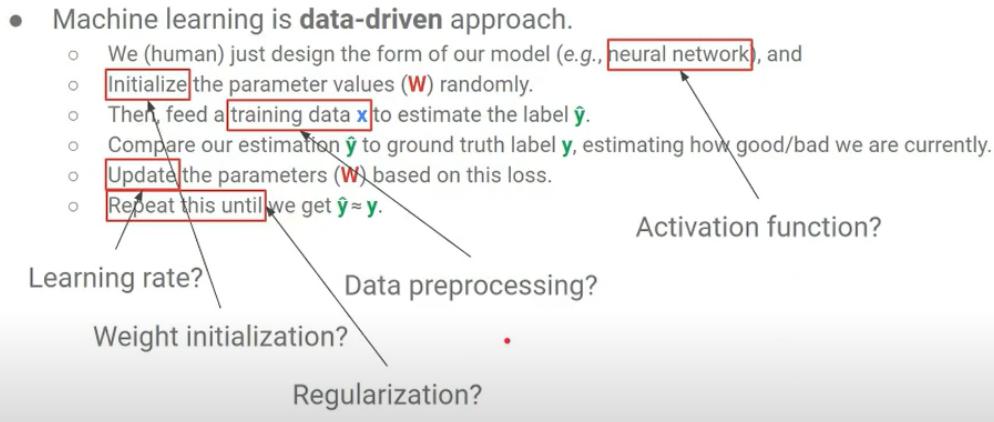


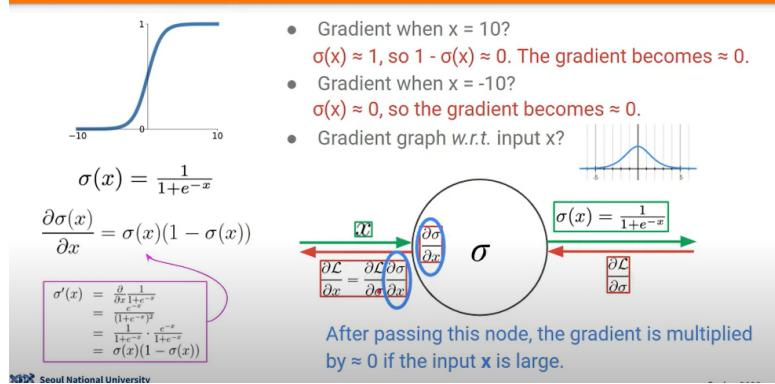
Lecture #6: Training Neural Networks



- Activation Functions

- sigmoid(초창기에 많이 쓰임)**
 - 문제점
 - nuerons kill the gradients
 - input값이 조금만 커져도 local gradients의 값이 0에 수렴해져서 gradient값이 죽어버림

Sigmoid Function: Killed Gradients



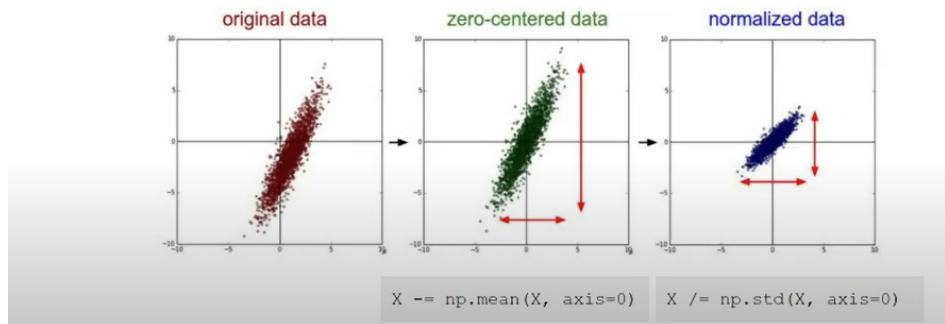
- outputs are not zero-centered[그렇게 큰 문제는 아님]
 - all gradient elements are either all-positive or all -negative → gradient update can go only to particular directions!
- $\exp()$ is computationally expensive

- Tanh Function
 - zero-centered
 - kill the gradients → 기울기→0
- ReLU(Rectified Linear Unit)
 - doesn't kill the gradients→기울기가 0으로가진 않음
 - doesn't saturate(수렴하지 않음)
 - 계산 효율적
 - 기울기가 단순하므로
 - but not zero-centered/ 0일 때 미분이 안됨
 - Dead ReLU problem
 - $x < 0$: 무조건 죽어버림(값이=0→ it's never updated)
 - 초기화 무작위 안됨 → slightly positive biases로 시작
- Leaky Relu
 - $\max(0.01x, x)$
 - 앞에 0.01이라는 값을 정해야 하는 overhead가 있음
- ELU(Exponential Linear Unit)
 - $x(x \geq 0), a(e^x - 1)(x < 0)$

Data Processing

- **zero-centering & normalization**
 -

- Recall what happens when all inputs are positive.
- It may help to **zero-centering** the data by subtracting global mean.
- It is also common to **normalize** the data by dividing by standard deviation.



원래 데이터 (original data):

- x축과 y축 데이터를 보면, 데이터 포인트들이 오른쪽 위로 치우쳐 있죠? → 이는 데이터의 평균값(중심점)이 원점(0, 0)이 아닌 어딘가 멀리 위치한다는 뜻입니다.
 - 예: x값의 평균 = 5, y값의 평균 = 7

2. Zero-centering 과정:

- 데이터를 "중심"으로 맞추기 위해 각 데이터 포인트에서 **평균값을 빼줍니다**.
 - 새로운 x값 = 원래 x값 - x값 평균(5)
 - 새로운 y값 = 원래 y값 - y값 평균(7)
- 결과적으로, 데이터 포인트들이 원점(0, 0)을 중심으로 배치됩니다.

3. 변환된 데이터 (zero-centered data):

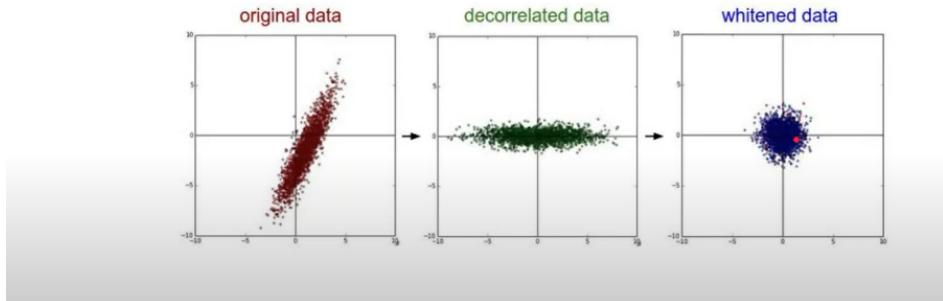
- 이제 데이터의 평균은 0이 되고, 데이터들이 x축과 y축의 중심(원점)에 더 가깝게 분포하게 됩니다.
- normalization
 - 표준편차로 나눠서 실제 영향력으로 다시 맞춰줌

- **PCA & Whitening**

-

PCA & Whitening

- Data becomes **zero-centered AND axis-aligned**.
- With whitening, the covariance matrix becomes identity matrix.
 - Each axis has the same importance.



•

1. PCA (Principal Component Analysis, 주성분 분석)

PCA는 데이터를 압축하거나 중요한 축을 기준으로 정렬하여 차원을 줄이거나 데이터의 구조를 더 잘 이해할 수 있게 만드는 기법입니다.

★★★ 벡터의 크기를 줄이면서 줄어드는 만큼 정보도 줄어들텐데 그 줄어드는 정보량을 최소화하는 것 → 줄어드는 정보량: 공간상에 분포돼 있는데 분산이 가장 큰 축을 첫번째 축으로 잡고 여기에 수직인 축을 그다음 분산이 큰 축으로 잡고... 이렇게 반복

PCA의 주요 단계

1. Zero-centering (평균을 0으로 맞추기):

- 데이터를 원점(0,0) 주변으로 재배치합니다. 이 단계는 PCA를 적용하기 위해 필수입니다.

2. 축 변경 (Decorrelation):

- 데이터를 새로운 축으로 변환하여, 각 축(변수)이 서로 독립적이게 만듭니다.
- 예: 원래 데이터의 x축과 y축이 서로 상관관계가 있다면, PCA는 새로운 축(주성분)으로 변경하여 상관관계를 없앱니다.
- 이미지에서 "decorrelated data"가 바로 이 상태입니다.

3. 데이터 압축:

- 변환된 축 중에서 가장 중요한 축(분산이 큰 축)만 선택하여 데이터를 줄입니다.
- 예를 들어, 2차원 데이터를 1차원으로 줄여도 큰 정보 손실이 없게 만듭니다.

•

Principal Component Analysis (PCA)

U is a kind of **rotation** matrix; the covariance matrix Σ is rotated to be axis-aligned. Then, the covariance matrix becomes **diagonal**, meaning that each dimension is **no longer correlated**. Each diagonal entry λ_i indicates the **variance** of data points according to the i -th axis.

$$U^\top \hat{\Sigma} U = \begin{bmatrix} \lambda_1 & 0 & 0 & 0 \\ 0 & \lambda_2 & 0 & 0 \\ 0 & 0 & \ddots & 0 \\ 0 & 0 & 0 & \lambda_d \end{bmatrix}$$

4. If we **choose the first $k < d$ eigenvectors** (with largest k eigenvalues) and discard the rest, we get a k -dimensional space such that the original data loses least amount of information (in terms of variance).

PCA is one of the most widely used dimension reduction technique in ML!

2. Whitening

Whitening은 PCA를 더 정교하게 변환한 것으로, 데이터를 정규화하여 모든 축이 동일한 중요도를 가지도록 만듭니다.

Whitening 과정

1. PCA 적용:

데이터를 새로운 독립된 축으로 변환합니다.

2. 스케일 조정:

각 축의 **분산(variance)을 동일하게 만들어** 모든 축이 같은 중요도를 가지게 합니다.

- 분산 = 1로 맞추는 작업을 합니다.

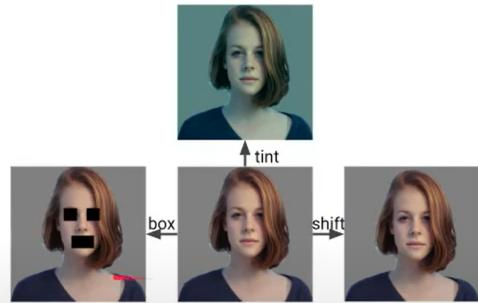
결과

- Covariance matrix(공분산 행렬)가 단위 행렬(identity matrix)이 됩니다.
 - 이는 모든 축이 독립적이고, 동일한 크기(분산)를 가진다는 것을 의미합니다.
- 이미지에서 "whitened data"처럼 모든 축이 고르게 분포된 상태가 됩니다.

Data Augmentation

Data Augmentation

- A dataset we have is just tiny subsamples of the reality.
- There are lots of ways to **slightly modifying** each datum **without affecting its semantics**.
- We want our classifier to be **invariant** to these slight modifications.
- More healthy way may be having a larger dataset, but it is costly.
- Instead, we'd like to **take full advantage of existing ones**.



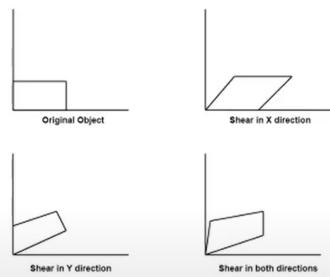
→ 한마디로 말하면, ★★★★ pixel level variation은 일으키되 semantic 변화는 일으키지 않음으로써 데이터를 최대한 확장/증강 시키는 것!

- 평행이동을 해도 여전히 원본 이미지와 같다고 판단돼야 한다.
- scaling: reasonable size → 여전히 원본 이미지와 같다고 판단해야 한다.
- HSL

Data Augmentation in Practice

There is no single answer. It depends on the problem and data domain. **Be creative!**

- Translation
- Rotation
- Stretching
- Shearing
- Random blocking
- Add noise
- Mix two images
- Apply a filter
- ...



Weight Initialization

- 초기값을 잘못 설정했을 때

◦

1) 초기값을 모두 0으로 설정한 경우

만약 데이터를 평균 0정도로 정규화시킨다면, 가중치를 0으로 초기화 시킨다는 생각은 꽤 합리적으로 보일 수 있다. 그러나 실제로 0으로 가중치를 초기화 한다면 모든 뉴런들이 같은 값을 나타낼 것이고, 역전파 과정에서 각 가중치의 update가 동일하게 이루어질 것이다. 이러한 update는 학습을 진행 해도 계속해서 발생할 것이며, 결국 제대로 학습하기 어려울 것이다. 또한 이러한 동일한 update는 여러 층으로 나누는 의미를 상쇄시킨다.

- Xavier Initialization

- $W = np.random.randn(d_{in}, d_{out}) / np.sqrt(d_{in})$

-

$$W = np.random.randn(d_{in}, d_{out}) / np.sqrt(d_{in})$$

$$x \in \mathbb{R}^{d_{in}}, W \in \mathbb{R}^{d_{out} \times d_{in}}, y = Wx \in \mathbb{R}^{d_{out}}$$

$$y_i = W_{i\cdot}^T x = \sum_{j=1}^{d_{in}} W_{ij} x_j$$

$$\text{Var}[y_i] = \text{Var} \left[\sum_{j=1}^{d_{in}} W_{ij} x_j \right] = d_{in} \text{Var}[W_{i\cdot} x]$$

↑
독립.

$$= d_{in} (E[W_{ij}^2 x_j^2] - E[W_{ij} x_j]^2)$$

$$= d_{in} \left(E[W_{ij}] E[x_j] - E[W_{ij}]^2 E[x_j]^2 \right)$$

$$\text{Var}[y_i] = d_{in} \left(\text{Var}[W_{ij}] \text{Var}[x_j] \right)$$

↓
if we set $\text{Var}[W_{ij}] = \frac{1}{d_{in}}$,

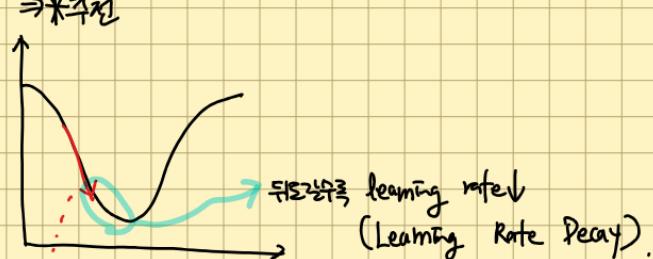
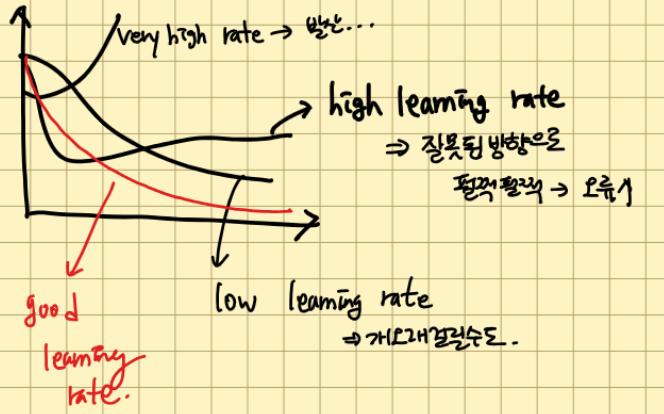
$$\text{Var}(y_i) = \text{Var}(x_j)$$

$$\begin{aligned} E[W] &= 0 \\ \text{Var}[x] &= E[X^2] - E[X]^2 \\ \text{Var}[x] &= E[X^2] \\ \text{Var}[W_{ij}] &= E[W_{ij}^2] \end{aligned}$$

Learning Rate

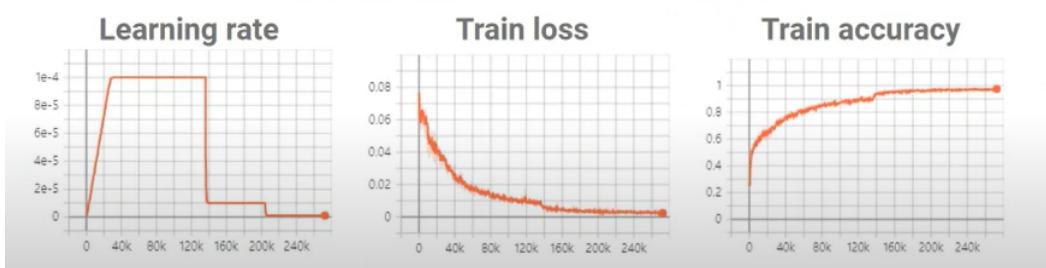
Update the parameters(W) based on this
loss. → Optimization/Batch Norm/ Transfer Learning

* Learning Rate Scheduling (학습률 조정) (몇 번째로 갈지...?)

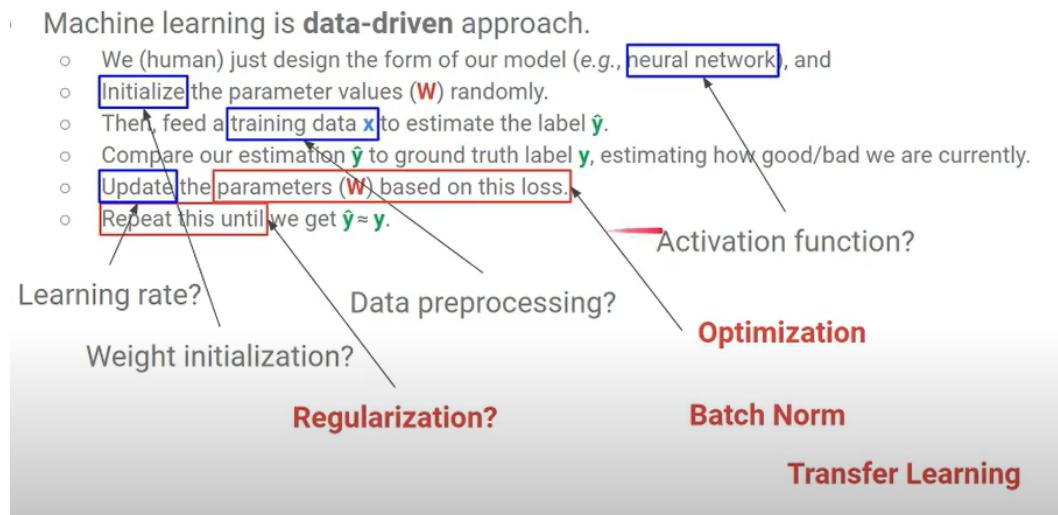


초반: 빨리
내려가도
그리다가
에너지도 날려으면
↑
촘촘히 차야한다.

global (coarse)
structure

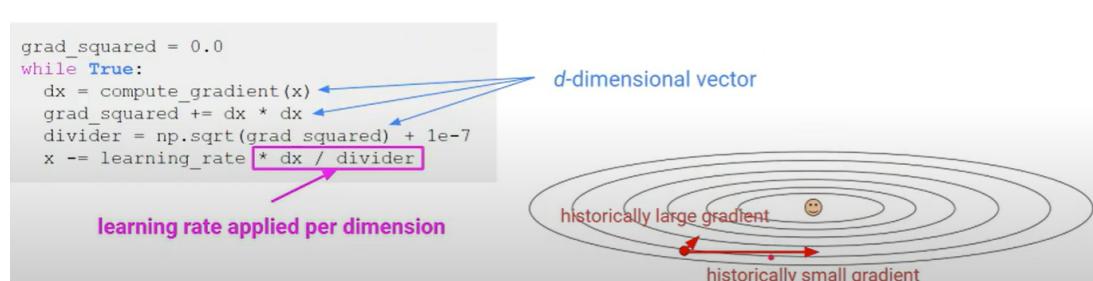


learning rate를 한 가지 값으로 주는 것이 아니라 이렇게 여러가지로 주면 loss를 줄이고 accuracy를 높일 수 있다.



Optimization beyond SGD

- SGD의 문제점
 - Jittering
 - 축의 한쪽이 완만하고 다른 한쪽이 가파르면 비효율적임
 - saddle point
 - 미분==0 → update가 안됨
 - mini-batch가 너무 작으면 전체 dataset을 대표할 수 없다
- 대안
 - AdaGrad



divider: 지금까지의 역사(경사가 그만큼 가파랐다면 갱률 더 예의주시)

- RMS Prob: Leaky AdaGrad

```

grad_squared = 0.0
while True:
    dx = compute_gradient(x)
    grad_squared = dr * grad_squared + (1 - dr) * dx * dx
    divider = np.sqrt(grad_squared) + 1e-7
    x -= learning_rate * dx / divider

```

decay rate

가장 최근 꺼를 반영을 많이 하고 예전꺼는 덜 반영한다.

- Adam(Adaptive Momentum)
- RMS Prop + SGD with Momentum

```

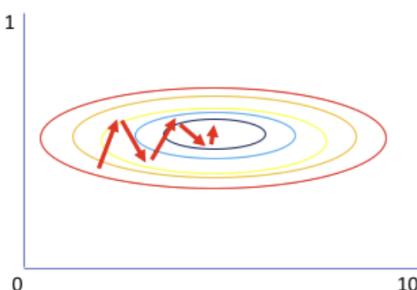
first_moment = 0.0
second_moment = 0.0
while True:
    dx = compute_gradient(x)
    first_moment = betal * first_moment + (1 - betal) * dx
    second_moment = beta2 * second_moment + (1 - beta2) * dx * dx
    divider = np.sqrt(second_moment) + 1e-7
    x -= learning_rate * first_moment / divider

```

momentum

AdaGrad
RMSProp

Batch Normalization



Gradient of larger parameter
dominates the update

데이터 값의 편차가 아주 클 경우, 편차가 큰 feature를 중심으로 학습하게 되므로 작은 애들은 무시되는 경향이 있음 → 정규화 필요

Batch: 데이터 묶음

batch마다 입력값의 범위를 scaling하는 것 = batch 정규화

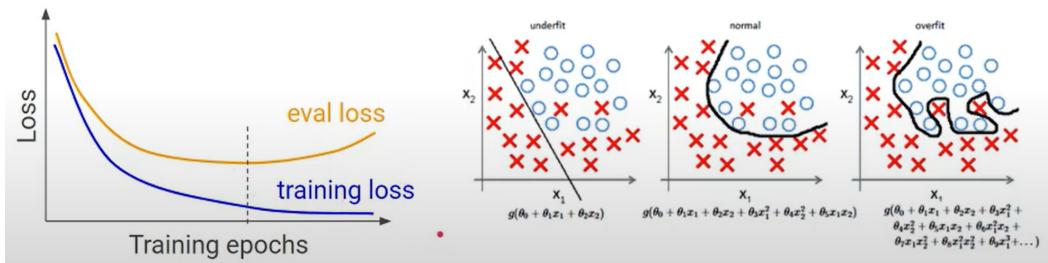
$$\hat{x}^{(k)} = \frac{x^{(k)} - \mathbb{E}[x^{(k)}]}{\sqrt{\text{Var}[x^{(k)}]}} \quad y^{(k)} = \gamma^{(k)} \hat{x}^{(k)} + \beta^{(k)}$$

왼쪽: 정규화/ 오른쪽: 원상 복구/ 감마, 베타: 배우는 값. 왼쪽값을 고정시켜놓고 오른쪽이 배우는 것

Regularization

optimization을 과하게 하다보면 overfitting이 일어날 수 있음. 이걸 방지하는 것

Repeat this until error→0



- **🚫 underfitting**
- **🚫 overfitting**
 - training loss는 계속 줄어드는데 test loss[eval loss]가 계속 올라가는 것
 - 즉, training set의 일반적인 패턴은 다 배웠는데 training set에서 '만' 나타나는 개별적인 고유의 패턴까지 (안 배워도 되는 부분까지) 배우기 시작했다는 뜻.
- **Regularization**
 - overfitting 방지
 - additional penalty term 을 주는 것!
 - 불필요한 weight에는 0이나 작은 값을 주는 것
 - 즉, 우리가 표현하는 세상(모델)은 그렇게 복잡하지 않을 것이다.의 철학을 반영한 것!

- 종류
 - Regularization for Machine Learning
 - Ridge Regression
 - norm 2개 씀
 - Lasso Regression
 - norm 1개 씀
 - 회귀계수가 0이 될 수 있으므로 변수 선택을 통해 더 간단한 모델 만들기가 가능
 - 덜 중요한 특징의 계수를 0으로 만들어서 overfitting을 방지한다.
 - Sparser representation of theta를 encourages!
 - Regularization for Neural networks
 - weight decay
 - ridge, lasso와 유사
 - Dropout
 - 어떤 뉴런들을 0으로 만드는 것
 - ex. 고양이의 특정 부분만 학습시키는 것이 아니라 전체를!
 - 특정 부분만 강하게 학습되면 이후에 그 특정 부분이 가려졌을 때 문제가 생김
 - p값이 확률상 0.5면, 0.5보다 확률이 작은 뉴런들을 2배만큼 곱해줘서 개체들의 중요도를 높인다. 단, 값이 0이면 당연히 0이 곱해지니깐 0으로...

Dropout Implementation

```
import numpy as np

def train_step(data, p):
    # p = dropout rate
    h1 = np.maximum(0, np.dot(w1, data) + b1)
    u1 = (np.random.rand(*h1.shape) < p) / p
    h1 *= u1
    h2 = np.maximum(0, np.dot(w2, h1) + b2)
    u2 = (np.random.rand(*h2.shape) < p) / p
    h2 *= u2
    return np.dot(w3, h2) + b3
```

```
def test_step(data):
    h1 = np.maximum(0, np.dot(w1, data) + b1)
    h2 = np.maximum(0, np.dot(w2, h1) + b2)
    return np.dot(w3, h2) + b3
```

Can we make the inference independent from dropout?

Magnify activation by $1/p$
(e.g., 2x for 50%).

With TensorFlow, just use the existing Dropout layer.

```
import tensorflow as tf

def __init__(self, config, **kwargs):
    self.dropout = tf.keras.layers.Dropout(0.5)

def call(self, inputs, training=True):
    embeddings = self.dropout(embeddings)
```

- Cutout