5조 | 최대상 조은정

BITAmin

week3

딥러닝과 최적화 이론

optimization in deep learning

Contents

A

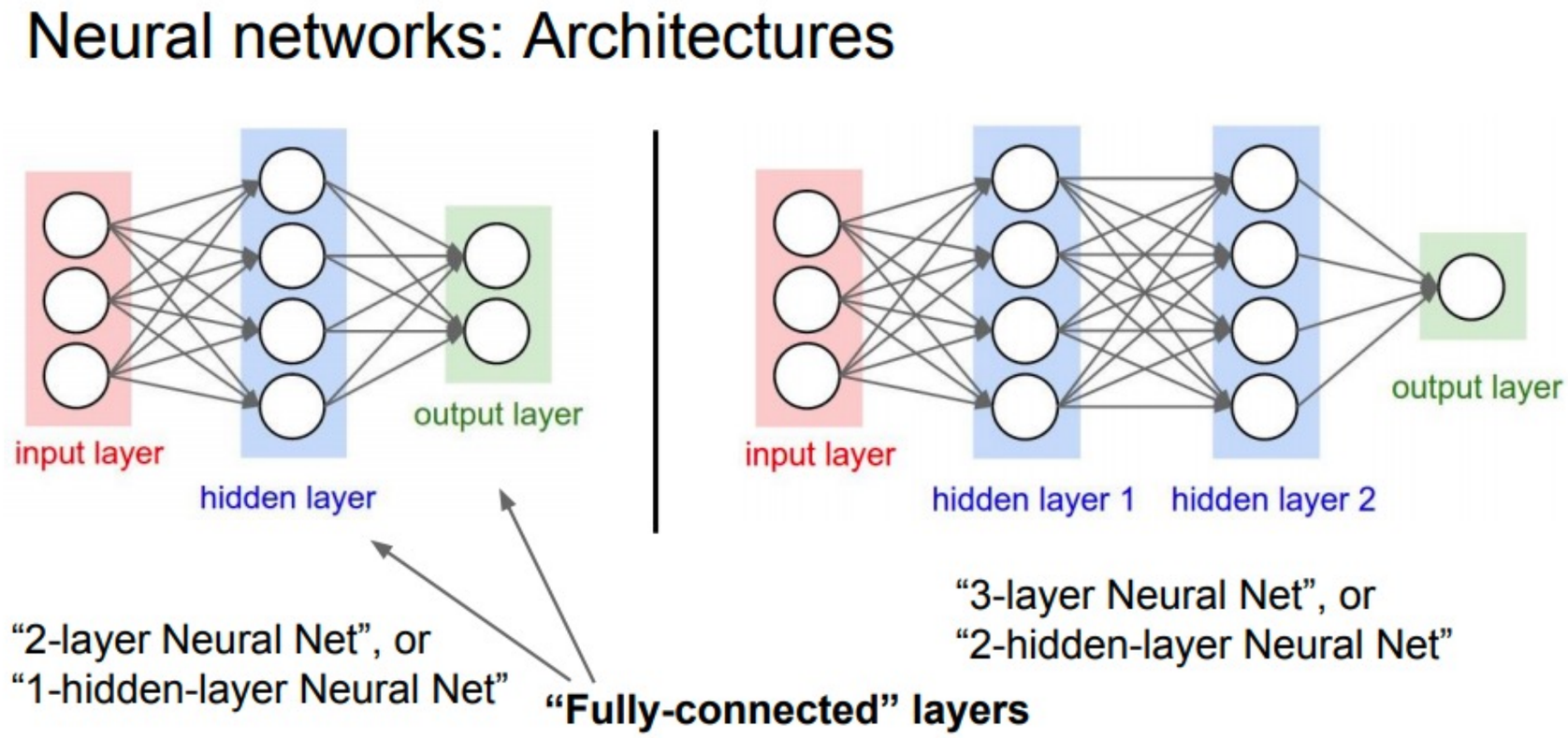
- 신경망 구조
- 활성화 함수
- 역전파

Review

B

- 최적화 개요
- 최적화 알고리즘
- 하이퍼파라미터 튜닝

Main



Check Point!

n 층 신경망 : 매끈한 활성화 함수를 사용하는 다층 퍼셉트론(MLP) → 퍼셉트론의 가중치 결정을 자동으로 학습

✓ 입력층 : 0 층

✓ 은닉층 : 1 ~ (n-1) 층

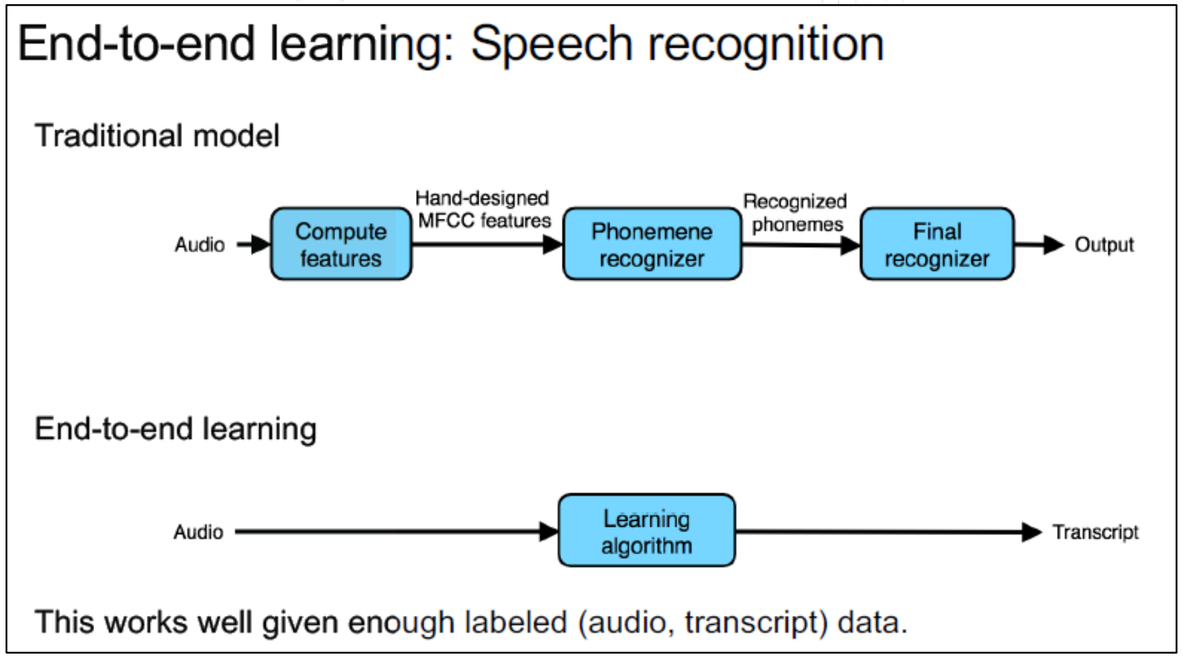
✓ 출력층 : n 층

퍼셉트론 수렴 정리

데이터가 선형 분리 가능 → 유한 번의 학습에 의해 단층 퍼셉트론의 정확도가 100%로 수렴

End-to-End

머신러닝에서 입력 데이터(처음)부터 목표한 결과(끝)까지 사람의 개입 없이 학습하는 방식 ; 종단 간 기계학습



Dense Layer

완전연결(Fully-Connected) 계층 ex. 일반 신경망에서 Affine 변환을 수행하는 계층

Affine 변환

신경망의 순전파에서 수행하는 행렬의 곱 → 행렬의 shape에 맞춰 연결

$$W^{(k)} = \left(w_{ji}^{(k)}\right), \quad B^{(k)} = \left(b_j^{(k)}\right)$$

- 가중치 행렬에서의 위치 : i 행 j 열
- k : 다음 층 번호 ; 1~(출력층 번호 n)
- j : 다음 층에서 뉴런 번호 ; 1~(다음 층 뉴런 개수)
- i : 앞 층에서 뉴런 번호 ; 1~(앞 층 뉴런 개수)

ex. 1층의 1번 뉴런으로 가는 신호

$$a_1^{(1)} = w_{11}^{(1)}x_1 + w_{12}^{(1)}x_2 + b_1^{(1)}$$

★ 신호를 다음 층으로 전달할 때의 가중치 행렬 계산 식

$$A^{(k)} = XW^{(k)} + B^{(k)}$$

```python
# Affine 계층
class Affine:
    def __init__(self, W, b):
        self.W = W
        self.b = b
        self.x = None
        self.dW = None
        self.db = None

    def forward(self, x):
        self.x = x
        out = np.dot(x, self.W) + self.b
        return out

    def backward(self, dout):
        dx = np.dot(dout, self.W.T)
        self.dW = np.dot(self.x.T, dout)
        self.db = np.sum(dout, axis=0)
        return dx
```

배치(Batch) 처리

미니배치 : 훈련 데이터의 일부 무작위 추출 → 여러 개의 작은 배열 대신 하나의 큰 배열을 계산하여 효율 증대

Epoch

훈련 데이터 전체에 대한 순회 단위 → 1 epoch의 경사 하강 횟수 = train size / batch size

계층

신경망의 기능 단위 → 일반적으로 프레임워크 내 클래스로 구현 ; 내부에서 순전파와 역전파 계산

순전파

학습한 매개변수(가중치와 편향)를 이용해 추론하는 과정

출력층 설정

활성화(출력값 반환) → 항등함수(회귀) | 시그모이드(이진분류) | 소프트맥스(다중분류) ; 뉴런 개수 결정

활성화 함수 $h(x)$

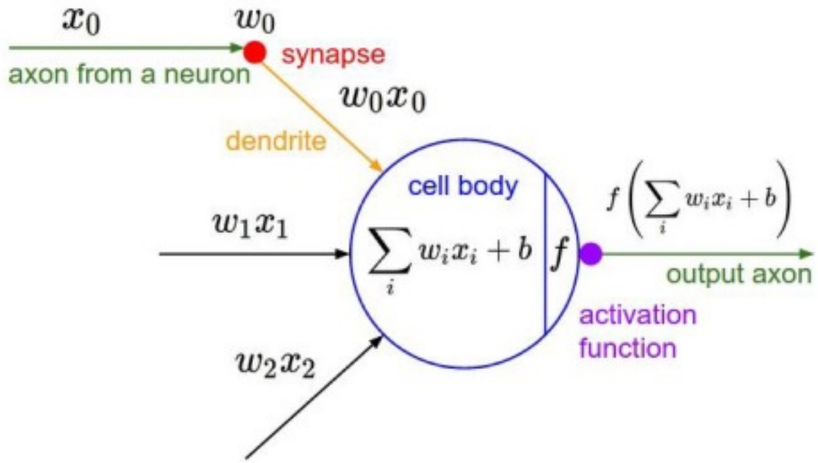
가중 입력 신호의 총합을 출력 신호로 변환

- 선형 함수 : $f(x) = ax + b$
- 비선형 함수 : 선형이 아닌 함수로, 하나의 직선으로 그려지지 않는 함수

★ 여러 선형 함수의 합성 함수는 하나의 선형 함수로 표현 가능

$\Longrightarrow$ 선형 활성화 함수를 사용하면 신경망의 은닉층을 구성하는 것이 무의미

$\Longrightarrow$ 다층 신경망은 항상 비선형 활성화 함수를 사용



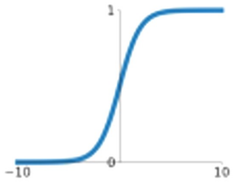
활성화 함수 복잡한 문제를 해결할 수 있도록 각 은닉층에서 비선형성 추가 ; 출력층에서는 문제에 적합한 출력값으로 변환 수행

Check Point!

활성화 함수 종류 : ~~Step~~, Sigmoid, ReLU, Tanh, Leaky ReLU, … → 전달 받은 신호를 얼마나 어떻게 활성화할지 결정(뉴런의 역치)

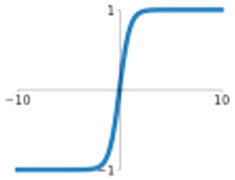
Sigmoid

$$\sigma(x) = \frac{1}{1+e^{-x}}$$



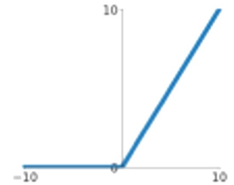
tanh

$$\tanh(x)$$



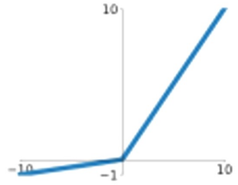
ReLU

$$\max(0, x)$$



Leaky ReLU

$$\max(0.1x, x)$$

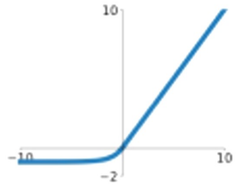


Maxout

$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



TLDR: In practice:

- Use ReLU. Be careful with your learning rates
- Try out Leaky ReLU / Maxout / ELU
- Try out tanh but don't expect much
- Don't use sigmoid

Check Point!

✓ saturated → vanishing gradient

✓ zero centered → 최적화 경로 탐색 효율 증대

✓ Maxout은 LU 계열의 일반화된 형태 ; 두 선형 함수의 매개변수는 모델에 의해 학습 → 매개변수 개수 2배

활성화 함수

S-O-T-A

BERT, GPT, Transformers…

paperswithcode.com

Activation	μ_{acc}	μ_{loss}	σ_{acc}
Mish	**87.48%**	4.13%	0.3967
Swish [37]	87.32%	4.22%	0.414
GELU [16]	87.37%	4.339%	0.472
ReLU [25, 34]	86.66%	4.398%	0.584
ELU [6]	86.41%	4.211%	0.3371
Leaky ReLU [32]	86.85%	**4.112%**	0.4569
SELU [23]	83.91%	4.831%	0.5995
SoftPlus	83%	5.546%	1.4015
SReLU [21]	85.05%	4.541%	0.5826
ISRU [4]	86.85%	4.669%	**0.1106**
TanH	82.72%	5.322%	0.5826
RReLU [48]	86.87%	4.138%	0.4478

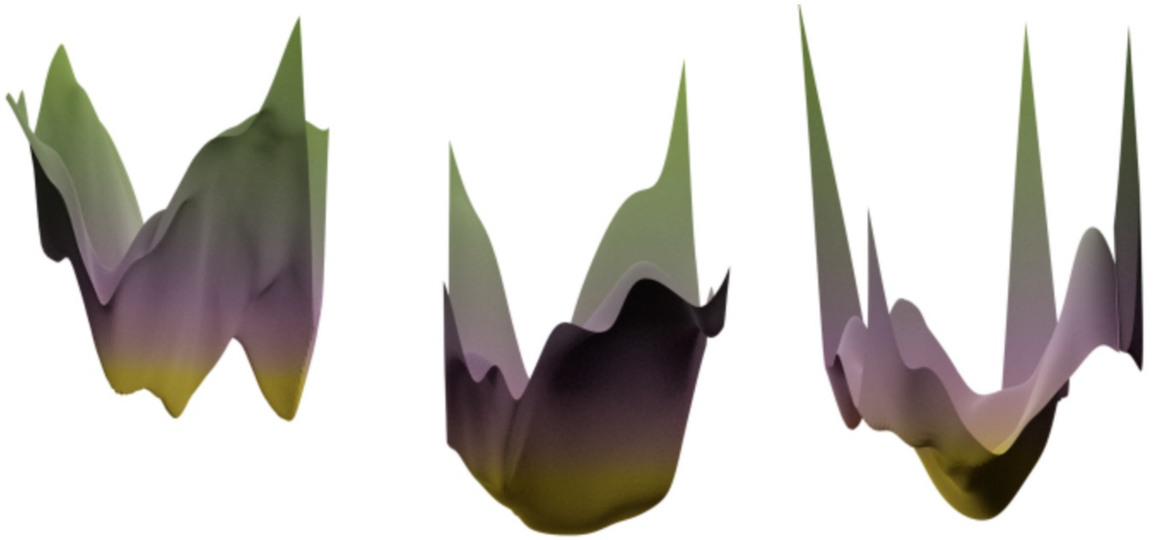
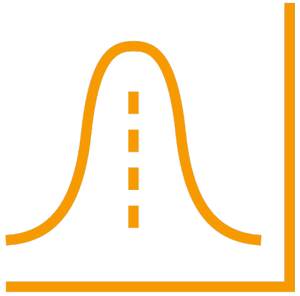


Figure 4: Comparison between the loss landscapes of (from left to right): (a) ReLU, (b) Mish and (c) Swish activation function for a ResNet-20 trained for 200 epochs on CIFAR-10.

Table 1. Properties Summary of Mish

Order of Continuity	C^{∞}
Monotonic	No
Monotonic Derivative	No
Saturated	No
Approximates Identity Near Origin	Yes



Gaussian Error Linear Unit

$$\mathrm{GELU}(x) = x * \Phi(x)$$

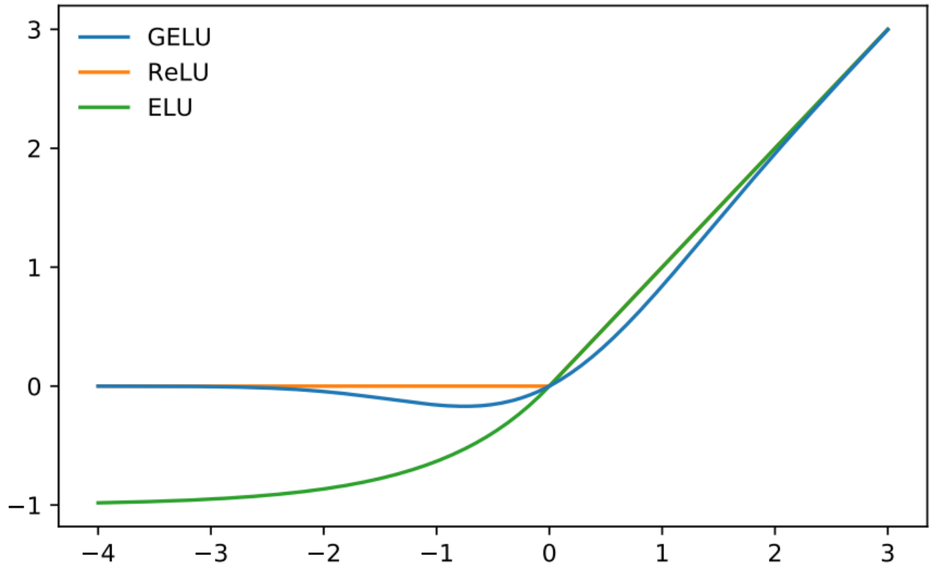
where $\Phi(x)$ is the Cumulative Distribution Function for Gaussian Distribution.

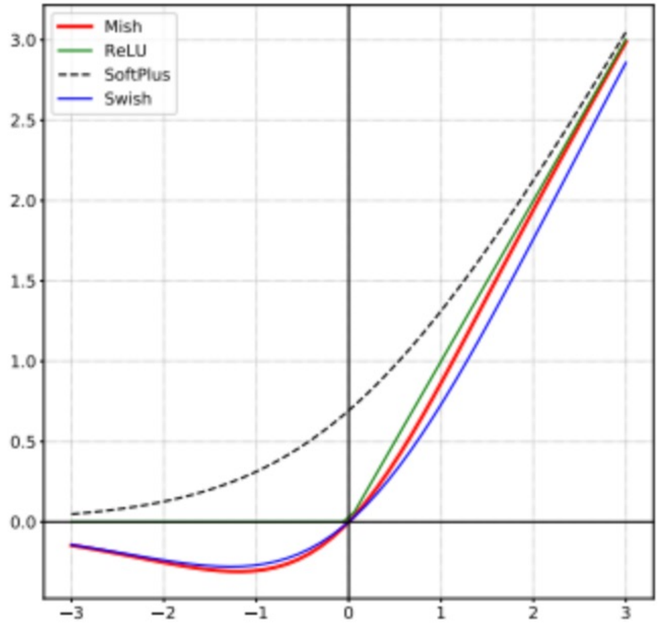
When the approximate argument is 'tanh', Gelu is estimated with:

$$\mathrm{GELU}(x) = 0.5 * x * (1 + \mathrm{Tanh}(\sqrt{(2/\pi)} * (x + 0.044715 * x^3)))$$

오차함수는 정규 분포의 누적분포함수와 본질적으로 동일하다.
Φ라고 쓰며 상수배하거나 평행이동하는 차이밖에 없다.

$$\Phi(x) = \frac{1}{2}\left[1 + \mathrm{erf}\left(\frac{x}{\sqrt{2}}\right)\right]$$





활성화 함수 + 가중치 초기화

가중치 초기화 설정 : 적절한 표준편차의 랜덤 값으로 초기화해야 함

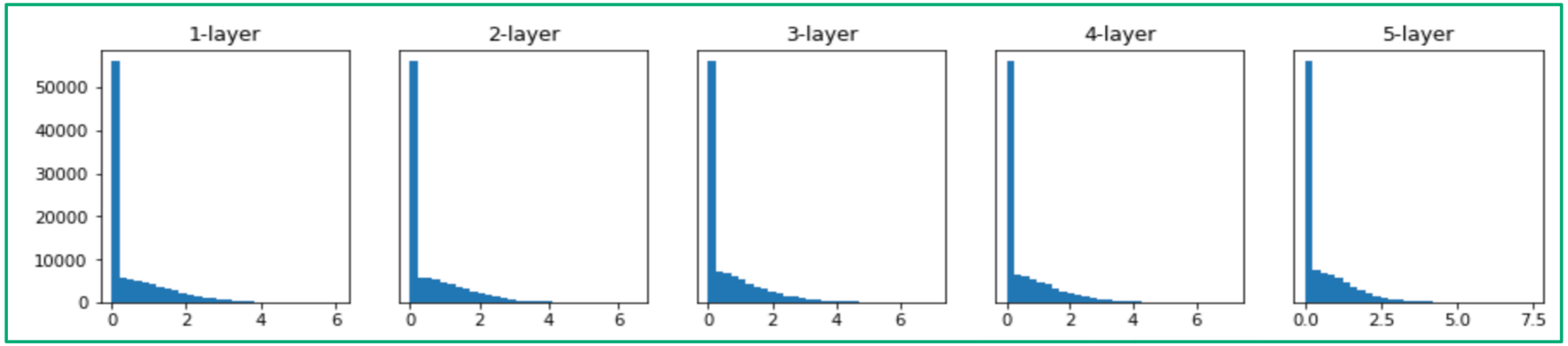
- 동일한 값으로 초기화
  → 여러 가중치를 사용하는 것이 무의미
- 활성화값의 밀집 현상
  → 표현력 제한(여러 노드를 사용하는 것이 무의미), 기울기 소실(ex. 활성화 함수가 sigmoid일 때 0과 1로 밀집되는 경우)

★ 신경망 학습의 효율성을 위해, 서로 다른 층 간에 적당히 다양한 데이터가 흐르도록 가중치를 초기화

1. **Xavier 초깃값** : 표준편차가 이전 층 노드 수 제곱근의 역수인 정규분포에서 랜덤 추출
   - 활성화 함수가 **sigmoid, tanh**인 경우 사용
2. **He 초깃값** : 표준편차가 Xavier 초깃값 표준편차의 2배가 되도록 정규분포에서 랜덤 추출
   - 활성화 함수가 **ReLU**인 경우 사용

ReLU 함수 + 가중치 초기화 실험 ⌥ 은닉층 활성화 값 분포 관찰

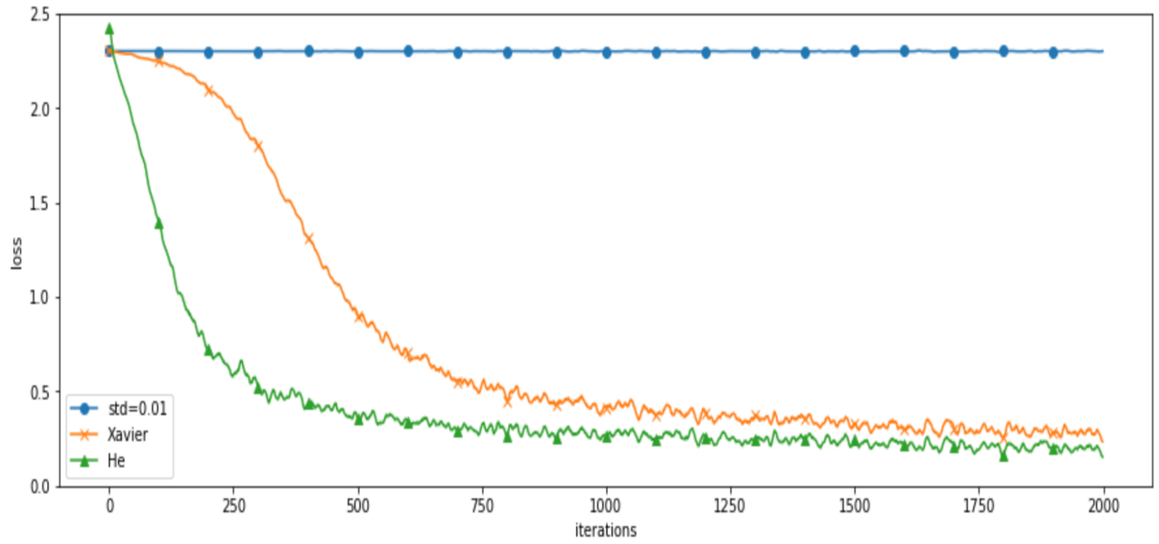
```
# 가중치 초기화 설정 ; 초깃값의 표준편차를 바꿔 실험
# w = np.random.randn(node_num, node_num) * 1
    # sigmoid인 경우 활성화값이 0과 1에 밀집되어 기울기 소실
# w = np.random.randn(node_num, node_num) * 0.01
    # sigmoid인 경우 활성화값이 0.5에 밀집되어 표현력 제한
# w = np.random.randn(node_num, node_num) * np.sqrt(1.0 / node_num)
    # Xavier 초깃값 : sigmoid, tanh
w = np.random.randn(node_num, node_num) * np.sqrt(2.0 / node_num)
    # He 초깃값 : ReLU

a = np.dot(x, w)

# 활성화 함수 설정
# z = sigmoid(a)
z = ReLU(a)
# z = tanh(a)
    # Xavier 초깃값에서 sigmoid인 경우 층이 깊어지면서 활성화값의 분포가 일그러지는 현상을 보완
```



ReLU 함수 + 가중치 초기화 실험 ⌥ 학습 효율 비교 그래프

```
for key in weight_init_types.keys():
    grads = networks[key].gradient(x_batch, t_batch)
    optimizer.update(networks[key].params, grads)
    loss = networks[key].loss(x_batch, t_batch)
    train_loss[key].append(loss)
```

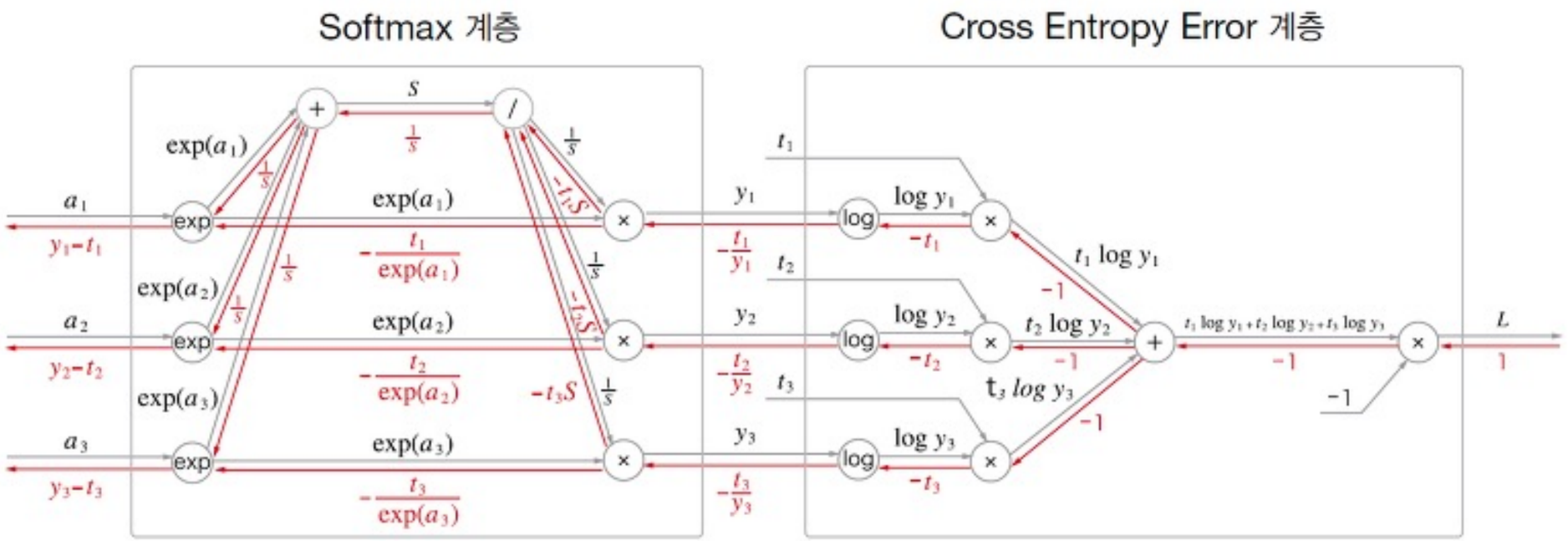
```
# 학습 진도 비교 그래프
markers = {'std=0.01': 'o', 'Xavier': 'x', 'He': '^'}
x = np.arange(max_iterations)
plt.figure(figsize=(15, 5))
for key in weight_init_types.keys():
    plt.plot(x, smooth_curve(train_loss[key]),
             marker=markers[key], markevery=100, label=key)
plt.xlabel("iterations")
plt.ylabel("loss")
plt.ylim(0, 2.5)
plt.legend()
plt.show()
# 가중치 초깃값의 표준편차를 0.01로 한 경우, 순전파의 활성화값이 대부분 0 근처로 밀집
#    ㄴ 역전파의 기울기가 작아지면서 가중치 갱신(학습)이 되지 않음
# Xavier, He 초깃값을 사용한 경우, 학습이 순조롭게 진행
# 활성화 함수가 ReLU이므로 He 초깃값에서 보다 효율적으로 학습
```



오차 역전파

목적함수에 대한 매개변수의 미분 계산을 위해, chain rule에 의해 편미분을 곱하여 오차를 역류

계산 그래프 ⌥ 출력층 역전파 예시(분류)



출력층에서 출력값에 대한 손실함수의 편미분

⌘ 분류 : Y-T = (yi-ti)

⌘ 회귀 : y-t

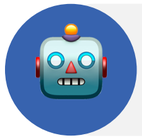
크로스엔트로피의 정교한 수식 설계

Check Point!

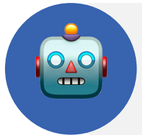
✓ 오차 역전파의 직관적 이해 참고 영상 : 3Blue 1Brown - What is backpropagation really doing?

최적화
Optimization

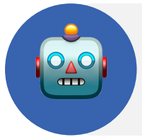
최적화가 뭐야? 

 손실 함수의 값이 최소가 되게 하는 매개변수를 결정하는 과정.

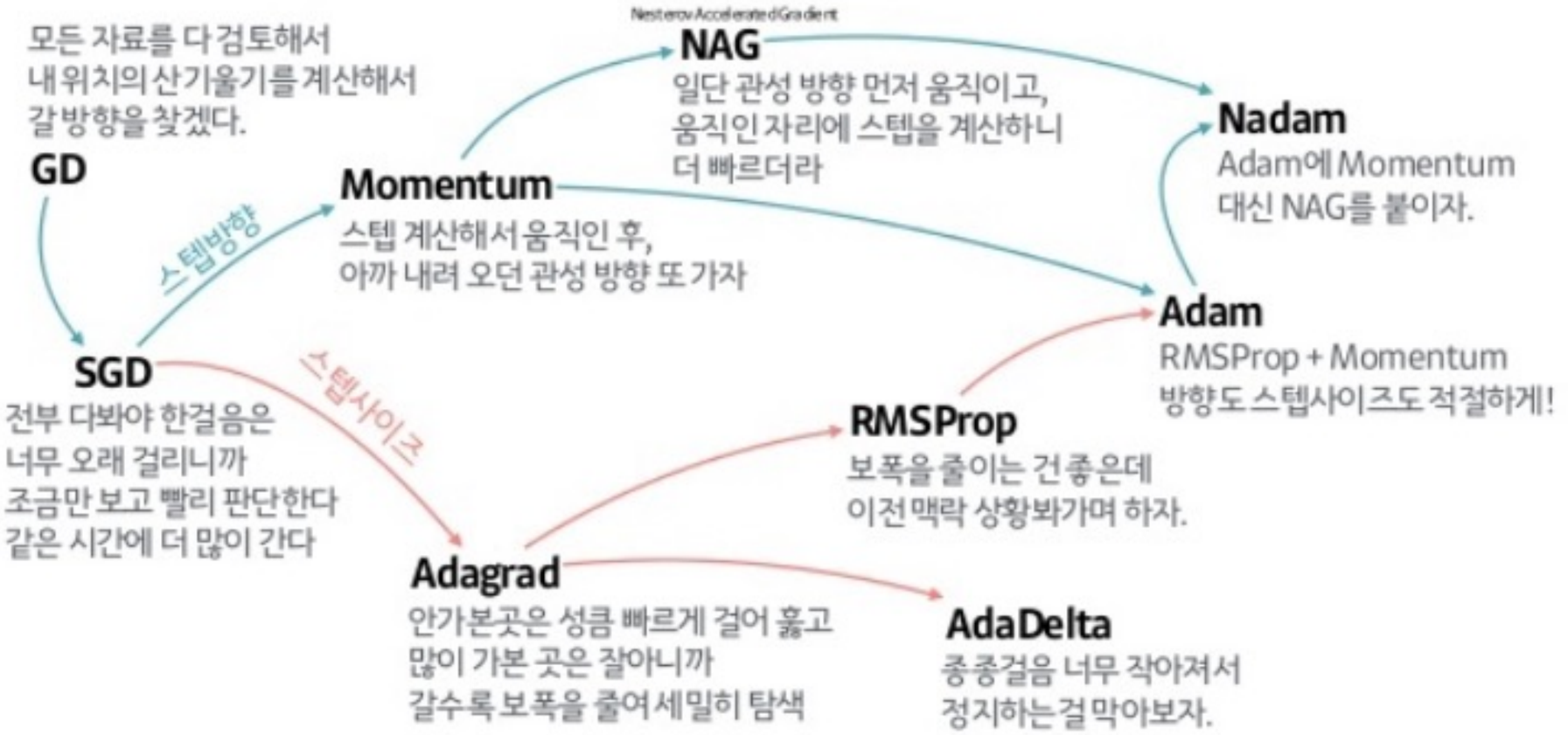
그럼 함수 미분해서 0인 것만 비교하면 되는 거 아냐? 

 근데 그 함수가 변수가 각각 수백 개 정도 되는 선형 및 비선형 함수 여러 개의 합성 함수라서 도함수를 구하는 게 어려워.

그럴 땐 오차 역전파로 경사 하강해서 최소로 근사하면 돼! 

 그 경사 하강에서 주어진 미분을 활용하는 알고리즘의 종류에 따라 최적화의 결과나 효율이 다를 수 있어.

최적화 알고리즘 ; Optimizer



경사 하강법의 기본 원리

SGD

수식이 직관적이고 이해하기 쉬움
→ 업데이트 방법이 일차원적임

손실 함수가 기울기에 대해 비등방성
→ 최적화 경로가 비효율적임

Batch Gradient Descent

$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta)$$

- train set의 모든 데이터 이용(타당성 확보) → 1회 업데이트
- 속도가 느리고 메모리 소모가 큼

Stochastic Gradient Descent

$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta; x^{(i)}, y^{(i)})$$

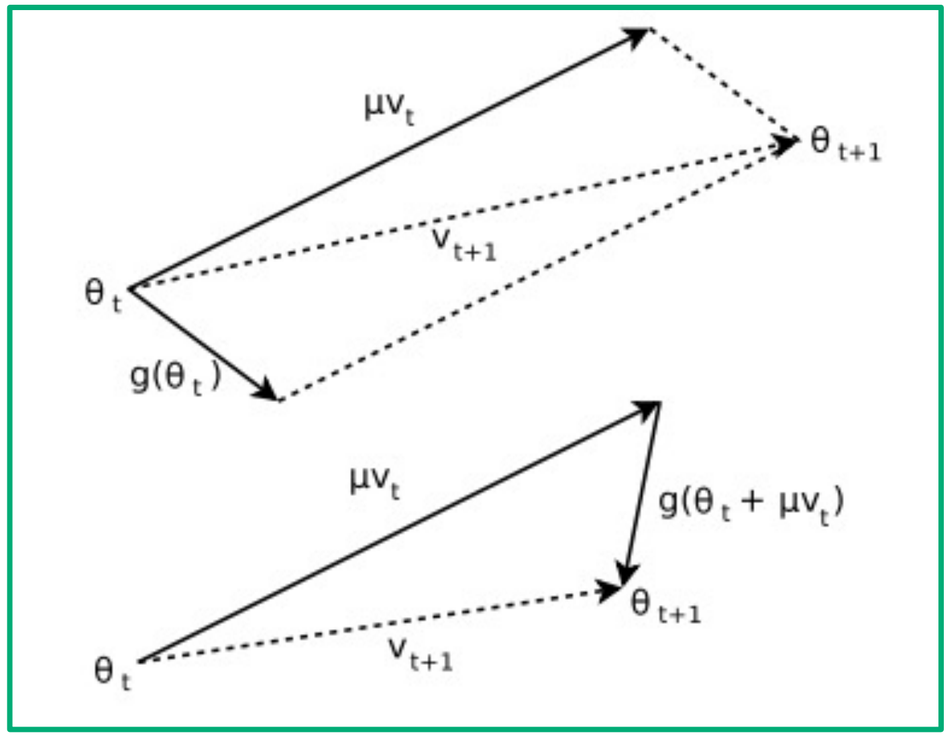
- train set에서 하나의 샘플 데이터 이용(논리적 비약) → 1회 업데이트
- 속도가 빠르고 메모리 소모가 적음

Mini-batch Gradient Descent

$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta; x^{(i:i+n)}, y^{(i:i+n)})$$

- train set의 부분집합 이용(유연성 확보) → 1회 업데이트
- 일반적으로 배치(train set의 부분집합) 크기 n은 64~256으로 설정 ; task별 상이함
- 흔히 사용되는 SGD는 대부분 미니 배치 학습에 해당

Momentum NAG



(Top) Classical Momentum
(Bottom) Nesterov Accelerated Gradient

Momentum

$$v_t = \mu v_{t-1} - \eta \cdot \nabla_\theta J(\theta)$$
$$\theta = \theta + v_t$$

- 이전 스텝의 관성 방향에 관한 항 추가 → SGD의 진동 완화, 기울기 방향 가속화
- gradient가 큰 흐름의 방향으로 갱신을 유도하여 성능 개선
- μ는 보통 0.9 주변 값 사용

Nesterov Accelerated Gradient (NAG)

$$v_t = \mu v_{t-1} - \eta \cdot \nabla_\theta J(\theta + \mu v_{t-1})$$
$$\theta = \theta + v_t$$

- 현재 매개변수를 μv_{t-1}만큼 이동했을 때의 gradient 계산 → 속도 항 v_t 업데이트
- 평균적으로 기존 momentum보다 안정적이고 빠름

Adagrad

Adaptive Gradient (Adagrad)

$$g_{t,i} = \nabla_{\theta_t} J(\theta_{t,i})$$
$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{G_{t,ii}+\epsilon}} \cdot g_{t,i} \quad \text{단, } G_{t,ii} = \Sigma_{i=1}^{t-1} g_{t,i}^2$$

- 매개변수별로 적응적인 학습률 조정 ; 각 매개변수의 갱신량에 반비례하는 학습률 부여
- 초기 학습률 default 값은 보통 0.01 → 튜닝 부담 완화
- 학습이 진행되면서 분모 부분 증가 → 매개변수가 더 이상 갱신되지 않는 문제 → Adadelta, RMSProp으로 개선

RMSProp

Root Mean Sqaure Propagation (RMSProp)

$$E[g^2]_{t,i} = \gamma E[g^2]_{t-1,i} + (1-\gamma) g_{t,i}^2$$
$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{E[g^2]_{t,i}+\epsilon}} \cdot g_{t,i}$$

- 보통 $\gamma = 0.9$, $\eta = 0.001$로 default 값 설정
- 학습률 조정에서 최근 gradient를 많이 반영하도록 지수이동평균 사용

Adam

Adaptive Moment Estimation (Adam)

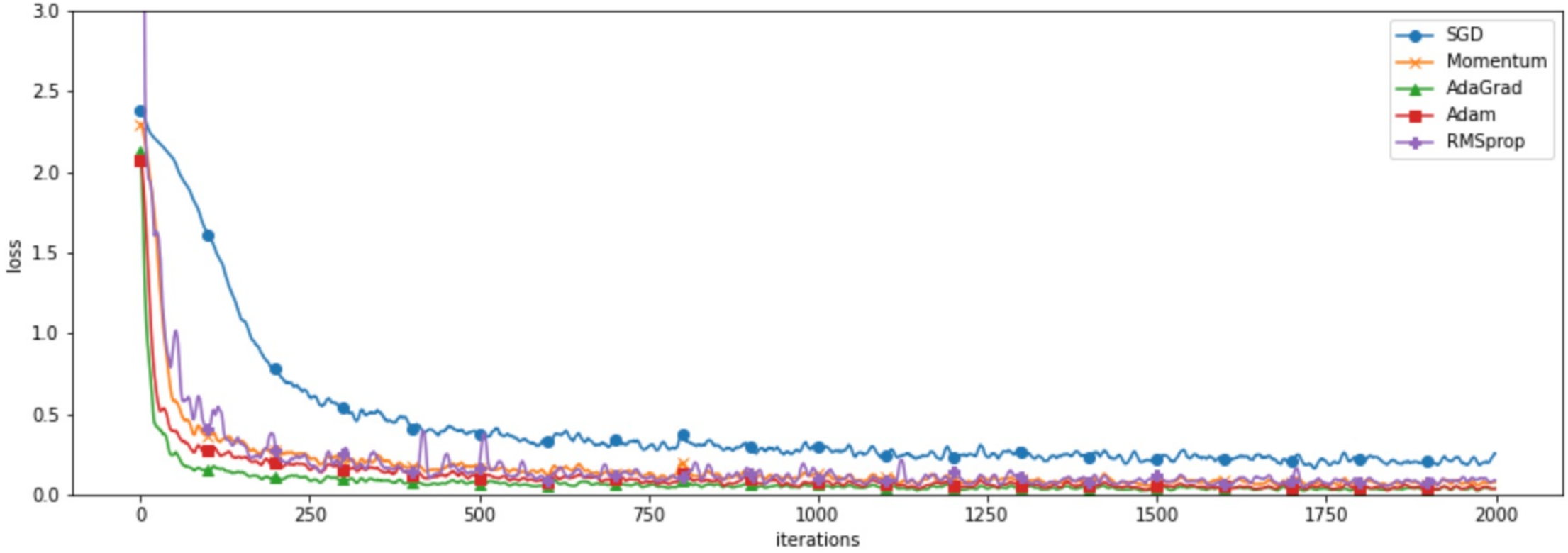
$$
\begin{aligned}
m_t &= \beta_1 m_{t-1} + (1-\beta_1) g_t \\
v_t &= \beta_2 v_{t-1} + (1-\beta_2) g_t^2 \\
\hat{m}_t &= \frac{m_t}{1-\beta_1^t} \\
\hat{v}_t &= \frac{v_t}{1-\beta_2^t} \\
\theta_{t+1} &= \theta_t - \frac{\eta}{\sqrt{\hat{v}_t}+\epsilon}\hat{m}_t
\end{aligned}
$$

- 보통 $\beta_1 = 0.9,\ \beta_2 = 0.999$로 default 값 설정
- gradient의 1차, 2차 적률 추정(및 편향 보정)하여 매개변수 업데이트에 사용
- 최근 gradient를 많이 반영하도록 지수이동평균 사용
- 대용량 고차원 데이터셋에서도 간단한 계산 → 효율 증대
- 메모리 소모가 적음
- non-convex 함수에서 최적화 성능 개선

Adam with decoupled weight decay (AdamW) : adamw.pdf / adamw.review

✱ 전반적으로 SGD에 비해 Momentum, AdaGrad, Adam의 학습 효율이 높음

MNIST 데이터셋에서 optimizer별 학습 진도 비교



하이퍼파라미터 튜닝

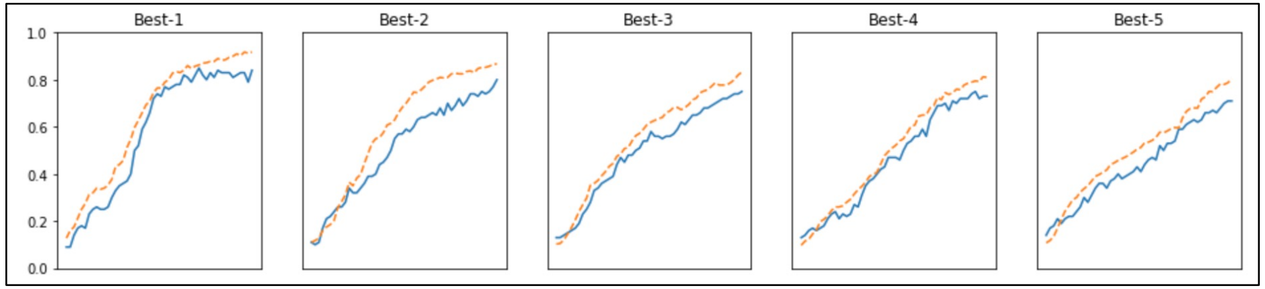
하이퍼파라미터 : 뉴런 개수, 배치 크기, 학습률, 가중치 감소의 규제 계수 등

- 검증 데이터를 통해 평가 및 조정

★ 하이퍼파라미터의 최적 값에 대한 범위를 좁혀가며 탐색

1. 하이퍼파라미터 값의 범위 설정 및 랜덤 추출
2. 추출된 하이퍼파라미터로 비교적 적은 epoch을 학습 후, 검증 데이터로 평가
3. 위 두 단계의 반복을 통해 범위 축소

- **베이즈 최적화** : 베이즈 정리 기반의 수학적 이론을 통한 하이퍼파라미터 최적화




```
=========== Hyper-Parameter Optimization Result ===========
Best-1 (val acc : 0.84) | lr : 0.009596486265083503, weight decay : 4.9002230847467255e-08
Best-2 (val acc : 0.8) | lr : 0.007697602267987423, weight decay : 1.1143736531746786e-08
Best-3 (val acc : 0.75) | lr : 0.00597486744835416, weight decay : 1.5110328199542916e-08
Best-4 (val acc : 0.73) | lr : 0.007485553832217598, weight decay : 6.298690881518333e-05
Best-5 (val acc : 0.71) | lr : 0.005539736351995695, weight decay : 2.0600111036864066e-05
```

THANK YOU