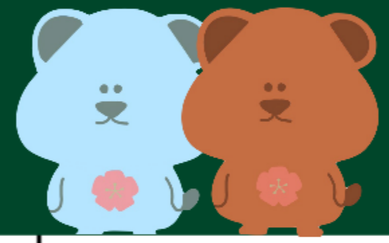
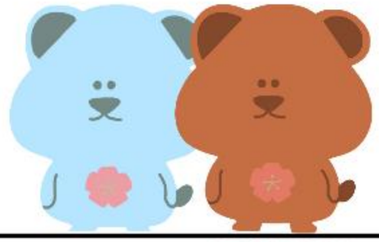


Training Neural Networks II

Week7_발표자: 하수민, 최예은

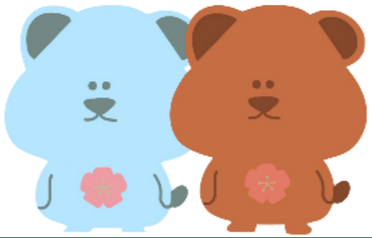
EWHA EURON

목차



EWHA EURON

Review

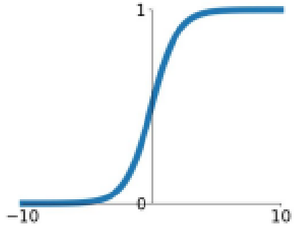


EWHA EURON

1. Review: Activation Function

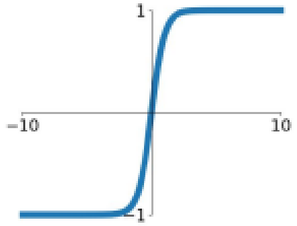
Sigmoid

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



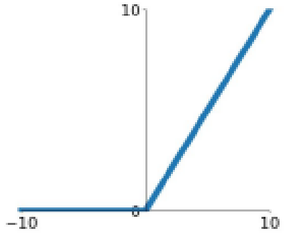
tanh

$$\tanh(x)$$



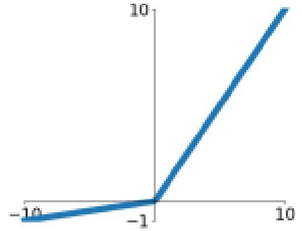
ReLU

$$\max(0, x)$$



Leaky ReLU

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

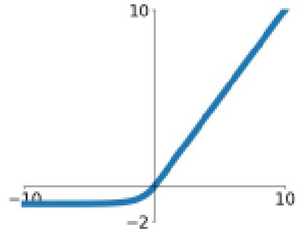


Maxout

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

ELU

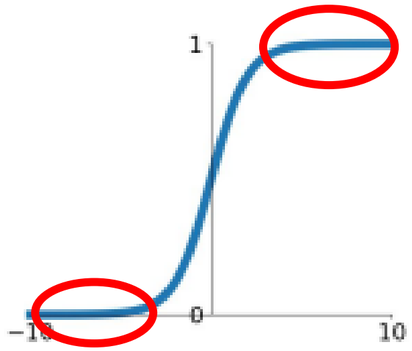
$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



1. Review: Activation Function

Sigmoid

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

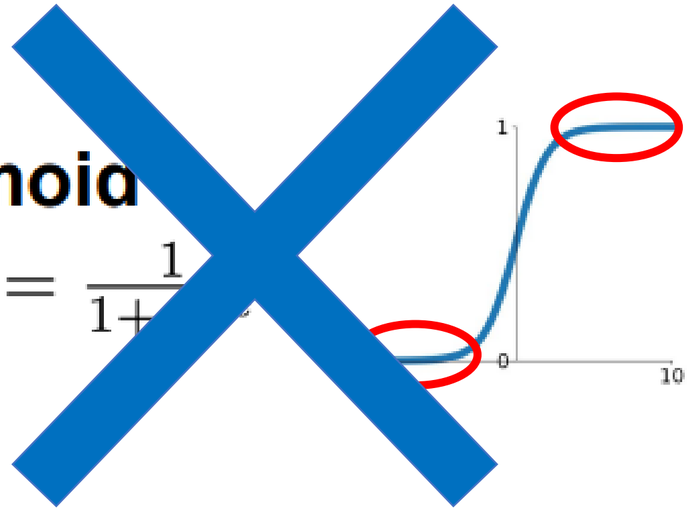


Vanishing Gradients

1. Review: Activation Function

Sigmoid

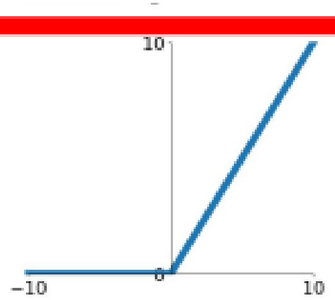
$$\sigma(x) = \frac{1}{1+}$$



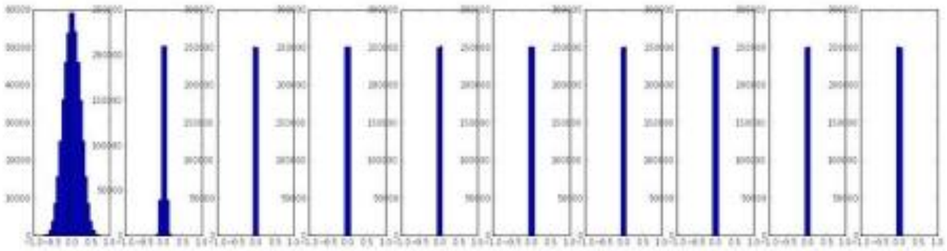
ReLU

$$\max(0, x)$$

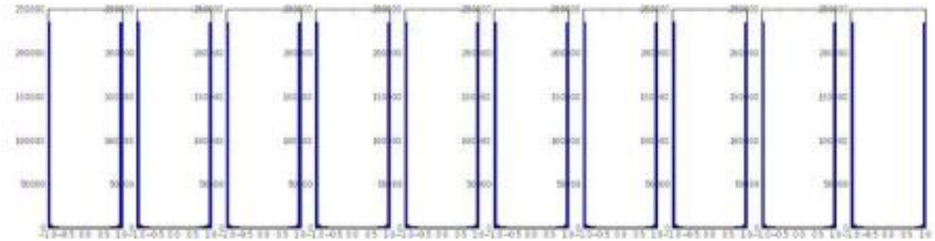
Good default choice



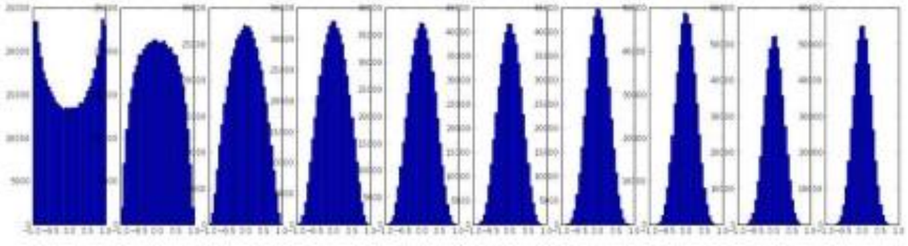
1. Review: Weight Initialization



가중치를 지나치게 작게 초기화하면?
➞ 작은 값이 계속해서 곱해지므로 gradient가 0이 된다.
➞ 모든 activation이 0이 되고 학습은 이루어지지 않는다.



가중치를 지나치게 크게 초기화하면?
➞ 큰 값이 계속해서 곱해지기 때문에 activation이 saturate
➞ gradient는 0이 되고 학습은 이루어지지 않는다.



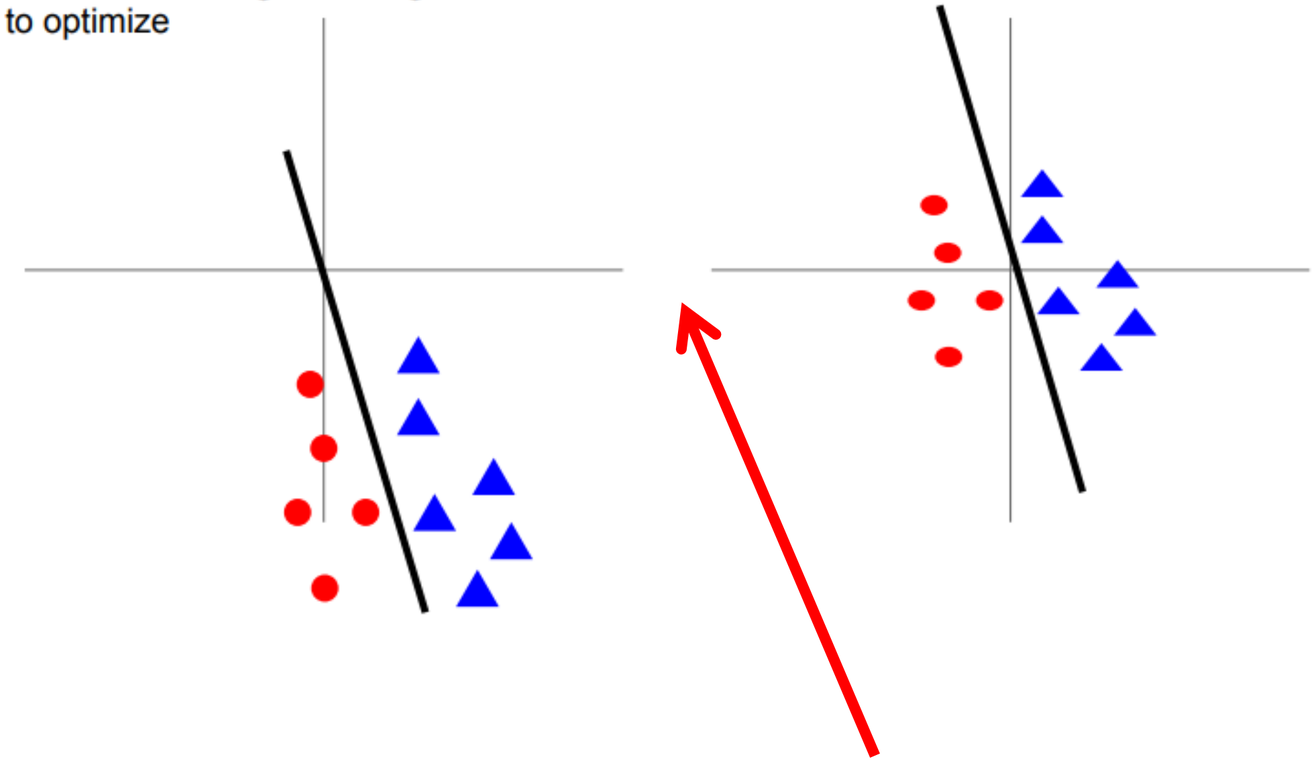
Xavier/MSRA Initialization을 사용해 가중치 초기화를 잘해주면?
➞ activation 분포를 좋게 유지할 수 있다.
➞ 학습이 잘 이루어진다.

Network가 깊어질수록 가중치를 더 많이 곱하게 된다.
➞ 가중치 초기화는 Network가 깊어질수록 더 중요!!

1. Review: 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



데이터 전처리를 통해 zero-mean, unit variance
➞ 손실 함수가 가중치의 변동에 덜 민감하다.
➞ 최적화, 학습이 더 쉽다.

1. Review: Batch Normalization

Input: $x : N \times D$

Learnable params:
$\gamma, \beta : D$

Intermediates: $\mu, \sigma : D$
$\hat{x} : N \times D$

Output: $y : N \times D$

$$\mu_j = \frac{1}{N}\sum_{i=1}^{N} x_{i,j}$$

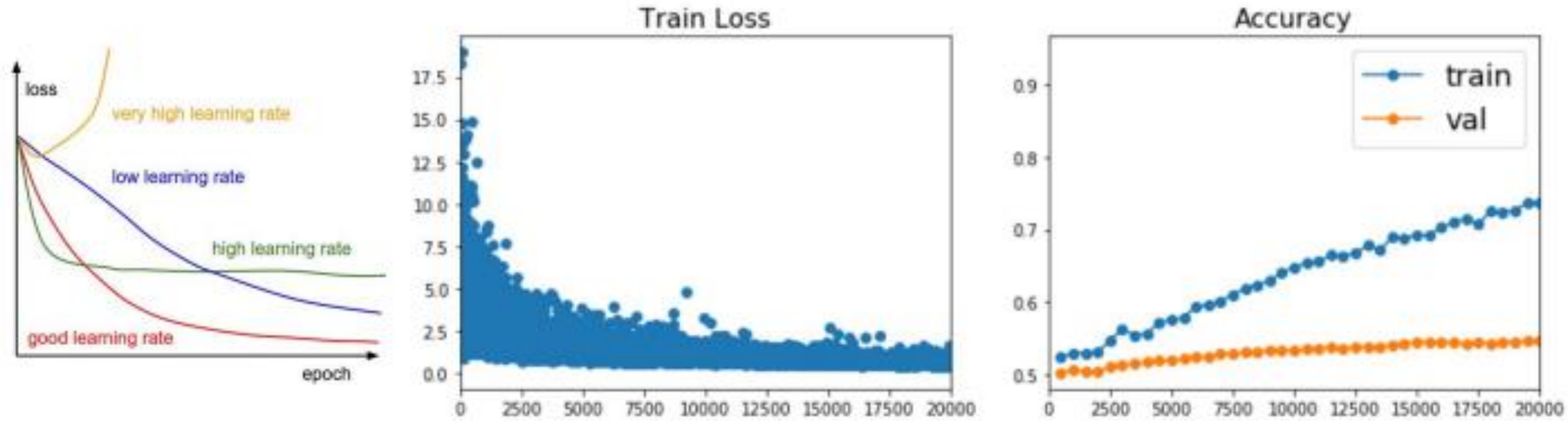
$$\sigma_j^2 = \frac{1}{N}\sum_{i=1}^{N}(x_{i,j} - \mu_j)^2$$

$$\hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \varepsilon}}$$

$$y_{i,j} = \gamma_j \hat{x}_{i,j} + \beta_j$$

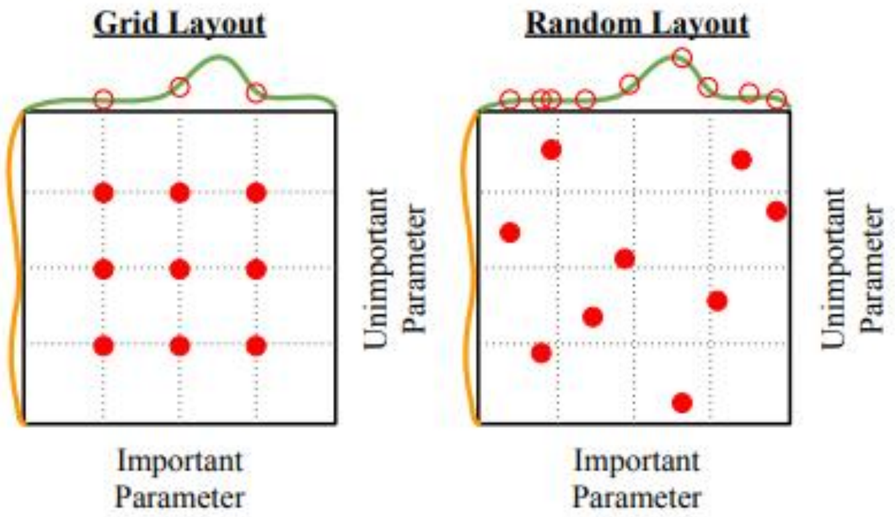
Gradient Vanishing이 발생하지 않도록 하기 위해 나온 아이디어
activation이 unit gaussian이 될 수 있도록 레이어를 하나 추가하는 방법
Mini-batch 마다 mean, variance를 계산하고, 이 값을 이용해 normalize
scale, shift 파라미터

1. Review: Babysitting Learning



train set 성능 계속 올라감 & loss 줄고 있음 but, val이 침체중이라면?
→ overfitting 된 것. Regularization이 필요!!

1. Review: Hyperparameter Search



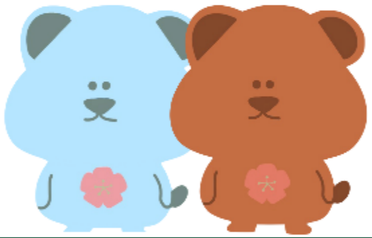
Coarse to fine search

```
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: 1.753380e-05, reg: 1.280424e+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: 3.461602e-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: 0.452000, lr: 2.273484e-04, reg: 9.991345e-04, (1 / 100)
val_acc: 0.512000, lr: 8.600827e-04, reg: 1.349727e-02, (2 / 100)
val_acc: 0.461000, lr: 1.028377e-04, reg: 1.220103e-02, (3 / 100)
val_acc: 0.468000, lr: 1.113730e-04, reg: 5.244389e-02, (4 / 100)
val_acc: 0.496000, lr: 9.477776e-04, reg: 2.001293e-03, (5 / 100)
val_acc: 0.469000, lr: 1.484102e-04, reg: 4.328113e-03, (6 / 100)
val_acc: 0.522000, lr: 5.506261e-04, reg: 2.312685e-04, (7 / 100)
val_acc: 0.520000, lr: 3.008183e-04, reg: 8.259964e-02, (8 / 100)
val_acc: 0.489000, lr: 1.979160e-04, reg: 1.018889e-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: 2.287887e-01, (11 / 100)
val_acc: 0.466000, lr: 1.135527e-04, reg: 3.905040e-02, (12 / 100)
val_acc: 0.515000, lr: 6.947860e-04, reg: 1.362888e-02, (13 / 100)
val_acc: 0.531000, lr: 9.471549e-04, reg: 1.439895e-03, (14 / 100)
val_acc: 0.509000, lr: 3.140886e-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: 3.921704e-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: 0.486000, lr: 1.319314e-04, reg: 1.189915e-02, (20 / 100)
val_acc: 0.516000, lr: 8.699527e-04, reg: 1.528291e-02, (21 / 100)
```

Grid search보다 Random search를 더 많이 사용
적절한 하이퍼파라미터를 잘 찾아야 train이 잘 이루어진다.

Optimization



EWHA EURON

2. Optimization: SGD 복습

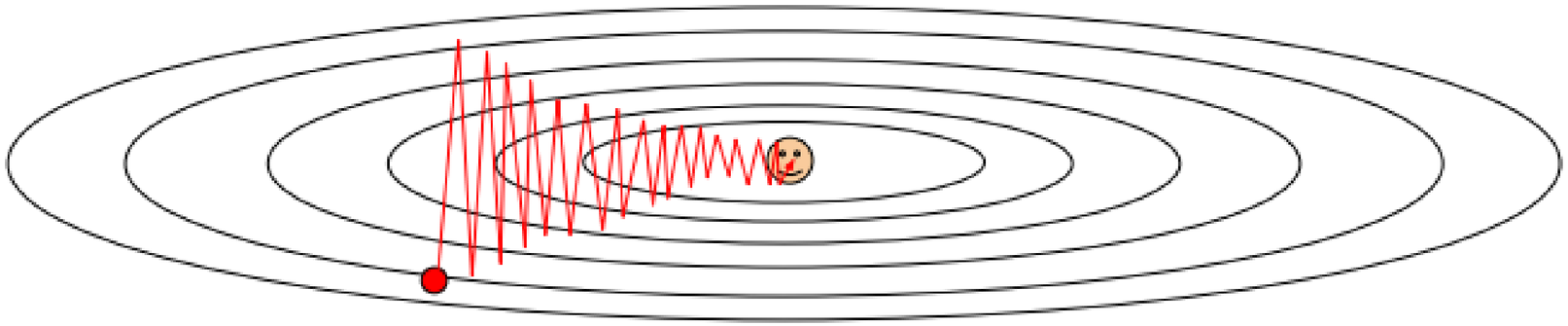
SGD(Stochastic Gradient Descent)

가장 간단한 최적화 알고리즘

1. Mini batch 안에 있는 data의 loss를 계산한다.
2. Gradient의 반대 방향을 이용하여 update를 한다.
3. 1,2번 과정을 반복한다.

2. Optimization: SGD 문제점

1. jittering

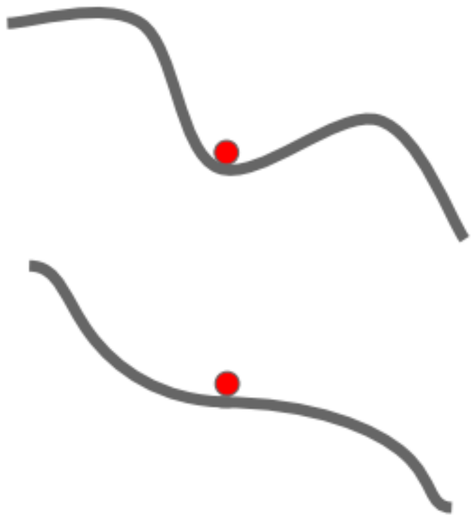


Loss 방향이 한 방향으로만 빠르게 바뀌고 반대 방향으로는 느리게 바뀌면?
➞ 불균형한 방향이 존재해 SGD는 잘 작동하지 않는다.

2. Optimization: SGD 문제점

2. Local Minima & Saddle Point

X축은 하나의 가중치, y축은 loss



휘어진 손실함수의 중간에 local minima가 있다.
➡ 순간적으로 기울기가 0이 되어 SGD는 멈춘다.

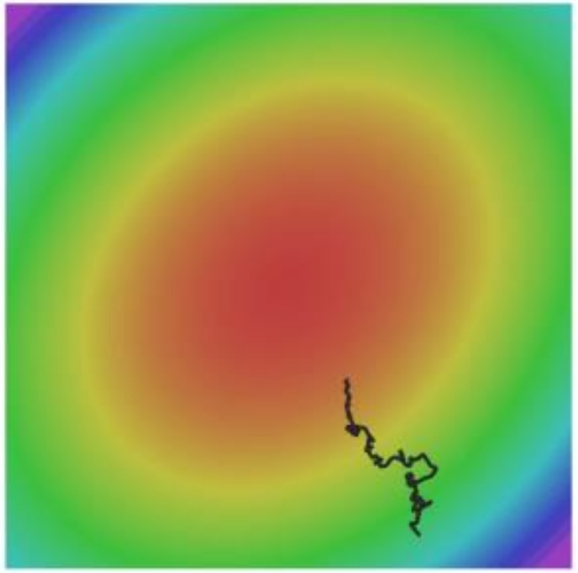
한쪽 방향으로는 증가하고 다른 방향으로는 감소하는 saddle point가 있다.
➡ 마찬가지로 순간적으로 기울기가 0이 되어 SGD가 멈춘다.

2. Optimization: SGD 문제점

3. Noise

$$L(W) = \frac{1}{N}\sum_{i=1}^{N} L_i(x_i, y_i, W)$$

$$\nabla_W L(W) = \frac{1}{N}\sum_{i=1}^{N} \nabla_W L_i(x_i, y_i, W)$$



Minibatches에서 gradient 값이 노이즈에 의해 많이 변할 수 있다.
오른쪽 그림처럼 gradient가 꼬불꼬불하게 update 될 수 있다.

2. Optimization: SGD + Momentum

앞서 본 SGD의 문제점들을 해결하기 위해 Momentum이라는 개념이 도입되었다.

SGD

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

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

SGD+Momentum

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

$$x_{t+1} = x_t - \alpha v_{t+1}$$

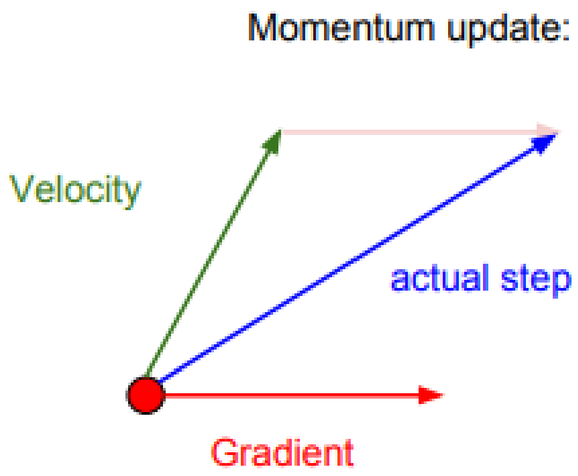
```
vx = 0
while True:
  dx = compute_gradient(x)
  vx = rho * vx + dx
  x += learning_rate * vx
```

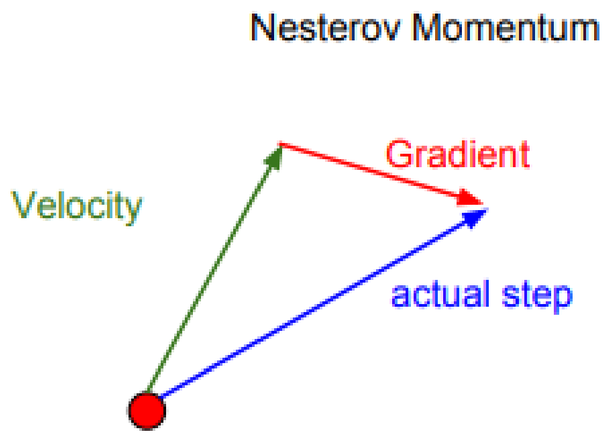
Momentum은 기울기가 0인 지점에 빠지더라도 가속도로 탐색을 진행하도록 SGD에서 gradient를 계산할 때 velocity를 추가하는 방법이다.

기존 SGD에서 momentum의 비율인 하이파퍼라미터 rho가 추가되었다.

2. Optimization: SGD + Momentum

Nesterov Momentum





기존의 momentum은 직전 지점의 gradient와 velocity 벡터의 합벡터로 다음 지점을 구했다.

Nesterov momentum 방식은 직전 지점의 velocity 벡터의 방향을 예측하여 그 지점에서의 gradient를 계산하고 원점으로 돌아가 합벡터로 다음 지점을 구한다.

Convex에서는 성능이 뛰어나나 NN과 같은 non-convex에서는 성능을 보장할 수 없다.

2. Optimization: AdaGrad

Velocity 대신에 grad squared term을 이용해서 gradient를 update하는 방법
학습율을 효과적으로 정하기 위해 제안된 방법

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

Gradient를 제곱해서 더해준 후, 업데이트할 때 앞서 계산한 gradient 제곱근 항으로 나눠준다.
➡ 이런 방식으로 업데이트를 계속하면 small dimension에서는 속도가 늘어나고, large dimension에서는 속도가 줄어든다.

학습이 진행될수록 step size는 점점 작아진다.(업데이트할 때 gradient의 제곱이 계속 더해지기 때문)
손실함수가 convex한 경우에는 step size가 작아지는 것이 좋다.
👎 non-convex한 경우에는 saddle point에 걸리게 되면 도중에 멈추게 될 수 있다. ➡ RMSProp 제안

2. Optimization: RMSProp

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)
```

RMSProp은 AdaGrad의 제곱항을 그대로 사용하지만 decay rate(보통 0.9~0.99)을 곱해준다.
gradient의 제곱을 계속해서 누적해 간다는 점에서 AdaGrad와 차이점이 있다.
➞ step의 속도를 효율적으로 가속/감속시킬 수 있다.

2. Optimization: Adam

Momentum + Ada 계열
: Velocity를 이용해 step을 조절하는 momentum과 gradient의 제곱을 나눠주는 방식으로 step을 조절하는 Ada계열(AdaGrad/RMSProp)

```
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

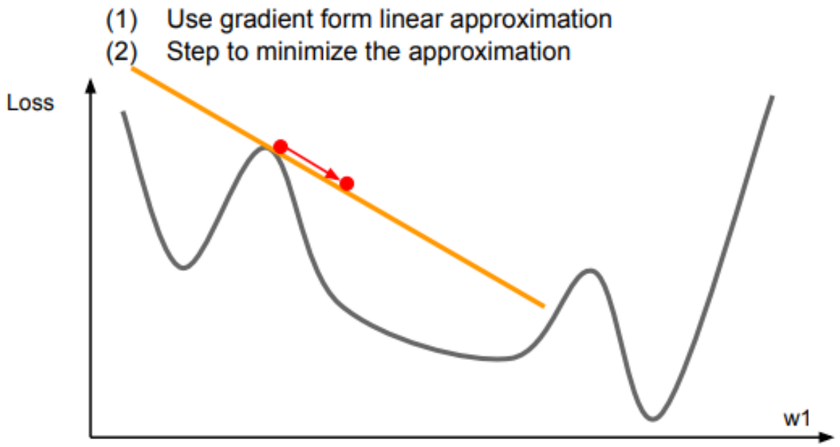
AdaGrad / RMSProp

first moment와 second moment를 이용한다.
- First moment: gradient의 가중합 ➞ velocity 담당
- Second moment: gradient의 제곱항

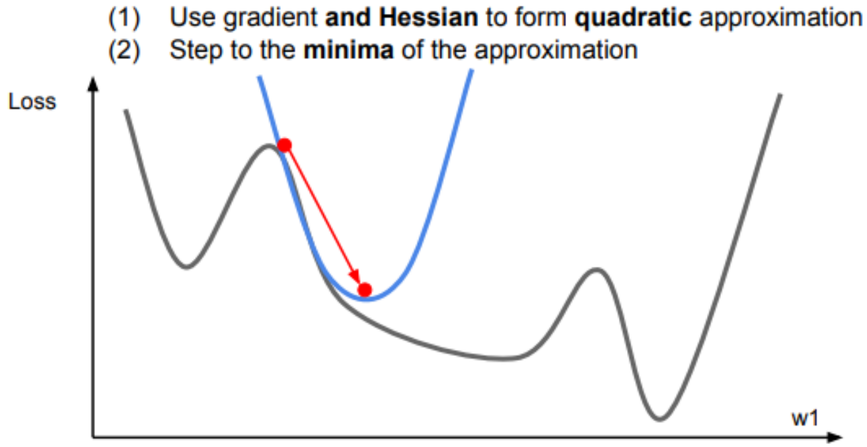
가장 많이 사용하는 최적화 방식
(beta_1 = 0.9, beta_2 = 0.999, learning_rate = 1e-3/5e-4)로 설정하면 대부분의 모델에서 잘 동작

2. Optimization: First-Order Opt. & Second-Order Opt.

First-Order Opt.



Second-Order Opt.



지금까지는 1차 미분으로 optimization ➡ 멀리 갈 수 없다는 단점

테일러 급수를 이용해 2차 근사(second-order opt.)하면?
👍 update시 learning rate을 설정해 주지 않아도 됨
👎 deep learning에서 사용불가
NxN 행렬(N은 network의 파라미터 수) ➡ 메모리에 다 저장할 수 없고 역행렬 계산도 불가

2. Optimization: First-Order Opt. & Second-Order Opt.

L-BFGS

- **Usually works very well in full batch, deterministic mode**
  i.e. if you have a single, deterministic f(x) then L-BFGS will probably work very nicely

- **Does not transfer very well to mini-batch setting**. Gives bad results. Adapting L-BFGS to large-scale, stochastic setting is an active area of research.

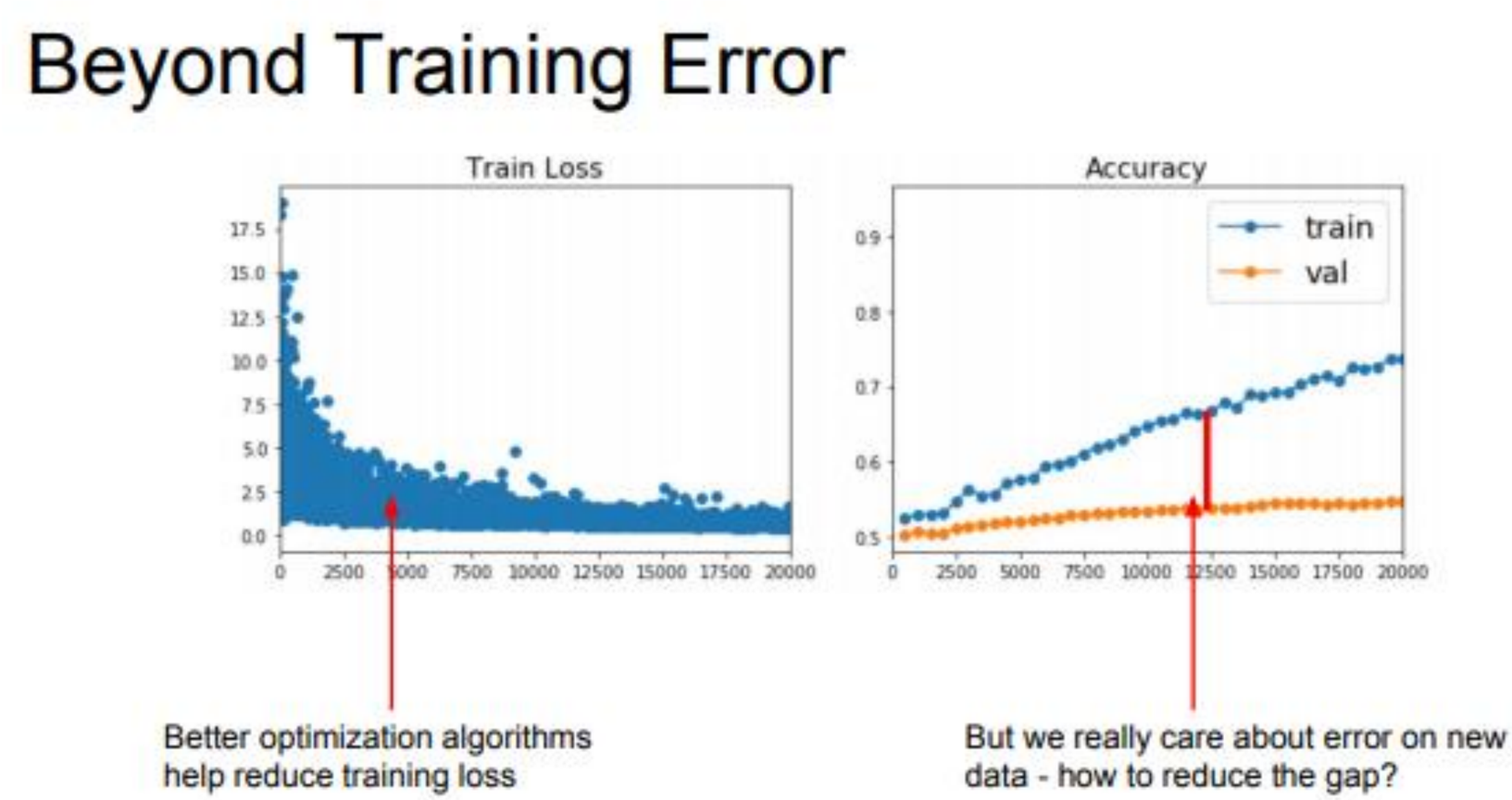
Full batch update가 가능하고 확률론적(stochastic) setting이 적은 경우에는
L-BFGS이 좋은 선택이 될 수 있다.

2. Optimization: In practice

- **Adam** is a good default choice in most cases

- If you can afford to do full batch updates then try out **L-BFGS** (and don't forget to disable all sources of noise)

2. Optimization: Regularization

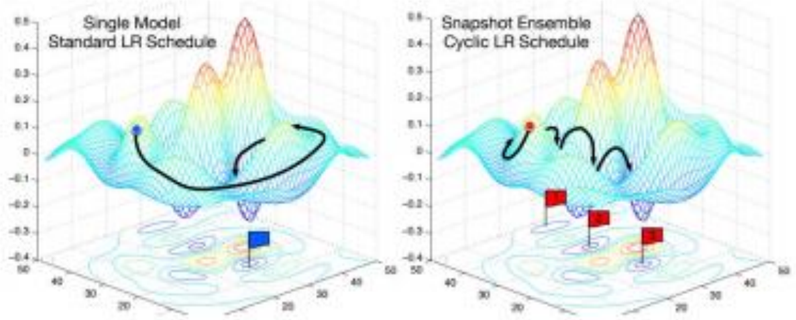


위의 내용은 Training error과 손실함수를 최소화 하기 위한 방법들이다.
그런데 우리는 이런 것들 말고도, Training error와 Test error 간의 격차를 줄이는 것도 중요하다.

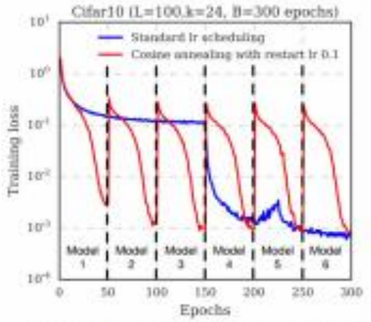
2. Optimization: Regularization

Model Ensembles: Tips and Tricks

Instead of training independent models, use multiple snapshots of a single model during training!



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!

```
while True:
  data_batch = dataset.sample_data_batch()
  loss = network.forward(data_batch)
  dx = network.backward()
  x += - learning_rate * dx
  x_test = 0.995*x_test + 0.005*x # use for test set
```

<모델 앙상블>

- 여러개의 모델을 독립적으로 학습 시킨 뒤 , 그 결과를 평균내어 테스트 타임에 이용한다.
- 학습률이 이미 높을 때 , 더 높이기 위해 사용한다.
- 파란색 선은 일반적인 학습률 스케쥴링이다. 빨간색 선은 모델 앙상블 방식으로, 트레이닝 로스가 낮아졌다, 다시 높아졌다를 반복하는 것을 볼 수 있다.
- 손실함수가 다양한 지역에 수렴할 수 있게 만들어 주는 것

2. Optimization: Regularization

Regularization: Add term to loss

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

In common use:

L2 regularization	$R(W) = \sum_k \sum_l W_{k,l}^2$	(Weight decay)
L1 regularization	$R(W) = \sum_k \sum_l \lvert W_{k,l}\rvert$	
Elastic net (L1 + L2)	$R(W) = \sum_k \sum_l \beta W_{k,l}^2 + \lvert W_{k,l}\rvert$	

➡ NN에서는 잘 사용 X

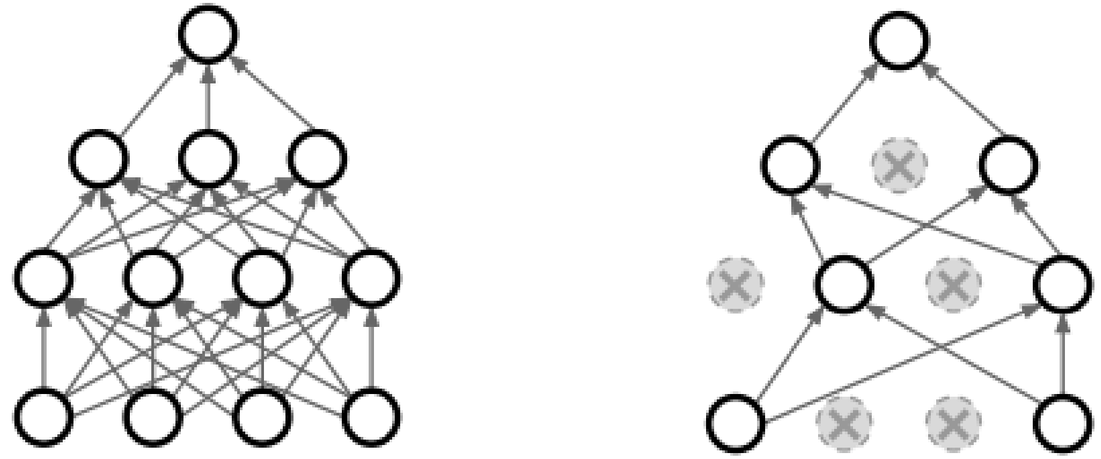
모델 앙상블은 여러 모델을 사용하여 성능을 향상시키는 방식
-> 효율 면에서 그리 좋지 않기 때문에 단일 모델을 사용하여 성능을 향상 시키는 방법이 필요

"Regularization"

2. Optimization: Regularization

Regularization: Dropout

In each forward pass, randomly set some neurons to zero
Probability of dropping is a hyperparameter; 0.5 is common



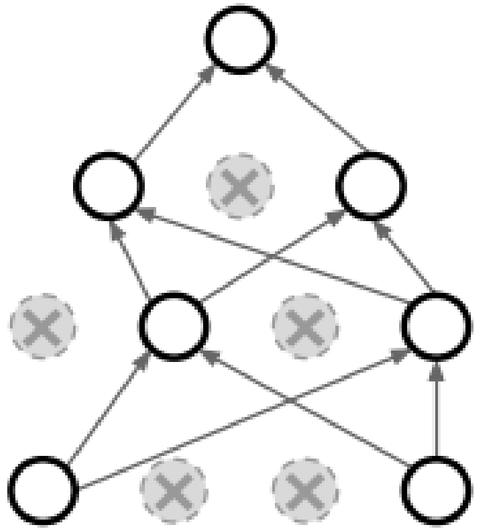
Srivastava et al. "Dropout: A simple way to prevent neural networks from overfitting". JMLR 2014

- **Dropout**이란, Forward pass 과정 중 몇몇 뉴런의 활동을 정지 시켜버리는 것이다.
- 매 pass를 진행할 때마다 정지 되는 뉴런은 계속 바뀐다.
- 레이어 단위 활동이므로, 다음 레이어로 넘어갈 때, 가중치와 곱해지는 이전 레이어의 활성화 함수 값을 0으로 만들어 버린다.

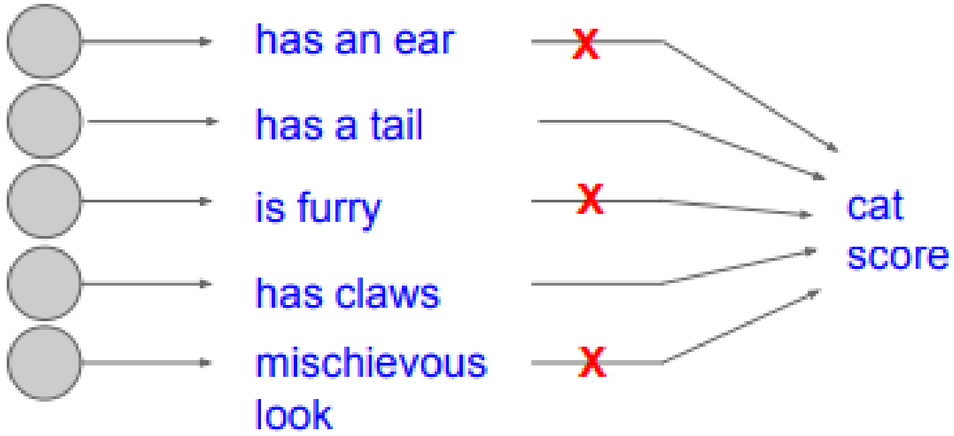
2. Optimization: Regularization

Regularization: Dropout

How can this possibly be a good idea?



Forces the network to have a redundant representation;
Prevents co-adaptation of features



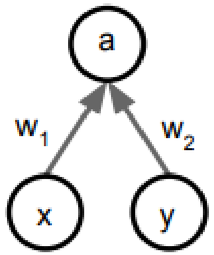
- 일부 뉴런들을 죽여버려, 노드가 어떤 일부 feature에만 의존하지 못하도록 하는 것이다.(overfitting)
- 또, 단일 모델로 앙상블 효과를 얻을 수 있다는 해석이 있다고 한다. forward pass마다 정지시키는 뉴런이 다르기에, 매번 다른 모델을 만드는 것처럼 효과가 나오기 때문이다.

2. Optimization: Regularization

Dropout: Test time

Want to approximate the integral

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



Consider a single neuron.

At test time we have: $E[a] = w_1 x + w_2 y$

During training we have:

$$E[a] = \frac{1}{4}(w_1 x + w_2 y) + \frac{1}{4}(w_1 x + 0y) + \frac{1}{4}(0x + 0y) + \frac{1}{4}(0x + w_2 y)$$

$$= \frac{1}{2}(w_1 x + w_2 y)$$

At test time, **multiply** by dropout probability

```
""" Vanilla Dropout: Not recommended implementation (see notes below) """

p = 0.5 # probability of keeping a unit active. higher = less dropout

def train_step(X):
  """ X contains the data """

  # forward pass for example 3-layer neural network
  H1 = np.maximum(0, np.dot(W1, X) + b1)
  U1 = np.random.rand(*H1.shape) < p # first dropout mask
  H1 *= U1 # drop!
  H2 = np.maximum(0, np.dot(W2, H1) + b2)
  U2 = np.random.rand(*H2.shape) < p # second dropout mask
  H2 *= U2 # drop!
  out = np.dot(W3, H2) + b3

  # backward pass: compute gradients... (not shown)
  # perform parameter update... (not shown)

def predict(X):
  # ensembled forward pass
  H1 = np.maximum(0, np.dot(W1, X) + b1) * p # NOTE: scale the activations
  H2 = np.maximum(0, np.dot(W2, H1) + b2) * p # NOTE: scale the activations
  out = np.dot(W3, H2) + b3
```

drop in forward pass

scale at test time

Dropout Summary

반면에, 테스트 타임 시의 기대값은 다르기 때문에 아까 있었던 dropout probability (0.5) 를 네트워크 출력에 곱해주어 기댓값을 같게 만들어준다.

2. Optimization: Regularization

More common: “Inverted dropout”

```
p = 0.5 # probability of keeping a unit active. higher = less dropout

def train_step(X):
  # forward pass for example 3-layer neural network
  H1 = np.maximum(0, np.dot(W1, X) + b1)
  U1 = (np.random.rand(*H1.shape) < p) / p # first dropout mask. Notice /p!
  H1 *= U1 # drop!
  H2 = np.maximum(0, np.dot(W2, H1) + b2)
  U2 = (np.random.rand(*H2.shape) < p) / p # second dropout mask. Notice /p!
  H2 *= U2 # drop!
  out = np.dot(W3, H2) + b3

  # backward pass: compute gradients... (not shown)
  # perform parameter update... (not shown)

def predict(X):
  # ensembled forward pass
  H1 = np.maximum(0, np.dot(W1, X) + b1) # no scaling necessary
  H2 = np.maximum(0, np.dot(W2, H1) + b2)
  out = np.dot(W3, H2) + b3
```

test time is unchanged!

<inverted dropout>

- 더 일반적인 방법으로, Test time의 계산 효율을 높이기 위해 p 계산 연산을 train으로 옮겨주는 것이다.
- test time시의 기댓값을 training time시의 기댓값과 같이 해주려면 트레이닝 타임의 기대값을 p로 나누어주면 된다. 보통 이렇게 많이 사용.

2. Optimization: Regularization

Regularization: A common pattern

Training: Add some kind of randomness

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

Testing: Average out randomness (sometimes approximate)

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

Example: Batch Normalization

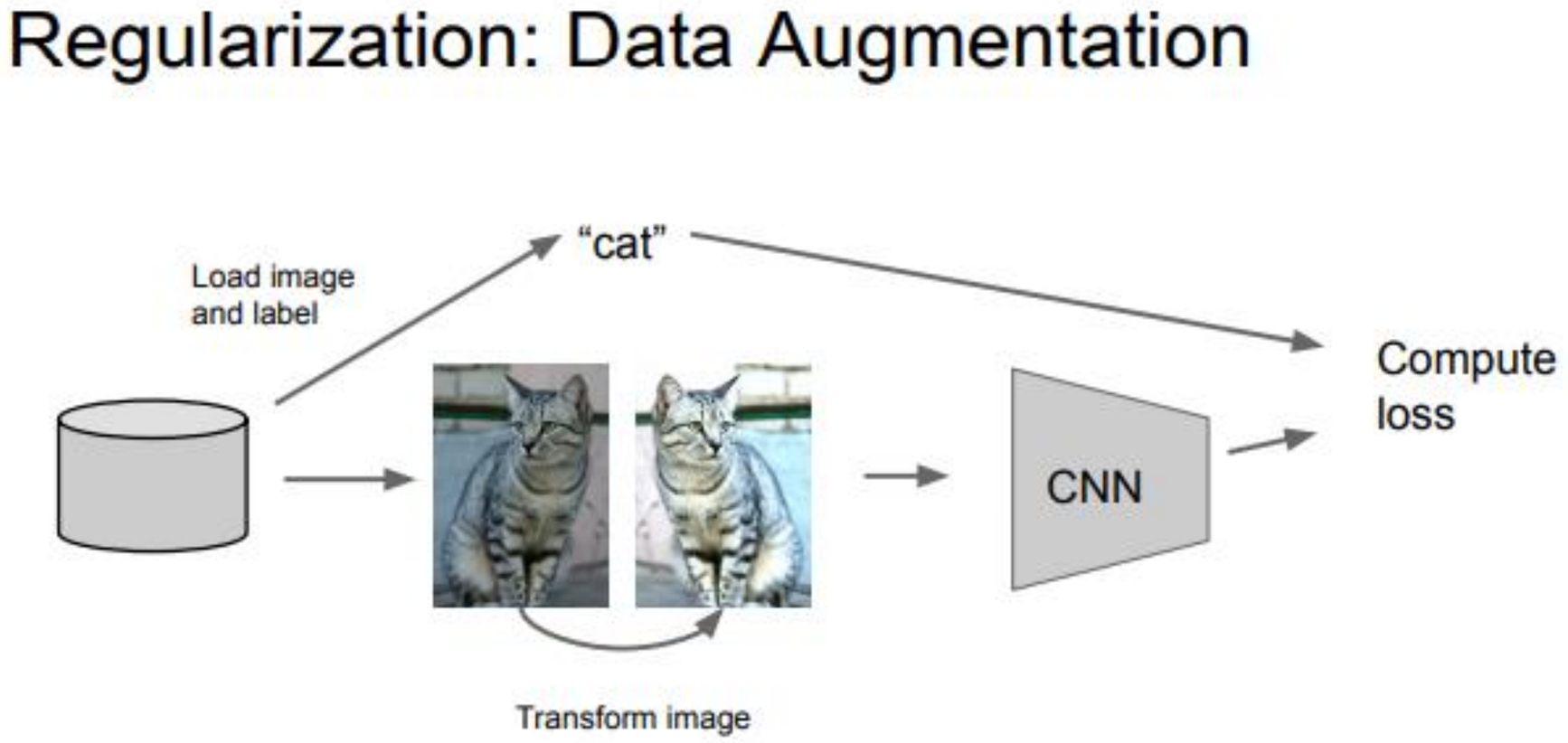
Training: Normalize using stats from random minibatches

Testing: Use fixed stats to normalize

<Batch Normalization>

- Train때 mini batch로 하나의 데이터가 샘플링 될 때 매번 서로 다른 데이터들과 만나게 됨.
- Train time에서는 각 데이터에 대해서 이 데이터를 얼마나 어떻게 정규화 시킬 것인지에 대한 stochasticity가 존재했다. 하지만 test time에서는 정규화를 mini batch가 아닌 global 단위로 수행해서 stochasticity를 평균화시킨다.
- 이런 특성 때문에 batch normalization은 regularization 효과를 얻을 수 있다.

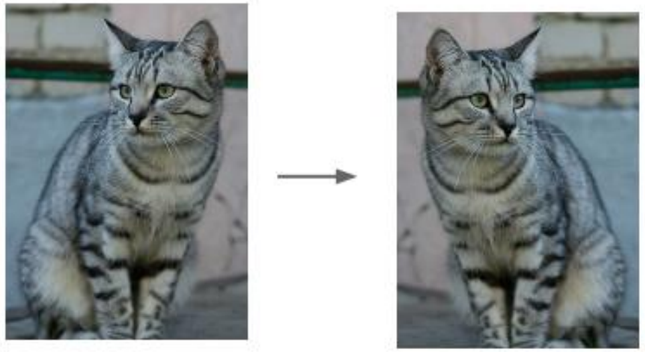
2. Optimization: Regularization



또 다른 Regularization 방법은 **data augmentation**이다.
train시에, 이미지를 약간씩 변형해봐서 무작위로 변형된 이미지를 학습하게 한다.

2. Optimization: Regularization

Data Augmentation
Horizontal Flips



Data Augmentation
Color Jitter

Simple: Randomize contrast and brightness

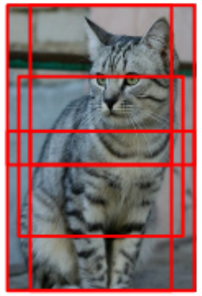
More Complex:

1. Apply PCA to all [R, G, B] pixels in training set
2. Sample a "color offset" along principal component directions
3. Add offset to all pixels of a training image

(As seen in *[Krizhevsky et al. 2012]*, ResNet, etc)

Data Augmentation
Random crops and scales

Training: sample random crops / scales
ResNet:
1. Pick random L in range [256, 480]
2. Resize training image, short side = L
3. Sample random 224 x 224 patch



Testing: average a fixed set of crops
ResNet:
1. Resize image at 5 scales: {224, 256, 384, 480, 640}
2. For each size, use 10 224 x 224 crops: 4 corners + center, + flips

고양이 사진을 반전시켜도 보고, 잘라보기도 하고, 밝기를 변형해 색감도 바꾸어보고, 여러 방식으로 바꾼 고양이 사진을 학습시키는 것이다.

하나의 이미지를 그냥 사용했을 때와 이렇게 10개(4개의 코너 + 중앙 이미지에다가 이게 그냥 일 때와 반전일 때 2가지가 있으니까 5 * 2)의 성능을 비교

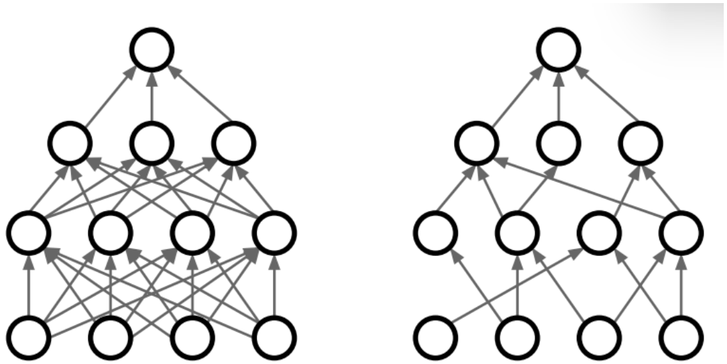
2. Optimization: Regularization

Data Augmentation 예제

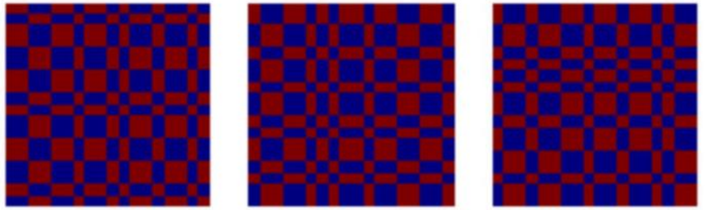
출처 - https://github.com/aleju/imgaug

2. Optimization: Regularization

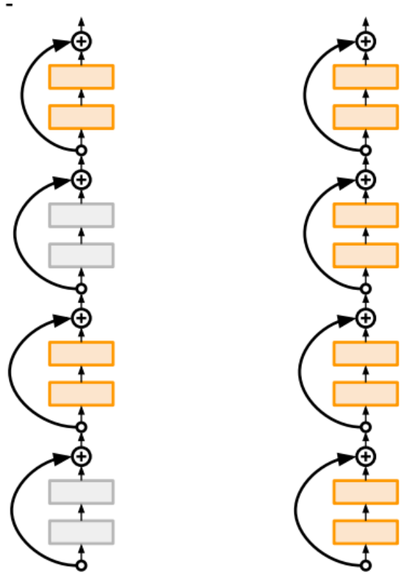
Others



<DropConnect>
activation 값을 0으로 만들었던 dropout과 반대로 weight matrix 값을 0으로 만들어주는 방식이다.

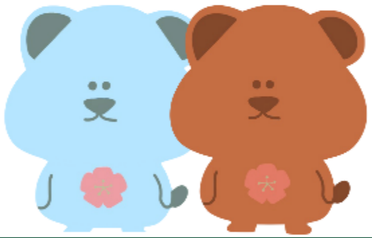


<Fractional max pooling>
풀링 연산 지역을 임의로 선정하여 진행하는 방식이다.



<Stochastic Depth>
위 사진과 같이 트레이닝 시에 네트워크 레이어를 랜덤하게 drop한다. 테스트 시에는 전체 네트워크를 모두 사용한다. dropout과 비슷한 효과가 있다고 한다.

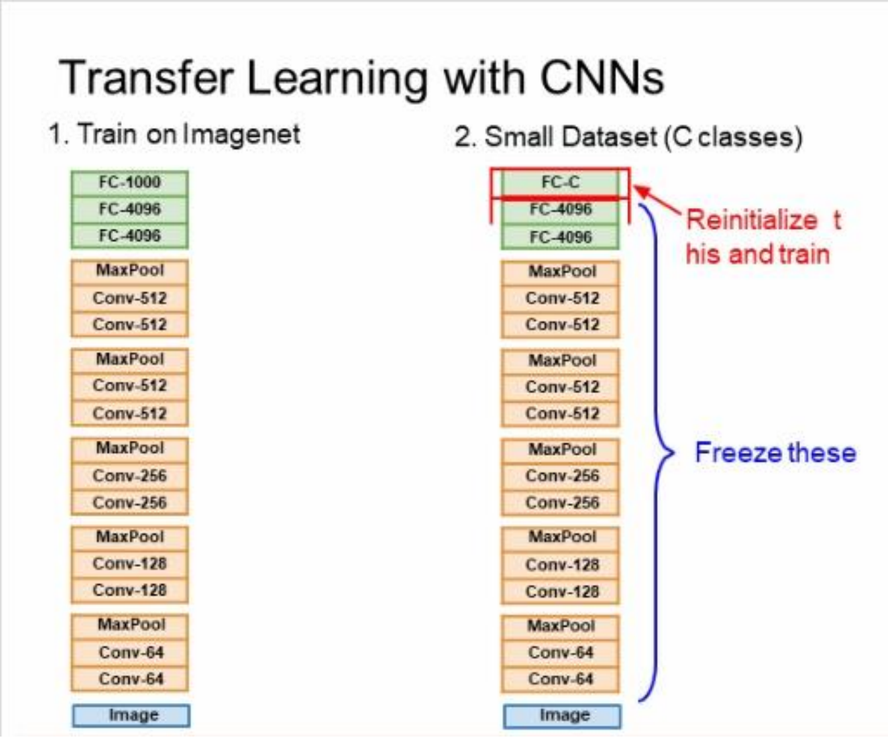
Transfer Learning



3. Transfer Learning

Trasfer Learning (전이학습)이란, 작은 데이터셋으로도 CNN을 학습할 수 있는 방법이다.
기존의 만들어진 모델을 사용하여 새로운 모델을 만들기 때문에, 학습이 빨라지며, 예측도 높아진다.

3. Transfer Learning



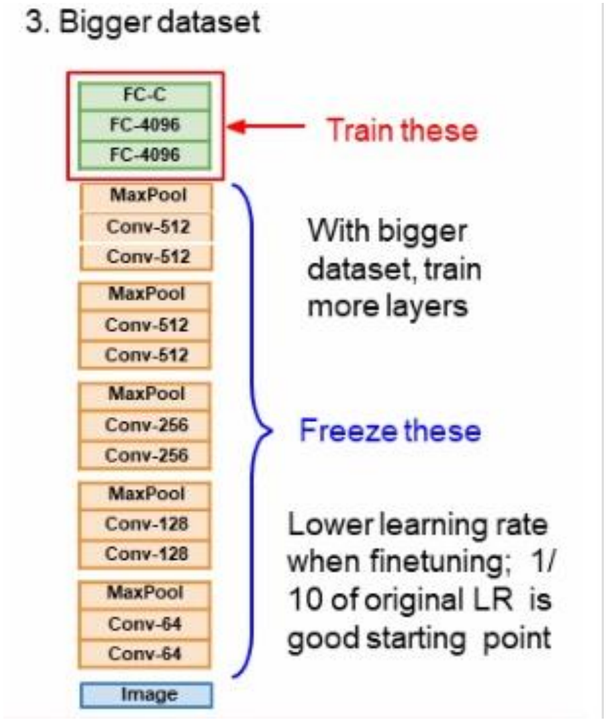
Ex. Imagenet에서 1000개의 카테고리를 분류하도록 하였다면 이제는 10종의 강아지를 분류하는 것

일반적인 절차는 가장 마지막의 FC Layer는 최종 feature와 class scores간의 연결인데 이걸 초기화 시켜주는 것이다.
기존의 ImageNet을 학습시킬 때는 4096 * 1000 차원의 행렬이었는데 이걸 이제 우리는 4096 * 4(혹은 클래스 개수 C)로 해준다.

그리고 가중치 방금 정의한 가중치 행렬은 초기화시키고 이전의 나머지 레이어들의 가중치는 freeze시킨다. 즉 우리는 초기화한 레이어만 재사용하는 것. 즉, 마지막 레이어만 가지고 데이터를 학습시키게 되는 것이다.

1. 우선 ImageNet과 같은 아주 큰 데이터셋으로 학습을 시킨다.
2. 훈련된 모델을 우리가 가진 작은 데이터셋에 적용시킨다.

3. Transfer Learning



Fine-tuning 이란?

•기존에 학습되어져 있는 모델을 기반으로, 아키텍쳐를 새로운 목적에 맞게 변형하고, 이미 학습된 모델의 파라미터를 업데이트 하는 방법을 말한다.

• " Fine-tuning 했다 " 라고 말하려면, 기존에 학습 된 layer에 데이터를 추가로 학습시켜, 파라미터를 업데이트 해야한다.

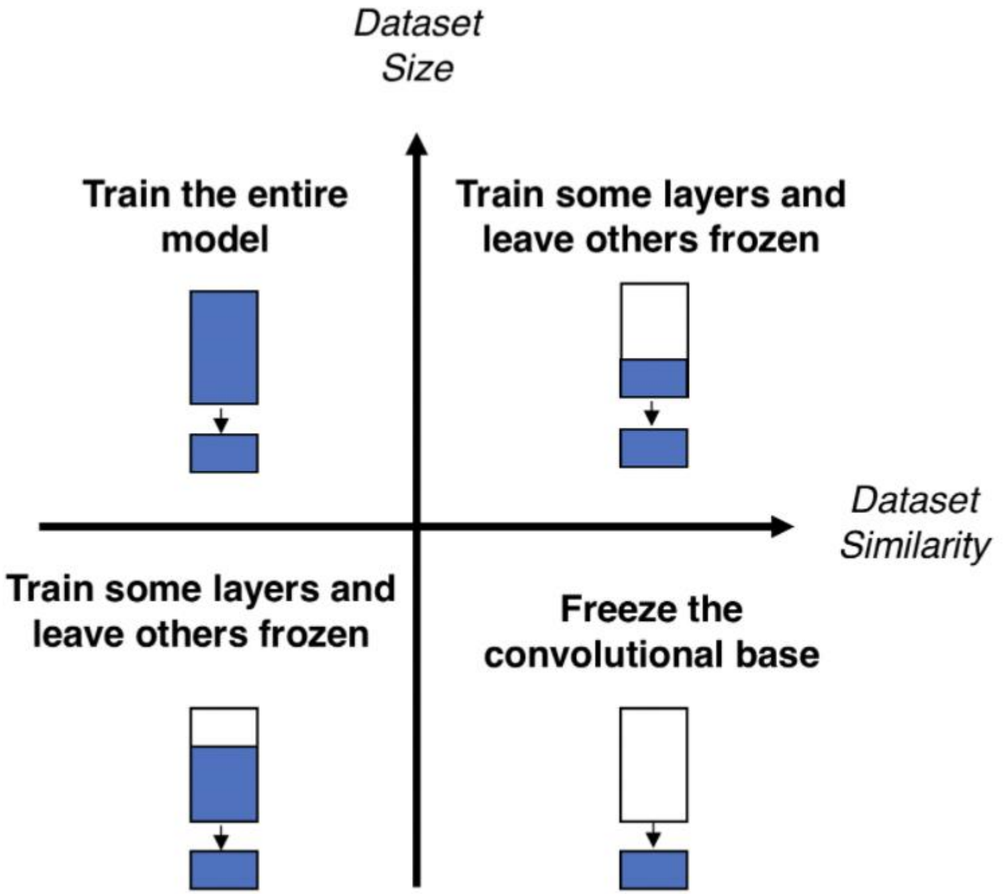
•이 때 주의할 점은, 완전히 랜덤한 초기 파라미터를 쓴다거나, 가장 아래쪽의 레이어의 파라미터를 학습하게 되면 오버피팅이 일어나거나 전체 파라미터가 망가지는 문제가 생긴다.

3. 데이터가 조금 많이 있다면 좀 더 위의 layer로 이동해서 fine tuning할 수 있다.

이때는 왜냐면 이미 잘 학습이 되어 있기 때문에 learning rate값을 조금 낮춰서 해주면 좋다.

3. Transfer Learning

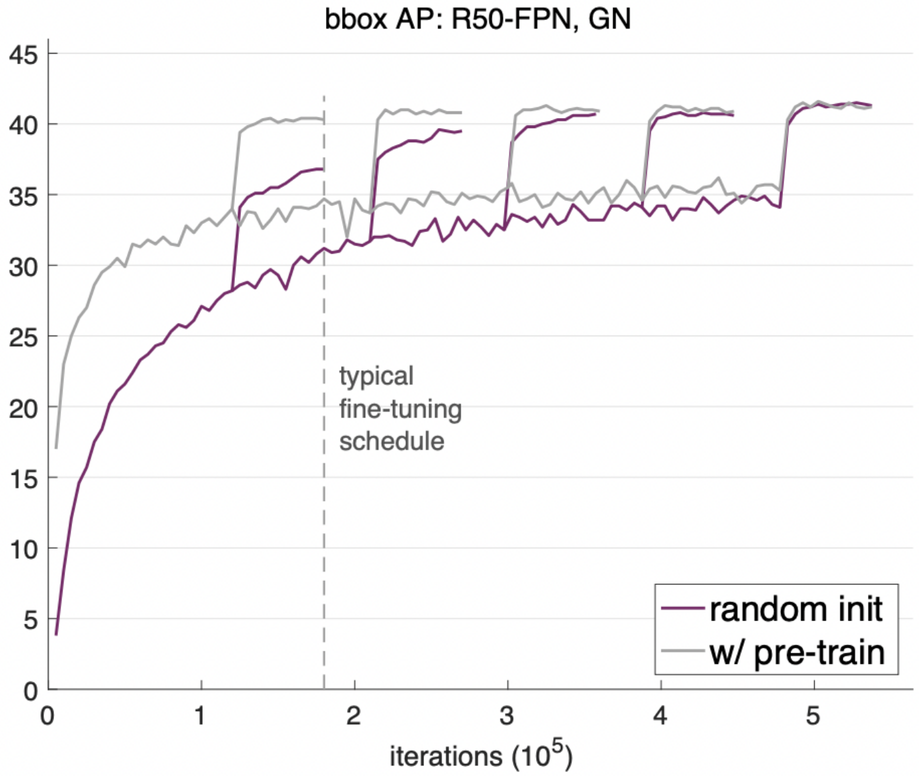
- **새로 훈련할 데이터가 적지만, original 데이터와 유사할 경우**
  - 데이터의 양이 적어, 전체 네트워크를 fine-tuning 하는 것은 over-fitting의 위험이 있기에 하지 않습니다.
  - 새로 학습할 데이터는 original 데이터와 유사하기 때문에
  - 이 경우 최종 linear classfier layer만 학습합니다.
- **새로 훈련할 데이터가 적으며, original 데이터와 다른 경우**
  - 이 경우 데이터의 양이 적고, 데이터가 서로 다르기 때문에, 네트워크의 마지막부분만 학습하는 것은 좋지 않습니다.
  - 따라서, 네트워크 초기 부분 어딘가 activation 이후에 특정 layer를 학습시키는게 좋습니다.
- **새로 훈련할 데이터가 많으며, original 데이터와 유사할 경우**
  - 새로 학습할 데이터의 양이 많다는 것은, over-fitting의 위험이 낮다는 뜻이므로
  - 전체 layer를 fine-tuning을 하거나
  - 마지막 몇 개의 layer만 fine-tuning하거나
  - 마지막 몇 개의 layer를 날려버리는 방법이 있습니다.
- **새로 훈련할 데이터가 많지만, original 데이터와와 다른 경우**
  - 데이터가 많기 때문에, 아예 새로운 ConvNet을 만들수도 있지만,
  - 실적적으로 transfer learning의 효율이 더 좋습니다.
  - 따라서, 전체 네트워크를 fine-tuning 해도 됩니다.



출처-https://wooono.tistory.com/230

3. Transfer Learning

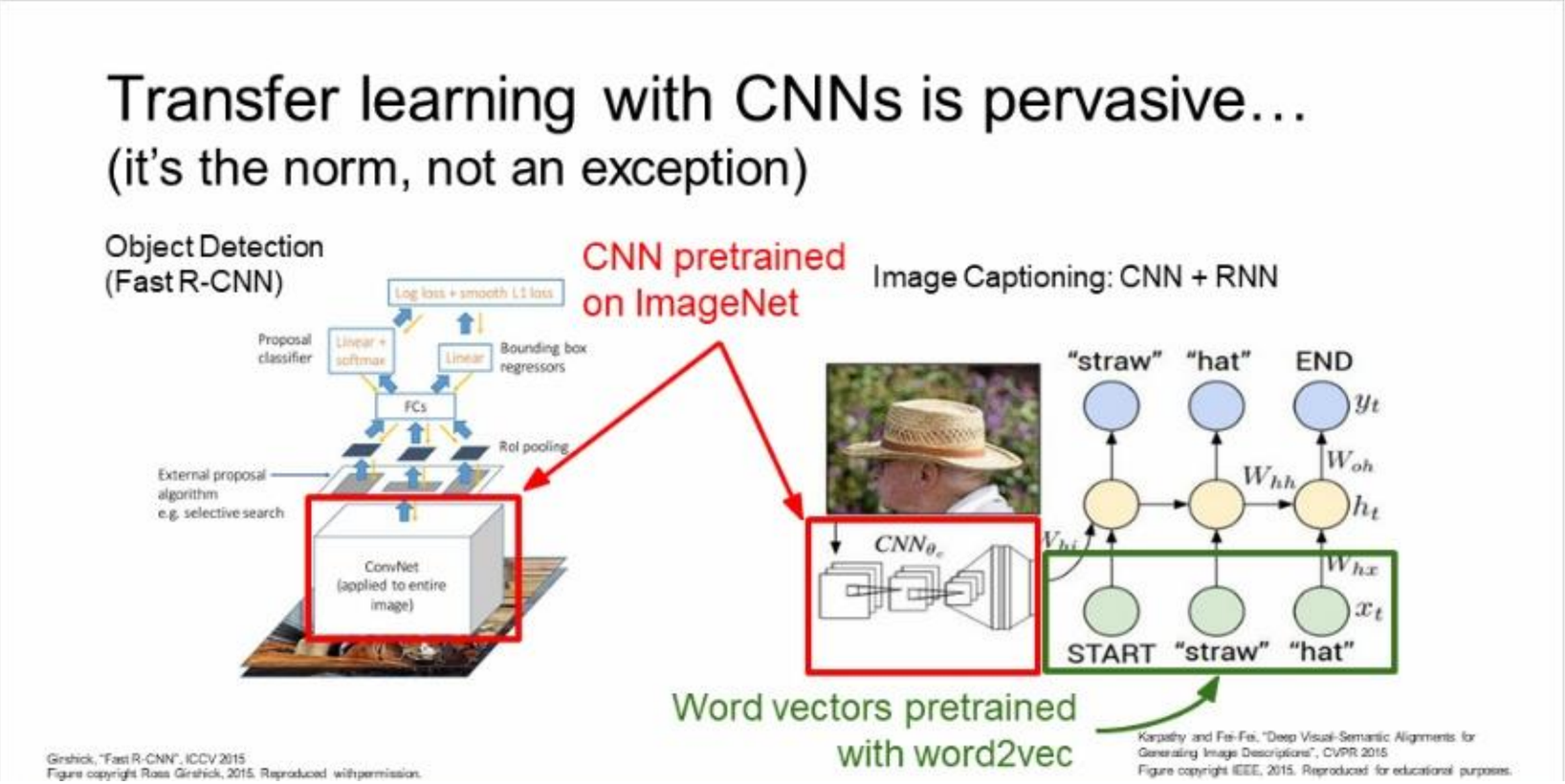
실제로 효과적인가?



- 2018년 FAIR(Facebook AI Research) 논문에서 실험을 통해 '전이학습이 학습 속도 면에서 효과가 있다'라는 것을 밝혀냄.
- 논문에선 동일한 조건에서 바닥부터 훈련시키는 것(train from the scratch)이 사전 학습된 모델을 사용하는 것보다 2~3배의 시간이 더 걸린다는 것을 보여주고 있다.
- 그림에서 랜덤으로 모델을 초기화를 시키는 것보다 사전 학습된 모델의 초기값을 사용하였을 때 더 빠르게 높은 정확도에 도달한다는 것을 보여주고 있다.

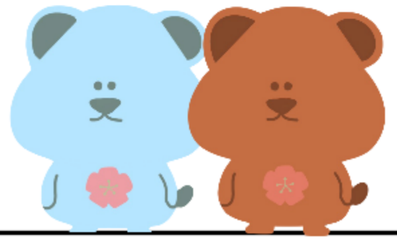
Rethinking ImageNet Pre-training [출처 : arxiv]

3. Transfer Learning



- object detection과 image captioning는 처음에 CNN으로 시작한다.
- 그리고 컴퓨터 비전(computer vision)관련 알고리즘들은 요즘 모델을 밑바닥부터 학습하지 않는다.
- 대부분 pretrained-model을 사용하고 우리 task에 맞게 fine-tunin하는 것.
- Captioning의 경우 word vectors를 pretrain하기도 한다.

THANK YOU



EWHA EURON