```
import tensorflow as tf
import numpy as np

model = tf.keras.models.load_model('model_perm_f64.h5')
x = tf.Variable([[1., 1.5, 2.]], dtype=tf.float64)

with tf.GradientTape() as tape:
    y = model(x)

grad = tape.gradient(y, x)
print(y.numpy(), grad.numpy())
```

텐서플로우의 그래디언트 테이프 함수를 사용한 자동 미분

Automatic Differentiation Using Gradient Tapes in Tensorflow

하 승 현
한국해양대학교 해양공학과 교수

1. 서론

최근 머신러닝 및 인공지능 분야의 급속한 발전과 더불어, 기존에 수치 해석 분야에서 널리 사용되었던 포트란이나 C 언어 대신에 파이썬 언어의 활용도가 점차 높아지고 있다. 또한 인공지능 개발 분야에서는 텐서플로우(Tensorflow)나 파이토치(PyTorch)와 같은 응용 프로그래밍 환경(application programming interface; API)이 널리 사용되고 있다. 그 과정에 있어서 주로 머신러닝 또는 기계학습을 수행하게 되는데, 일반적으로 여러 입력 변수의 조합으로 구성된 손실함수(loss function)를 최소화하는 연산을 무수히 많이 반복 수행하게 된다. 이 과정에서 구배(gradient) 기반의 최소화를 효율적으로 진행하기 위해서는 정확한 미분값 또는 설계 민감도 계산이 필수적으로 요구된다.

텐서플로우에서는 이와 같은 미분값을 계산하기 위해서 자동 미분(automatic differentiation) 연산을 사용한다. 특히 자동 미분은 여러 층(layer)으로 구성된 네트워크를 학습하는데 있어서 역전파(back propagation)와 같은 머신러닝 알고리즘을 구현하는데 매우 유용한 것으로 알려져 있다. 먼저 자동 미분에 대한 보다 구체적인 내용은 지난 2022년도 전산구조공학 학회지 제35권 4호에서도 자세히 다뤄진 바가 있다[1]. 이에 본 기사에서는 텐서플로우 환경에서 자동 미분을 위한 그래디언트 테이프의 사용법에 대해서 보다 구체적으로 살펴보고자 한다.

2. 그래디언트 테이프(Gradient Tape)

텐서플로우에서는 자동 미분 연산을 위해서 순방향(forward)으로 어떤 연산이 어떤 순서로 진행되었는지를 기억하고, 이 연산 목록을 역방향(backward)으로 진행해 나가며 연산의 미분값을 계산한다. 이를 위해서 텐서플로우에 내장된 그래디언트 테이프(GradientTape) API를 사용하게 된다[2]. 그 간단한 사용 예시는 다음과 같다.

```
import numpy as np
import tensorflow as tf

x = tf.Variable(4.0)

with tf.GradientTape() as tape:
    y = x**3 + x**2

dy_dx = tape.gradient(y, x)
dy_dx.numpy()
```

스칼라 값에 대한 그래디언트 테이프 사용 예시

이에 대한 실행 결과는 56.0인데, 이는 $y = x^3 + x^2$의 미분식인 $y' = 3x^2 + 2x$에 $x = 4.0$을 대입한 당연한 결과이다. 이와 같이 주어진 함수식에 대한 미분식을 명시적으로 입력하지 않더라도 그래디언트 테이프를 통한 자동 미분 연산을 통해서 그 미분값을 정확하게 계산할 수 있다.

한편 이와 같은 연산은 간단한 스칼라 값에만 적용되는 것이 아니라 모든 텐서(tensor)에서 동일하게 작동된다. 이에 대한 사용 예시는 다음과 같다.

```
t1 = tf.Variable([2, 3, 4], dtype=tf.float32)
t2 = tf.Variable([5, 6, 7], dtype=tf.float32)

with tf.GradientTape() as tape:
    t3 = t1 * t2
    t4 = 2 * t1 + 3 * t2 + 4 * t3

gradients = tape.gradient(t4, [t1, t2, t3])
dt1, dt2, dt3 = gradients[0], gradients[1], gradients[2]

print('t3: ', t3.numpy())
print('t4: ', t4.numpy())
print('dt1: ', dt1.numpy())
print('dt2: ', dt2.numpy())
print('dt3: ', dt3.numpy())
```

텐서 값에 대한 그래디언트 테이프 사용 예시

먼저 $t_1 = [2,3,4]$와 $t_2 = [5,6,7]$로 각각 선언된 두 텐서를 이용해서, 새로운 텐서 $t_3 = t_1 \times t_2$와 $t_4 = 2 \times t_1 + 3 \times t_2 + 4 \times t_3$를 각각 정의하였다. 참고로 두 텐서의 곱은 내적이 아니라 텐서의 각 요소별(element-wise) 곱으로 정의된다. 그 결과 t_3와 t_4는 각각 $t_3 = [10,18,28]$ 및 $t_4 = [59,96,141]$로 계산된다. 위의 스칼라 경우와 마찬가지로 t_3와 t_4에 대한 연산이 그래디언트 테이프 함수 안에서 이루어지기 때문에 그 연산 순서 및 과정이 기억된다. 그 후에 t_1, t_2, t_3에 대한 t_4의 미분값을 tape.gradient 함수를 이용해 호출하게 되면 dt_1, dt_2, dt_3라는 변수에 각각 그 미분값이 저장되게 되고, 그 결과는 각각 다음과 같다.

$$dt_1 = 2 \times [1,1,1] + 4 \times t_2 = [22,26,30],$$
$$dt_2 = 3 \times [1,1,1] + 4 \times t_1 = [11,15,19],$$
$$dt_3 = 4 \times [1,1,1] = [4,4,4]$$

이와 같이 단순한 스칼라 연산뿐만 아니라 다양한 텐서 연산에서도 그래디언트 테이프 함수를 통해서 미분식의 명시적인 표현없이도 그 미분값을 정확하게 계산할 수 있음을 확인하였다.

3. 학습된 인공 신경망에서의 미분값 추출

머신러닝 또는 딥러닝 분야에서 가장 널리 사용되고 있는 기법 중에 하나가 인공 신경망(artificial neural network; ANN) 학습 기법이다. 내부에 수많은 노드들로 구성된 은닉층(hidden layer)을 여러 층으로 겹겹이 쌓아 결합하여 만든 인공 신경망에 대해서 입력값과 출력값 사이의 최적의 관계식을 얻기 위해서 신경망을 구성하고 있는 모든 가중치(weight) 및 편향(bias) 값들을 무수히 많은 반복 계산 과정을 통해서 업데이트해 나가게 된다. 이와 같은 과정을 통해서 한번 학습된 인공 신경망은 추후에 임의의 입력값에 대해서 복잡한 해석 과정없이도 그 출력값을 높은 정확도로 예측하게 되는데, 그 과정에서 그래디언트 테이프 함수를 활용하게 되면 각 입력값에 대한 출력값의 미분값 또는 설계 민감도 값을 복잡한 과정없이 계산할 수 있다.

먼저 본문에서 예시로 사용되는 인공 신경망 모델은 필자의 연구에 사용되었던 3차원 엮임 재료 물성치 추정 모델이다[3]. 우선 수치 모델은 3차원 엮임 재료에서 각 축방향 와이어 사이의 간격을 정의하는 세 개의 설계 변수인 x_1, x_2, x_3로 매개변수화되어 있으며, 인공 신경망 학습 과정을 통해서 설계 변수와 재료 물성치(밀도, 체적탄성계수, 열전도계수, 유체투과율) 사이의 관계가 0.99 이상의 높은

결정계수를 가지도록 미리 학습되어 있다.

```
import tensorflow as tf
import numpy as np

model = tf.keras.models.load_model('model_perm_f64.h5')
x = tf.Variable([[1., 1.5, 2.]], dtype=tf.float64)

with tf.GradientTape() as tape:
    y = model(x)

grad = tape.gradient(y, x)
print(y.numpy(), grad.numpy())
```

학습된 인공신경망 모델에 대한 그래디언트 테이프 사용 예시

```
h = 1.e-2

x1f = x + tf.Variable([[h,0,0]], dtype=tf.float64)
x2f = x + tf.Variable([[0,h,0]], dtype=tf.float64)
x3f = x + tf.Variable([[0,0,h]], dtype=tf.float64)
x1b = x - tf.Variable([[h,0,0]], dtype=tf.float64)
x2b = x - tf.Variable([[0,h,0]], dtype=tf.float64)
x3b = x - tf.Variable([[0,0,h]], dtype=tf.float64)

dx1 = (model(x1f) - model(x1b)) / (2.*h)
dx2 = (model(x2f) - model(x2b)) / (2.*h)
dx3 = (model(x3f) - model(x3b)) / (2.*h)
print(dx1.numpy(), dx2.numpy(), dx3.numpy())
```

자동 미분의 정확도를 검증하기 위한 유한차분 사용 예시

먼저 그래디언트 테이프 사용에 앞서 인공 신경망 모델의 학습이 필요한데, 본 원고에서는 텐서플로우에 내장되어 있는 'load_model' 함수를 사용해서 이미 학습된 모델(model_perm_f64.h5)을 불러왔다. 참고로 텐서플로우에 마찬가지로 내장되어 있는 'model.save' 함수를 사용하면 학습된 전체 모델을 텐서플로우 SavedModel 형식 또는 과거의 Keras H5 형식으로 디스크에 저장할 수 있다[4]. 다음으로는 입력 변수 x를 정의하였는데, 임의로 $x_1 = 1.0$, $x_2 = 1.5$, $x_3 = 2.0$에서의 재료 물성치 값을 추정하고 그 위치에서의 각 설계 변수별 미분값을 계산하였다. 앞서 살펴본 임의의 스칼라 또는 텐서 연산의 경우와 달리 신경망 모델로 정의된 경우에는 명시적인 미분 관계식을 얻어내는 것이 거의 불가능하지만, 그래디언트 테이프 함수를 사용하게 되면 신경망 내부에서 일어나는 연산의 순서와 과정을 기억하기 때문에 자동 미분의 역방향 연산을 통해서 미분값을 계산할 수 있다. 그리고 그 결과값은 tape.gradient 함수를 통해서 입력 변수 x와 동일한 크기를 가지는 배열로 저장된다.

끝으로 자동 미분으로 계산된 미분값의 정확도를 검증하기 위해서 유한차분을 통해 계산된 미분값과 비교해 보았다. 다음의 표 1에서 AD_64와 FD_64는 배정밀도(double precision)에서 각각 자동 미분과 유한 차분으로 계산된 미분값을 나타내며, AD_32와 FD_32는 단정밀도(single precision)에서 마찬가지로 각각 계산된 자동 미분 및 유한 차분 미분값을 나타낸다. 참고로 배정밀도가 아닌 단정밀도의 결과값을 얻기 위해서는 위의 코드 식에서 'tf.float64'를 'tf.float32'로 변경해 주어야 한다. 유한 차분을 이용한 미분값을 계산하기 위해서 현재 설계 위치에서 각 설계 변수마다 $\pm h$ 만큼 이동된 위치에서 모델의 물성치 값을 호출해 주었으며, 그 차이값을 $2h$ 로 나누어 주었다. 그 결과 배정밀도 계산에서는 유한 차분과 정확하게 100% 일치하는 자동 미분 결과값이 얻어진 반면에, 단정밀도 계산에서는 그 정밀도가 조금 떨어지는 것을 확인할 수 있었다. 추가적으로 표 2와 같이 단정밀도 계산에서는 h 값의 크기에 따라서 유한 차분 미분값의 정확도가 달라지는 현상도 확인할 수 있었는데, 이는 수치 모델의 비선형성이 증가하게

표 1 배정밀도와 단정밀도 계산에서 미분값의 정확도 비교

	AD_64	FD_64	Accuracy	AD_32	FD_32	Accuracy
$\frac{dy}{dx_1}$	10.55335639	10.55335639	100.00%	10.872234	10.657883	98.03%
$\frac{dy}{dx_2}$	3.54598202	3.54598202	100.00%	3.3432283	3.252411	97.28%
$\frac{dy}{dx_3}$	4.54459815	4.54459815	100.00%	4.8318615	4.7807693	98.94%

표 2 h 값의 변화에 따른 단정밀도 유한 차분 미분값의 정확도 비교

	AD_32	$h=10^{0}$	$h=10^{-1}$	$h=10^{-2}$	$h=10^{-3}$	$h=10^{-4}$
$\frac{dy}{dx_1}$	10.872234	10.719289	10.851669	10.657883	10.393142	0.0
$\frac{dy}{dx_2}$	3.3432283	3.4288673	3.3495903	3.252411	5.073547	0.0
$\frac{dy}{dx_3}$	4.8318615	5.3208714	4.815674	4.7807693	4.594803	0.0

된다면 미분값의 정확도는 더 떨어질 것으로 예측할 수 있다. 따라서 정확한 미분값을 필요로 하는 곳에서는 계산 비용이 더 소모되더라도 배정밀도 계산이 필수적이라고 할 수 있다. 참고로 신경망 모델이 배정밀도를 가지도록 정의하기 위해서는 신경망 모델의 정의 및 학습에 앞서서 “tf.keras.backend.set_floatx('float64')”의 명령어를 사용해서 신경망 모델 구성에 사용되는 모든 변수들이 배정밀도로 정의되고 내부 연산이 수행되도록 명시해야 한다.

4. 결론

경사도(gradient) 기반의 최적 설계에서는 정확하고 효율적인 최적화를 위해 목적함수와 제약조건의 미분값 또는 설계 민감도의 정확한 계산이 매우 중요하다. 일반적으로 구조물의 컴플라이언스 최소화와 같은 위상최적설계 문제들은 보조 방정식(adjoint equation)을 사용해서 빠르고 정확하게 그 설계 민감도를 구하고 최적설계를 수행하는 방법이 널리 알려져 있지만, 보다 일반화된 문제에서 유한 차분이 아닌 해석적인 방법으로 설계 민감도를 구하는 것은 여전히 쉽지 않은 문제이다.

하지만 서론에서 언급한 바와 같이 기계 학습 또는 인공지능 학습에서 학습이라고 불리는 과정이 다름이 아니라 문제에 따라 정의된 손실함수를 최소화시켜 나가는 과정이기 때문에 다양하고 효율적인 최적화 알고리즘이 텐서플로우나 파이토치와 같은 API에 이미 내장되어 있으며, 또한 사용자가 개발한 알고리즘을 직접 적용시킬 수도 있다. 따라서 예를 들어 회귀 분석을 통한 반응 표면(response surface)을 생성하는 문제 등에서는 미분값 또는 설계 민감도를 구하는 과정에서 본문에 소개된 텐서플로우 코드를 바로 적용시켜 볼 수 있다. 그 밖에도 다양한 최적 설계 문제에서 그래디언트 테이프 함수를 사용한 직접 미분 기능을 사용해서 텐서플로우 API를 활용한 연구를 확장시켜 나갈 수 있을 것으로 기대한다.

참고 문헌

1. 구본용 (2022), 자동 미분의 공학문제 적용, 전산구조공학 35(4), 23~26.
2. https://www.tensorflow.org/guide/autodiff
3. 김병모, 하승현 (2023), 시뮬레이션 기반 3차원 엮임 재료의 물성치 분석 및 인공 신경망 해석, 한국전산구조공학회 논문집 36(4), 256~264.
4. https://www.tensorflow.org/guide/keras/serialization_and_saving