Kuggle DL study

CS231n, Lecture7

정성준

INDEX

Activation function

Neural network

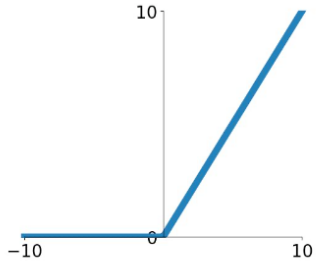
- Parameter: 학습가능한 숫자
- Activation: 계산 결과 숫자

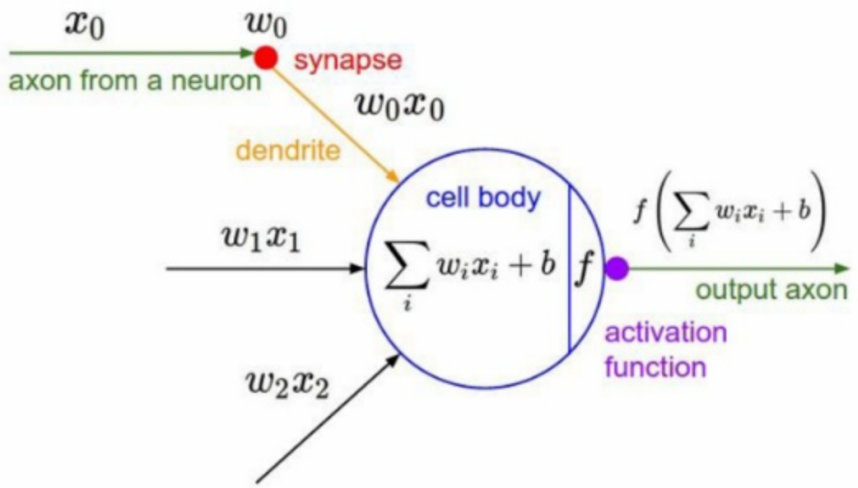
Activation에 적용하는 함수 -> activation function

-> 주로 **ReLU(max(0, x))**를 사용함.

ReLU

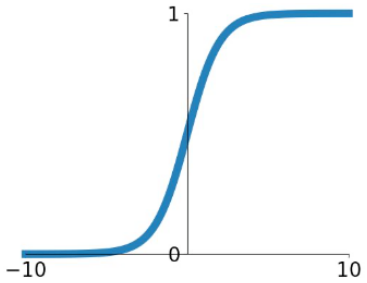
$$\max(0, x)$$





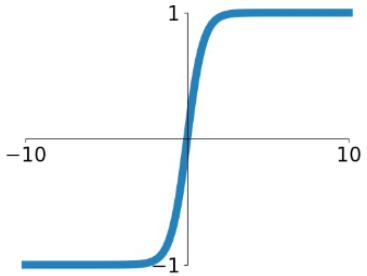
Sigmoid

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



tanh

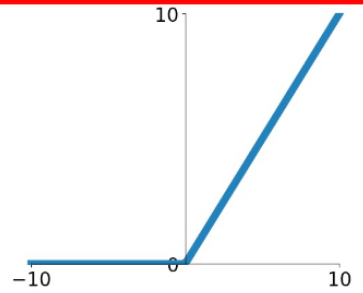
$$\tanh(x)$$



ReLU

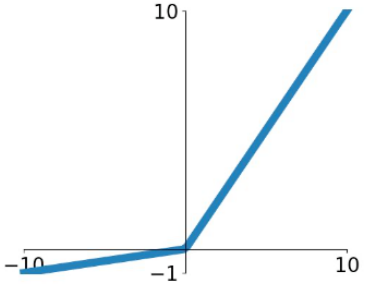
$$\max(0, x)$$

Good default choice



Leaky ReLU

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

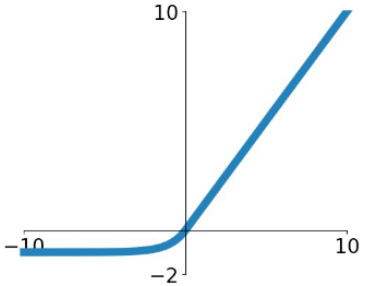


Maxout

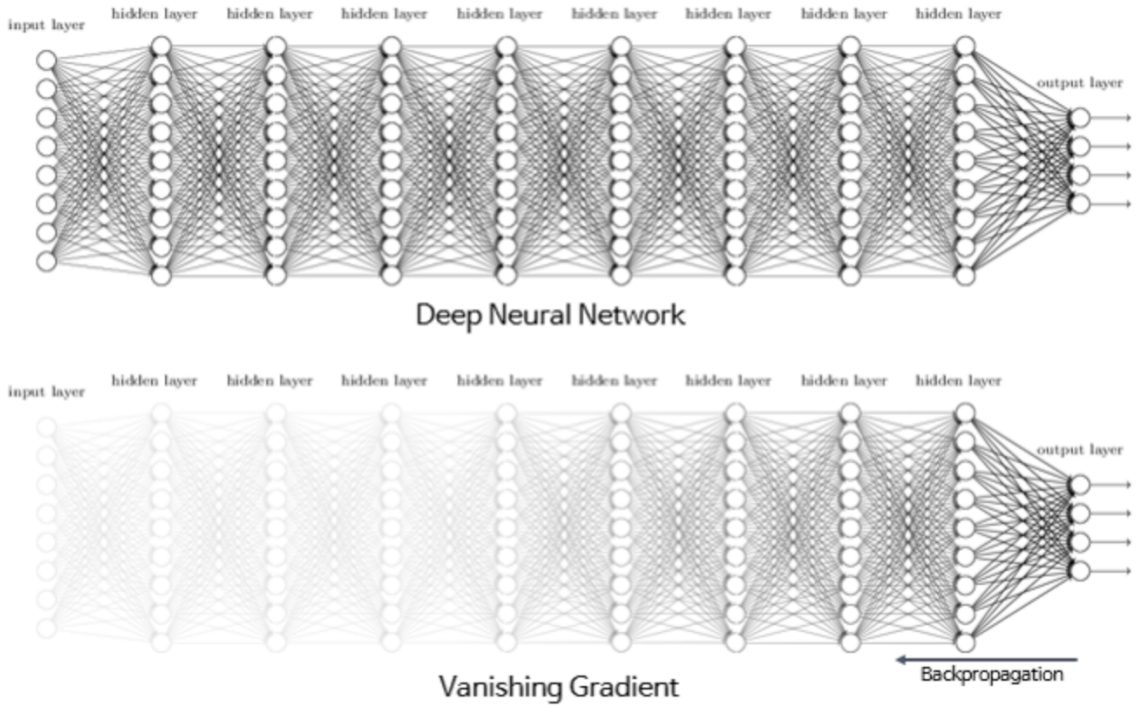
$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

ELU

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

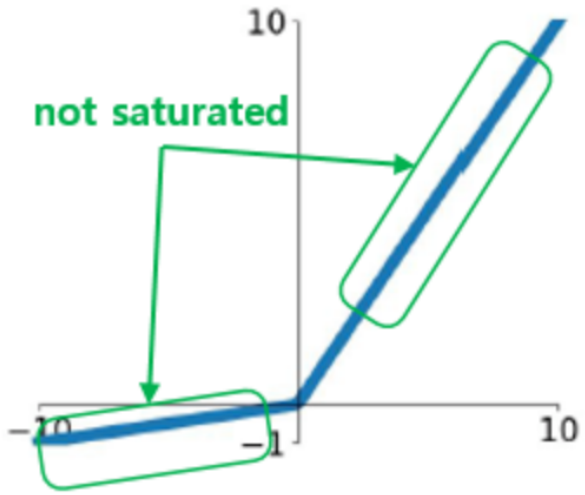


- 하지만 **ReLU**는 음수의 공간에서 gradient가 0이 되기 때문에 **dead ReLU**가 발생할 수 있음
- Gradient vanishing은 깊은 모델의 back prop과정에서 끝으로 갈수록 gradient가 소실되어 parameter가 업데이트되지 못하는 현상을 말함.



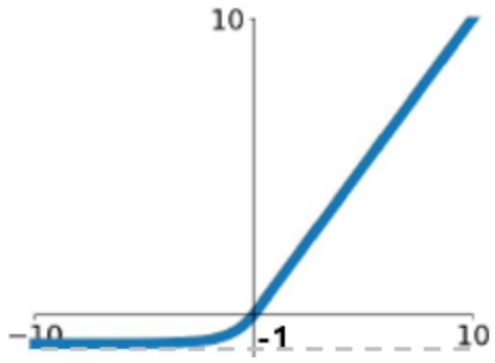
- Dead ReLU문제를 해결하기 위해 ReLU를 변형한 activation function들이 만들어졌음.
- LeakyReLU, PReLU(Parameteric ReLU), ELU 등이 그 종류임.

LeakyReLU



ELU

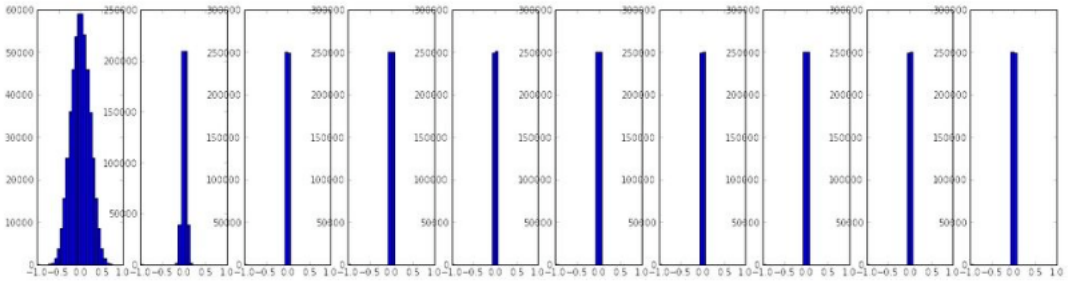
$$\mathrm{ELU}_\alpha = \begin{cases} \alpha\left(\exp\left(x\right) - 1\right) & \text{if } x < 0 \\ x & \text{if } x \geq 0 \end{cases}$$



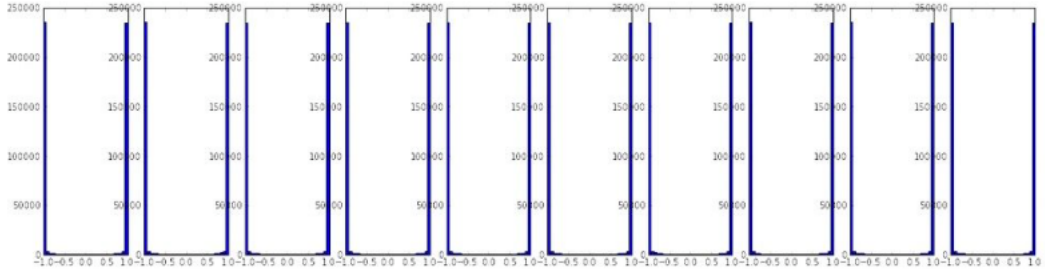
- 강의에서는 ReLU를 먼저 쓰되 주로 ReLU의 변형들을 사용하고 sigmoid를 사용하지 말라고 함.
- ReLU와 비슷한 함수들은 activation function의 변화가 성능에 중요한 영향을 미치지 않음.

Weight Initialization

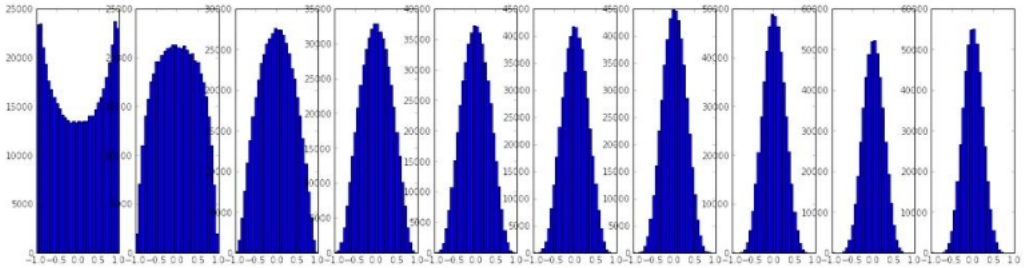
- **Initialization too small**
- 가중치(parameter)의 초기값을 0에 가깝게 매우 작게 초기화한 경우
  - Activation들이 0에 가까운 값을 가져 gradient도 0에 가까운 값을 가져 학습에 도움이 되지 않음
  - Activation이 가중치에 따라 다양한 값을 가져야 가중치 업데이트에 도움
- **Initialization too big**
  - Activation들이 큰 값을 갖게됨.
  - activation function을 통과할때 . Activations saturate 문제가 발생하여 gradientrk 0이 될 수 있음.
  - 따라서 학습이 일어나지 않게 됨
- **Initialization just right**
  - 각 층의 activation이 고른 분포를 가지게 되어 학습에 도움됨.



small



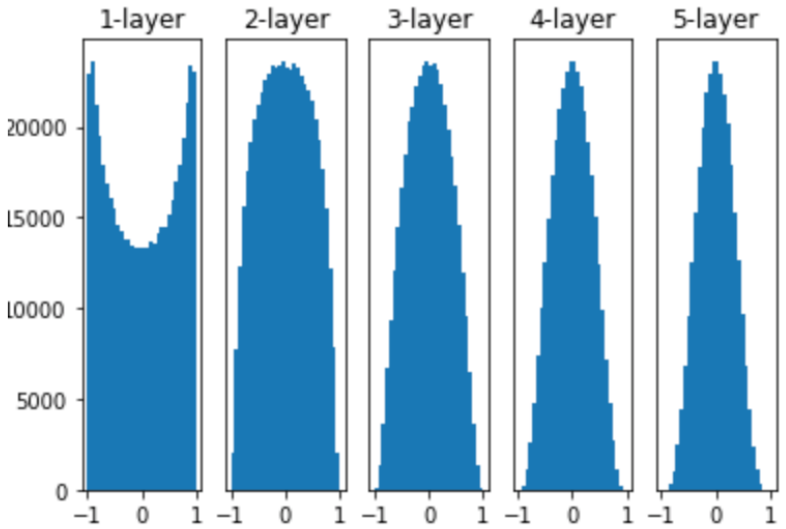
big



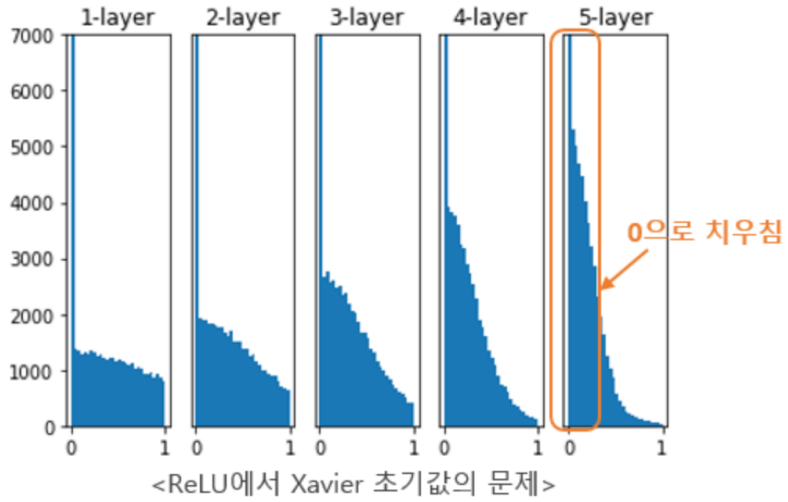
right

Weight Initialization

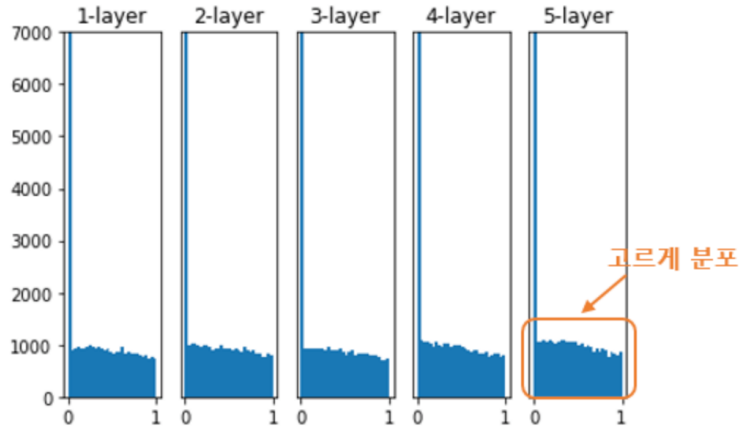
- Xavier initialization
  - 각 레이어의 입력의 분산과 출력의 분산을 같게 만들고,
  - 각 layer의 gradient분산을 같게 만들어 학습을 돕는다는 아이디어
  - 자세한 내용은 논문 참고 ..(Understanding the difficulty of training deep feedforward neural networks)
- He initialization
  - Xavier는 ReLU를 사용하면 activation이 0으로 치우치는 문제발생 가능.
  - ReLU에 더 적합한 parameter 초기화 방법
  - 자세한 내용은 논문 참고..(Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classifcation)



Tanh를 사용할 때 Xavier를 사용할 때 각 층의 activation분포

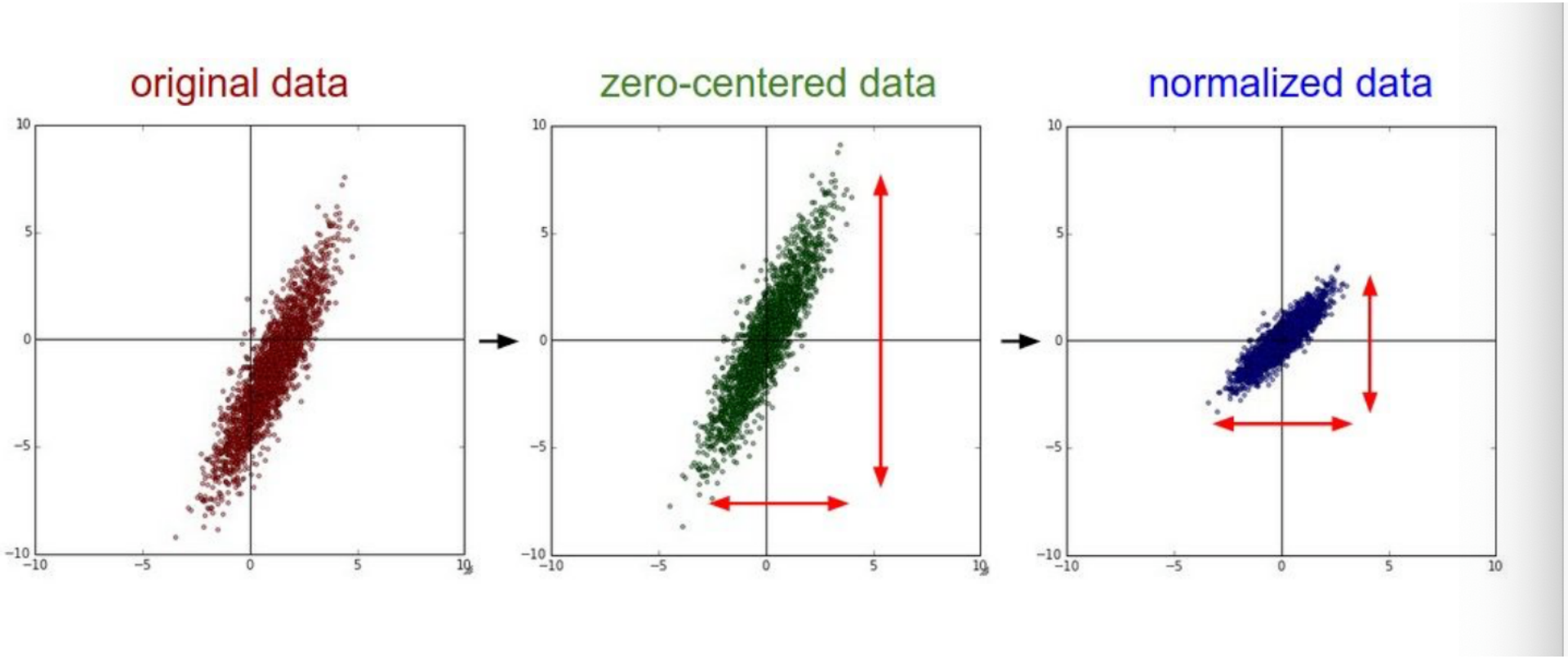


<ReLU에서 Xavier 초기값의 문제>

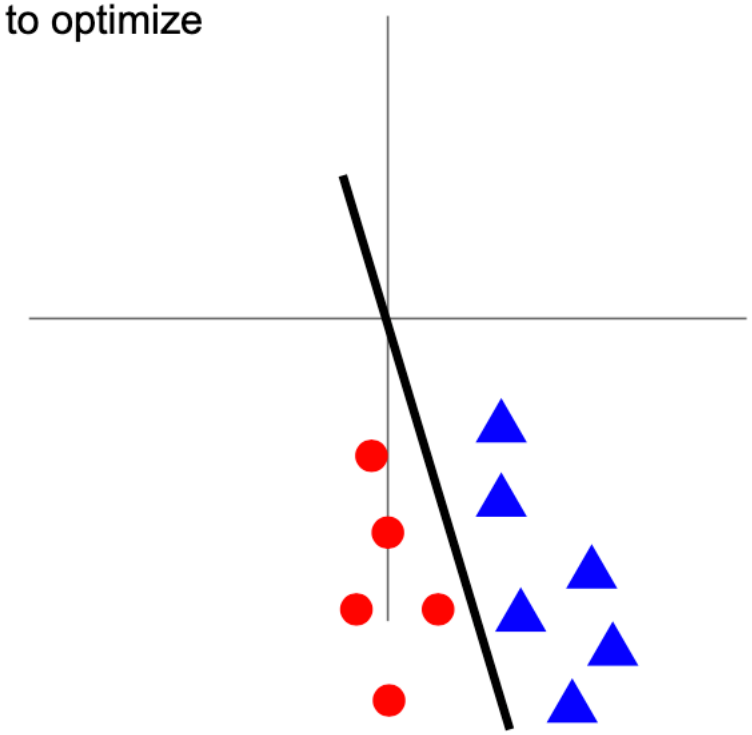


<ReLU에서 He 초기값 사용>

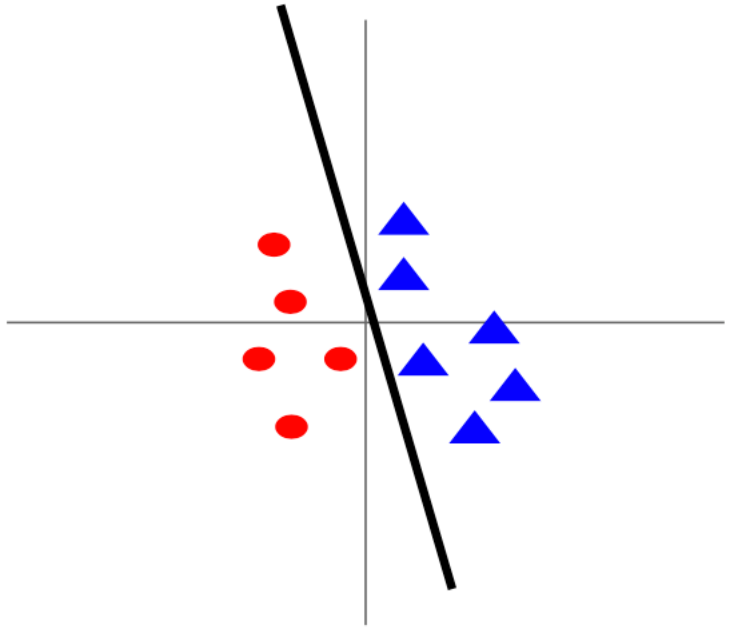
Data Preprocessing



Before normalization: classification loss very sensitive to changes in weight matrix; hard to optimize



After normalization: less sensitive to small changes in weights; easier to optimize



정규화를 통해 가중치의 최적화를 더 쉽게 할 수 있는 장점이 있음.

Batch Normalization

- Batch Normalization(https://arxiv.org/pdf/1502.03167.pdf)
- Batch normarlization accelerate training by reducing internal covariate shift.
- internal covariate shift는 연구자들이 정의한 개념이고 아래와 같음.
- We define Internal Covariate Shift as the change in the distribution of network activations due to the change in network parameters during training.
- 하지만 How Does Batch Normalization Help Optimization? (https://arxiv.org/pdf/1805.11604.pdf) 에서 internal covariate shift가 학습을 돕는 원인이 아니고 batch normalization이 loss의 분산을 줄이기 때문에 학습을 돕는다는 내용을 밝힘.

- 또 다른 주장은 batch-normalization의 gamma와 beta를 통해 수많은 parameter를 활용하지 않고 scale과 mean을 변경할 수 있기 때문에 학습이 빨라진다는 주장도 존재.
- 어쨌든 neural-net을 학습할 때 batch-normalization을 사용하면 학습이 용이할 수 있다.

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

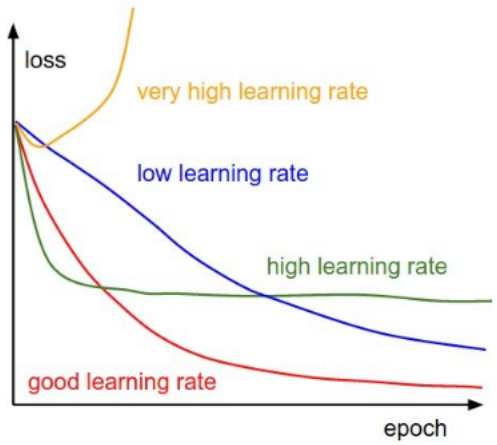
$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

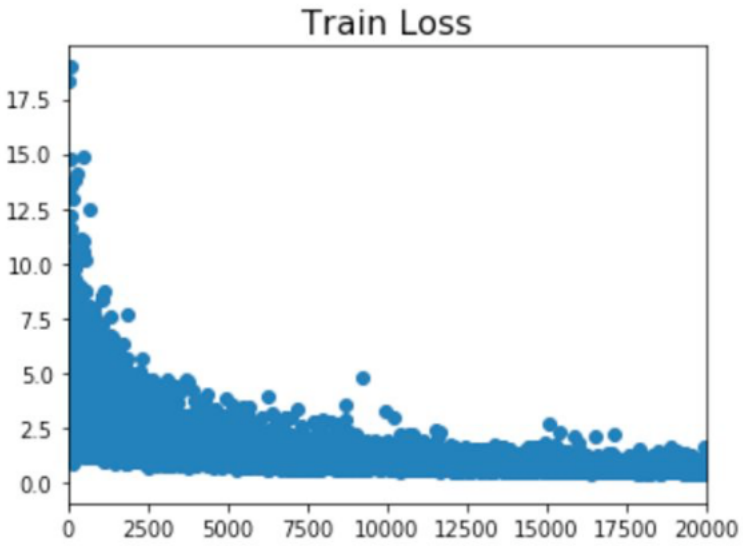
$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \qquad \text{// normalize}$$

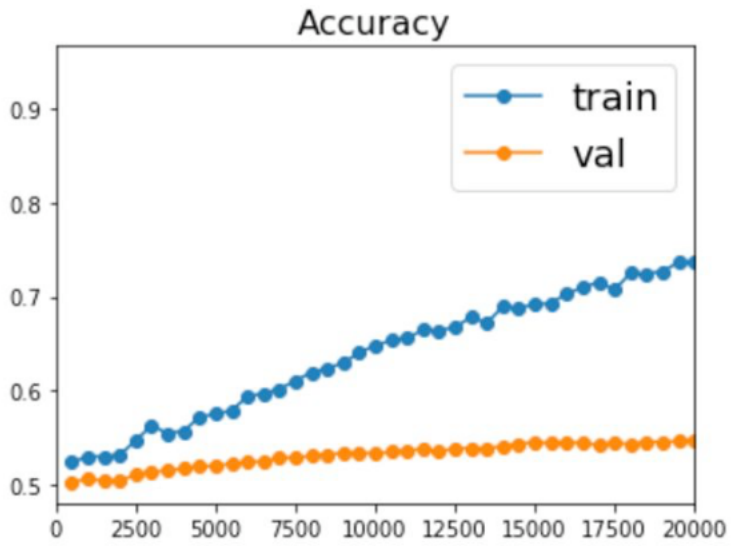
$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Algorithm 1: Batch Normalizing Transform, applied to activation x over a mini-batch.

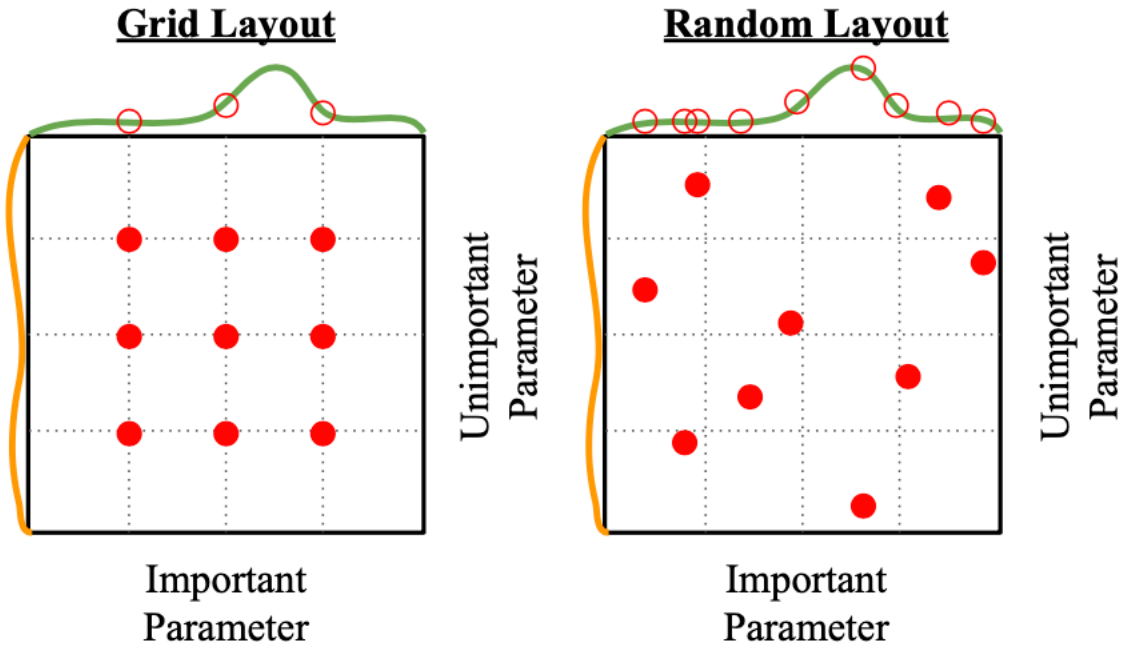
Babysitting Learning







Hyperparameter Search



Coarse to fine search

```
val_acc: 0.412000, lr: 1.405206e-04, reg: 4.793564e-01, (1 / 100)
val_acc: 0.214000, lr: 7.231888e-06, reg: 2.321281e-04, (2 / 100)
val_acc: 0.208000, lr: 2.119571e-06, reg: 8.011857e+01, (3 / 100)
val_acc: 0.196000, lr: 1.551131e-05, reg: 4.374936e-05, (4 / 100)
val_acc: 0.079000, lr: 1.753300e-05, reg: 1.200424e+03, (5 / 100)
val_acc: 0.223000, lr: 4.215128e-05, reg: 4.196174e+01, (6 / 100)
val_acc: 0.441000, lr: 1.750259e-04, reg: 2.110807e-04, (7 / 100)
val_acc: 0.241000, lr: 6.749231e-05, reg: 4.226413e+01, (8 / 100)
val_acc: 0.482000, lr: 4.296863e-04, reg: 6.642555e-01, (9 / 100)
val_acc: 0.079000, lr: 5.401602e-06, reg: 1.599828e+04, (10 / 100)
val_acc: 0.154000, lr: 1.618508e-06, reg: 4.925252e-01, (11 / 100)
```

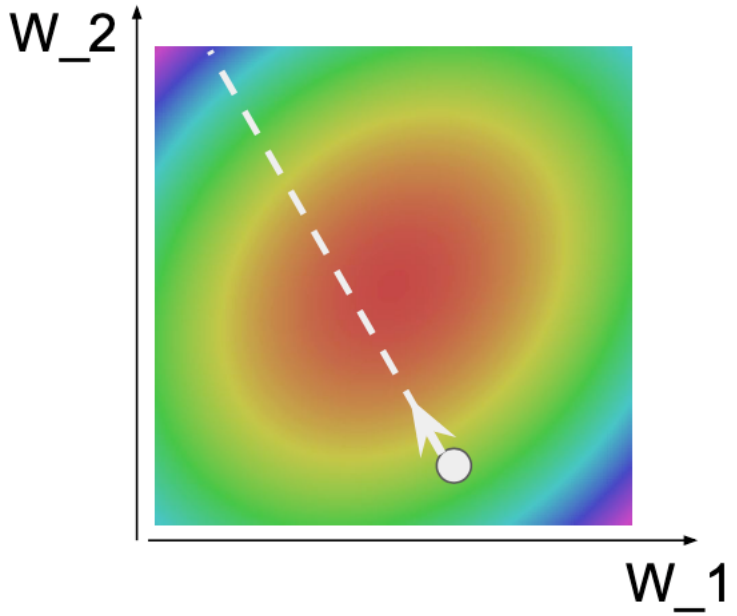
```
val_acc: 0.527000, lr: 5.340517e-04, reg: 4.097824e-01, (0 / 100)
val_acc: 0.492000, lr: 2.279484e-04, reg: 9.991345e-04, (1 / 100)
val_acc: 0.512000, lr: 8.680827e-04, reg: 1.349727e-02, (2 / 100)
val_acc: 0.461000, lr: 1.028377e-04, reg: 1.220193e-02, (3 / 100)
val_acc: 0.460000, lr: 1.113730e-04, reg: 5.244309e-02, (4 / 100)
val_acc: 0.498000, lr: 9.477776e-04, reg: 2.001293e-03, (5 / 100)
val_acc: 0.469000, lr: 1.484369e-04, reg: 4.328313e-01, (6 / 100)
val_acc: 0.522000, lr: 5.586261e-04, reg: 2.312685e-04, (7 / 100)
val_acc: 0.530000, lr: 5.808183e-04, reg: 8.259964e-02, (8 / 100)
val_acc: 0.489000, lr: 1.979168e-04, reg: 1.010889e-04, (9 / 100)
val_acc: 0.490000, lr: 2.036031e-04, reg: 2.406271e-03, (10 / 100)
val_acc: 0.475000, lr: 2.021162e-04, reg: 2.287807e-01, (11 / 100)
val_acc: 0.460000, lr: 1.135527e-04, reg: 3.905040e-02, (12 / 100)
val_acc: 0.515000, lr: 6.947668e-04, reg: 1.562808e-02, (13 / 100)
val_acc: 0.531000, lr: 9.471549e-04, reg: 1.433895e-03, (14 / 100)
val_acc: 0.509000, lr: 3.140888e-04, reg: 2.857518e-01, (15 / 100)
val_acc: 0.514000, lr: 6.438349e-04, reg: 3.033781e-01, (16 / 100)
val_acc: 0.502000, lr: 3.921784e-04, reg: 2.707126e-04, (17 / 100)
val_acc: 0.509000, lr: 9.752279e-04, reg: 2.850865e-03, (18 / 100)
val_acc: 0.500000, lr: 2.412048e-04, reg: 4.997821e-04, (19 / 100)
val_acc: 0.466000, lr: 1.319314e-04, reg: 1.189915e-02, (20 / 100)
val_acc: 0.516000, lr: 8.039527e-04, reg: 1.528291e-02, (21 / 100)
```

Optimization

- Loss를 최소화하는 최적의 parameter를 찾는 과정

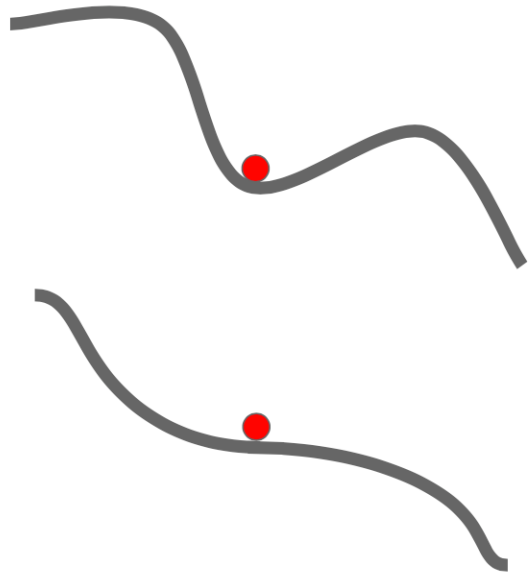
```
# Vanilla Gradient Descent

while True:
  weights_grad = evaluate_gradient(loss_fun, data, weights)
  weights += - step_size * weights_grad # perform parameter update
```



Optimization: Problems with SGD

- Vanilia SGD는 loss landscape의 상황에 따라 크게 영향을 받음
- 평평한 곳에서는 아주 천천히 움직이고 가파른 곳에서는 변덕이 심함.
- 또한 local minima나 saddle point를 만나면 global optimization을 실패해 일반화 성능이 나빠질 수 있음
- 또한 mini-batch를 사용하기 때문에 각 batch에 따라 변덕이 심할 수 있음.



SGD + Momentum, Nesterov Momentum

- 이전 gradient를 반영해서 가중치 업데이트
- 즉, 이전의 방향을 고려해서 업데이트
- 비율은 주로 이전 gradient를 0.9 정도로 준다고 함.

SGD+Momentum

$$v_{t+1} = \rho v_t + \nabla f(x_t)$$
$$x_{t+1} = x_t - \alpha v_{t+1}$$

```
vx = 0
while True:
  dx = compute_gradient(x)
  vx = rho * vx + dx
  x += learning_rate * vx
```

Nesterov Momentum

$$v_{t+1} = \rho v_t - \alpha \nabla f(x_t + \rho v_t)$$
$$x_{t+1} = x_t + v_{t+1}$$

Annoying, usually we want update in terms of $x_t, \nabla f(x_t)$

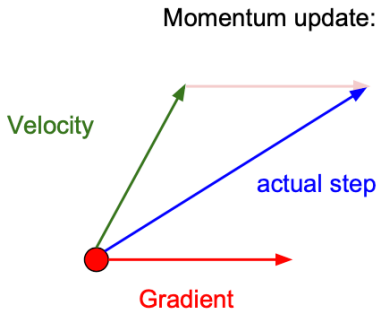
Change of variables $\tilde{x}_t = x_t + \rho v_t$ and rearrange:

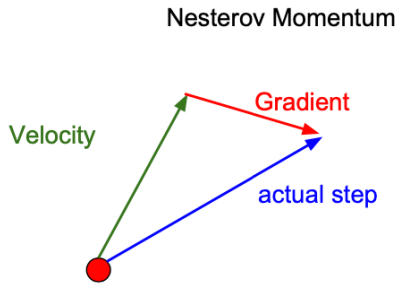
$$v_{t+1} = \rho v_t - \alpha \nabla f(\tilde{x}_t)$$
$$\tilde{x}_{t+1} = \tilde{x}_t - \rho v_t + (1+\rho) v_{t+1}$$
$$= \tilde{x}_t + v_{t+1} + \rho(v_{t+1} - v_t)$$

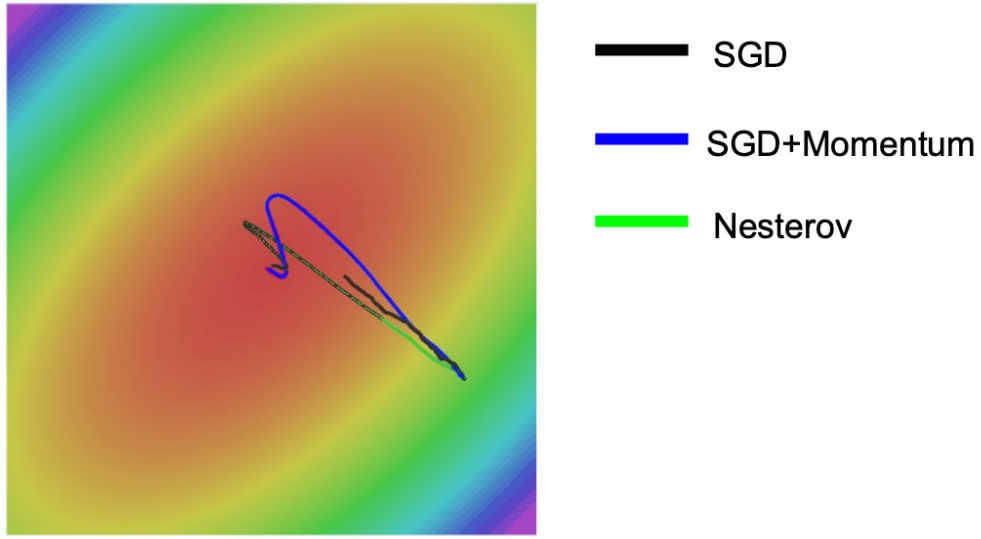
```
dx = compute_gradient(x)
old_v = v
v = rho * v - learning_rate * dx
x += -rho * old_v + (1 + rho) * v
```

- Build up “velocity” as a running mean of gradients
- Rho gives “friction”; typically rho=0.9 or 0.99

SGD + Momentum, Nesterov Momentum







- Momentum을 사용하면 Local minima나 saddle points를 만나도 이전 gradient의 추진력을 이용해 벗어날 수 있게 만듦

AdaGrad

- Learning rate가 작거나 커지면 모델 피팅이 어려워 진다.
- Ada grad는 학습이 진행될 수록 learning rate를 줄여 학습이 잘 되도록 만듦.

```
grad_squared = 0
while True:
  dx = compute_gradient(x)
  grad_squared += dx * dx
  x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```

RMSProp

- Adagrad는 grad_squared를 너무 크게 만들어 학습률을 너무 빨리 줄여버릴 수 있음.
- 그래서 RMSProp은 grad_squared를 단순 합하지 않고 momentum 처럼 가중평균을 이용해 너무 커지는 것을 방지함.

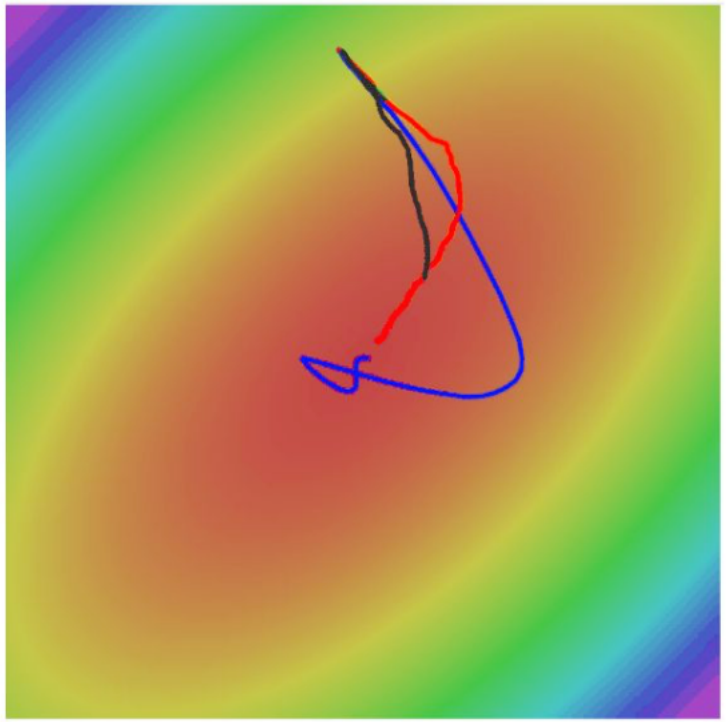
AdaGrad

```
grad_squared = 0
while True:
  dx = compute_gradient(x)
  grad_squared += dx * dx
  x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```

RMSProp

```
grad_squared = 0
while True:
  dx = compute_gradient(x)
  grad_squared = decay_rate * grad_squared + (1 - decay_rate) * dx * dx
  x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```

RMSProp



SGD

SGD+Momentum

RMSProp

Adam

- Momentum과 RMSProp을 융합한 방법
- Beta1,2 / learning rate는 hyperparameter

```
first_moment = 0
second_moment = 0
for t in range(1, num_iterations):
  dx = compute_gradient(x)
  first_moment = beta1 * first_moment  + (1 - beta1) * dx
  second_moment = beta2 * second_moment + (1 - beta2) * dx * dx
  first_unbias = first_moment / (1 - beta1 ** t)
  second_unbias = second_moment / (1 - beta2 ** t)
  x -= learning_rate * first_unbias / (np.sqrt(second_unbias) + 1e-7))
```

Momentum

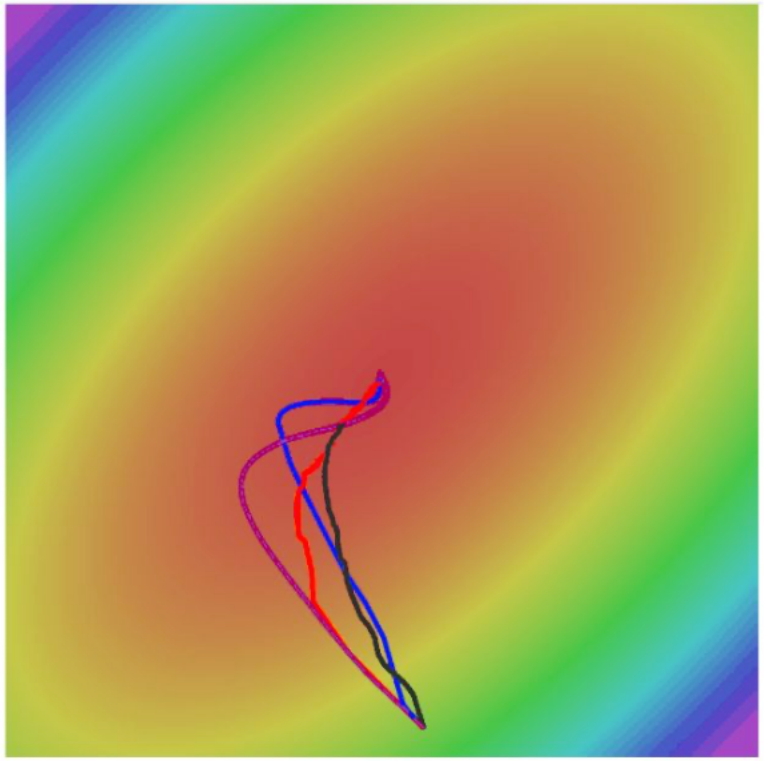
Bias correction

AdaGrad / RMSProp

Bias correction for the fact that first and second moment estimates start at zero

Adam with beta1 = 0.9, beta2 = 0.999, and learning_rate = 1e-3 or 5e-4 is a great starting point for many models!

Adam



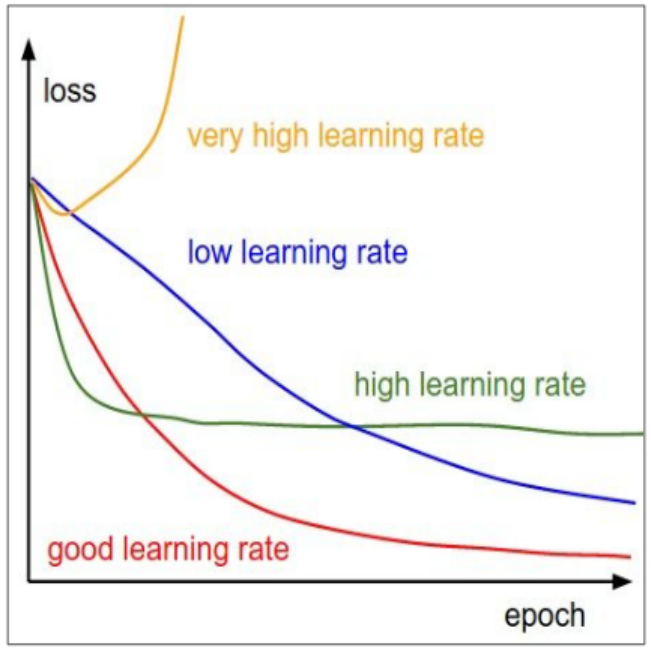
SGD

SGD+Momentum

RMSProp

Adam

SGD, SGD+Momentum, Adagrad, RMSProp, Adam all have **learning rate** as a hyperparameter.



=> Learning rate decay over time!

step decay:
e.g. decay learning rate by half every few epochs.

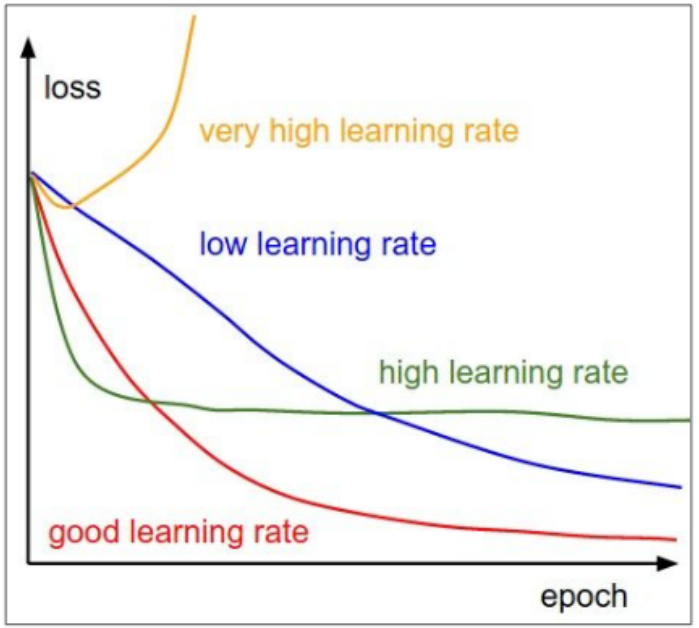
exponential decay:

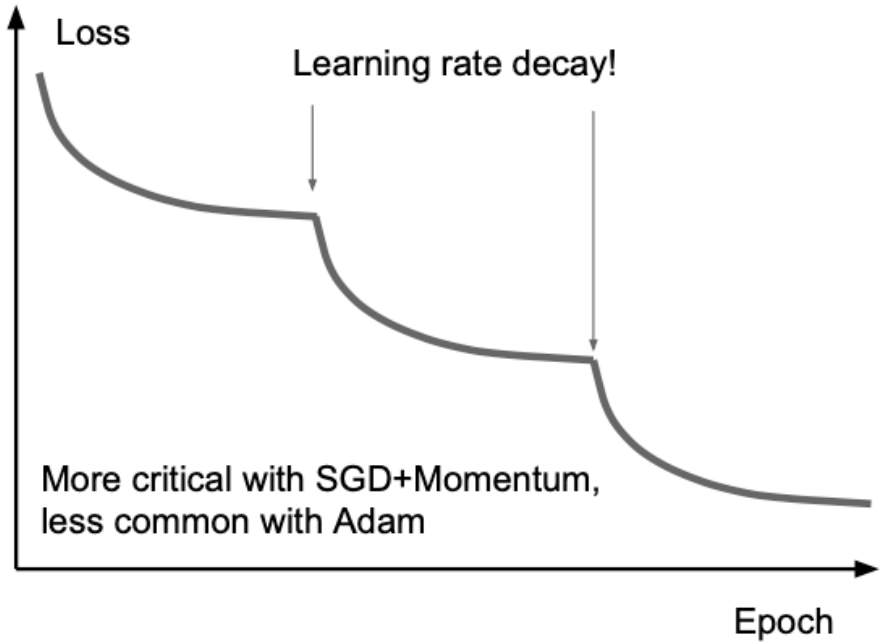
$$\alpha = \alpha_0 e^{-kt}$$

1/t decay:

$$\alpha = \alpha_0 / (1 + kt)$$

SGD, SGD+Momentum, Adagrad, RMSProp, Adam all have **learning rate** as a hyperparameter.





First-Order/Second-Order Optimization

- First-Order
  - 선형 함수의 gradient를 이용해 가중치 업데이트
- Second-Order
  - Loss function의 테일러급수를 이용해 다항식을 만든 후 newton's method를 이용해 gradient를 구한 후 업데이트
  - hyperparameter(learning rate 등)이 필요없지만 계산 비용이 높음.

$$f(x_k + t) \approx f(x_k) + f'(x_k)t + \frac{1}{2}f''(x_k)t^2.$$

$$0 = \frac{\mathrm{d}}{\mathrm{d}t}\left(f(x_k) + f'(x_k)t + \frac{1}{2}f''(x_k)t^2\right) = f'(x_k) + f''(x_k)t,$$

$$t = -\frac{f'(x_k)}{f''(x_k)}.$$

$$x_{k+1} = x_k + t = x_k - \frac{f'(x_k)}{f''(x_k)}.$$

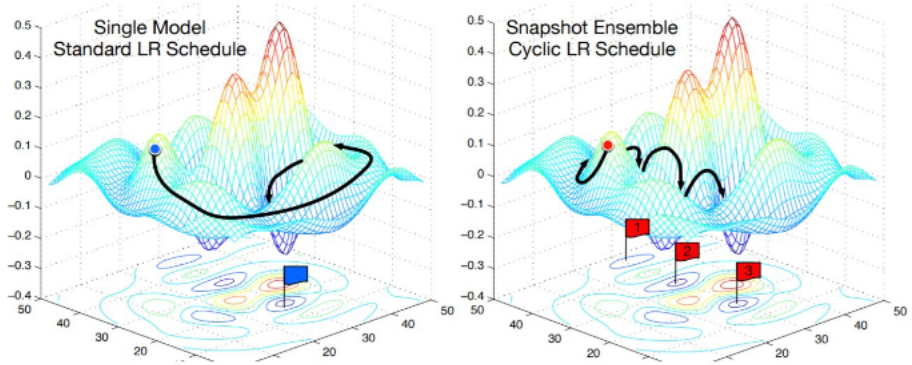
First-Order/Second-Order Optimization

- Second-Order
  - L-BFGS – full batch를 사용할 때 좋을 수 있음
  - BGFS

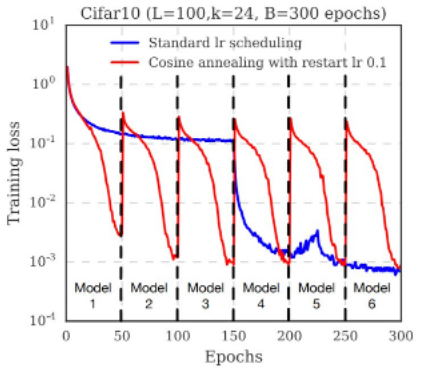
- **Adam**을 실제로 가장 많이 사용하게 됨

Model Ensemble

- 여러 개의 모델을 동시에 예측에 사용하는 전략
- 여러 개의 독립된 모델을 사용해 각각 예측하고 평균내는 방법
- 하나의 신경망 모델을 학습할 때 학습과정에서 생기는 각 단계의 모델들을 각각 사용해 ensemble기법으로 사용할 수 있음(snap shot), 각각의 독립된 모델을 만드는 것보다 계산 비용이 적게 든다.
- 성능에 큰 영향을 주지 않음



Loshchilov and Hutter, "SGDR: Stochastic gradient descent with restarts", arXiv 2016
Huang et al, "Snapshot ensembles: train 1, get M for free", ICLR 2017
Figures copyright Yixuan Li and Geoff Pleiss, 2017. Reproduced with permission.



Cyclic learning rate schedules can make this work even better!

Regularization: Add term to loss

- parameter가 많을 때 overfitting을 방지하기 위함

$$L = \frac{1}{N}\sum_{i=1}^{N}\sum_{j\neq y_i}\max(0, f(x_i;W)_j - f(x_i;W)_{y_i} + 1) + \boxed{\lambda R(W)}$$

In common use:

L2 regularization $R(W) = \sum_k \sum_l W_{k,l}^2$ (Weight decay)

L1 regularization $R(W) = \sum_k \sum_l |W_{k,l}|$

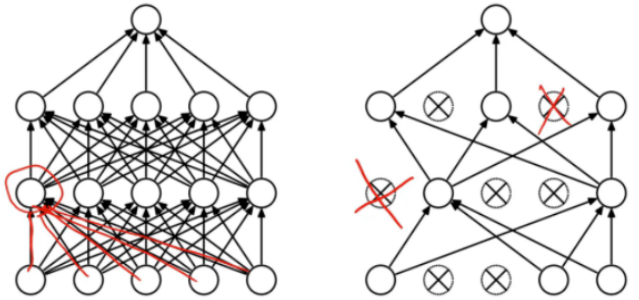
Elastic net (L1 + L2) $R(W) = \sum_k \sum_l \beta W_{k,l}^2 + |W_{k,l}|$

Regularization: Dropout

- 모델 학습 정해진 비율의 activation들을 랜덤을 0을 만들어 overfitting을 방지.
- Dropout비율은 사용자가 정하는 hyperparameter

Dropout: A Simple Way to Prevent Neural Networks from Overfitting

Nitish Srivastava NITISH@CS.TORONTO.EDU
Geoffrey Hinton HINTON@CS.TORONTO.EDU
Alex Krizhevsky KRIZ@CS.TORONTO.EDU
Ilya Sutskever ILYA@CS.TORONTO.EDU
Ruslan Salakhutdinov RSALAKHU@CS.TORONTO.EDU



(a) Standard Neural Net (b) After applying dropout.

Figure 1: Dropout Neural Net Model. **Left**: A standard neural net with 2 hidden layers. **Right**: An example of a thinned net produced by applying dropout to the network on the left. Crossed units have been dropped.

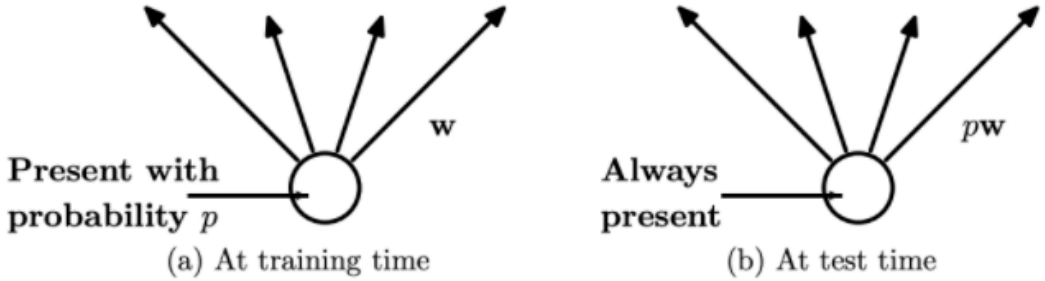
Regularization: Dropout

- Activation이 특정 데이터의 특징을 기억하는 것을 방지함으로써 overfitting을 방지.

I went to my bank. The tellers kept changing and I asked one of them why. He said he didn't know but they got moved around a lot. I figured it must be because it would require cooperation between employees to successfully defraud the bank. This made me realize that randomly removing a different subset of neurons on each example would prevent conspiracies and thus reduce overfitting.
Hinton: Reddit AMA

Regularization: Dropout

- 강의에서는 activation의 scale을 맞춰주기 위해 dropout 비율을 곱해준다고 함



(a) At training time (b) At test time

Figure 2: **Left**: A unit at training time that is present with probability p and is connected to units in the next layer with weights $\mathbf{w}$. **Right**: At test time, the unit is always present and the weights are multiplied by p. The output at test time is same as the expected output at training time.

- Test의 경우에는 dropout을 사용하지 않고 가중치에 dropout 비율을 곱해준다.
- This ensures that for any hidden unit the expected output (under the distribution used to drop units at training time) is the same as the actual output at test time. By doing this scaling, 2^n networks with shared weights can be combined into a single neural network to be used at test time. We found that training a network with dropout and using this approximate averaging method at test time leads to significantly lower generalization error on a wide variety of classification problems compared to training with other regularization methods.
- Dropout을 사용하면 같은 가중치를 가지는 2^n개의 신경망을 학습한 것과 같다 그래서 parameter에 dropout비율을 곱해줘 하나의 신경망 모델로 만들어 준다...?

Regularization: A common pattern

Training: Add some kind of randomness

$$y = f_W(x, z)$$

Testing: Average out randomness (sometimes approximate)

$$y = f(x) = E_z\left[f(x, z)\right] = \int p(z) f(x, z) dz$$

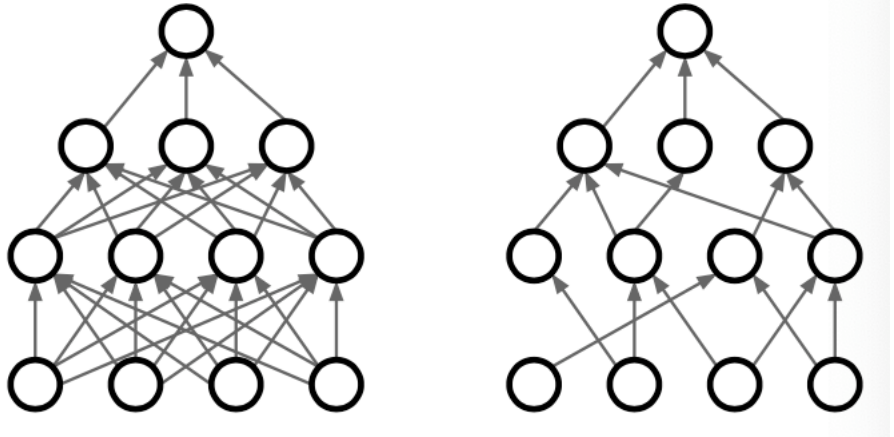
Regularization: A common pattern

- Dropout
- Batch Normalization
  - Training : 각 mini-batch의 통계량을 이용해 batch normalization
  - Test : 고정된 통계량을 사용해 normalization
- Data Augmentation
  - Test에서는 고정된 augmentation을 적용
  - 다양한 방법 적용 가능(zoom, padding, noising..)

Regularization: A common pattern

Training: Add random noise
Testing: Marginalize over the noise

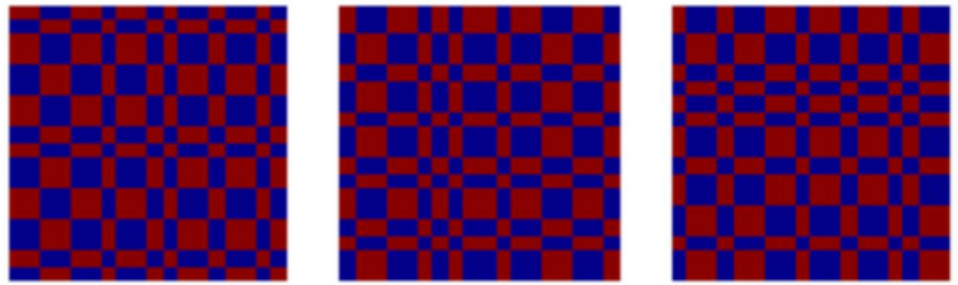
Examples:
Dropout
Batch Normalization
Data Augmentation
DropConnect



Regularization: A common pattern

Training: Add random noise
Testing: Marginalize over the noise

Examples:
Dropout
Batch Normalization
Data Augmentation
DropConnect
Fractional Max Pooling

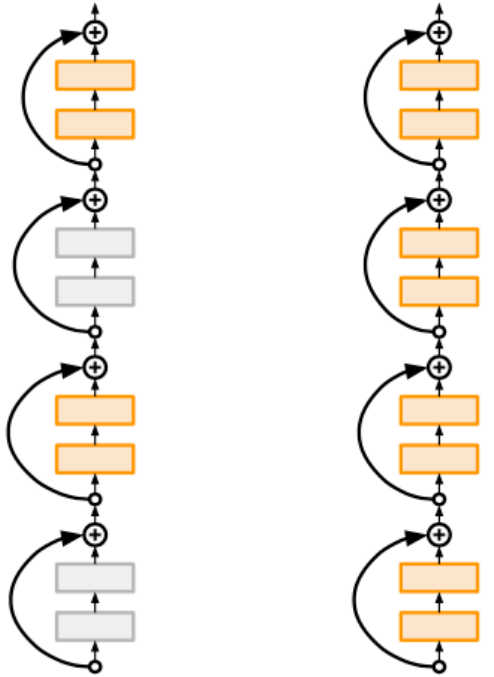


Regularization: A common pattern

Training: Add random noise
Testing: Marginalize over the noise

Examples:
Dropout
Batch Normalization
Data Augmentation
DropConnect
Fractional Max Pooling
Stochastic Depth

Huang et al, "Deep Networks with Stochastic Depth", ECCV 2016



Regularization: A common pattern

Training: Add some kind of randomness

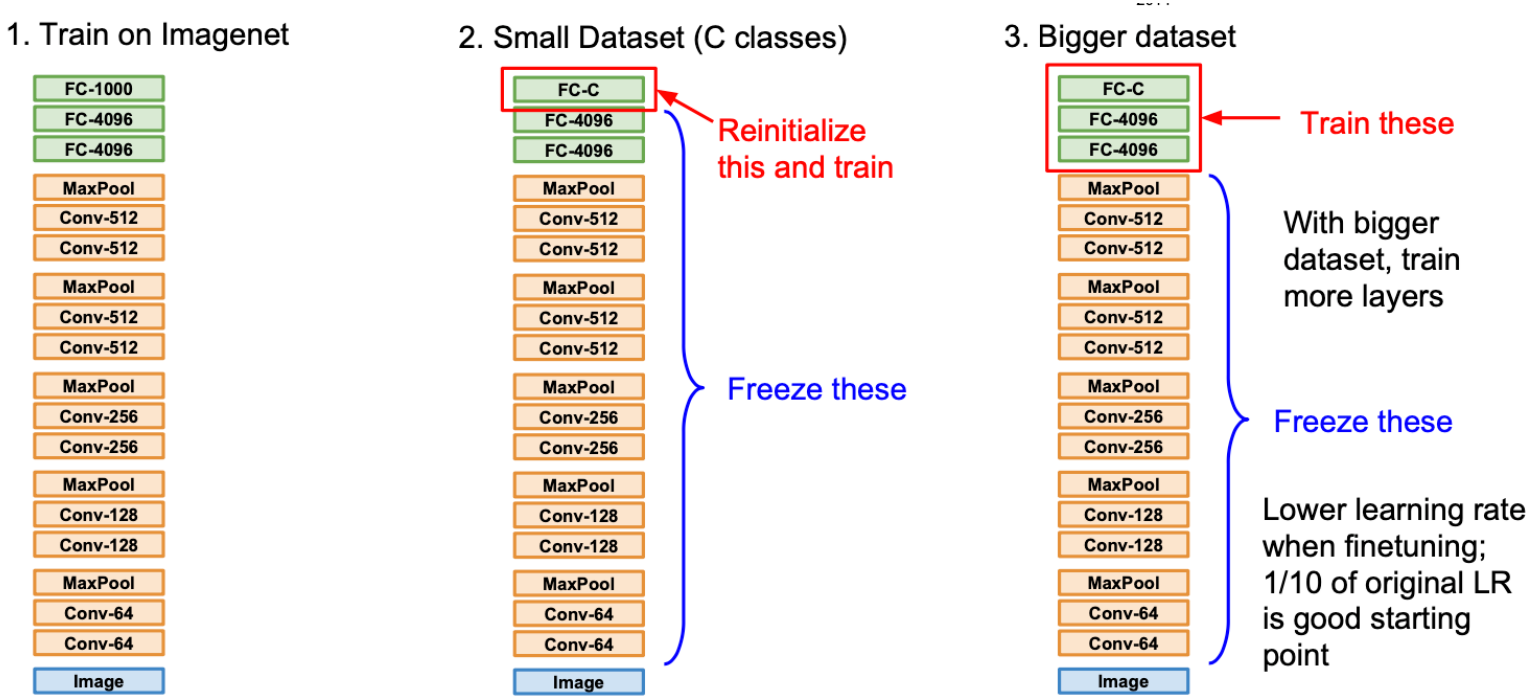
$$y = f_W(x, z)$$

Testing: Average out randomness (sometimes approximate)

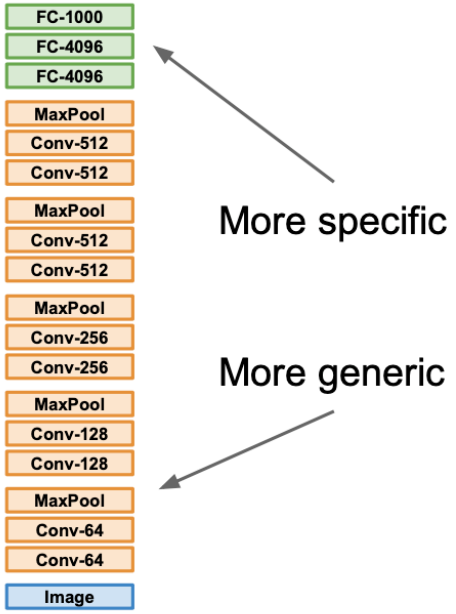
$$y = f(x) = E_z\left[f(x, z)\right] = \int p(z) f(x, z) dz$$

Transfer Learning with CNNS

- 미리 많은 양의 데이터에 fitting되어있는 모델의 parameter를 이용하는 방법.



Transfer Learning



	very similar dataset	very different dataset
very little data	Use Linear Classifier on top layer	You're in trouble… Try linear classifier from different stages
quite a lot of data	Finetune a few layers	Finetune a larger number of layers

- 대부분의 application에서 pretrained model을 사용하는 추세

- Optimization
  - Momentum, RMSProp, Adam, etc
- Regularization
  - Dropout, etc
- Transfer learning
  - Use this for your projects!

감 사 합 니 다 !

Thank you!