Nengo to ONNX(SNN) Part

2020.09.24. 한성대학교 최현웅

□ Nengo 모델 가중치 추출 및 분석

○ Nengo Simulator를 이용하여, MNIST 데이터 기반 Fully-Connected Layer를 학습하여 가중치 추출

○ 학습이 종료된 후에, 얻게 되는 가중치 파일 .npz를 분석하기 위함

○ 실제 학습을 위해 사용한 모델링 코드는 다음과 같음

```
inp = nengo.Node(np.zeros(784))
# Activation Function = LIF
x = nengo_dl.Layer(tf.keras.layers.Dense(784))(inp)
x = nengo_dl.Layer(tf.keras.layers.Dense(128))(x)
out = nengo_dl.Layer(tf.keras.layers.Dense(units=10))(x)
```

○ 추출된 npz 가중치 정보는 다음과 같음

```
layer 1 : (784, 784) <class 'numpy.ndarray'>
layer 2 : (784,) <class 'numpy.ndarray'>
layer 3 : (784, 128) <class 'numpy.ndarray'>
layer 4 : (128,) <class 'numpy.ndarray'>
layer 5 : (128, 10) <class 'numpy.ndarray'>
layer 6 : (10,) <class 'numpy.ndarray'>
```

○ 실제 딥러닝 관련 SNS 커뮤니티에 질의하여 얻은 응답은 다음과 같음

Q1.

Nengo_dl를 이용하여, 3단 Fully-connected Layer를 학습 시킨 후, 얻은 가중치 파일에 대한 구성 요소를 점검한 결과, **당연히 3단 정보가 추출될 줄 알았는데, 6단으로 이루어진 정보가 추출이 되었다. 혹시, SNN에서 Activation function 등에도 가중치가 적용이 되는지 궁금하다.**

A1.

activation function이나 단순 Matmul 과 같은 경우에는 가중치나 따로 파라미터가 없는 것으로 알고 있고, 아마 **NengoDL에서 tf.keras.layers 안에 있는 걸 Casting 해서 LIF Layer를 구성하다보니 실질적으로 히든 레이어 갯수가 2배가 된 것 같다.**

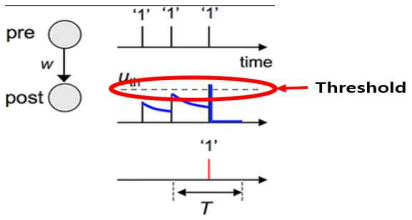
※ 즉, Nengo가 DNN 기반의 Keras 모델링 코드를 SNN으로 변환하는 과정에서 SNN Activation function를 추가함으로써 발생된 레이어 정보

Q2.

그렇다면, 예를 들어 **Layer 2와 3은 실제로는 같은 레이어에서 나온 가중치**라고 볼 수 있는데, 서로의 값을 출력하여 비교해보면 값이 많이 다른 것을 확인할 수 있다. 그럼 여기서는 **Layer 3가 Nengo에서 SNN를 사용하기 위해 만든 Layer이기 때문에, Layer 3이 실제 SNN Layer 가중치인가?**

A2.

정확하게 말하자면, Layer 3에 들어 있는 정보들은 **가중치가 아니라, LIF에서 사용하는 현재 막전위 값 혹은 역치를 의미**하는 것 같다. 그렇기 때문에 **실질적으로는 Layer 2만 존재한다고 보아야 한다.**



※ Layer 3은 LIF에서 다음 Layer로 신호 전달을 위해, 입력 값들을 모아놓는 Threshold 값 등을 의미한다.

○ 전문가의 질의응답 결과, Weight에 대한 Layer 다음에 붙는 Layer는 SNN Activation function을 운영하기 위해, 사용되는 Threshold라고 볼 수 있음

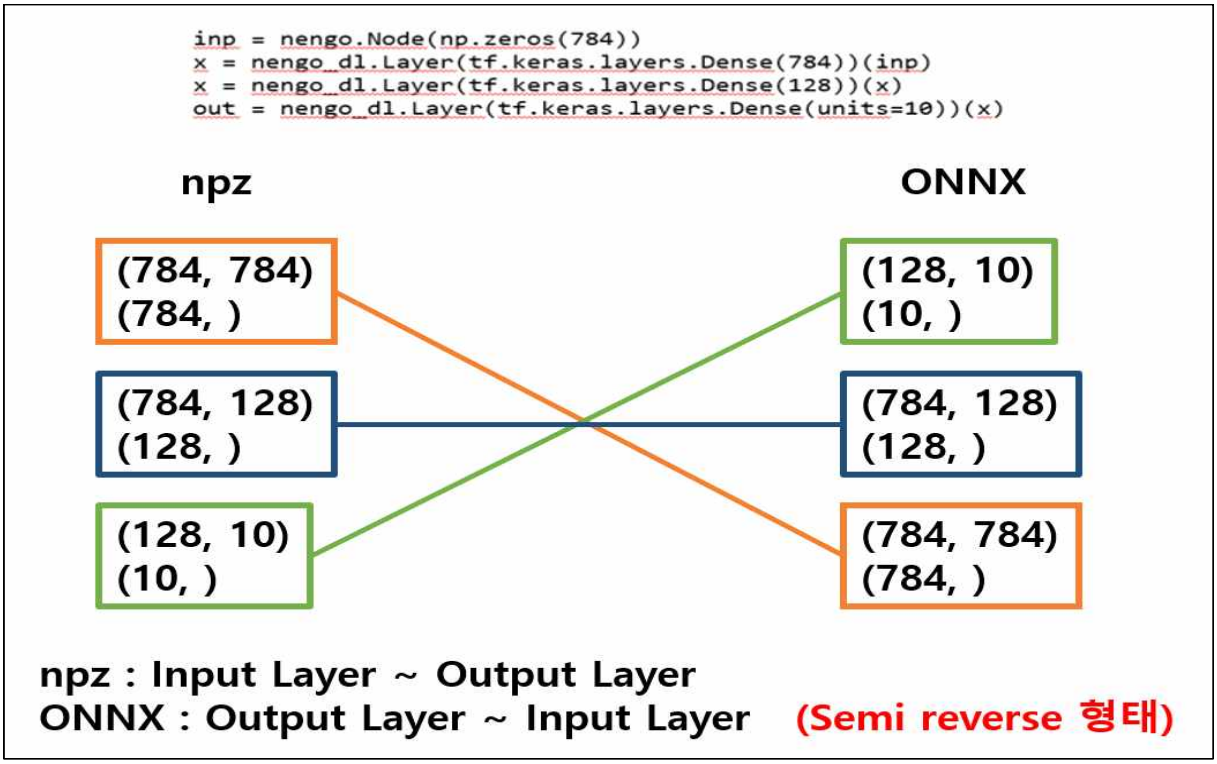
○ 최종적으로, **실제 Weight에 대한 Layer 다음에 붙는 Threshold의 값은 DNN에서 Bias에 대한 정보로 대응이 됨**

□ npz와 ONNX 간의 차이점 분석

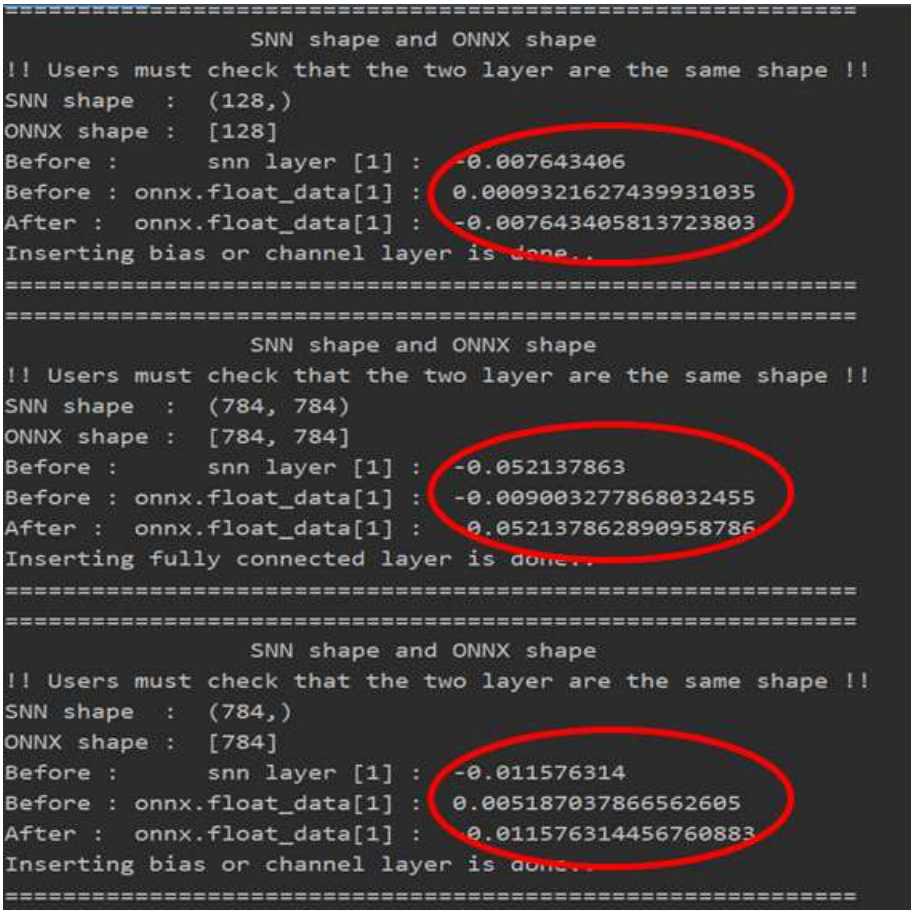
○ npz로 저장된 가중치들을 분석한 결과, 프로그래머가 모델링한 코드 순서 그대로 저장이 되는 것을 확인할 수 있음

○ 이에 해당하는 ONNX 모델을 분석하기 위해, Keras로 임시 모델을 만든 뒤, keras2onnx 모듈을 이용하여 똑같은 레이어를 갖는 ONNX 모델 생성

○ 같은 레이어 정보를 갖는 npz shape과 ONNX graph shape은 다음 그림과 같음



□ SNN to ONNX 변환 - Fully Connected Layer




```
# SNN Fully Connected Layer Version #
# Fully Connected Layer was trained by nengo_dl.simulator,
# this model consists of 784 – 128 – 10 and have activation function named LIF.
# Unfortunately, ONNX is not supporting SNN activation function, this temporary converting program uses DNN activation function named Relu.
# However, It will be enable to that the final version will be not using DNN activation function such as Relu, but will be supporting SNN activation function in ONNX.

# ! Temporary version is not automatically converting SNN model to ONNX model.
import numpy as np
import onnx

ONNX_MODEL_PATH = './document/model.onnx'
NEW_ONNX_MODEL_PATH = './adjusted_model.onnx'

snn_weights = np.load('./document/nengo-snn_fullyconnectedtest.npz')

def insert_snn_weight_2_onnx_model_fully_connected(snn_layer, onnx_layer):
    '''
    this method insert snn weigths to onnx model, which supports fully connected layer. For example, [128, 10] flattened to 1280.
```

```
    If you want to use this method, snn_layer and onnx_layer must be consisted of [None, None]
    :param snn_layer: snn layer about fully connected layer
    :param onnx_layer: onnx layer about fully connected layer

    '''
    print('============================================================')
    print('                  SNN shape and ONNX shape')
    print('!! Users must check that the two layer are the same shape !!')
    print('SNN shape  : ', snn_layer.shape)
    print('ONNX shape : ', onnx_layer.dims)

    count = 0

    print('Before :       snn layer [1] : ', snn_layer[0][1])
    print('Before : onnx.float_data[1] : ', onnx_layer.float_data[1])

    # In fully connected layer, SNN is consisted of a two-dimensional array, but, ONNX is flatten without distinction of dimension

    for i in range(0, len(snn_layer)):
        weights = snn_layer[i]

        for weight in weights:
            onnx_layer.float_data[count] = weight
            count += 1

    print('After :  onnx.float_data[1] : ', onnx_layer.float_data[1])
    print('Inserting fully connected layer is done..')
    print('============================================================')


def insert_snn_weight_2_onnx_model_biases_channels(snn_layer, onnx_layer):
    '''
        this method insert snn weigths to onnx model, which supports bias or channel.

        For example, snn layer (10,) corresponds onnx layer [10,] , so snn_layer and onnx_layer must be consisted of [None,]
        :param snn_layer: snn layer about bias or channel
        :param onnx_layer: onnx layer about bias or channel

        '''
    print('============================================================')
    print('                  SNN shape and ONNX shape')
    print('!! Users must check that the two layer are the same shape !!')
    print('SNN shape  : ', snn_layer.shape)
    print('ONNX shape : ', onnx_layer.dims)

    print('Before :       snn layer [1] : ', snn_layer[1])
```

```
    print('Before : onnx.float_data[1] : ', onnx_layer.float_data[1])
    for i in range(0, len(snn_layer)):
        weight = snn_layer[i]

        onnx_layer.float_data[i] = weight

    print('After :  onnx.float_data[1] : ', onnx_layer.float_data[1])
    print('Inserting bias or channel layer is done..')
    print('============================================================')


# ------------ snn weight components display ------------
for k in snn_weights.keys():
    print(k)
# -------------------------------------------------------

# ------- each layer weight information extraction ------
layer1 = snn_weights.get('arr_0')
layer2 = snn_weights.get('arr_1')
layer3 = snn_weights.get('arr_2')
layer4 = snn_weights.get('arr_3')
layer5 = snn_weights.get('arr_4')
layer6 = snn_weights.get('arr_5')

# It is very important to check numpy arrays shape
print('layer 1 : ', len(layer1), layer1.shape, type(layer1))
print('layer 2 : ', len(layer2), layer2.shape, type(layer2))
print('layer 3 : ', len(layer3), layer3.shape, type(layer3))
print('layer 4 : ', len(layer4), layer4.shape, type(layer4))
print('layer 5 : ', len(layer5), layer5.shape, type(layer5))
print('layer 6 : ', len(layer6), layer6.shape, type(layer6))

# onnx model load
onnx_model = onnx.load_model(ONNX_MODEL_PATH)

# onnx_graph information extraction
onnx_weights = onnx_model.graph.initializer
```

```
# Also, Checking ONNX model's weights shape is very important,
# this is because they have some different shape both.
# ------------------------------------
# ------ Layer shape information ------
#   SNN                    ONNX
# 784, 784               128, 10
# 784                    10
# 784, 128               784, 128
# 128                    128
# 128, 10                784, 784
# 10                     784
# ------------------------------------
print('** Checking ONNX weights dims... **')
for i in onnx_weights:
    print(i.dims)
print('** Done.... **')

# !! It is possible to directly modify float_data in onnx_model.graph.initializer though python code

## Modifying ONNX model's first layer
# This corresponds to the last of second layer of SNN
snn_first_graph = layer5
onnx_first_graph = onnx_weights[0]

insert_snn_weight_2_onnx_model_fully_connected(snn_first_graph, onnx_first_graph)


## Modifying ONNX model's second layer
# This corresponds to the last layer of SNN
snn_second_graph = layer6
onnx_second_graph = onnx_weights[1]

insert_snn_weight_2_onnx_model_biases_channels(snn_second_graph, onnx_second_graph)


## Modifying ONNX model's third layer
# This corresponds to the third layer of SNN
snn_third_graph = layer3
onnx_third_graph = onnx_weights[2]

insert_snn_weight_2_onnx_model_fully_connected(snn_third_graph, onnx_third_graph)

## Modifying ONNX model's fourth layer
# This corresponds to the fourth layer of SNN
snn_fourth_graph = layer4
onnx_fourth_graph = onnx_weights[3]

insert_snn_weight_2_onnx_model_biases_channels(snn_fourth_graph, onnx_fourth_graph)
```

```
## Modifying ONNX model's fifth layer
# This corresponds to the first layer of SNN
snn_fifth_graph = layer1
onnx_fifth_graph = onnx_weights[4]

insert_snn_weight_2_onnx_model_fully_connected(snn_fifth_graph, onnx_fifth_graph)

## Modifying ONNX model's sixth layer
# This corresponds to the second layer of SNN
snn_sixth_graph = layer2
onnx_sixth_graph = onnx_weights[5]

insert_snn_weight_2_onnx_model_biases_channels(snn_sixth_graph, onnx_sixth_graph)

# ONNX model save.
onnx.save_model(onnx_model, NEW_ONNX_MODEL_PATH)
```