, 각 파라미터들은 conv_props라는 멤버 변수에 저장된다. 이후 forwarding(), calcGradient(), calcDerivative() 등의 함수에서 사용된다.

5-b. Conv2DTranspose Layer output

Conv2DTranspose layer의 주요 output은 역컨볼루션 과정을 거쳐 생성된 output tensor나, 학습과정에서의 그래디언트 값 및 미분 값들이다. Output Tensor는 입력 텐서의 차원에 따라 확장된 텐서로, [Batch, Output Channels, Output Height, Output Width] 형식을 따른다. 이 텐서는 forwarding() 함수에서 필터와 입력 텐서를 기반으로 계산되며, 활성화 함수에 따라 비선형 변환을 거쳐 모델의 출력으로 전달된다. Gradient는 학습 중 calcGradient() 함수를 통해 계산되는 그래디언트 값으로, 각 필터와 바이어스에 대한 그래디언트를 포함하여 역전파 과정에서 필터의 가중치를 업데이트하는 데 사용된다. Derivative는 calcDerivative() 함수에서 계산되는 미분 값으로, 다음 레이어로 전달되어 학습이 진행될 수 있도록 한다.

b. Inter Module Communication Interface

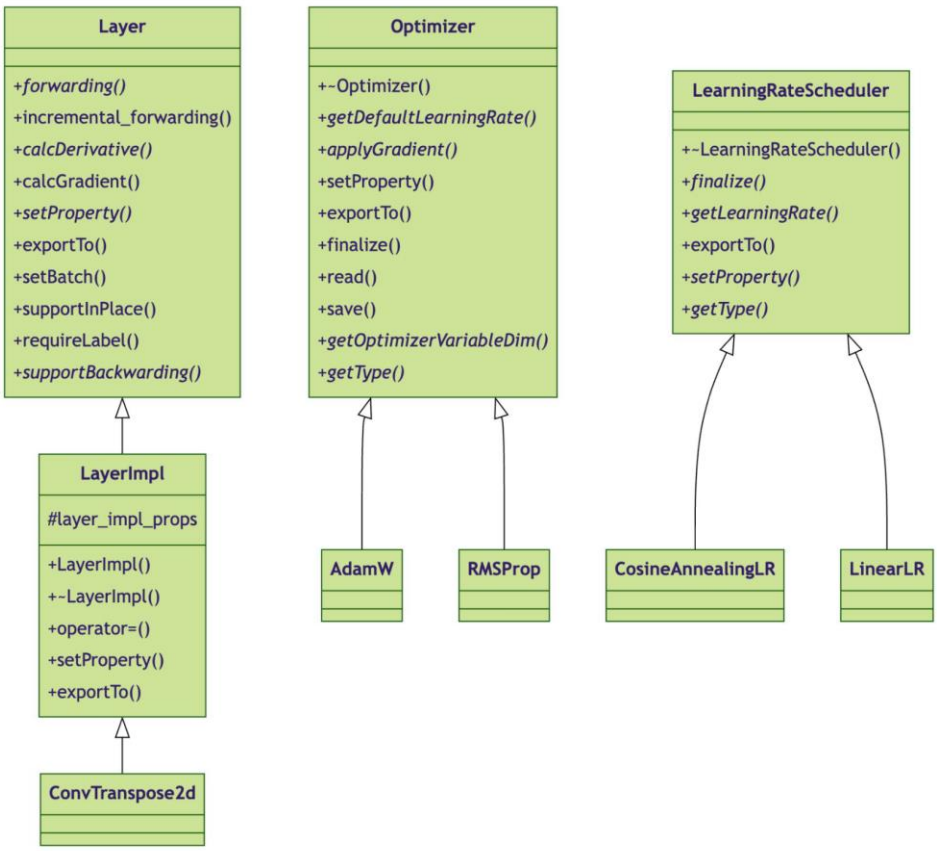
NNTrainer 프레임워크에서 Scheduler, Optimizer, Layer 모듈 간의 상호작용은 신경망 모델의 학습과정에서 유기적으로 연결되어 동작하며, 각 모듈은 서로 다른 역할을 담당하여 효율적인 학습을 지원하게 된다. NNTrainer에서 이러한 모듈들의 상호작용을 통해 학습이 진행되며, 각각의 모듈은 서로 필요할 때 서로의 정보를 주고받으며 협력한다. 각 모듈이 서로 통신하는 방식은 아래와 같다.

먼저, optimzer와 scheduler는 학습률 조정이라는 목적을 가지고 통신한다. optimizer는 학습 과정에서 가중치 업데이트를 담당하며, scheduler는 이러한 업데이트를 위한 학습률을 조정하는 역할을 한다.

scheduler는 getLearningRate() 함수를 통해 현재 단계에 적절한 학습률을 계산하여 optimizer에 전달한다. optimizer는 applyGradient() 함수에서 scheduler로부터 받은 학습률을 사용해 그래디언트를 기반으로 가중치를 업데이트 한다. 예를 들자면, AdamW optimizer는 각 단계에서 임의의 scheduler로부터 제공받은 학습률을 이용해 모멘텀과 이동 평균을 조정해 최적화 과정을 수행할 수 있다.

layer와 optimizer는 학습 과정에서 모델 파라미터의 업데이트를 위해 통신한다. 각 layer는 학습 중에 활성화된 그래디언트 정보를 optimizer로 전달하며, optimizer는 이를 통해 모델 파라미터를 조정한다. 더 자세히 보자면, optimizer는 학습 과정에서 각 layer의 가중치와 바이어스 파라미터를 업데이트 한다. 각 layer는 forward 및 backward propagation 과정에서 가중치와 바이어스의 그래디언트를 계산하고, 이 정보를 optimizer로 전달한다. optimizer는 applyGradient()함수를 호출해, layer로부터 받은 그래디언트를 이용해 각 파라미터를 업데이트하게 된다.

c. Modules



앞선 A. Input/output interface에서 설명했듯이, 이번 프로젝트에서는 NNTrainer의 모듈 중 3종류인 Layer, Optimizer, LearningRateScheduler를 개발한다. 각 모듈은 독립적인 역할을 수행하며, 학습 과정을 최적화하기 위해 서로 상호작용한다. 이들은 각각 신경망의 계층, 학습률 조정, 최적화 알고리즘을 담당한다.

먼저 Layer 모듈은 신경망의 계층을 정의하며, 데이터를 전달하고 처리하는 주요 역할을 한다. Layer는 주로 입력 데이터를 받아 forward 및 backward 연산을 수행한다. forwarding() 함수는 입력 데이터를 전방향으로 전파하여 중간 활성화 값을 계산하고, calcDerivative() 함수와 calcGradient() 함수를 통해 역방향 전파 시 계산된 그래디언트를 업데이트한다. 각 Layer는 exportTo() 함수를 통해 학습된 파라미터를 외부로 내보낼 수 있으며, 필요한 경우 setProperty() 함수를 통해 학습 파라미터를 설정할 수 있다.

다음으로 Optimizer 모듈은 학습 과정에서 가중치를 최적화하는 역할을 담당한다. 이 모듈은 각 Layer의 가중치와 그래디언트를 받아 이를 학습률에 따라 조정한다. 각 optimizer의 종류에 따른 특정한 알고리즘을 적용해 가중치 업데이트를 관리한다. applyGradient() 함수는 매 학습 단계에서 계산된 그래디언트를 기반으로 가중치를 조정하며, getDefaultLearningRate() 함수를 통해 기본 학습률을 반환한다. Optimizer는 학습률을 기반으로 가중치를 업데이트하고, 필요시 getOptimizerVariableDim() 함수를 통해 필요한 최적화 변수를 할당하여 학습 안정성을 높인다. Optimizer는 LearningRateScheduler와 연동하여 현재 학습 단계에 맞는 최적의 학습률을 적용한다.

LearningRateScheduler 모듈은 학습률을 동적으로 조정하는 역할을 한다. 주어진 학습 단계에 따라 학습률을 조정하여 학습 속도를 제어하고, 과적합을 방지한다. getLearningRate() 함수는 현재 학습 단계에 맞는 학습률을 반환하며, 이를 optimizer가 받아 가중치를 업데이트 하는 데 사용한다. setProperty() 함수를 통해 학습률 조정에 필요한 파라미터를 설정할 수 있다. 이를 통해 학습 전반적인 과정에서 적절히 학습률을 적용해 모델의 수렴 속도를 조절할 수 있다.

H. Solution

a. Implementations Details

1) Layer 관련 함수들

forwarding : Conv2DTranspose Layer의 주요 연산 함수로, 입력 텐서와 필터를 기반으로 출력 텐서를 생성한다. 이 함수는 im2col_transpose를 이용해 입력 데이터를 행렬로 변환한 뒤, 필터와의 행렬 곱셈 연산을 수행하고, 결과를 다시 4D 텐서로 변환하여 반환한다.

calcGradient : 역전파 과정에서 필터의 그래디언트를 계산하는 함수이다. 저장된 im2col 데이터를

활용하여 출력 gradient와 행렬 곱셈을 수행해 필터의 gradient를 계산한다.

calcDerivative : 출력 gradient와 필터를 활용해 이전 레이어로 전달할 입력 gradient를 계산한다. 행렬 연산을 수행한 뒤, col2im_transpose를 통해 4D 텐서로 복원하여 반환한다.

im2col_transpose : Conv2DTranspose 연산을 효율적으로 수행하기 위해 입력 텐서를 행렬로 변환하는 함수이다.

col2im_transpose : 역전파 과정에서 사용되는 행렬 데이터를 4D 텐서 형태로 복원하는 함수이다.

2) Optimizer 관련 함수들

applyGradient : 입력된 gradient를 기반으로 가중치를 업데이트하는 함수이다. 1차 및 2차 모멘텀을 업데이트하고, bias correction을 적용한 뒤 weight decay를 반영하여 최종적으로 가중치를 수정한다.

getOptimizerBariableDim : 옵티마이저가 사용하는 추가 변수들의 차원을 반환하는 함수이다. 모멘텀 및 관련 변수들의 크기를 정의한다.

exportTo : 현재 옵티마이저의 상태와 속성을 외부로 내보내는 함수로, 학습 상태를 저장하거나 재사용할 수 있도록 지원한다.

setProperty : 사용자가 제공한 설정 값을 파싱하여 옵티마이저 속성을 초기화하거나 업데이트하는 함수이다.

3) Scheduler 관련 함수들

getLearningRate : 현재 iteration에 적합한 학습률을 반환하는 함수이다. 스케줄러 종류에 따라 다른 방식으로 학습률을 계산한다. 예를 들어, CosineAnnealingLR은 cosine 함수를 기반으로 학습률을 주기적으로 조정하며, LinearLR은 학습률을 선형적으로 감소시킨다.

finalize : 스케줄러의 설정 값을 검증하고 초기화하는 함수이다. 설정된 파라미터의 유효성을 확인하며, 필수 파라미터가 누락되었을 경우 오류를 발생시킨다.

setProperty : 학습률 스케줄러의 속성을 사용자가 제공한 값으로 설정하거나 업데이트하는 함수이다.

exportTo : 학습률 스케줄러의 현재 속성을 외부로 내보내는 함수로, 학습 상태를 저장하거나 재현하는 데 사용된다.

b. Implementations Issues

1) im2col 최적화

convolution은 필터의 가중치를 이미지 그리드의 각 부분과 곱하여 이미지의 local feature를 얻어내는 연산이다. convTranspose도 이미지의 지역적 특성을 이용하는데 convolution과는 다르게 이미지를 확장한다.

연산의 설명 그대로 구현하면 필터의 각 가중치와 이미지의 각 그리드를 도는 반복문을 이용하게 된다. 그러나 for문으로 생기는 수많은 branch들은 CPU의 hardware적으로 최적화된 pipeline 등의 특성을 최대로 이용하지 못하게 한다. 따라서 이를 제거하고 하나의 행렬곱으로 convolution을 대체하는 연산이 im2col이다.

각 필터의 가중치는 이미지 feature와 곱해져 하나의 수를 만든다. 이때 각 필터를 하나의 벡터로 보고 곱해지는 이미지 feature를 골라 하나의 벡터로 만든다. 이를 반복하면 이미지 feature를 하나의 행렬로 만들 수 있고 결국 필터와의 행렬곱으로 convolution 연산을 할 수 있다.

2) NNTrainer code review 및 CI 통과 문제

NNTrainer는 대규모 오픈 소스인 만큼, CI/CD 파이프라인에서 특정 코드 스타일이나 unit test 조건을 강제하였다. 또, 많은 code review가 올라오는 공간인만큼, CI/CD checker에 걸릴만한 문제는 모두 local 환경에서 해결하는 것이 중요하였다. 이를 준수하기 위해, git commit 및 push 시에 나의 code style, commit sign, unit test 등 다양한 조건을 자동으로 체크하도록 hook을 설정하였다. 이를 통해 git consistency를 유지하는 commit만을 NNTrainer repository로 보낼 수 있었다.

3) Lack of Online References

NNTrainer 는 많은 기능을 제공하는 오픈 소스 인공 신경망 훈련 라이브러리이긴 하지만, 특정한 환경에 최적화된 라이브러리인데다가, PyTorch 나 TensorFlow 와 같은 라이브러리에 비해서 만들어진지도 오래되지 않아서, 이들 라이브러리를 사용하는 것에 익숙한 프로그래머로서 NNTrainer 를 이용하여 프로그래밍을 하게 된다면, 주로 라이브러리에서 제공하는 documentation 과 예제에만 의존해야하는, 전반적으로 다소 급격한 개발 환경의 변화를 겪게 된다. 온라인 자료가 거의 전무한 환경에서 프로그래밍을 하는 연습을 톡톡히 하며, 라이브러리에서 제공하는 documentation 과 예제를 자세히 읽어봄으로써 극복할 수 있었다.

I. Results

a. Experiments

Layer name	Layer type	Output dimension	Input layer
input0	input	1:1:28:28	
conv2d1	conv2d	1:4:14:14	input0
batch_normalization	batch_normalization	1:4:14:14	conv2d1
batch_normalization	activation	1:4:14:14	batch_normalization
conv2d3	conv2d	1:8:7:7	batch_normalization
batch_normalization	batch_normalization	1:8:7:7	conv2d3
batch_normalization	activation	1:8:7:7	batch_normalization
h	flatten	1:1:1:392	batch_normalization
h/generated_out_0	multiout	1:1:1:392	h
fc2	fully_connected	1:1:1:16	h/generated_out_0
fc2/generated_out_0	multiout	1:1:1:16	fc2
fc1	fully_connected	1:1:1:16	h/generated_out_0
fc1/generated_out_0	multiout	1:1:1:16	fc1
reparametrize	bottleneck	1:1:1:16	fc1/generated_out_0 fc2/generated_out_0
fully_connected5	fully_connected	1:1:1:392	reparametrize
fully_connected5/ac	activation	1:1:1:392	fully_connected5
reshape6	reshape	1:8:7:7	fully_connected5/ac
conv2dtranspose7	conv2dtranspose	1:4:14:14	reshape6
batch_normalization	batch_normalization	1:4:14:14	conv2dtranspose7
batch_normalization	activation	1:4:14:14	batch_normalization
conv2dtranspose9	conv2dtranspose	1:1:28:28	batch_normalization
recon_x	activation	1:1:28:28	conv2dtranspose9
vae_loss10	vae_loss	1:1:28:28	recon_x fc1/generated_out_0 fc2/generated_out_0

위는 NNTrainer를 이용하여, 프로젝트 기간 동안 구현한 conv2dTranspose를 이용하여 VAE 모델을 구현한 레이어 구조이다.

```
#1/15 - Training Loss: 3753.66 >> [ Accuracy: 0% - Validation Loss : 197177 ]
#2/15 - Training Loss: 2751.18 >> [ Accuracy: 0% - Validation Loss : 21264.3 ]
#3/15 - Training Loss: 2115.3 >> [ Accuracy: 0% - Validation Loss : 9363.32 ]
#4/15 - Training Loss: 1706.62 >> [ Accuracy: 0% - Validation Loss : 5956.27 ]
#5/15 - Training Loss: 1432.92 >> [ Accuracy: 0% - Validation Loss : 4417.4 ]
#6/15 - Training Loss: 1238.51 >> [ Accuracy: 0% - Validation Loss : 3584.28 ]
#7/15 - Training Loss: 1093.09 >> [ Accuracy: 0% - Validation Loss : 3039.6 ]
#8/15 - Training Loss: 977.455 >> [ Accuracy: 0% - Validation Loss : 2634.01 ]
#9/15 - Training Loss: 885.587 >> [ Accuracy: 0% - Validation Loss : 2320.18 ]
#10/15 - Training Loss: 808.74 >> [ Accuracy: 0% - Validation Loss : 2075.16 ]
#11/15 - Training Loss: 745.437 >> [ Accuracy: 0% - Validation Loss : 1860.18 ]
#12/15 - Training Loss: 689.879 >> [ Accuracy: 0% - Validation Loss : 1672.6 ]
#13/15 - Training Loss: 642.173 >> [ Accuracy: 0% - Validation Loss : 1506.49 ]
#14/15 - Training Loss: 600.107 >> [ Accuracy: 0% - Validation Loss : 1359.64 ]
#15/15 - Training Loss: 562.951 >> [ Accuracy: 0% - Validation Loss : 1228.59 ]
```

학습을 수행하면, 위와 같이 진행된다.

b. Result Analysis and Discussion

먼저, 프로젝트 기간 동안 구현한 OP들의 작동성을 보증하기 위해, NNTrainer convention에 맞는 다양한 unit test들을 구현하였다. 해당 unit test들이 문제없이 실행되었고, CI pass 및 code review를 받아 주어진 OP들을 전부 잘 구현할 수 있었다. 그 다음으로는, 프로젝트 기간동안 구현한 OP들을 실제로 사용해보기 위해, 기존에 NNTrainer에 구현되어 있던 모델의 scheduler나 optimizer를 변경해보는 방식으로 실험을 진행해보았다. 실험 결과. 문제없이 optimizer나 scheduler가 작동하는 것을 확인할 수 있었다.

그 다음으로는, 프로젝트 기간 동안 구현한 Layer인 Conv2dTranspose를 테스트해보기 위해, VAE 모델을 실제로 NNTrainer를 이용해서 구현해보았다. MNIST 데이터셋을 이용해서 VAE 모델의 학습 및 인퍼런스를 수행하였으며, NNTrainer 상에서 구현한 Conv2dTranspose Layer가 정확히 작동함을 확인하였다.

또한, VAE 모델의 학습 결과를 평가하기 위해 Inception Score를 활용하여 생성된 이미지의 품질을 정량적으로 분석하였다. 실험 결과, NNTrainer 기반 VAE 모델의 생성 성능이 만족스러운 수준으로 나타났으며, Pytorch로 구현된 동일한 모델과 비교했을때에도, 유사한 수준의 성능을 보임을 확인할 수 있었다. 또, VAE 모델의 학습 속도를 Pytorch 버전과 비교하였을 때, 학습 속도에 있어서 크게 떨어지지 않고, 특히 메모리는 매우 적게 사용하는 것을 확인할 수 있었다. 이를 통해 프로젝트에서 구현한 OP들이 NNTrainer의 주요 목표인 on-device 학습 최적화에 부합한다는 것을 입증하였다.

J. Division & Assignment of Work

항목	담당자
CosineAnnealingLR 스케줄러 로직 구현	이현우, 장민혁

CosineAnnealingLR 스케줄러 unit test 구현	이현우
CosineAnnealingLR 스케줄러 최적화 수행	이현우, 송우경
CosineAnnealingLR 스케줄러 디버깅 및 리펙토링	이현우
AdamW optimizer 로직 구현	장민혁, 송우경
AdamW optimizer unit test 구현	이현우, 장민혁
AdamW optimizer 최적화 수행	장민혁, 송우경
AdamW optimizer 디버깅 및 리펙토링	장민혁
Conv2DTranspose Layer 로직 구현	송우경, 장민혁
Conv2DTranspose Layer unit test 구현	이현우, 송우경
Conv2DTranspose Layer 최적화 수행	송우경
Conv2DTranspose Layer 디버깅 및 리펙토링	송우경
LinearLR 스케줄러 로직 구현	이현우, 장민혁
LinearLR 스케줄러 unit test 구현	이현우
LinearLR 스케줄러 최적화 수행	이현우, 송우경
LinearLR 스케줄러 디버깅 및 리펙토링	이현우
RMSProp optimizer 로직 구현	장민혁, 송우경
RMSProp optimizer unit test 구현	이현우, 장민혁
RMSProp optimizer 최적화 수행	장민혁, 송우경
RMSProp optimizer 디버깅 및 리펙토링	장민혁
Demo : VAE 로직 구현	장민혁, 이현우
Demo : VAE 학습	장민혁, 이현우
Demo : VAE 테스트	장민혁, 이현우

K. Conclusion

본 프로젝트를 통하여, NNTrainer에 Con2DTranspose Layer, CosineAnnealingLR 및 LinearLR 학습률 스케줄러, AdamW 및 RMSProp 옵티마이저를 성공적으로 추가하며, 딥러닝 프레임워크로서의 다양한 기능을 강화할 수 있었다. 특히, 온디바이스 학습 환경의 제약 조건인 메모리와 연산 자원 제한 속에서도

효율적인 모델 학습을 가능하게 하는 최적화된 연산을 설계하고 구현하는 과정을 통해, 시스템적인 최적화와 알고리즘적인 최적화를 함께 고려해야 함을 깊이 이해할 수 있었다. 또한, 우리가 구현한 OP들을 VAE 모델 학습에 적용하여 NNTrainer가 실제 환경에서 학습 성능을 발휘할 수 있음을 확인함으로써, 프로젝트의 실질적인 기여를 입증할 수 있었다.

이 프로젝트를 통해 단순한 코드 작성을 넘어, 오픈소스 기여 과정을 통해 협업의 중요성과 코드 품질 유지 및 최적화의 필요성을 배울 수 있었다. 또한, VAE 모델 학습 및 Inception Score 분석을 통해 NNTrainer가 기존 프레임워크와 비교해 경쟁력 있는 성능을 보임을 확인하며, 우리의 작업이 실질적인 가치를 창출함을 깨닫을 수 있었다. 이러한 경험은 온디바이스 AI 개발뿐 아니라, 다양한 딥러닝 연구 및 개발 프로젝트에 있어서도 유의미한 밑거름이 될 것이라고 생각한다.

◆ [Appendix] User Manual

NNTrainer의 프로젝트 전체 코드를 확인할 수 있는 github link는 아래와 같다.

https://github.com/nnstreamer/nntrainer

NNTrainer를 PC(linux)에 설치하기 위해서는, 아래와 같은 과정을 따르면 된다.

```
$ sudo add-apt-repository ppa:nnstreamer/ppa
$ sudo apt-get update

$ meson build
$ ninja -C build install
```

그 외 환경에서 NNTrainer를 사용하고 싶다면, 아래의 리드미를 참고하면 된다.

https://github.com/nnstreamer/nntrainer/blob/main/docs/getting-started.md

해당 NNTrainer를 이용해 직접 Model을 만들어보고 싶다면, 아래의 리드미를 참고하여 모델을 생성하면 된다.

https://github.com/nnstreamer/nntrainer/blob/main/docs/how-to-create-model.md

우리가 만든 VAE 모델은 다음 링크를 참고하면 볼 수 있고
https://github.com/2024-2-CID-TEAM-A/nntrainer/tree/VAE
우리가 작업한 전체 github group organization link는 다음과 같다.

https://github.com/2024-2-CID-TEAM-A

또한, 라이브러리를 위 순서에 따라 빌드한 뒤, repo root directory 에서

```
$ build/Applications/VAE/jni/nntrainer_VAE Applications/MNIST/res/mnist_trainingSet.dat
```

로써 demo 를 실행할 수 있다.