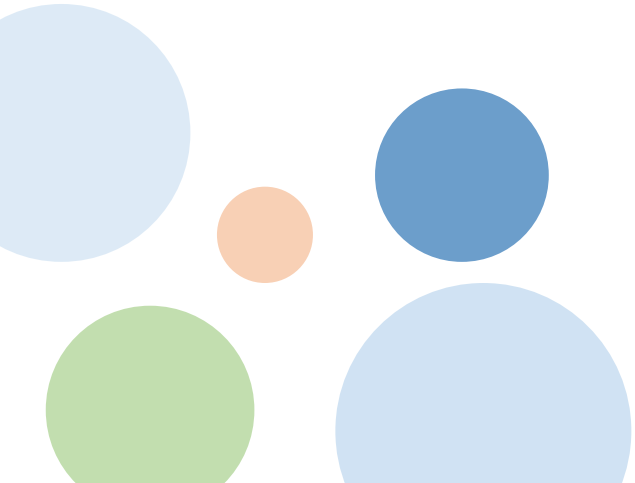
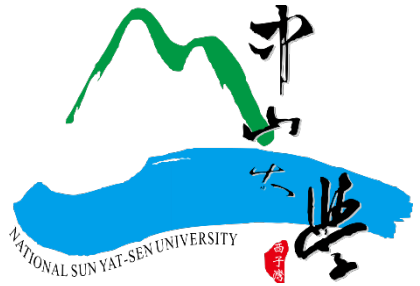
Training of Neural Network

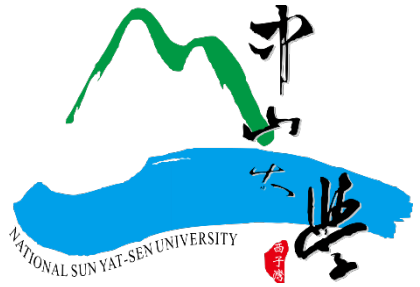




Outlines

- Optimization Algorithms
- Regularization
- Model Compression and Quantizatiion

Optimization Algorithms



Training

- Machine Learning = Training (Learning) + Inference (Testing, Prediction)
- Training = forward propagation + backward propagation
  - determine weights in the models
- Inference = forward propagation
  - use trained weights to predict
- Mini-batch SGD training

  loop:

  1. **Sample** a mini-batch of data
  2. **forward** propagate it through network, and get loss
  3. **backward** propagate to calculate gradients
  4. **update** parameters using gradients

Deep Neural Network

- neural network model
  - different model structures (CNN, RNN, GAN, ..) and number of layers d

$$f(x; w_{1,2,\cdots,d}) = f_d(f_{d-1}(\cdots f_2(f_1(x; w_1), w_2)\cdots, w_{d-1}), w_d)$$

- learning the model's weights by minimizing loss function L

$$w^* = \underbrace{\arg\min}_{w} \sum_{(x,y)\subseteq(X,Y)} L(y, f_d(x; w_{1,2,\cdots,d}))$$

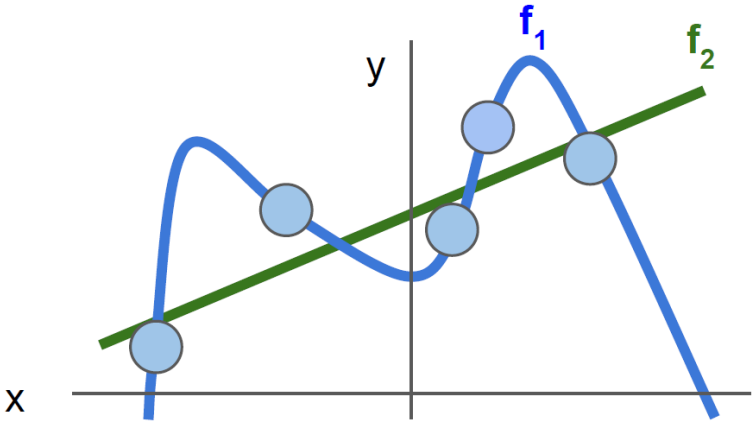
- optimizing with stochastic gradient descent (SGD)

$$w(t+1) = w(t) - \varepsilon \nabla_w L(y, w)$$

Pure Optimization vs. Machine Learning Training

- Pure Optimization
  - Find the optimal solution (usually in explicit forms) based on all the data points
  - e.g. least square (LS) estimation in normal equation

$$w^* = (X^T X)^{-1} X^T y$$



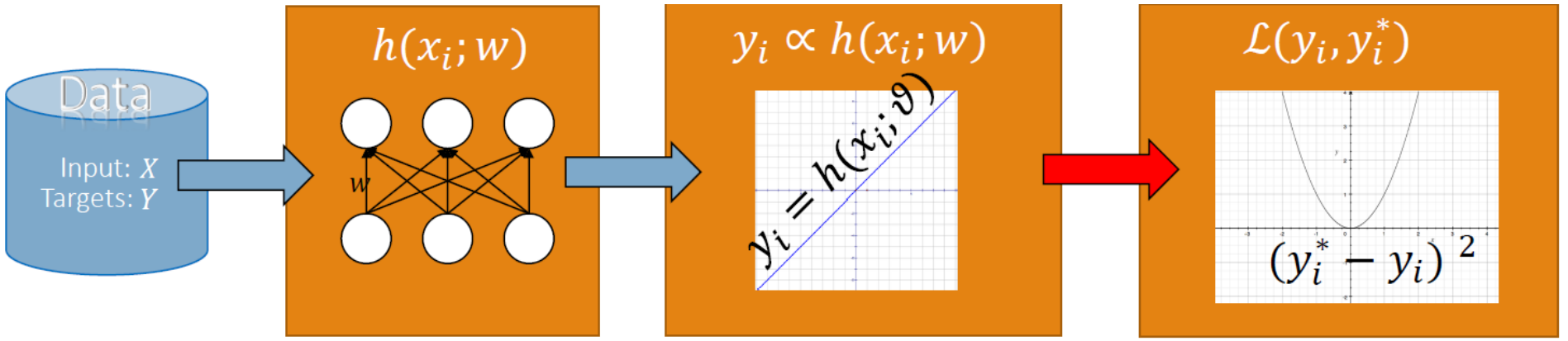
- Machine Learning Training
  - Find optimal parameters based on the training data points
  - not necessarily could be **generalized** to "new" data points

$$w^* = \arg\min_{w} \sum_{(x,y)\subseteq(X,Y)} L(y, f_d(x; w_{1,2,\cdots,d}))$$

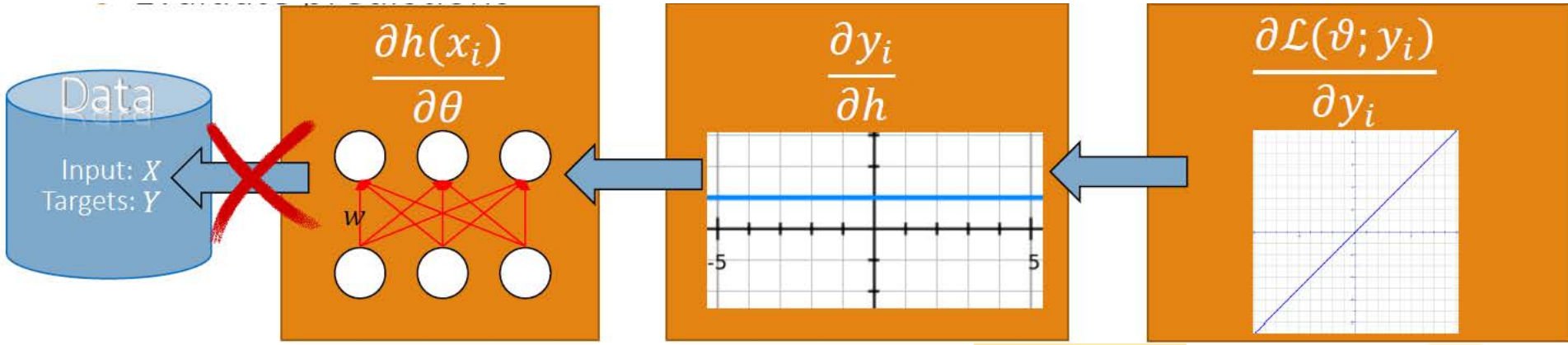
$$L(x, y, w) = \|\hat{y} - y\|^2 + \lambda \|w\|^2$$

Forward and Backward Propagation

- forward
  - collect annotated data
  - define model
  - score/predict/output
  - objective/loss/cost



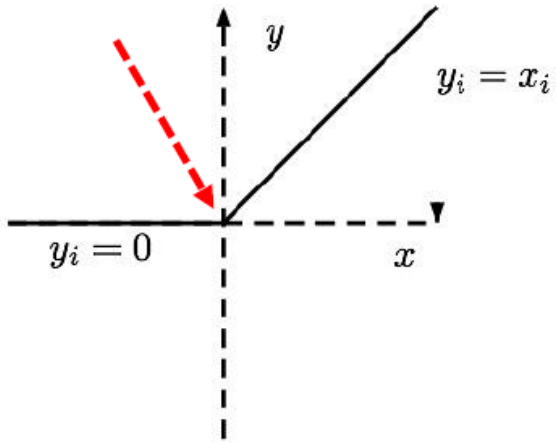
- backward
  - compute gradients backward
  - update model parameters



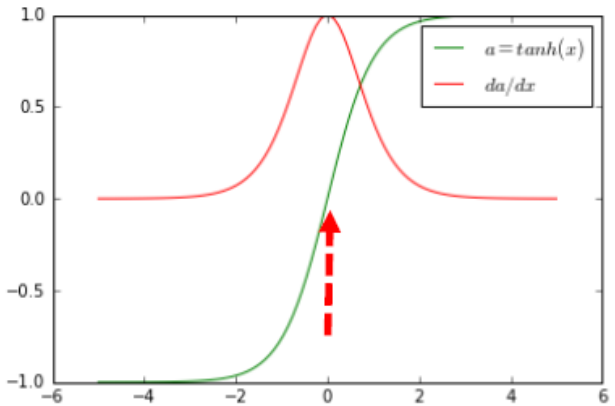
data pre-processing

- center data to be roughly zero
  - activation functions usually centered around 0
  - convergence usually faster (most non-linearities have high gradient near center)
- when input dimensions have similar ranges, and with right non-linearity
  - centering might be enough
  - e.g., in images all dimensions are pixels. and all pixels have more or less the same range
  - just make sure that images have mean zero by subtracting the average

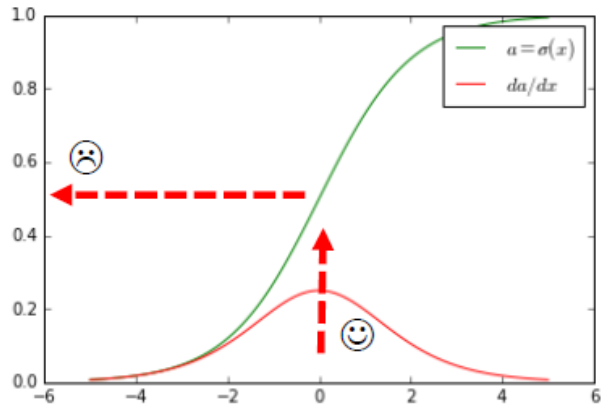
ReLU ☺



$\tanh(x)$ ☺

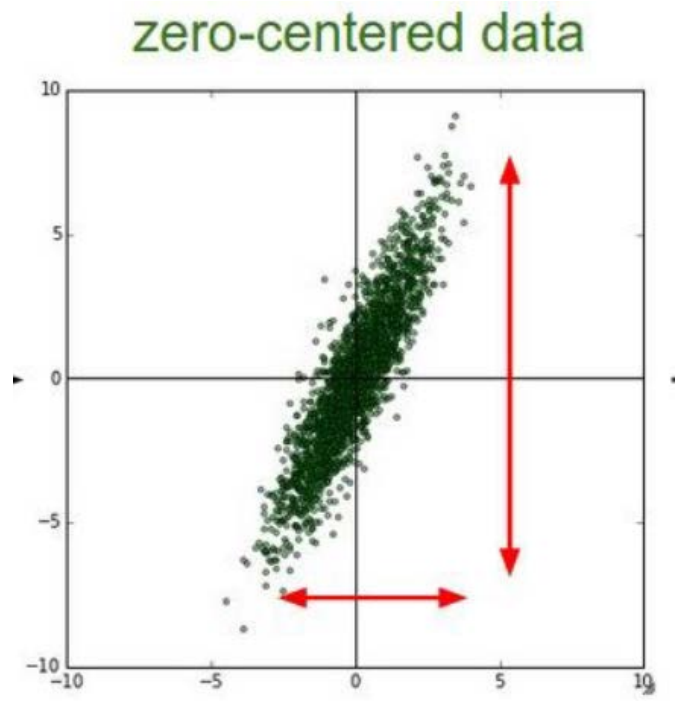


$\sigma(x)$ ☹

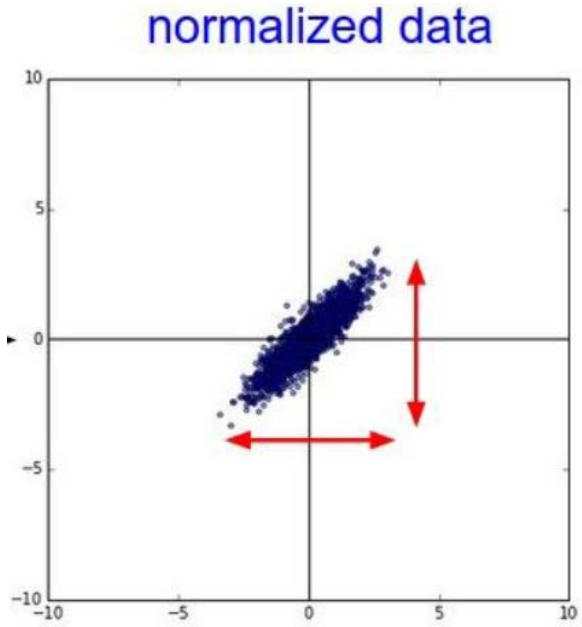


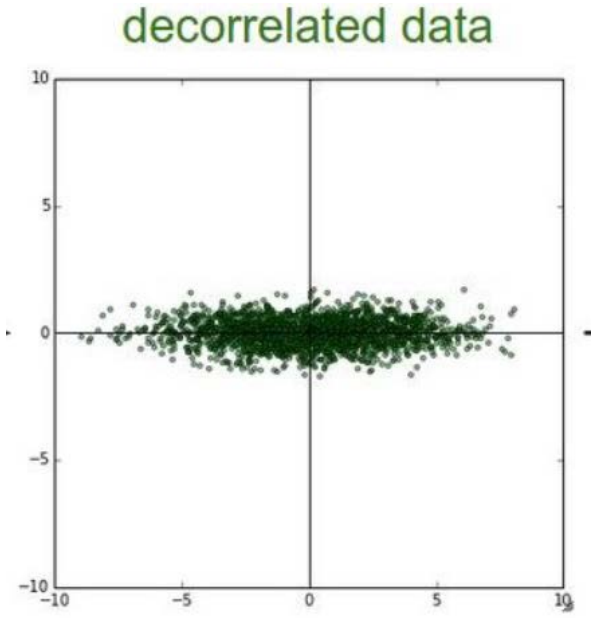
Centered Data is Common

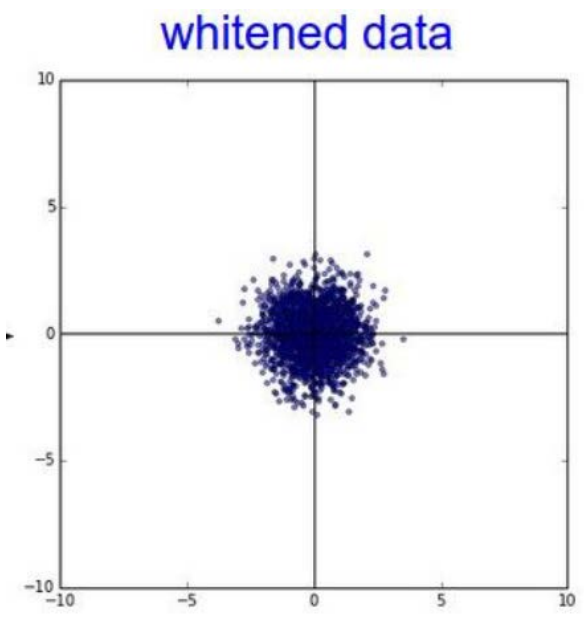
- consider CIFAR-10 example with [32, 32, 3] images
  - subtract the mean image (e.g., AlexNet)
  - subtract per-channel mean (e.g., VGG)
    - mean along each channel = 3 numbers



- normalize variance, do PCA, or whitening is NOT common

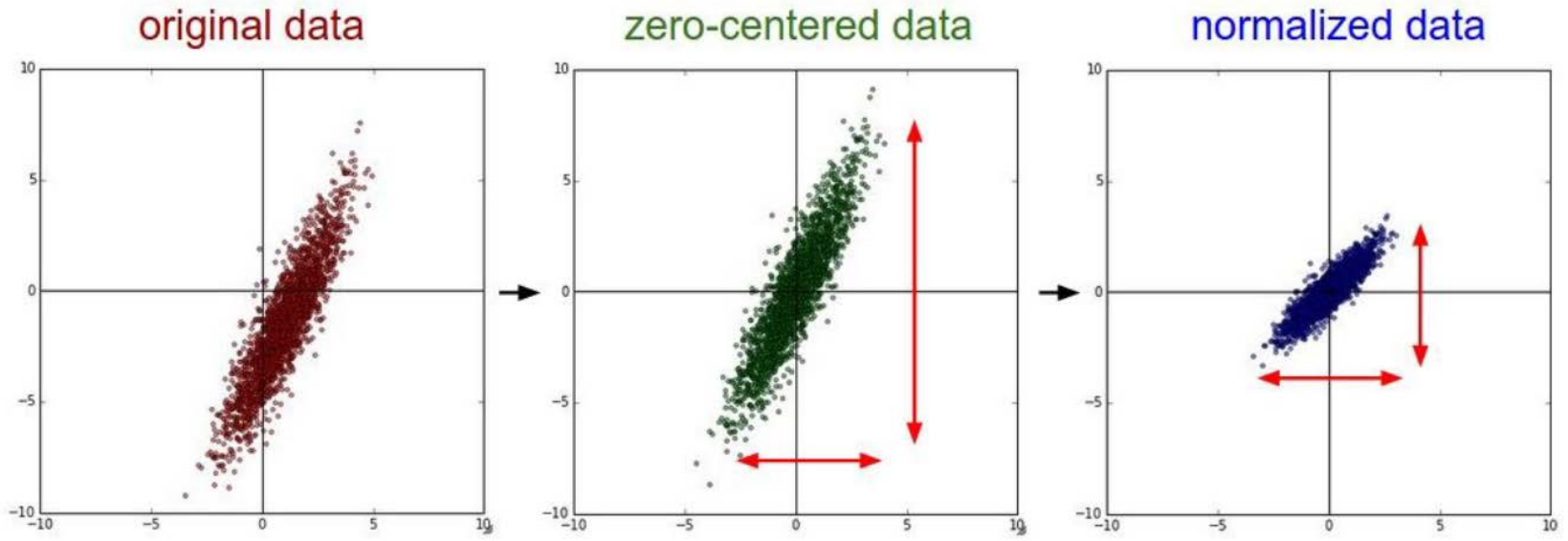




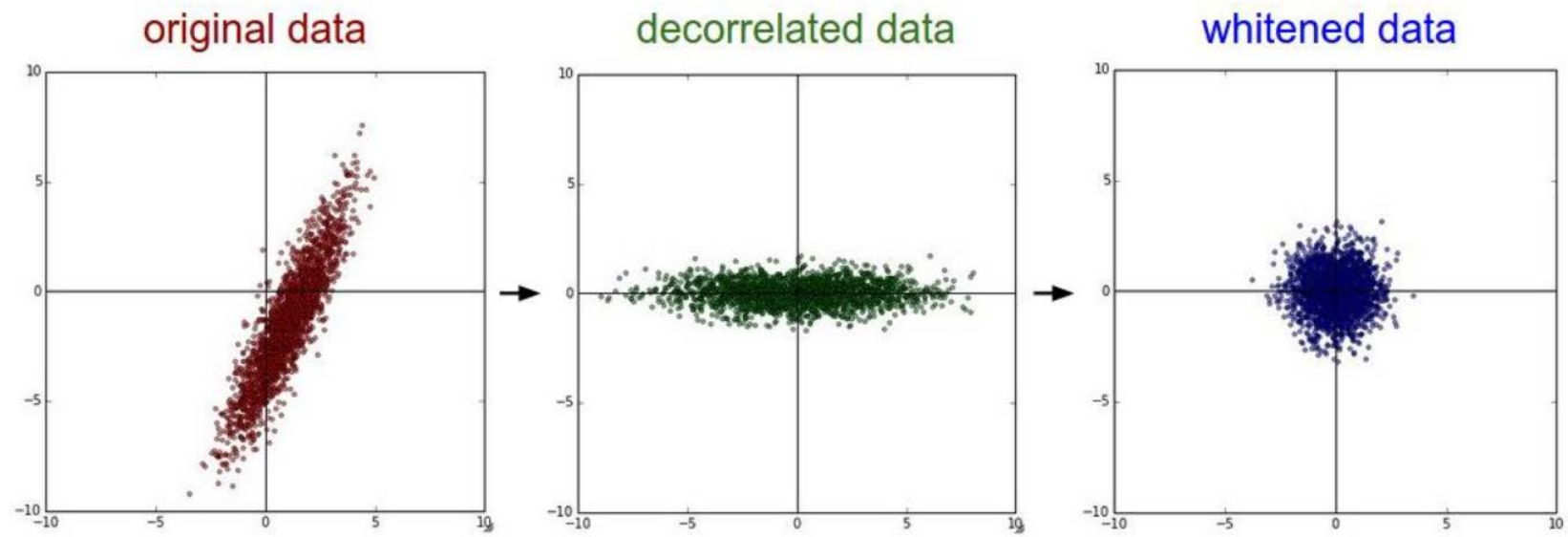


Data Preprocessing

- zero-centered
  - subtract mean
- normalized
  - divided by standard deviation



- de-correlated
  - covariance is diagonal
- whitened
  - covariance is a scaled identity matrix



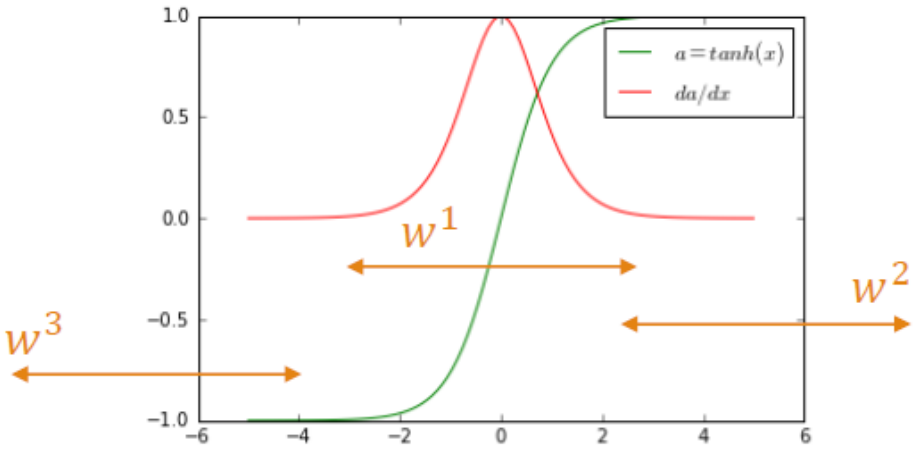
(data has diagonal covariance matrix)

(covariance matrix is the identity matrix)

data augmentation: Similar Covariance

- scale input variables to have similar diagonal covariance
  - similar covariance -> more balanced rate of learning for different weights
  - e.g. (below) x1, x2, x2 have much different covariances -> much different gradients w.r.t. w1, w2, w3
  - but learning rate is applied to all the gradients

$x = [x_1, x_2, x_3]^T, w = [w_1, w_2, w_3]^T, a = \tanh(w^T x)$



x_1, x_2, x_3 → much different covariances

↓

Generated gradients $\left.\frac{d\mathcal{L}}{d\theta}\right|_{x_1,x_2,x_3}$: much different

↓

Gradient update harder: $w_{t+1} = w_t - \eta_t \begin{bmatrix} d\mathcal{L}/dw_1 \\ d\mathcal{L}/dw_2 \\ d\mathcal{L}/dw_3 \end{bmatrix}$

data augmentation: De-correlated

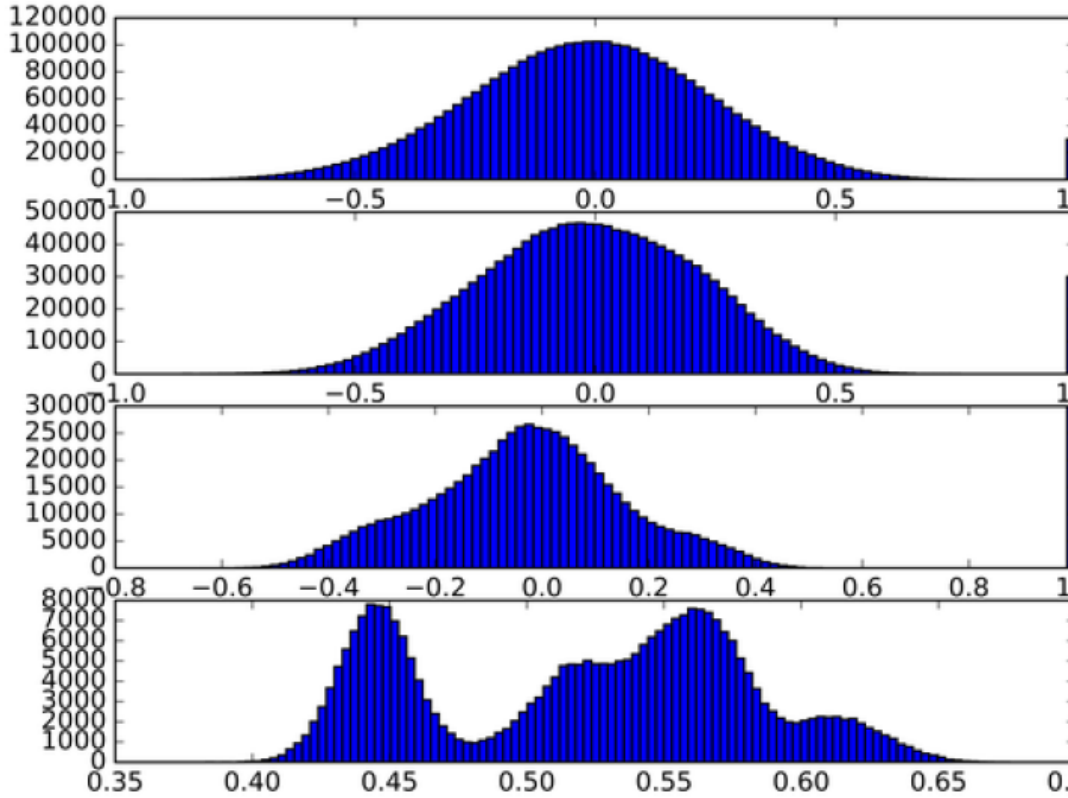
- input variables should be as de-correlated as possible
  - input variables are more “independent”
  - model is forced to find **non-trivial** connections among inputs
- de-correlated inputs -> better optimization
- unit normalization
  - input follows a Gaussian zero mean=0 and variance=1
    - from training set, compute mean and standard deviation
    - subtract the mean from training samples
    - divide the result by standard deviation

Weight Initialization

- small random numbers

```
W = 0.01* np.random.randn(D,H)
```

  - Gaussian with zero mean and 1e-2 standard deviation
  - OK for small networks, but problems with deeper networks
- transfer learning
  - initialization from pre-trained models
  - discussed later

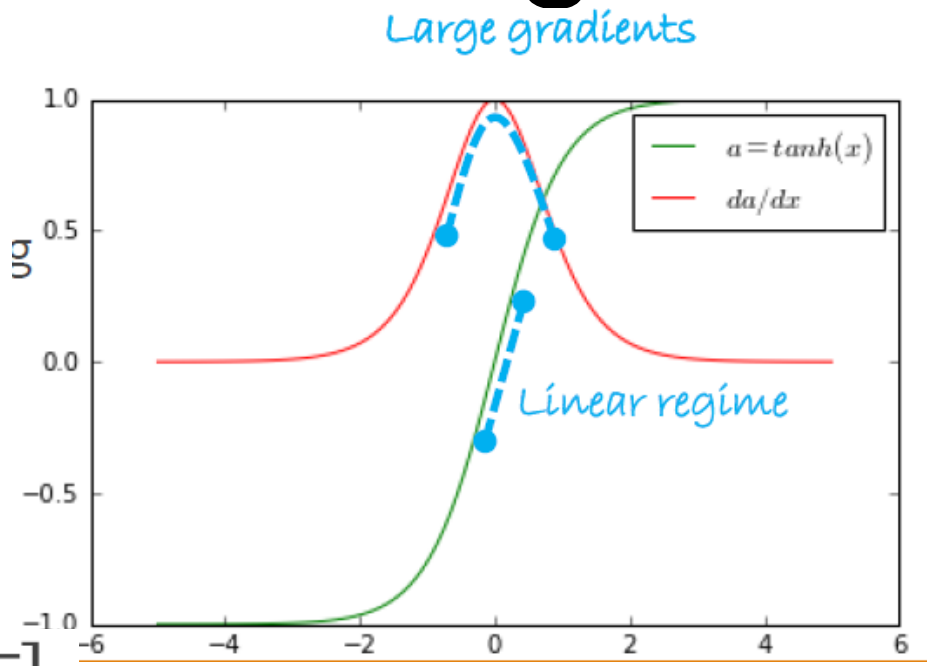


preserve variance of activations

- weights must be initialized to preserve the variance of the activations during forward and backward computations
  - all neurons operate in their full capacity (output range)
- **input variance = output variance**
  - because the output of one layer is the input of another
- good practice: initialize weights to be asymmetric
  - DO NOT give saved values to all weights (like all zeros), because in this case, all neurons generate same gradient -> no learning
- generally speaking, initialization depends on
  - non-linear functions
  - data normalization

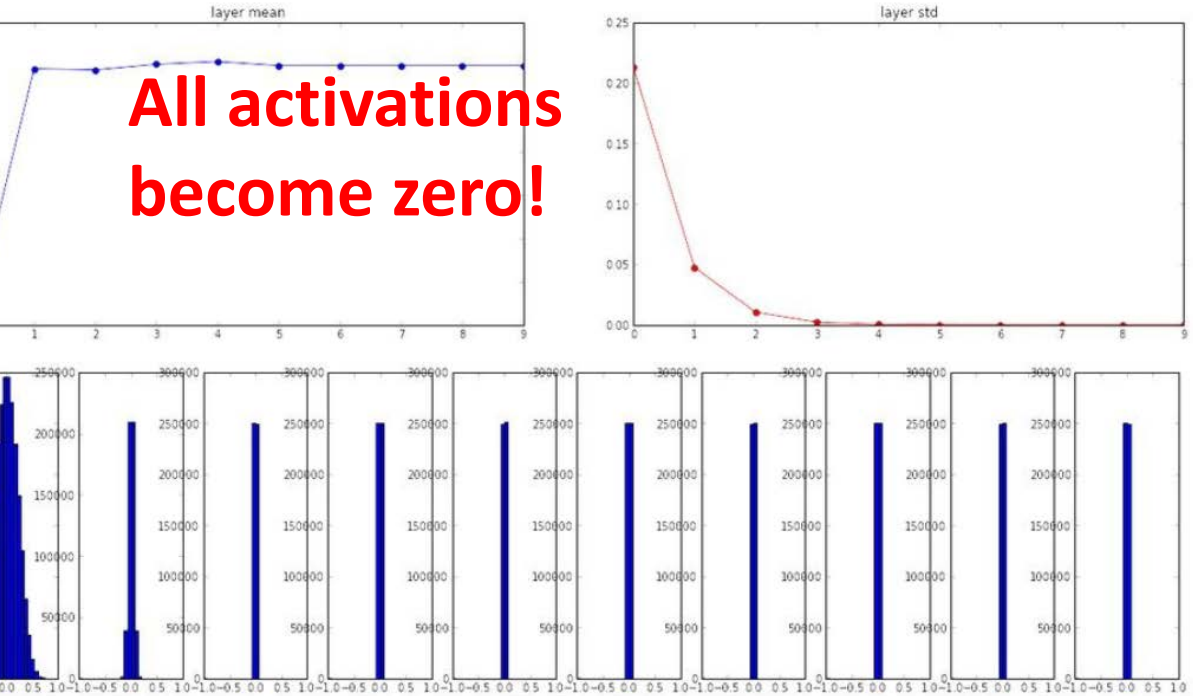
contradictory requirements for weights

- weights need to be **small** enough
  - around origin for symmetric functions (tanh, sigmoid)
  - when training starts, better stimulate activation functions near their linear regime
  - large gradients -> faster training
- weights need to be **large** enough
  - otherwise, signal is too weak for any serious learning
- for tanh, initialize weights from $\left[-\sqrt{\frac{6}{d_{l-1}+d_l}}, \sqrt{\frac{6}{d_{l-1}+d_l}}\right]$
  - d_{l-1}, d_l are respectively the numbers of input and output variables
- for sigmoid, initialize weights from $\left[-4\cdot\sqrt{\frac{6}{d_{l-1}+d_l}}, 4\cdot\sqrt{\frac{6}{d_{l-1}+d_l}}\right]$



Examples of Activation Statistics

- consider 10-layer ANN with 500 neurons/layer, with tanh, and weight initialization W=**0.01***np.random.randn(D,H)



All activations become zero!

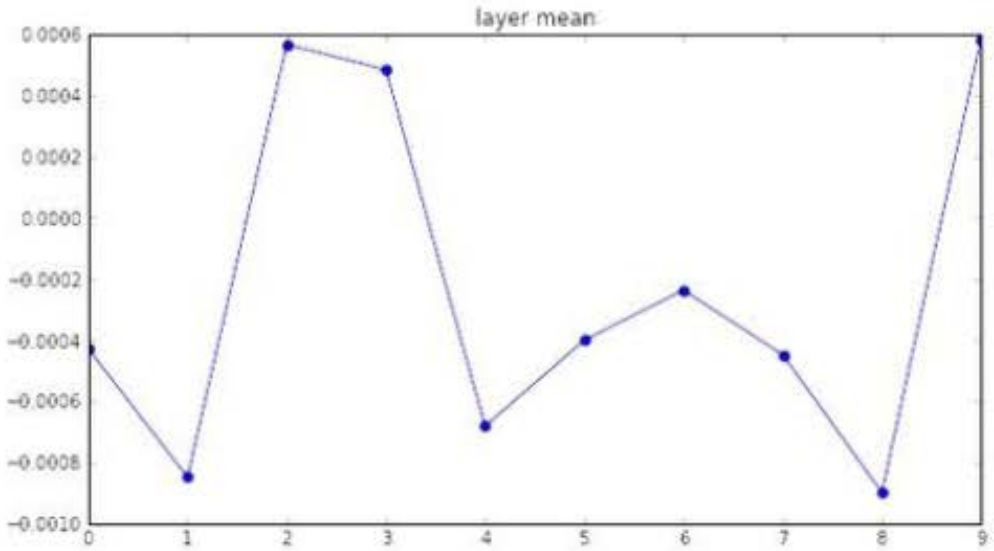
```
# assume some unit gaussian 10-D input data
D = np.random.randn(1000, 500)
hidden_layer_sizes = [500]*10
nonlinearities = ['tanh']*len(hidden_layer_sizes)
```

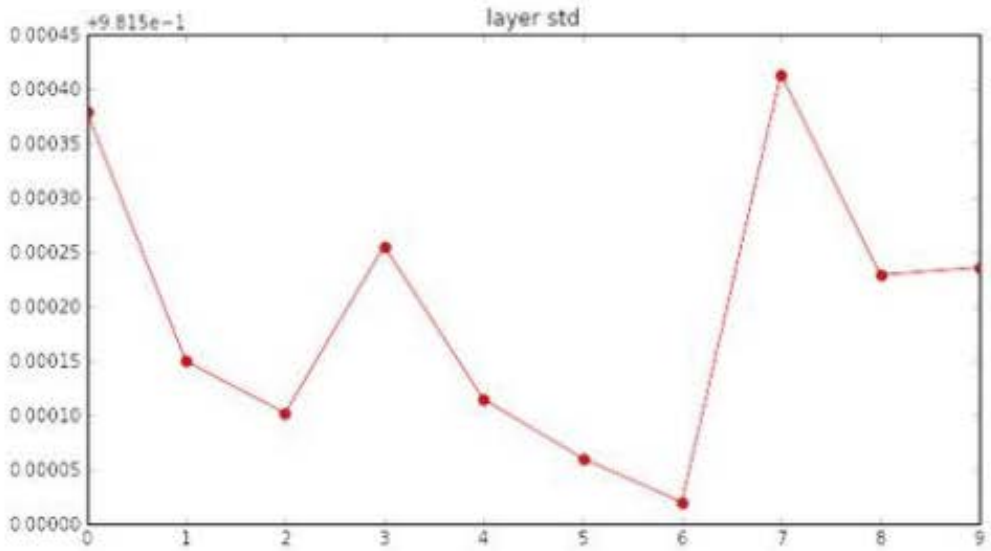
```
act = {'relu':lambda x:np.maximum(0,x), 'tanh':lambda x:np.tanh(x)}
Hs = {}
for i in xrange(len(hidden_layer_sizes)):
    X = D if i == 0 else Hs[i-1] # input at this layer
    fan_in = X.shape[1]
    fan_out = hidden_layer_sizes[i]
    W = np.random.randn(fan_in, fan_out) * 0.01 # layer initialization

    H = np.dot(X, W) # matrix multiply
    H = act[nonlinearities[i]](H) # nonlinearity
    Hs[i] = H # cache result on this layer
```

```
# look at distributions at each layer
print 'input layer had mean %f and std %f' % (np.mean(D), np.std(D))
layer_means = [np.mean(H) for i,H in Hs.iteritems()]
layer_stds = [np.std(H) for i,H in Hs.iteritems()]
for i,H in Hs.iteritems():
    print 'hidden layer %d had mean %f and std %f' % (i+1, layer_means[i], layer_stds[i]

# plot the means and standard deviations
plt.figure()
plt.subplot(121)
plt.plot(Hs.keys(), layer_means, 'ob-')
plt.title('layer mean')
plt.subplot(122)
plt.plot(Hs.keys(), layer_stds, 'or-')
plt.title('layer std')

# plot the raw distributions
plt.figure()
for i,H in Hs.iteritems():
    plt.subplot(1,len(Hs),i+1)
    plt.hist(H.ravel(), 30, range=(-1,1))
```

Another Weight Initialization

```
W = np.random.randn(fan_in, fan_out) * 1.0 # layer initialization
```

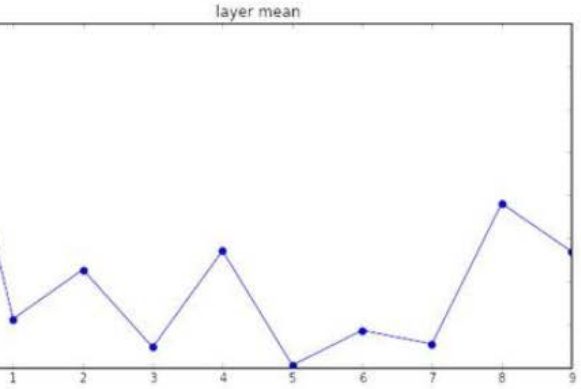


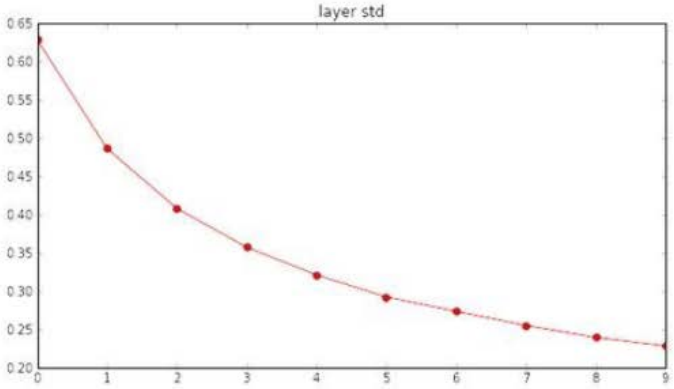


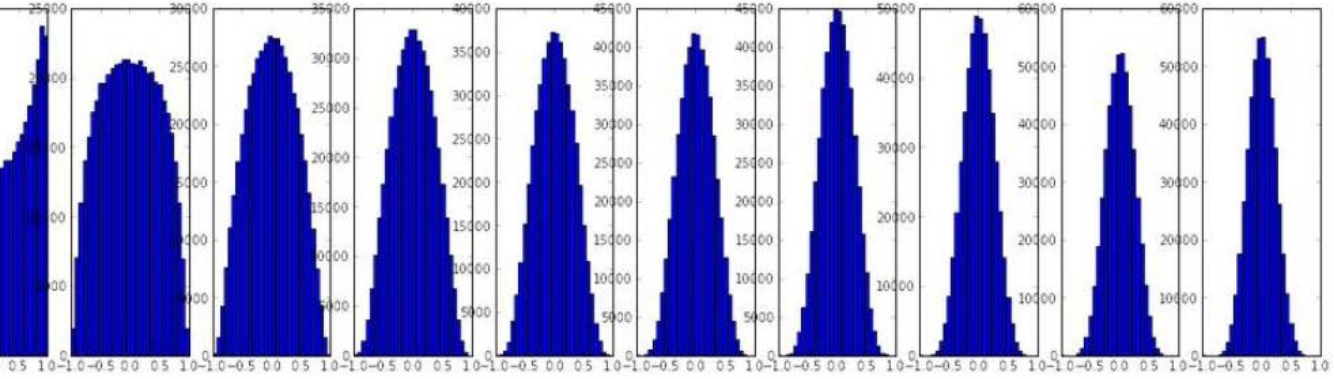
Almost all activations saturate (either 1 or -1)
Gradients will be all zero

Xavier Initialization [Glorot 2010]

```
W = np.random.randn(fan_in, fan_out) / np.sqrt(fan_in) # layer initialization
```





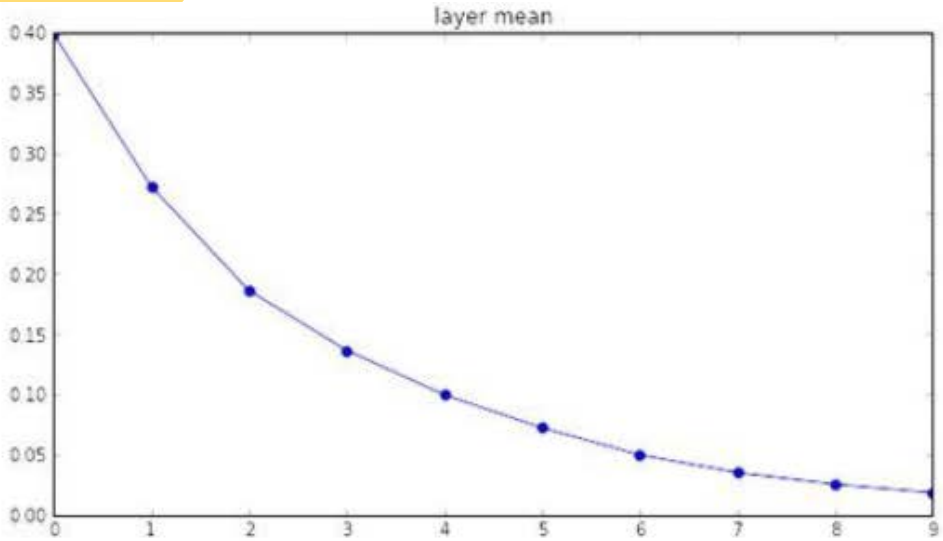


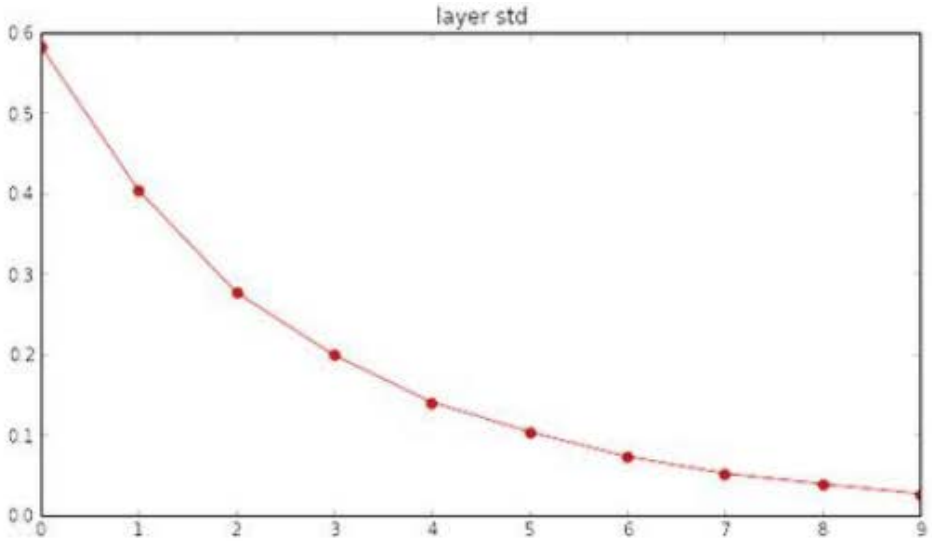
- For $a = wx$ the variance is
  $$var(a) = E[x]^2 var(w) + \mathrm{E}[w]^2 var(x) + var(x)var(w)$$
- Since $E[x] = E[w] = 0$
  $$var(a) = var(x)var(w) \approx d \cdot var(x_i)var(w_i)$$
- For $var(a) = var(x) \Rightarrow var(w_i) = \frac{1}{d}$
- Draw random weights from
  $$w \sim N\left(0, \sqrt{1/d}\right)$$
  where d is the number of neurons in the input

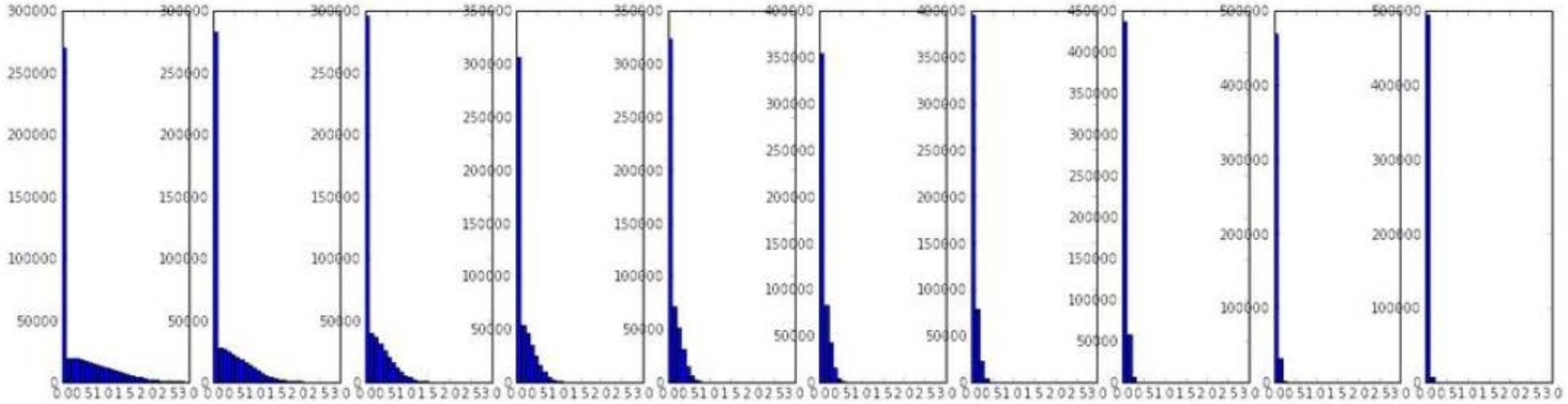
OK, But …

Use ReLU as Nonlinearity

- activation approaches zero again







Initialization for ReLU[[He 2015]

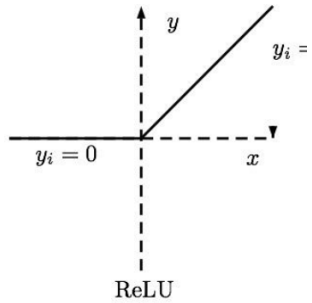
He et al., 2015
(note additional 2/)

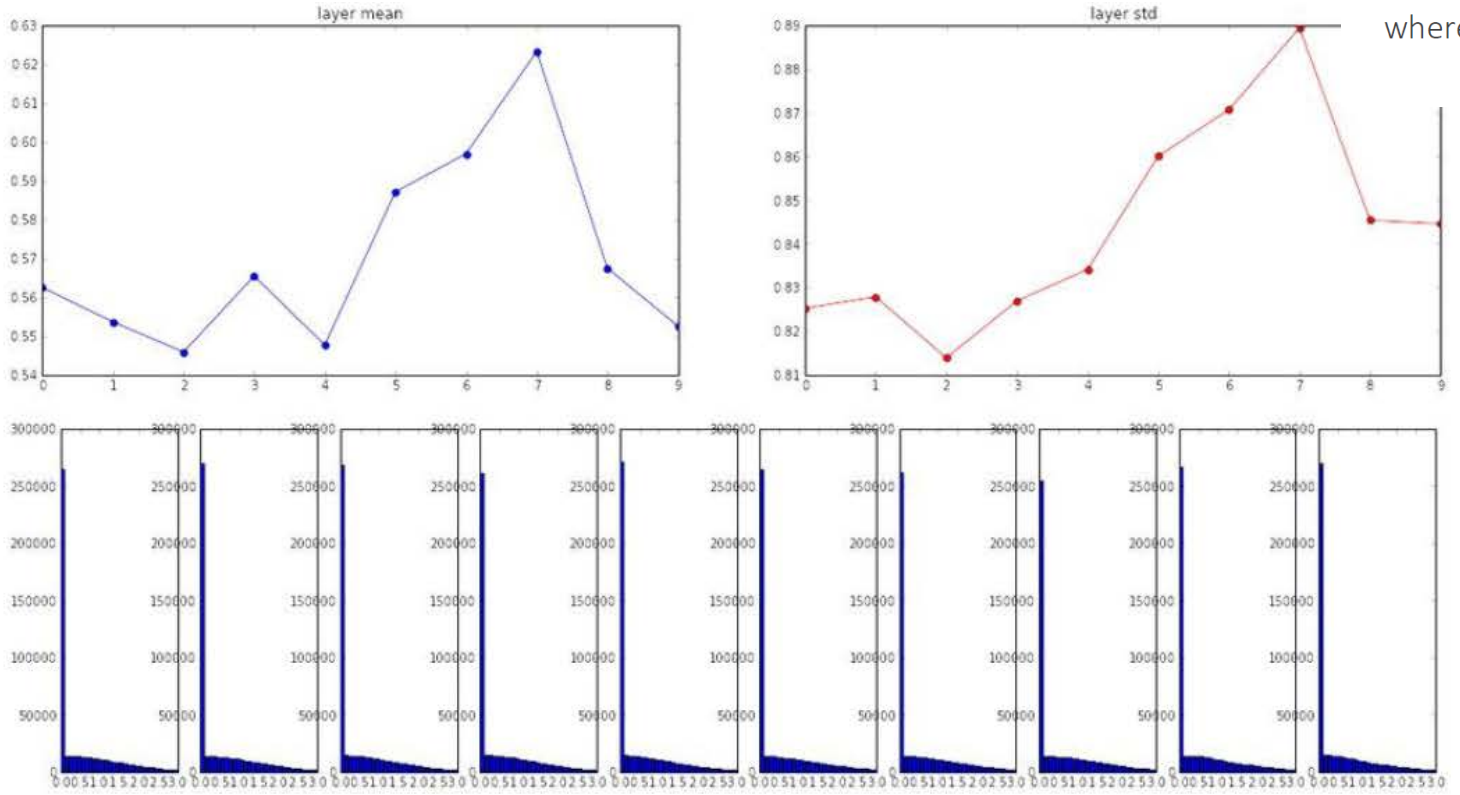
```
W = np.random.randn(fan_in, fan_out) / np.sqrt(2/fan_in) # layer initialization
```

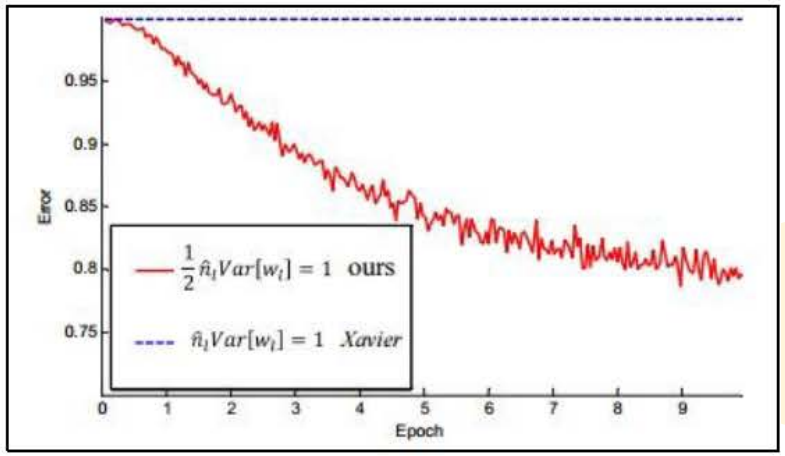
- better activation distribution
- much better accuracy

- Unlike sigmoids, ReLUs ground to 0 the linear activations half the time
- Double weight variance
  - Compensate for the zero flat-area →
  - Input and output maintain same variance
  - Very similar to Xavier initialization
- Draw random weights from $w \sim N\left(0, \sqrt{2/d}\right)$

  where d is the number of neurons in the input







Summary of Weight Initialization

- still an active research topic

Understanding the difficulty of training deep feedforward neural networks by Glorot and Bengio, 2010

Exact solutions to the nonlinear dynamics of learning in deep linear neural networks by Saxe et al, 2013

Random walk initialization for training very deep feedforward networks by Sussillo and Abbott, 2014

Delving deep into rectifiers: Surpassing human-level performance on ImageNet classification by He et al., 2015

Data-dependent Initializations of Convolutional Neural Networks by Krähenbühl et al., 2015

All you need is a good init, Mishkin and Matas, 2015

…

Back Propagation (scalar)

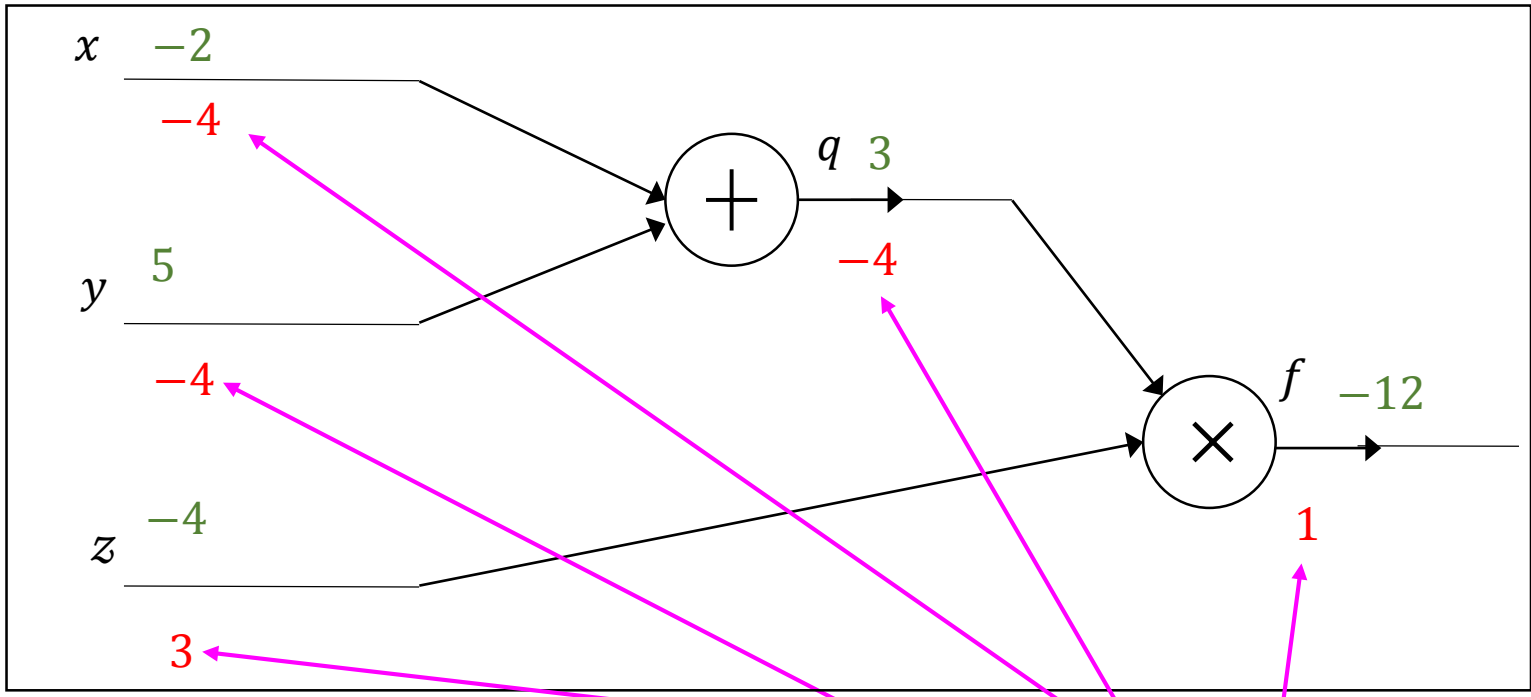
$$f(x, y, z) = (x + y)z$$

e.g. x = -2, y = 5, z= -4

$$q = x + y \quad \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$

$$f = qz \quad \frac{\partial f}{\partial q} = z, \frac{\partial f}{\partial z} = q$$

Want : $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$



Chain rule: Chain rule:

$$\frac{\partial f}{\partial y} = \frac{\partial f}{\partial q}\frac{\partial q}{\partial x}\frac{\partial f}{\partial x} = \frac{\partial f}{\partial q}\frac{\partial q}{\partial x}$$

Upstream gradient

Local gradient

Upstream gradient

Local gradient

$\frac{\partial f}{\partial y}$ $\frac{\partial f}{\partial f}$ $\frac{\partial f}{\partial z}$

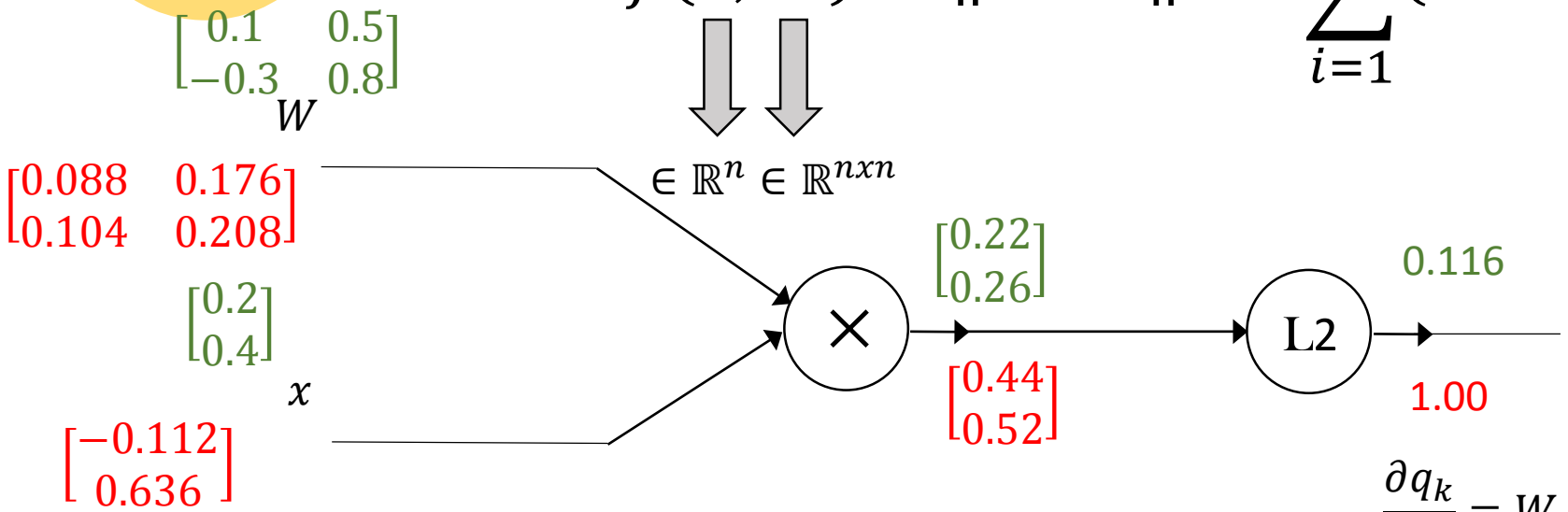
$\frac{\partial f}{\partial q}$ $\frac{\partial f}{\partial x}$

Back Propagation (vector)

$$f(x, W) = \|W \cdot x\|^2 = \sum_{i=1}^{n} (W \cdot x)_i^2$$

$$\nabla_W f = 2q \cdot x^T$$

$$\nabla_x f = 2W^T \cdot q$$



Always check: The gradient with respect to a variable should have the same shape as the variable

$$q = W \cdot x = \begin{pmatrix} W_{1,1}x_1 + \cdots + W_{1,n}x_n \\ \vdots \\ W_{n,1}x_1 + \cdots + W_{n,n}x_n \end{pmatrix}$$

$$f(q) = \|q\|^2 = q_1^2 + \cdots + q_1^2$$

$$\frac{\partial q_k}{\partial W_{i,j}} = 1_{k=i} x_j$$

$$\frac{\partial q_k}{\partial x_i} = W_{k,i}$$

$$\frac{\partial f}{\partial x_i} = \Sigma_k \frac{\partial f}{\partial q_k} \frac{\partial q_k}{\partial x_i} = \Sigma_k (2q_k)(W_{k,i})$$

$$\frac{\partial f}{\partial W_{i,j}} = \Sigma_k \frac{\partial f}{\partial q_k} \frac{\partial q_k}{\partial W_{i,j}} = \Sigma_k (2q_k)(1_{k=i} x_j) = 2q_i\, x_j$$

$$\frac{\partial f}{\partial q_i} = 2q_i$$

$$\nabla_q f = 2q$$

Stochastic Gradient Descent (SGD)

- Gradient Descent (GD) is computed from all available training samples
- SGD uses a mini-batch of *m* samples to compute the gradient
  - randomness helps avoid overfitting
  - random sampling allows to be much faster than GD, and better accuracy
  - mini-batch sampling is suitable for datasets that change over time
  - variance of gradients increases when mini-batch size decreases

SGD Algorithm

1 . Initialize weights randomly $\sim \mathcal{N}(0, \sigma^2)$

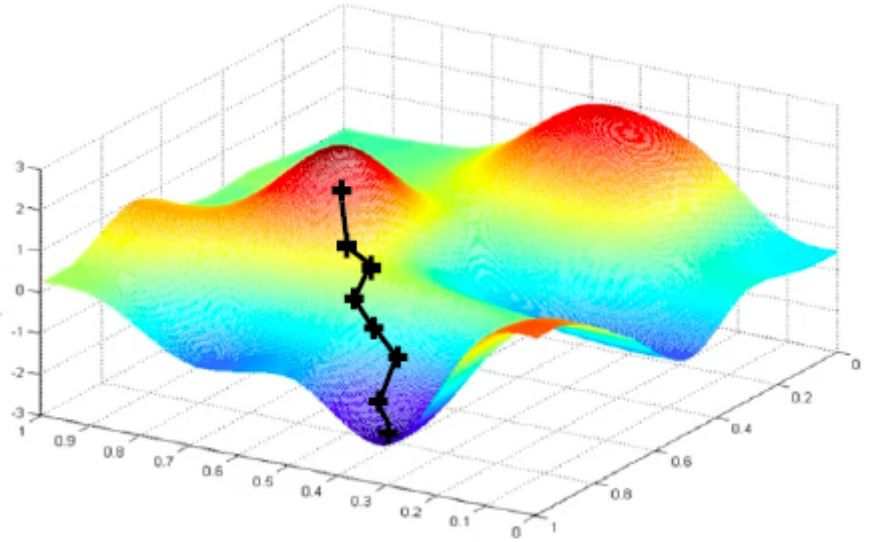
2. Loop until convergence:

3. Pick batch of *m* data points

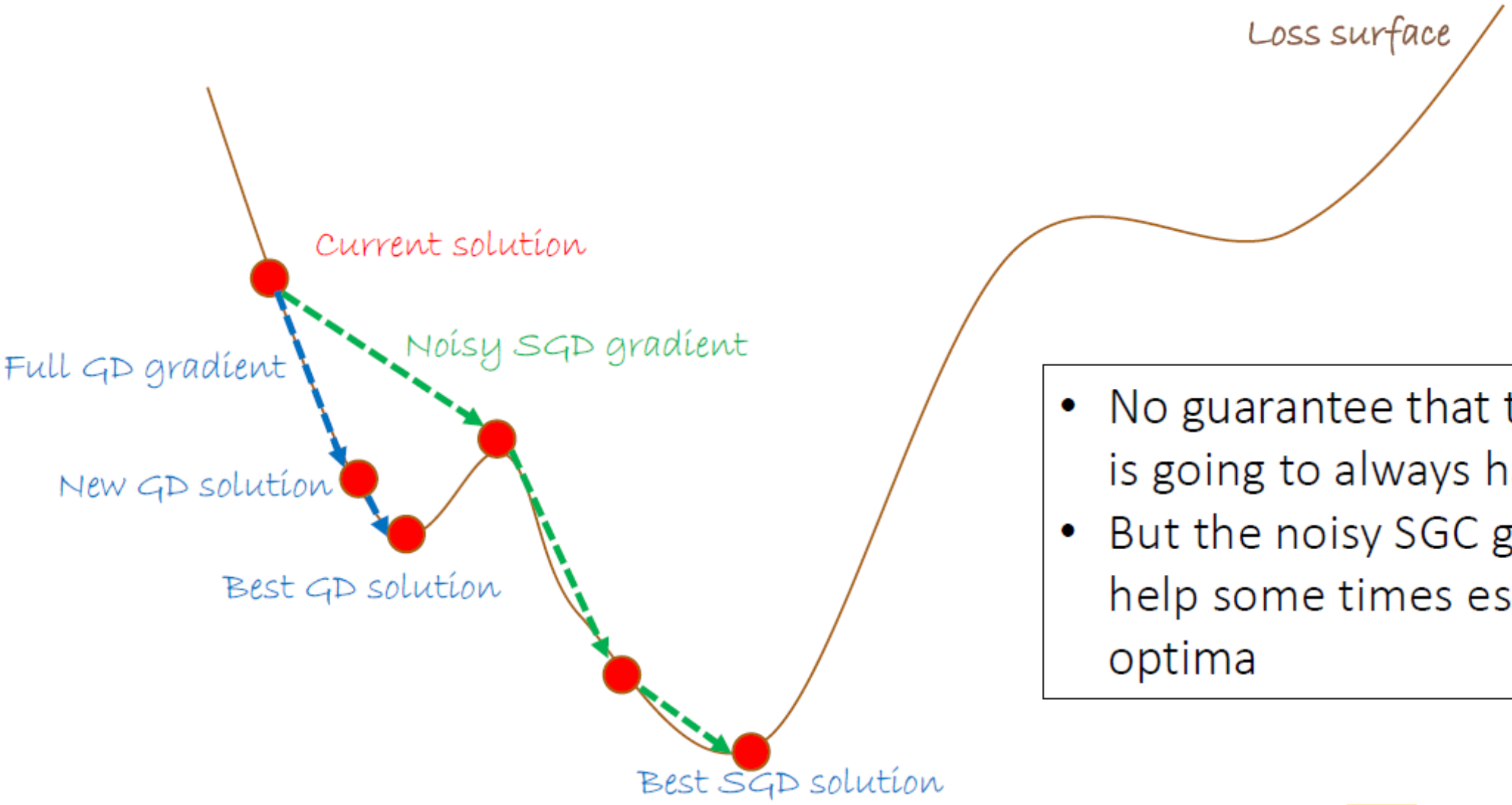
4. Compute gradient, $\frac{\partial L(\theta)}{\partial \theta} = \frac{1}{m} \Sigma_{k=1}^{m} \frac{\partial L_k(\theta)}{\partial \theta}$

5. Update weights, $\theta \leftarrow \theta - \epsilon \frac{\partial L(\theta)}{\partial \theta}$

6. Return weights

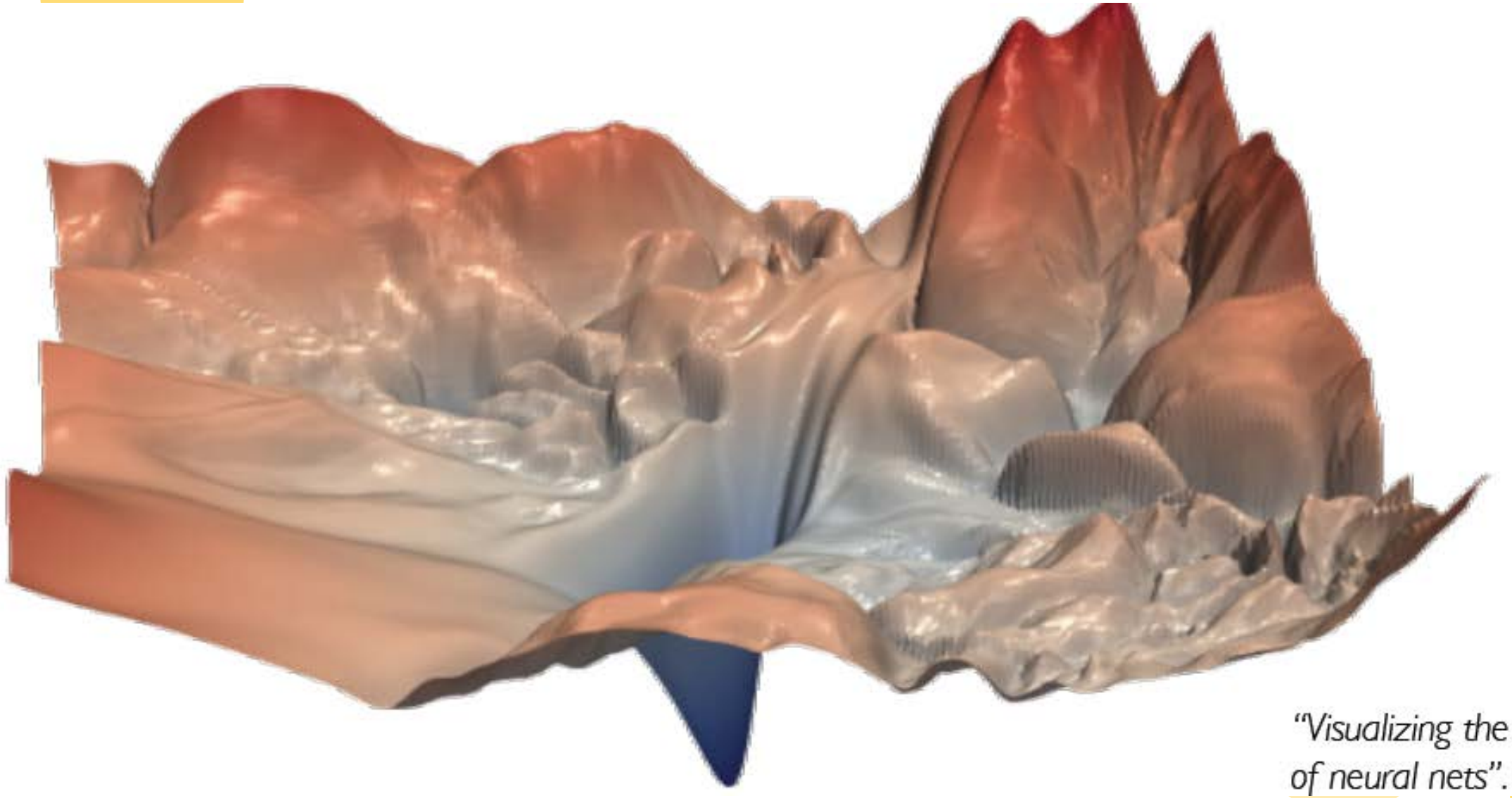


SGD is often better and faster



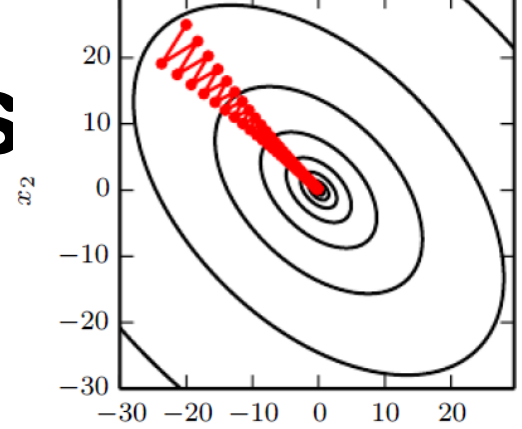
- No guarantee that this is what is going to always happen.
- But the noisy SGC gradients can help some times escaping local optima

Optimization of Loss Function

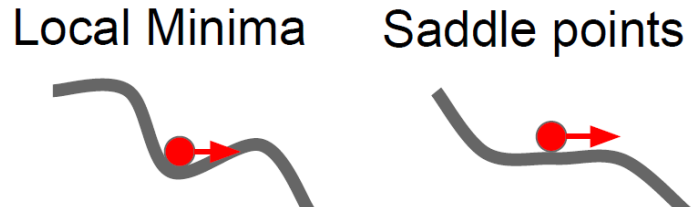


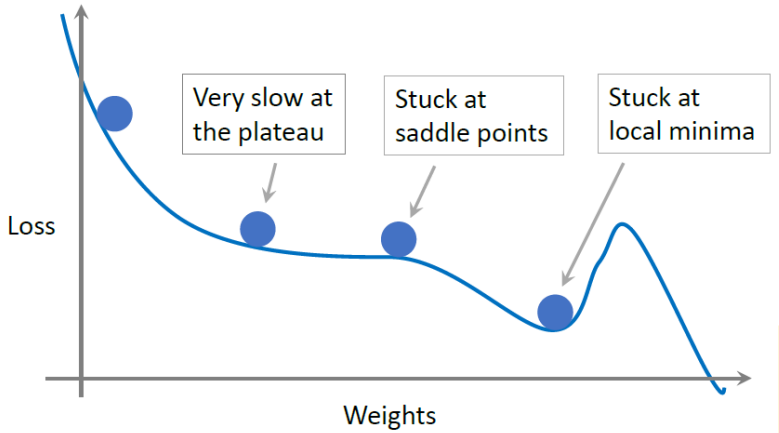
"Visualizing the loss landscape of neural nets". Dec 2017.

Optimization Algorithms



- SGD $\hat{g} \leftarrow +\frac{1}{m}\nabla_\theta \Sigma_i L\left(f\left(x^{(i)};\theta\right), y^{(j)}\right)$ $\theta \leftarrow \theta - \epsilon\hat{g}$
- SGD+Momentum $v \leftarrow \alpha v - \epsilon g$ $\theta \leftarrow \theta + v$
- Nesterov Momentum $\tilde{\theta} \leftarrow \theta + \alpha v$ $g \leftarrow \frac{1}{m}\nabla_{\tilde{\theta}} \Sigma_i L\left(f\left(x^{(i)};\tilde{\theta}\right), y^{(i)}\right)$
  $v \leftarrow \alpha v - \epsilon g$ $\theta \leftarrow \theta + v$
- AddGrad $g \leftarrow \frac{1}{m}\nabla_{\tilde{\theta}} \Sigma_i L\left(f\left(x^{(i)};\tilde{\theta}\right), y^{(i)}\right)$ $\boldsymbol{r} \leftarrow \boldsymbol{r} + g \odot g$
  $\Delta\theta \leftarrow -\frac{\epsilon}{\delta+\sqrt{r}} \odot g$ $\theta \leftarrow \theta + \Delta\theta$
- RMSProp
  $\boldsymbol{r} \leftarrow \rho\boldsymbol{r} + (1-\rho) g \odot g$
  $\Delta\theta = -\frac{\epsilon}{\sqrt{\delta+r}} \odot g$ $\theta \leftarrow \theta + \Delta\theta$
- Adam $\boldsymbol{s} \leftarrow \rho_1 \boldsymbol{s} + (1-\rho_1) g$ $\tilde{s} \leftarrow \frac{s}{1-\rho_1^t}$
  $\boldsymbol{r} \leftarrow \rho_2 \boldsymbol{r} + (1-\rho_2) g \odot g$ $\tilde{\boldsymbol{r}} \leftarrow \frac{\boldsymbol{r}}{1-\rho_2^t}$
  $\Delta\theta = -\epsilon \frac{s}{\sqrt{\tilde{\boldsymbol{r}}} + \delta}$ $\theta \leftarrow \theta + \Delta\theta$





Stochastic Gradient Descent (SGD)

- mini-batch SGD
  - **sample** a batch of data (not entire training dataset)
  - **forward propagate** it through the network, and get loss
  - **back propagate** to calculate the gradients
  - **update the parameters** using the gradients

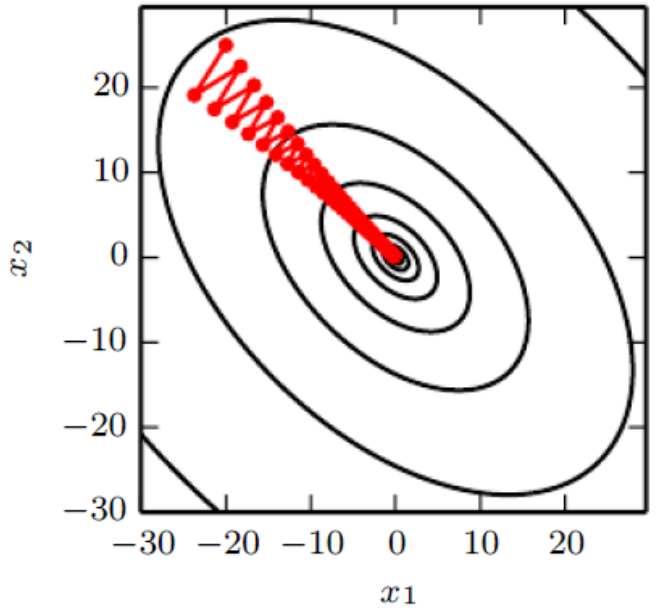


Fig. 4.6 (p. 90, Goodfellow)

Algorithm 8.1 Stochastic gradient descent (SGD) update at training iteration k

Require: Learning rate ϵ_k.
Require: Initial parameter $\boldsymbol{\theta}$
while stopping criterion not met **do**
  Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.
  Compute gradient estimate: $\hat{\boldsymbol{g}} \leftarrow +\frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
  Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \epsilon \hat{\boldsymbol{g}}$
end while

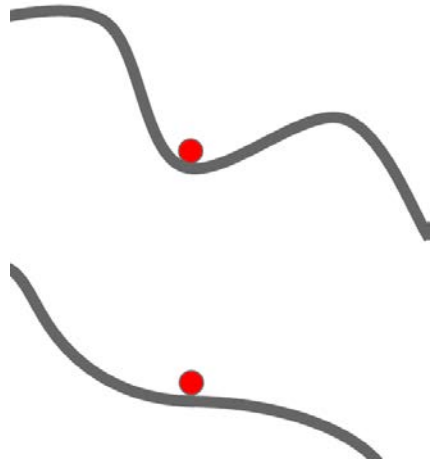
$$\theta_{t+1} = \theta_t - \varepsilon \nabla f(\theta_t)$$

Problems with SGD

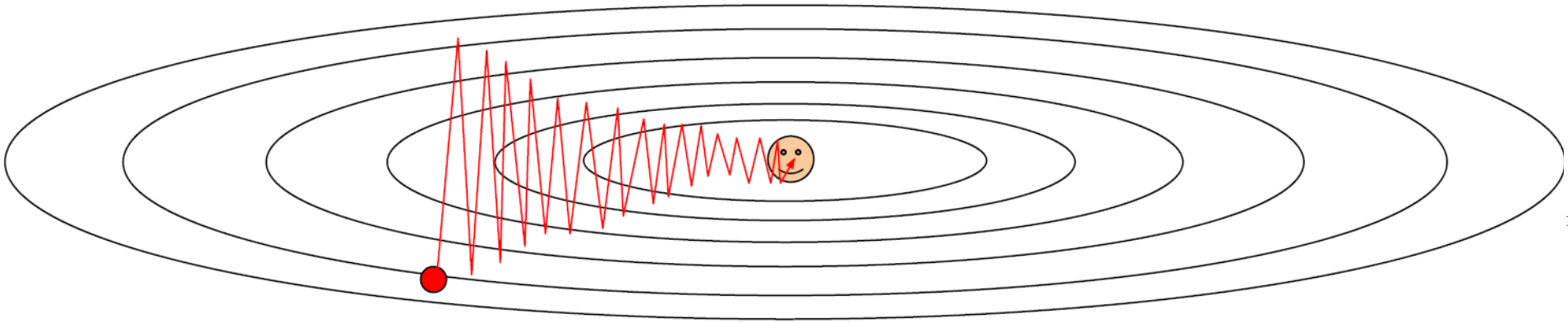
- gradients come from min-batches
  - could be noisy
- get stuck at local minima or saddle point

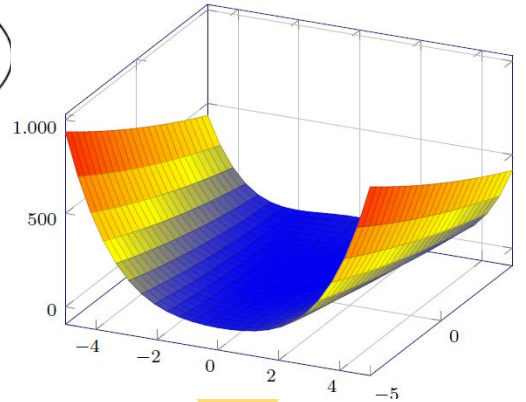
$$L(W) = \frac{1}{N}\sum_{i=1}^{N} L_i(x_i, y_i, W)$$

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



What if loss changes quickly in one direction and slowly in another?
What does gradient descent do?
Very slow progress along shallow dimension, jitter along steep direction

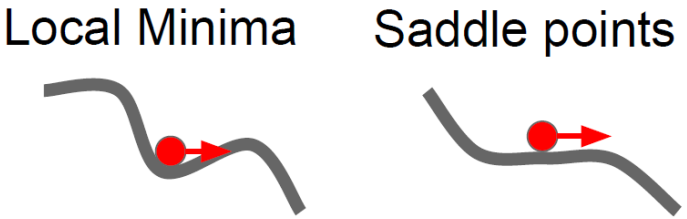




Loss function has high **condition number**: ratio of largest to smallest singular value of the Hessian matrix is large

SGD + Momentum

- different formula, but have the same result

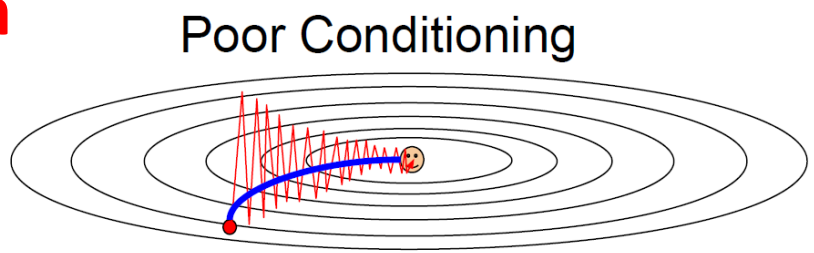


SGD+ Momentum

$$\mathbf{v}_{t+1} = \alpha \mathbf{v}_t - \varepsilon \nabla_\theta f(\boldsymbol{\theta}_t)$$
$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \mathbf{v}_{t+1}$$

SGD+ Nesterov Momentum

$$\mathbf{v}_{t+1} = \alpha \mathbf{v}_t - \varepsilon \nabla_\theta f(\boldsymbol{\theta}_t + \alpha \mathbf{v}_t)$$
$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \mathbf{v}_{t+1}$$



Algorithm 8.2 Stochastic gradient descent (SGD) with momentum

Require: Learning rate ϵ, momentum parameter α.
Require: Initial parameter $\boldsymbol{\theta}$, initial velocity $\boldsymbol{v}$.
while stopping criterion not met **do**
  Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.
  Compute gradient estimate: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
  Compute velocity update: $\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} - \epsilon \boldsymbol{g}$
  Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}$
end while

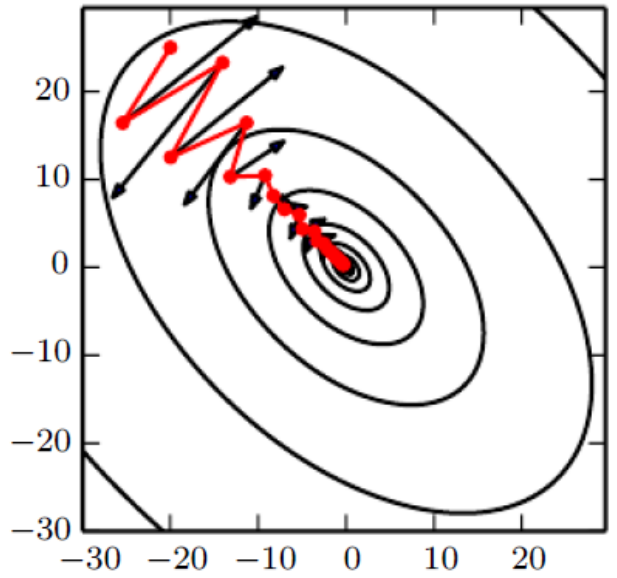
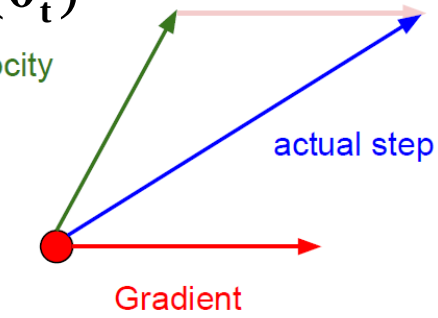


Fig. 8.5 (p. 297, Goodfellow)

Nesterov Momentum

Momentum update:

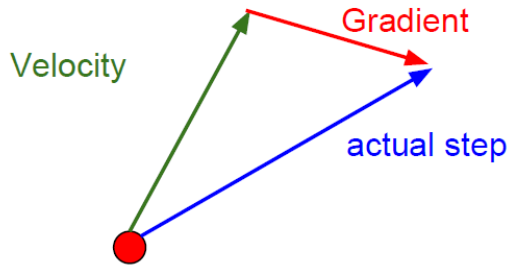
$$\mathbf{v}_{t+1} = \alpha \mathbf{v}_t - \varepsilon \nabla_\theta f(\boldsymbol{\theta}_t)$$
$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \mathbf{v}_{t+1}$$



Combine gradient at current point with velocity to get step used to update weights

Nesterov, "A method of solving a convex programming problem with convergence rate O(1/k^2)", 1983
Nesterov, "Introductory lectures on convex optimization: a basic course", 2004
Sutskever et al, "On the importance of initialization and momentum in deep learning", ICML 2013

Nesterov Momentum



$$\mathbf{v}_{t+1} = \alpha \mathbf{v}_t - \varepsilon \nabla_\theta f(\boldsymbol{\theta}_t + \alpha \mathbf{v}_t)$$
$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \mathbf{v}_{t+1}$$

"Look ahead" to the point where updating using velocity would take us; compute gradient there and mix it with velocity to get actual update direction

Algorithm 8.3 Stochastic gradient descent (SGD) with Nesterov momentum

Require: Learning rate ϵ, momentum parameter α.
Require: Initial parameter $\boldsymbol{\theta}$, initial velocity $\boldsymbol{v}$.
while stopping criterion not met **do**
Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding labels $\boldsymbol{y}^{(i)}$.
Apply interim update: $\tilde{\boldsymbol{\theta}} \leftarrow \boldsymbol{\theta} + \alpha \boldsymbol{v}$
Compute gradient (at interim point): $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\tilde{\boldsymbol{\theta}}} \sum_i L(f(\boldsymbol{x}^{(i)}; \tilde{\boldsymbol{\theta}}), \boldsymbol{y}^{(i)})$
Compute velocity update: $\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} - \epsilon \boldsymbol{g}$
Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}$
end while

$$\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} - \epsilon \nabla_{\boldsymbol{\theta}} \left(\frac{1}{m} \sum_{i=1}^{m} L(\boldsymbol{f}(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)}) \right),$$
$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}.$$

momentum

$$\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} - \epsilon \nabla_{\boldsymbol{\theta}} \left[\frac{1}{m} \sum_{i=1}^{m} L\left(\boldsymbol{f}(\boldsymbol{x}^{(i)}; \boldsymbol{\theta} + \alpha \boldsymbol{v}), \boldsymbol{y}^{(i)} \right) \right]$$
$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v},$$

Nesterov momentum

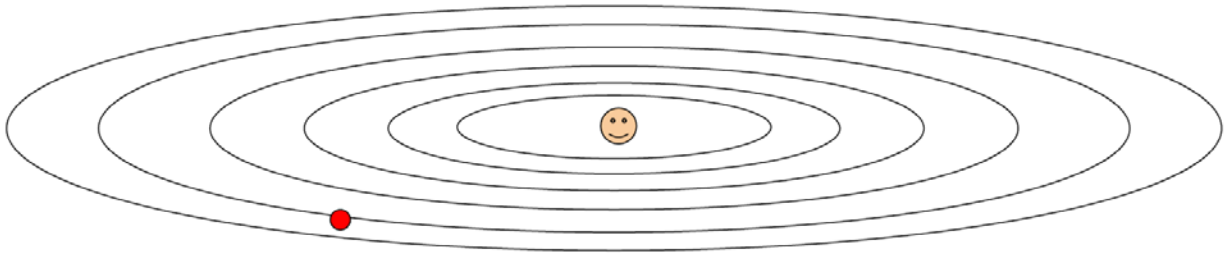
Adaptive Learning Rate

- till now, we assign same learning rate to all features
- if features vary in importance and frequency, it is better to adaptively change learning

AdaGrad

- add element-wise scaling of the gradient based on historical sum of squares in each dimension
  - per-parameter learning rate, or
  - adaptive learning rate

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



progress along "steep" direction is damped;
progress along "flat" direction is accelerated

Algorithm 8.4 The AdaGrad algorithm

Require: Global learning rate ϵ

Require: Initial parameter $\boldsymbol{\theta}$

Require: Small constant δ, perhaps 10^{-7}, for numerical stability

Initialize gradient accumulation variable $\boldsymbol{r} = \mathbf{0}$

while stopping criterion not met **do**

Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.

Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$

Accumulate squared gradient: $\boldsymbol{r} \leftarrow \boldsymbol{r} + \boldsymbol{g} \odot \boldsymbol{g}$

Compute update: $\Delta\boldsymbol{\theta} \leftarrow -\frac{\epsilon}{\delta + \sqrt{\boldsymbol{r}}} \odot \boldsymbol{g}$. (Division and square root applied element-wise)

Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$

end while

$$\mathbf{g} = \nabla_{\theta} f(\theta_t)$$

$$\mathbf{r}_t = \mathbf{r}_{t-1} + \mathbf{g} \odot \mathbf{g}$$

$$\Delta\boldsymbol{\theta} = \varepsilon \times \frac{1}{\delta + \sqrt{\mathbf{r}_t}} \odot \mathbf{g}$$

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \Delta\boldsymbol{\theta}$$

RMSProp

$$\mathbf{g} = \nabla_{\theta} f(\theta_t)$$

$$\mathbf{r}_{t+1} = \rho \times \mathbf{r}_t + (1-\rho) \times \mathbf{g} \odot \mathbf{g}$$

$$\mathbf{v}_{t+1} = \alpha \times \mathbf{v}_t - \varepsilon \times \frac{1}{\sqrt{\mathbf{r}_{t+1}}} \odot \mathbf{g}$$

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \mathbf{v}_{t+1}$$

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)
```

Algorithm 8.5 The RMSProp algorithm

Require: Global learning rate ϵ, decay rate ρ.
Require: Initial parameter $\boldsymbol{\theta}$
Require: Small constant δ, usually 10^{-6}, used to stabilize division by small numbers.
Initialize accumulation variables $\boldsymbol{r} = 0$
while stopping criterion not met **do**
  Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.
  Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
  Accumulate squared gradient: $\boldsymbol{r} \leftarrow \rho \boldsymbol{r} + (1-\rho) \boldsymbol{g} \odot \boldsymbol{g}$
  Compute parameter update: $\Delta\boldsymbol{\theta} = -\frac{\epsilon}{\sqrt{\delta + \boldsymbol{r}}} \odot \boldsymbol{g}$. ($\frac{1}{\sqrt{\delta + \boldsymbol{r}}}$ applied element-wise)
  Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$
end while

Algorithm 8.6 RMSProp algorithm with Nesterov momentum

Require: Global learning rate ϵ, decay rate ρ, momentum coefficient α.
Require: Initial parameter $\boldsymbol{\theta}$, initial velocity $\boldsymbol{v}$.
Initialize accumulation variable $\boldsymbol{r} = \boldsymbol{0}$
while stopping criterion not met **do**
  Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.
  Compute interim update: $\tilde{\boldsymbol{\theta}} \leftarrow \boldsymbol{\theta} + \alpha \boldsymbol{v}$
  Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\tilde{\boldsymbol{\theta}}} \sum_i L(f(\boldsymbol{x}^{(i)}; \tilde{\boldsymbol{\theta}}), \boldsymbol{y}^{(i)})$
  Accumulate gradient: $\boldsymbol{r} \leftarrow \rho \boldsymbol{r} + (1-\rho) \boldsymbol{g} \odot \boldsymbol{g}$
  Compute velocity update: $\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} - \frac{\epsilon}{\sqrt{\boldsymbol{r}}} \odot \boldsymbol{g}$. ($\frac{1}{\sqrt{\boldsymbol{r}}}$ applied element-wise)
  Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}$
end while

Adam

Sort of like RMSProp with momentum

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!

Algorithm 8.7 The Adam algorithm

Require: Step size ϵ (Suggested default: 0.001)
Require: Exponential decay rates for moment estimates, ρ_1 and ρ_2 in $[0, 1)$. (Suggested defaults: 0.9 and 0.999 respectively)
Require: Small constant δ used for numerical stabilization. (Suggested default: 10^{-8})
Require: Initial parameters $\boldsymbol{\theta}$
  Initialize 1st and 2nd moment variables $\boldsymbol{s} = \mathbf{0}$, $\boldsymbol{r} = \mathbf{0}$
  Initialize time step $t = 0$
  while stopping criterion not met **do**
    Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.
    Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
    $t \leftarrow t + 1$
    Update biased first moment estimate: $\boldsymbol{s} \leftarrow \rho_1 \boldsymbol{s} + (1 - \rho_1)\boldsymbol{g}$
    Update biased second moment estimate: $\boldsymbol{r} \leftarrow \rho_2 \boldsymbol{r} + (1 - \rho_2)\boldsymbol{g} \odot \boldsymbol{g}$
    Correct bias in first moment: $\hat{\boldsymbol{s}} \leftarrow \frac{\boldsymbol{s}}{1 - \rho_1^t}$
    Correct bias in second moment: $\hat{\boldsymbol{r}} \leftarrow \frac{\boldsymbol{r}}{1 - \rho_2^t}$
    Compute update: $\Delta\boldsymbol{\theta} = -\epsilon \frac{\hat{\boldsymbol{s}}}{\sqrt{\hat{\boldsymbol{r}}} + \delta}$ (operations applied element-wise)
    Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$
  end while

```
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))
```

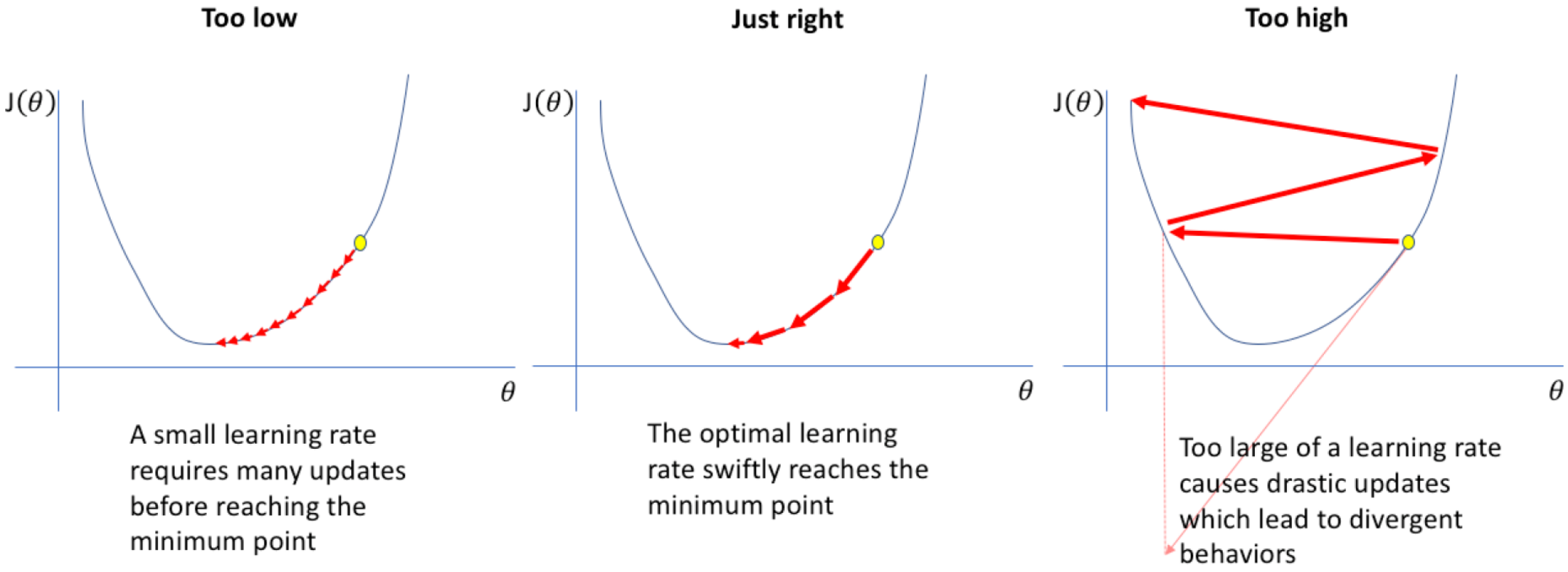
Momentum

Bias correction

AdaGrad / RMSProp

Learning Rate

- too strong: gradients overshoot and bounce
- too weak: too small gradients -> slow training
- rule of thumb:
  - earning rate of (shared) weights square root of weight connections
- learning rate per weight is advantageous
  - some weights are near convergence, others not

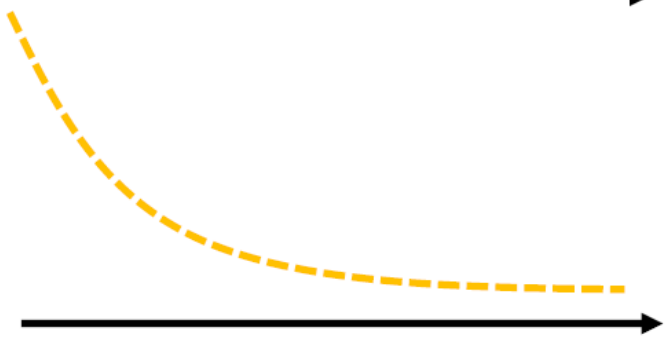


learning rate schedules

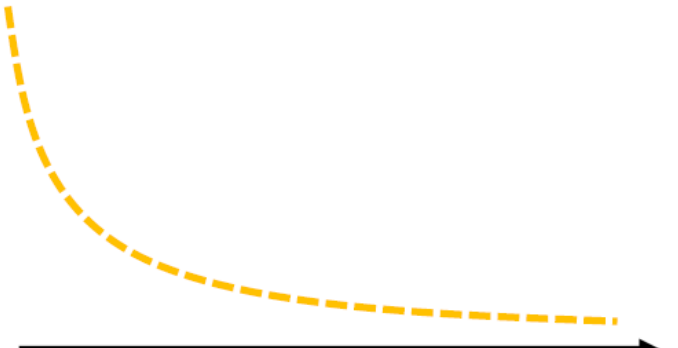
- constant
  - learning rate remains the same for all epochs

- step decay
  - decrease by 1/T for every T number of epochs
  - preferred
    - because simple, intuitive, work well and only a single extra hyper-parameter T (T=2, or 10)

- inverse decay: $\epsilon_t = \epsilon_0 /* (1 + \delta t)$



- exponential decay: $\epsilon_t = \epsilon_0 e^{-\delta t}$



learning rate

- SGD, SGD+Momentum, Adagrad, RMSProp, Adam all have **learning rate** as a hyper-parameter

SGD

$$\theta \leftarrow \theta - \epsilon \hat{g}$$

Momentum

$$v \leftarrow \alpha v - \epsilon g$$

$$\theta \leftarrow \theta + v$$

AdaGrad

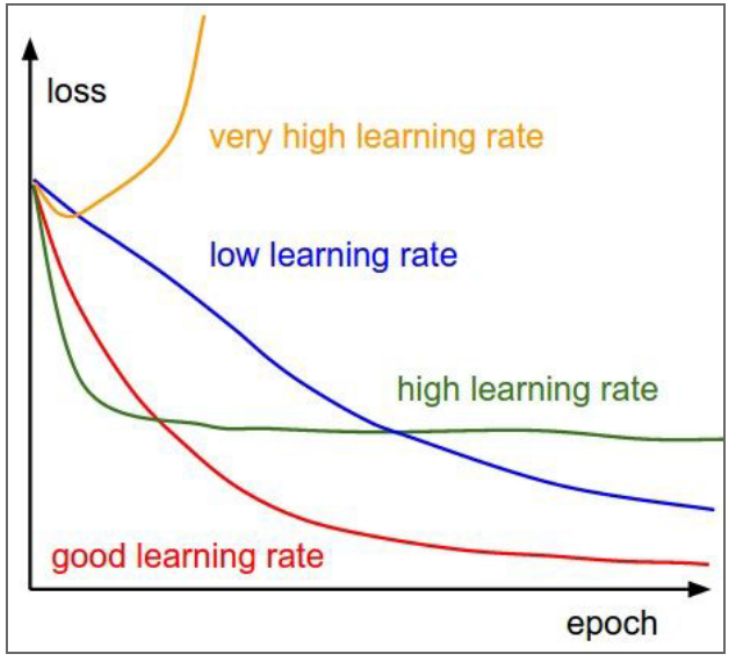
$$\Delta\theta \leftarrow -\frac{\epsilon}{\delta + \sqrt{r}} \odot g$$

RMSProp

$$\Delta\theta = -\frac{\epsilon}{\sqrt{\delta + r}} \odot g$$

Adam

$$\Delta\theta = -\epsilon \frac{\tilde{\boldsymbol{s}}}{\sqrt{\tilde{\boldsymbol{r}}} + \delta}$$



=> Learning rate decay over time!

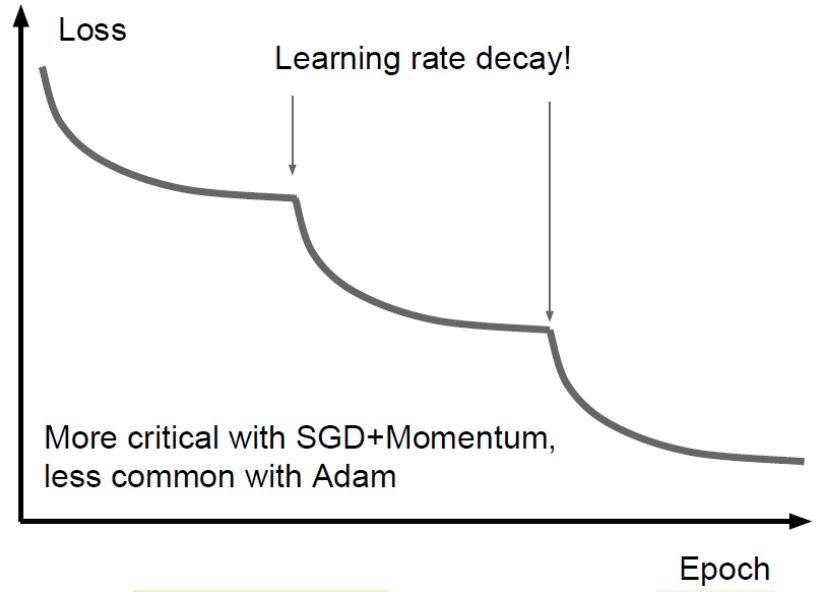
step decay:
e.g. decay learning rate by half every few epochs.

exponential decay:

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

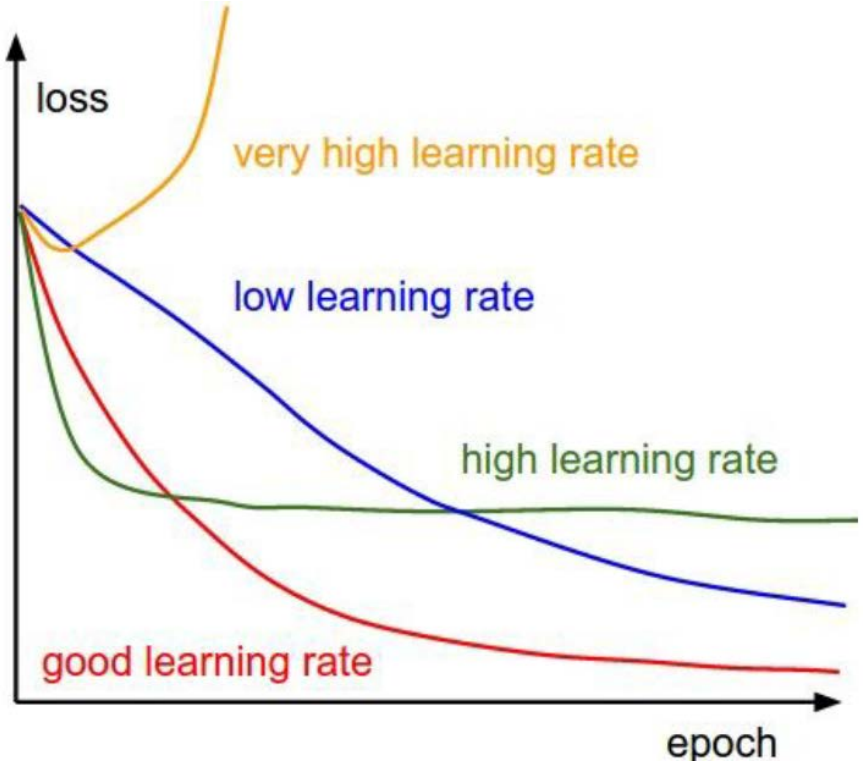
1/t decay:

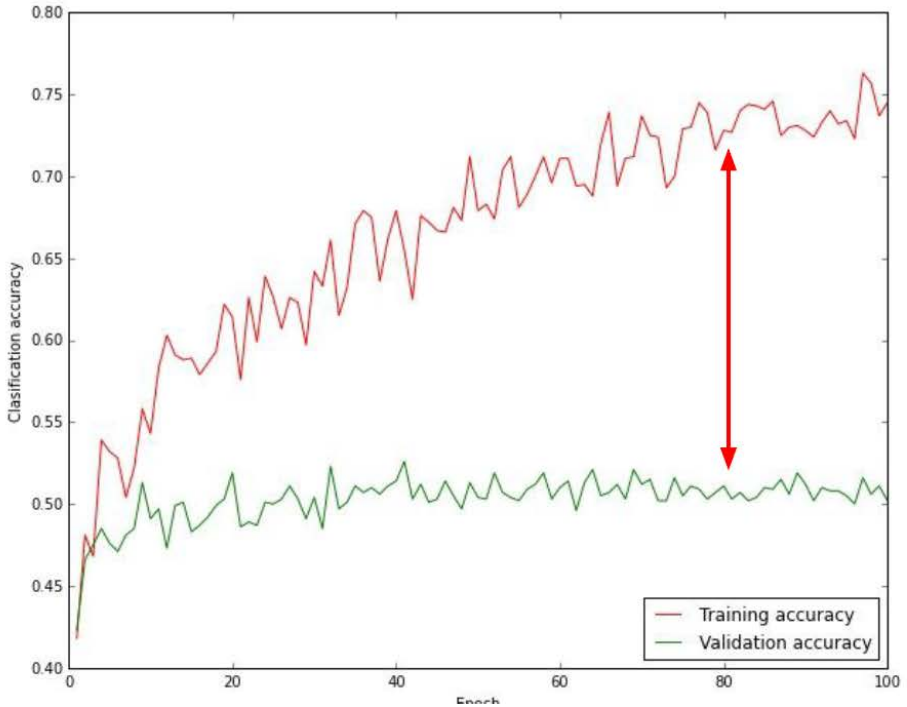
$$\alpha = \alpha_0 / (1 + kt)$$



Hyper-parameter Optimization

- hyper-parameters
  - network architecture
  - learning rate, its decay schedule, update type
  - regularization (L1/L2, dropout strength), weight decay
- coarse-to-fine cross-validation in stages



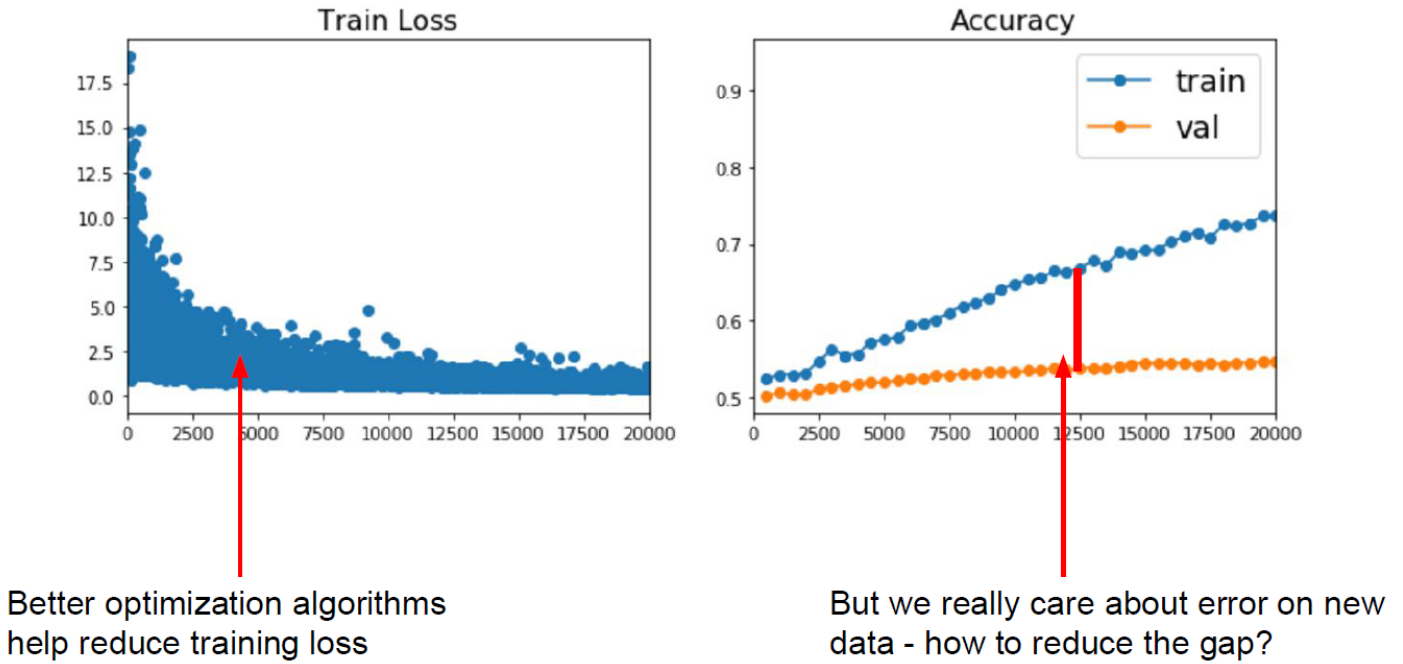


big gap = overfitting
=> increase regularization strength?

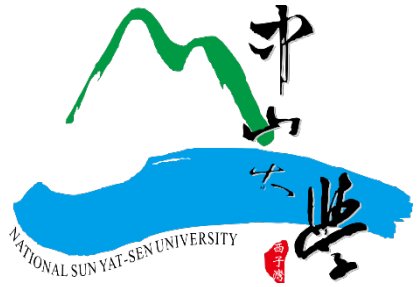
no gap
=> increase model capacity?

Summary of Optimization Algorithms

- Adam is a good default choice in many cases
- SGD + Momentum with learning rate decay often outperforms Adam by a bit, but requires more tuning
- But we care about error rate on “new” data
  - regularization methods



Regularization



Regularization

- early stop
- L1/L2 regularization term
- dropout
- batch normalization
- data augmentation

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

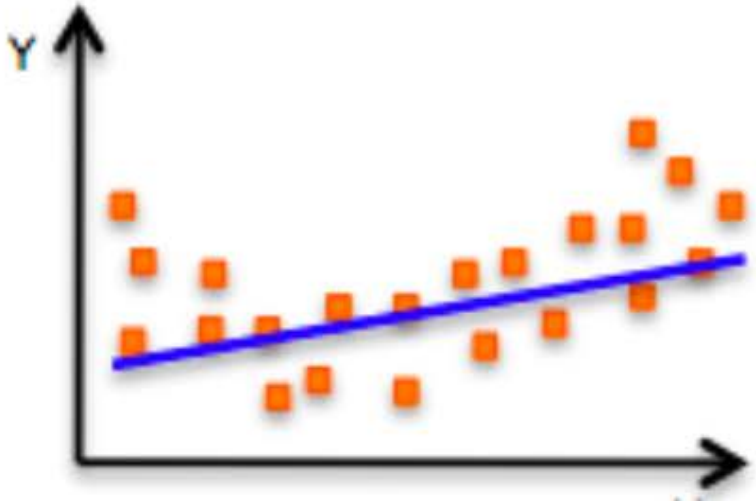
Testing: Use fixed stats to normalize

Early Stop

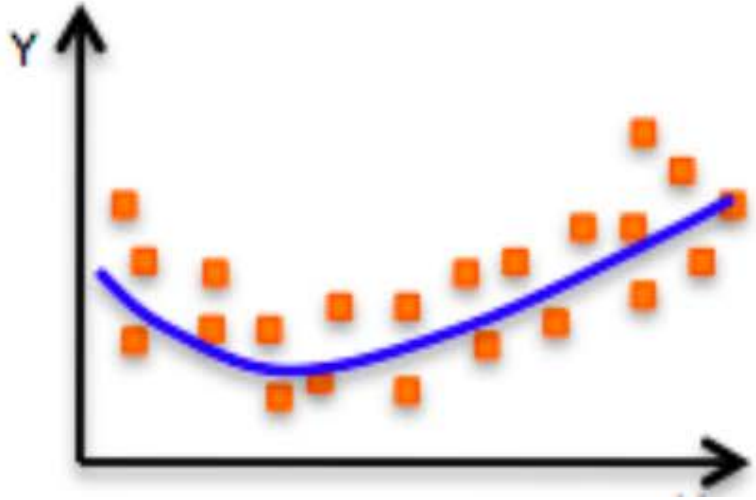
- stop training before overfitting

Problem of Overfitting

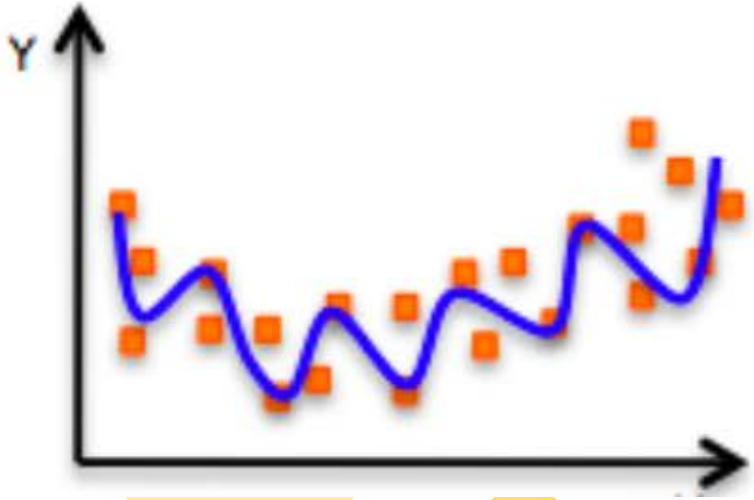
Under-fitting:
model does not have capacity to fully learn the data



Ideal-fitting:



Over-fitting:
too complex, extra parameters does not generalize



add regularization term to loss

- proper weight regularization to avoid overfitting
  - add (L1/L2) regularization term with weight decay (λ) parameter to loss function

$$L = \frac{1}{N}\Sigma_{i=1}^{N} L_i(f(x_i\ ;\ W), y_i) + \boxed{\lambda R(W)}$$

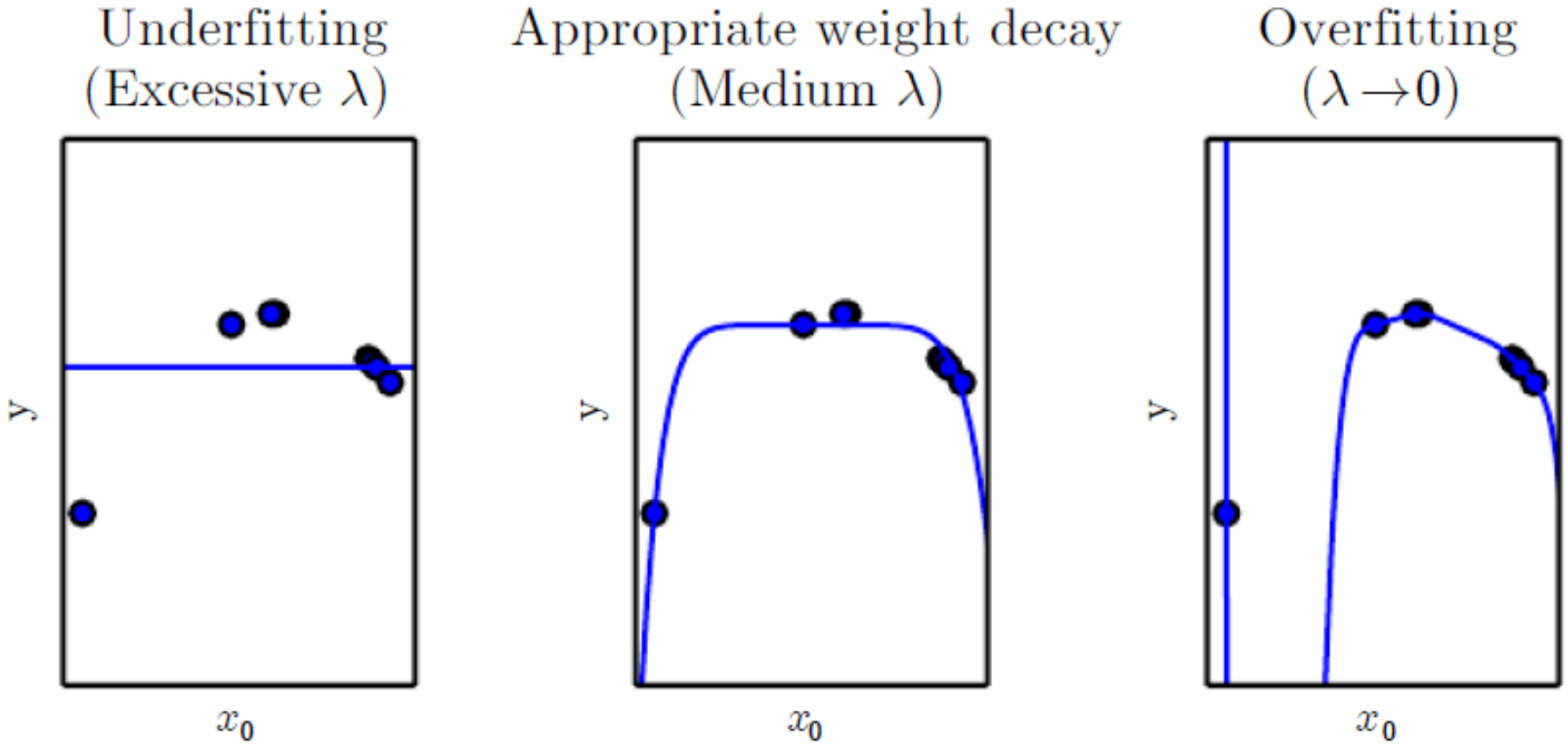
L2 regularization $R(W) = \Sigma_k \Sigma_l W_{k,l}^2$ (Weight decay)

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

L2 Regularization (1-D example)

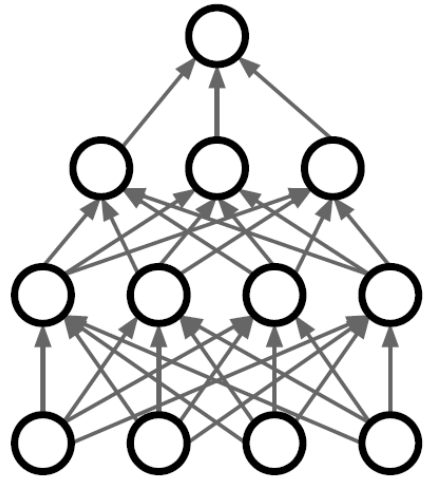
$$L(w) = \frac{1}{m^{(train)}} \left\| X^{(train)} w - y^{(train)} \right\|_2 + \underbrace{\lambda w^T w}_{\text{L2 regularization term}}$$

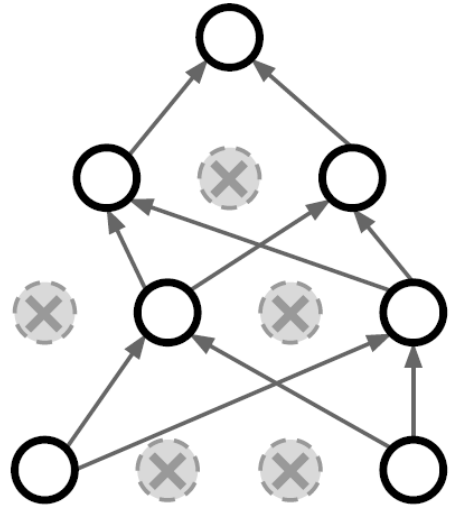
λ : weight decay coefficient



dropout

- force network to not rely on any one node
  - forces network to have a redundant representation
  - prevent co-adaptation of features
- In each forward pass, randomly set some neurons to zero Probability of dropping is a hyper-parameter; 0.5 is common
- dropout is training a large **ensemble** of models with shared parameters
  - each binary mask is one model
- dropout makes our output random during tra
- at test time, want to “average out” the randomness
  - all neurons are always active at test time
    - must scale the activations so that for each neuron, output at test time = expected output at training time
  - multiply by dropout probability





Output (label) Input (image) Random mask

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

Srivastava et al, “Dropout : A simple way to prevent neural networks from overfitting”, JMLR 2014

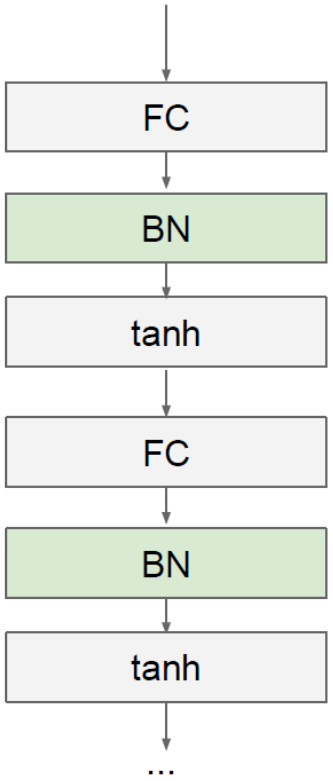
Dropout Summary

```
""" 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

Batch Normalization (BN)



Usually inserted after Fully Connected or Convolutional layers, and before nonlinearity.

Input: Values of x over a mini-batch:$\boldsymbol{B}$ ={$x_{1\ldots m}$};
Parameters to be learned:γ,β

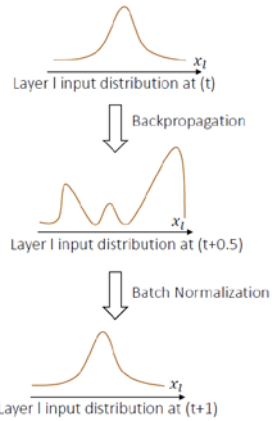
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

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

$\sigma^2_{\boldsymbol{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\boldsymbol{B}})^2$ //mini-batch variance

$\hat{x}_i \leftarrow \frac{x_i - \mu_{\boldsymbol{B}}}{\sqrt{\sigma^2_{\boldsymbol{B}} + \varepsilon}}$ // normalize

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



- Improves gradient flow through the network
- Allows higher learning rates
- Reduces the strong dependence on initialization
- Acts as a form of regularization in a funny way, and slightly reduces the need for dropout, maybe

Note: at test time BatchNorm layer functions differently:

The mean/std are not computed based on the batch. Instead, a single fixe empirical mean of activations during training is used.
(e.g. can be estimated during training with running averages)

Normalize:

$$\hat{x}^{(k)} = \frac{x^{(k)} - E[x^{(k)}]}{\sqrt{Var[x^{(k)}]}}$$

And then allow the network to squash the range if it wants to:

$$y^{(k)} = \gamma^{(k)}\hat{x}^{(k)} + \beta^{(k)}$$

Batch Normalization (Training Time)

- estimate mean and variance from mini-batch
  - cannot do this at test time

Input: $x : N \times D$

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

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

Output: $y : N \times D$

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

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

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

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

Batch Normalization (Test Time)

Input: $x : N \times D$

Learnable params:

$\gamma, \beta : D$

Intermediates:

$\mu, \sigma : D$

$\hat{x} : N \times D$

Output: $y : N \times D$

$\mu_j =$ (Running) average of values seen during training

$\sigma_j^2 =$ (Running) average of values seen during training

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

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

Batch Normalization for ConvNets

Batch Normalization for
fully-connected networks

$x: N \times D$

Normalize

$\mu, \sigma: 1 \times D$

$\gamma, \beta: 1 \times D$

$y = \gamma(x-\mu)/\sigma+\beta$

Batch Normalization for
convolutional networks
(Spatial Batchnorm, BatchNorm2D)

$x: N \times C \times H \times W$

Normalize

$\mu, \sigma: 1 \times C \times 1 \times 1$

$\gamma, \beta: 1 \times C \times 1 \times 1$

$y = \gamma(x-\mu)/\sigma+\beta$

Layer Normalization

Layer Normalization for fully-connected networks
Same behavior at train and test!
Can be used in recurrent networks

$x: N \times D$

Normalize

$\mu, \sigma: N \times 1$

$\gamma, \beta: 1 \times D$

$y = \gamma(x-\mu)/\sigma+\beta$

cp.

Batch Normalization for fully-connected networks

$x: N \times D$

Normalize

$\mu, \sigma: 1 \times D$

$\gamma, \beta: 1 \times D$

$y = \gamma(x-\mu)/\sigma+\beta$

Ba, Kiros, and Hinton, "Layer Normalization", arXiv 2016

Instance Normalization

Instance Normalization for convolutional networks
Same behavior at train / test!

x: N×C×H×W

Normalize

μ, σ: N×C×1×1

γ, β: 1×C×1×1

$y = \gamma(x-\mu)/\sigma+\beta$

cp.

Batch Normalization for convolutional networks

x: N×C×H×W

Normalize

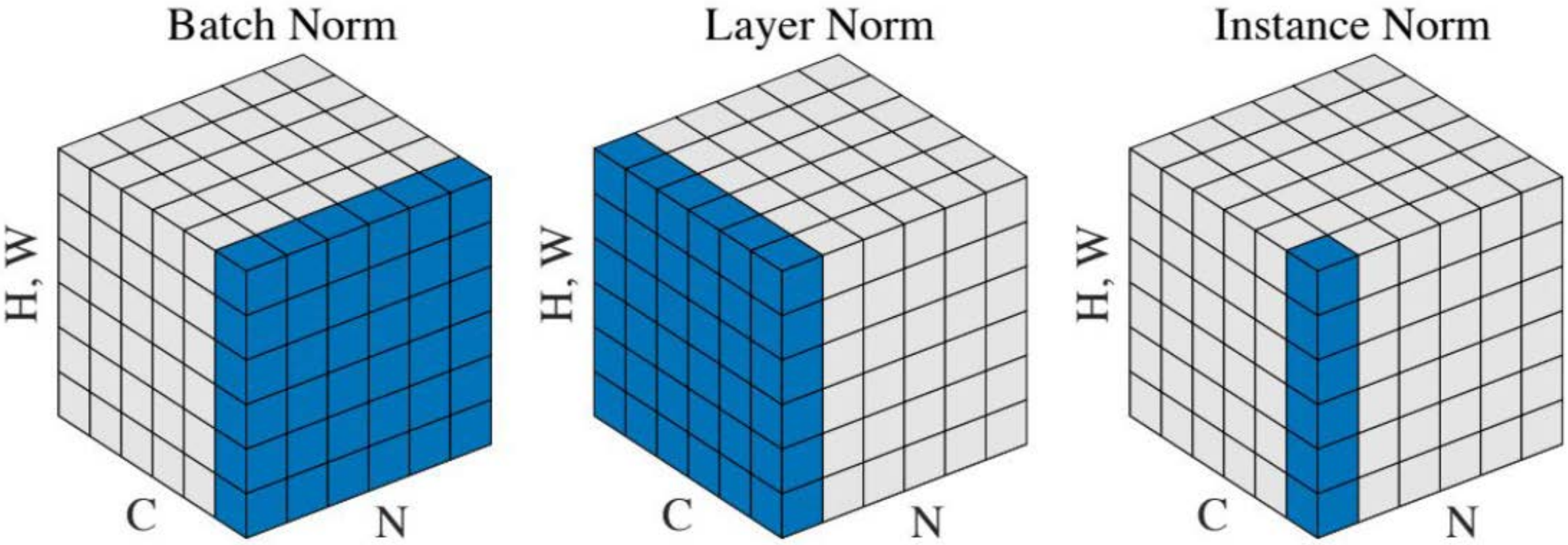
μ, σ: 1×C×1×1

γ, β: 1×C×1×1

$y = \gamma(x-\mu)/\sigma+\beta$

Ulyanov et al, Improved Texture Networks: Maximizing Quality and Diversity in Feed-forward Stylization and Texture Synthesis, CVPR 2017

Group Normalization



Wu and He, “Group Normalization”, arXiv 2018

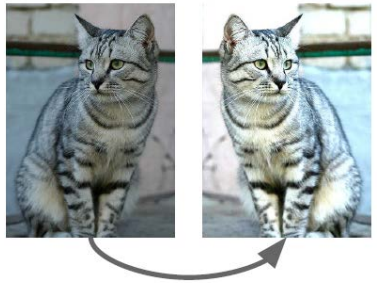
Local Response Normalization (LRN)

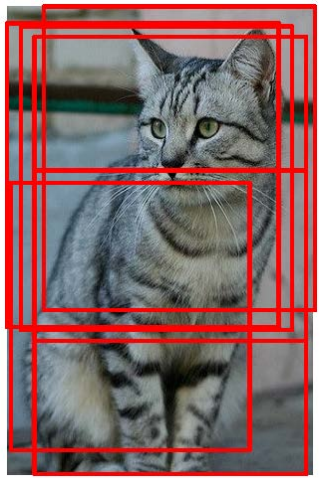
- only used in AlexNet CNN model
- sum over n "adjacent" kernel maps at the same spatial location
  - N is the total number of kernels in the layer
  - k, n, α, β are hyper-parameters whose values are determined by validation set
- implement a form of lateral inhibition inspired by the real neurons
  - create competition for big activities among neuron outputs computed using different kernels

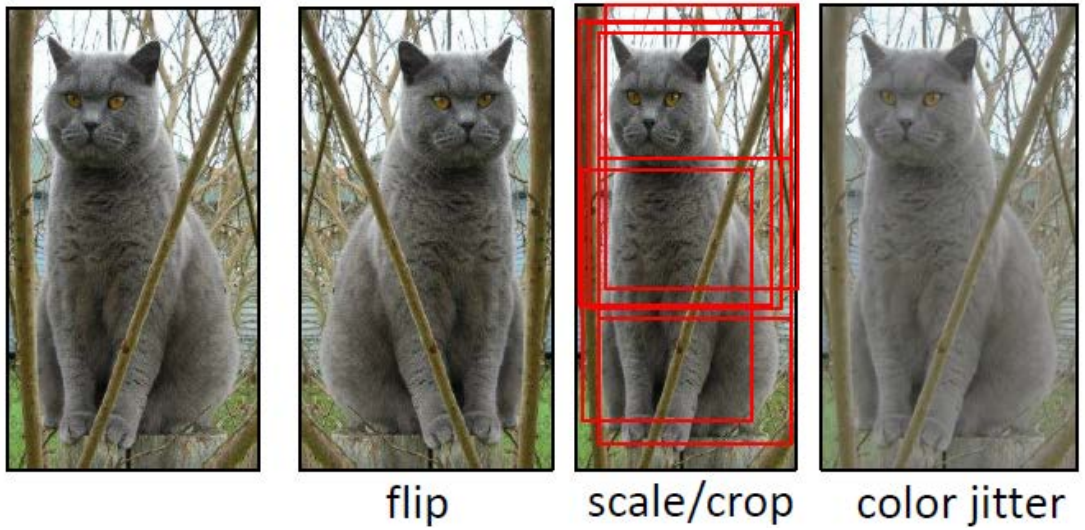
$$b_{x,y}^{i} = a_{x,y}^{i} / \left(k + \alpha \sum_{j=\max(0,i-n/2)}^{\min(N-1,i+n/2)} (a_{x,y}^{j})^2 \right)^{\beta}$$

data augmentation

- horizontal flip

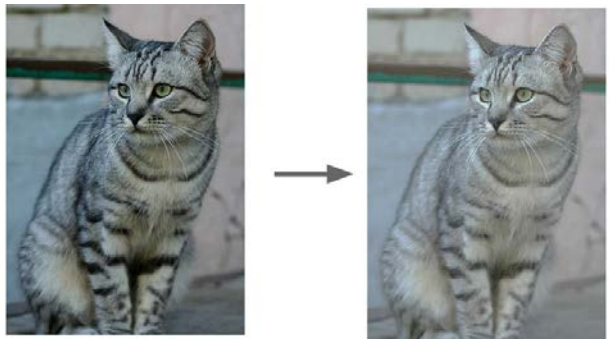






- crops and scales

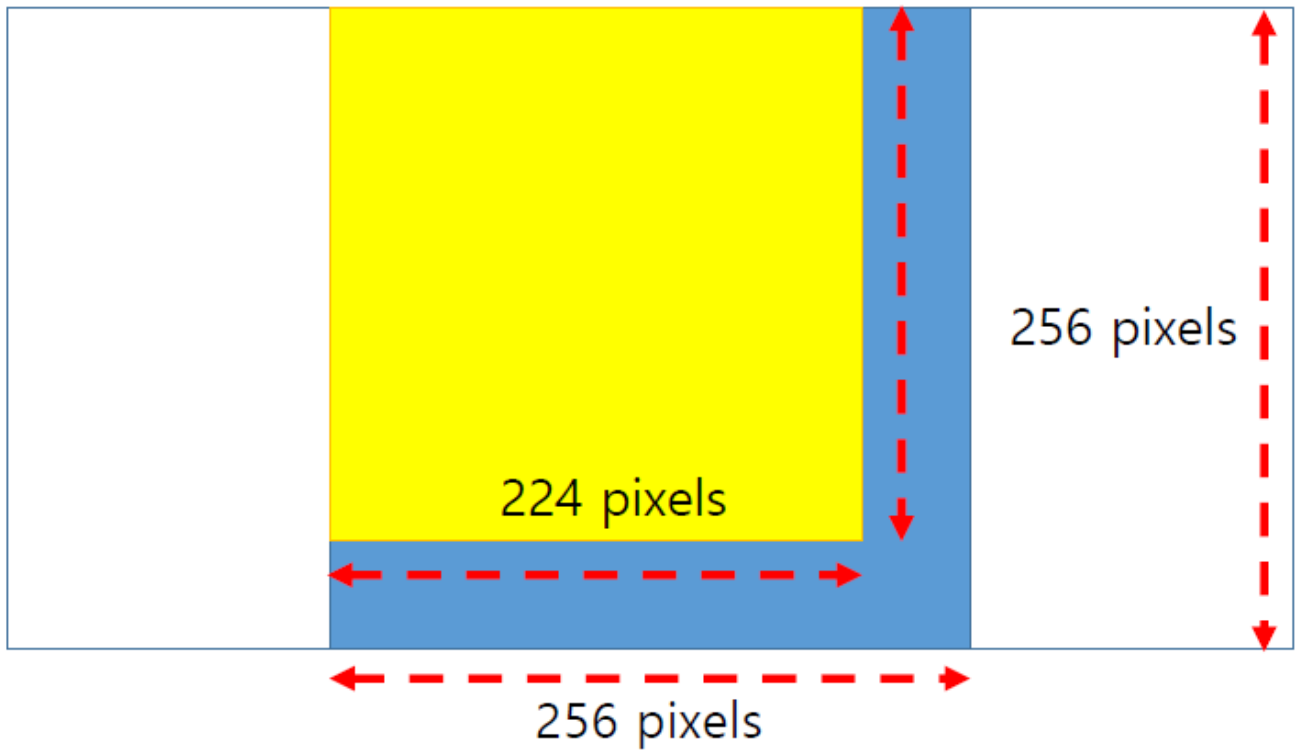
- color jitters

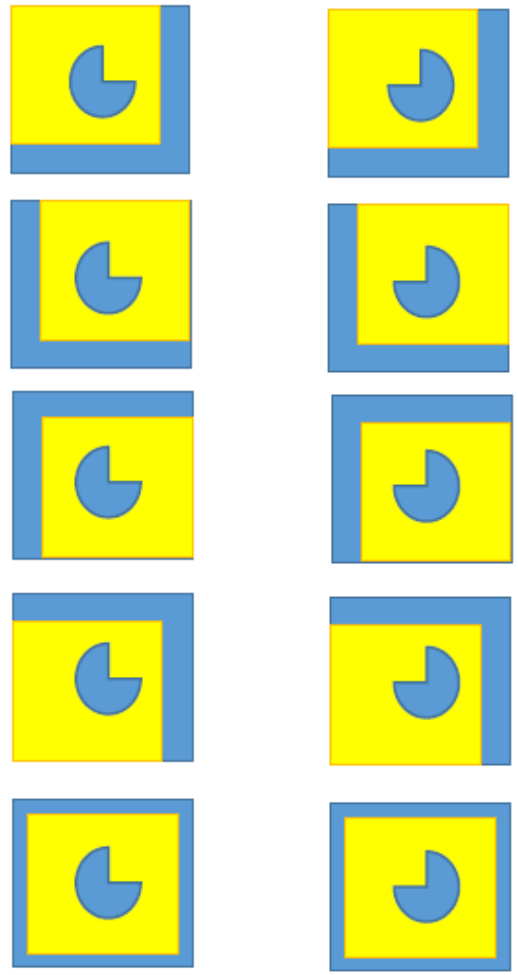


- translation, rotation, stretching, shearing, lens distortion, …

Data Augmentation in AlexNet

- Given an original image, first resize it to fit 256 pixel to the smaller side
- Crop 10 224x224 patches (4 corner and 1 center crops)
- Prediction based on the average of 10 softmax outputs

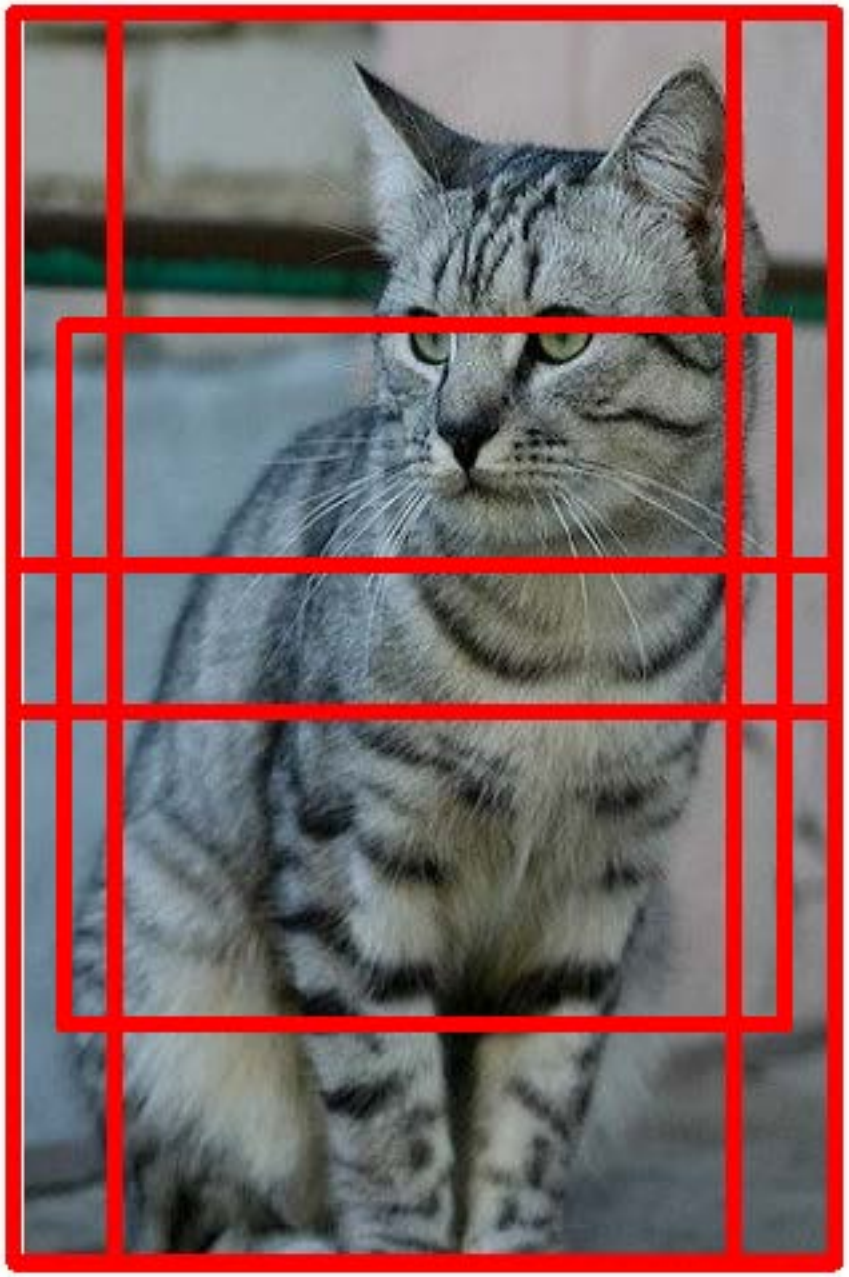




data augmentation in ResNet

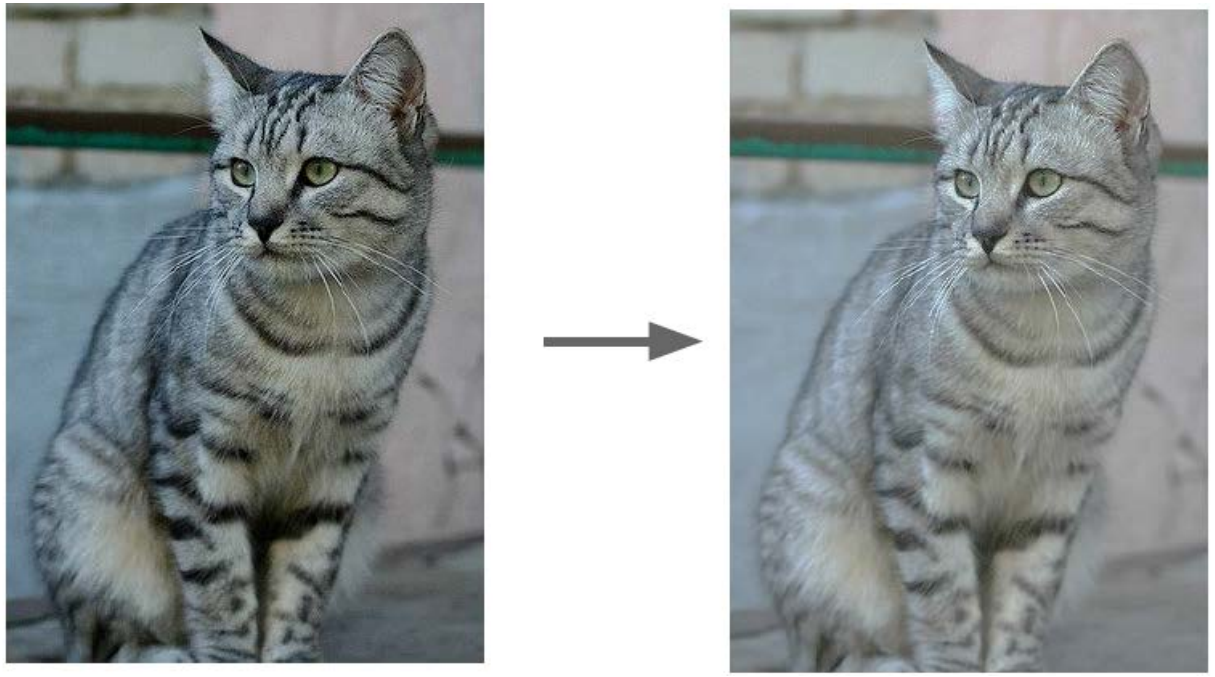
- training : sample random crops/scales
  - pick random L in range [256, 480]
  - resize training image, short side=L
  - sample random 224x224 patch

- testing: average a fixed set of crops
  - resize image at 5 scales: {224, 256, 384, 480, 640}
  - for each size, use 10 224x224 crops: 4 corners+center, +flips



data augmentation (color jitter)

- simple:
  - randomize contrast and brightness
- more complex:
  - apply PCA to all [R,G,B] pixels in training set
  - sample a “color offset” along principle component direction
  - add offset to all pixels of a training image
  - as seen in AlexNet and ResNet



other data augmentation methods

- translation
- rotation
- stretching
- shearing
- lens distortion
- …

Transfer Learning

- a CNN has millions of parameters, but our datasets are not always as large
  - wish to still train CNN without overfitting problems
- pre-train a network on the dataset S, then
  - solution I: fine-tune with dataset T, or
  - solution II: CNN as feature extractor
- Solution I: fine-tuning
  - Assume two datasets, T and S
  - Dataset S is
    - fully annotated, plenty of images
    - We can build a model h_S
  - Dataset T is
    - Not as much annotated, or much fewer images
    - The annotations of T do not need to overlap with S
  - We can use the model h_S to learn a better h_T
  - This is called transfer learning

Imagenet: 1million

h_A

My dataset: 1,000

h_B

Solution I: Fine-tune h_T using h_S as initialization

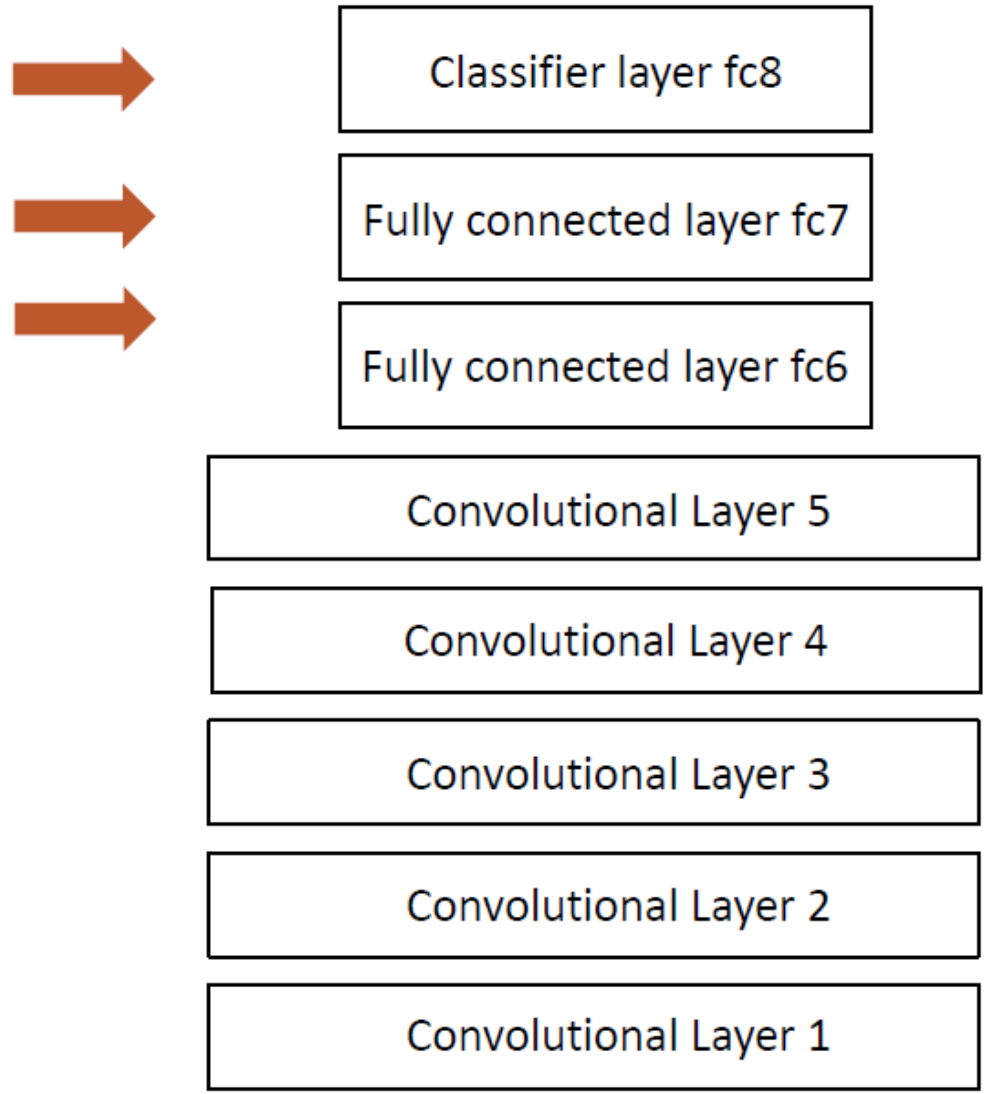
- Assume the parameters of S are already a good start near our final local optimum
- Use them as the initial parameters for our new CNN for the target dataset

$$w^l_{\mathrm{T},\, t=0} = w^l_{\mathrm{S},\, t=\infty} \text{ for layers } l = 1, 2, \ldots$$

- Use when the target dataset T is relatively big
  - E.g. for Imagenet S with approximately 1 million images a dataset T with more than a few thousand images should be ok
- What layers to initialize and how?

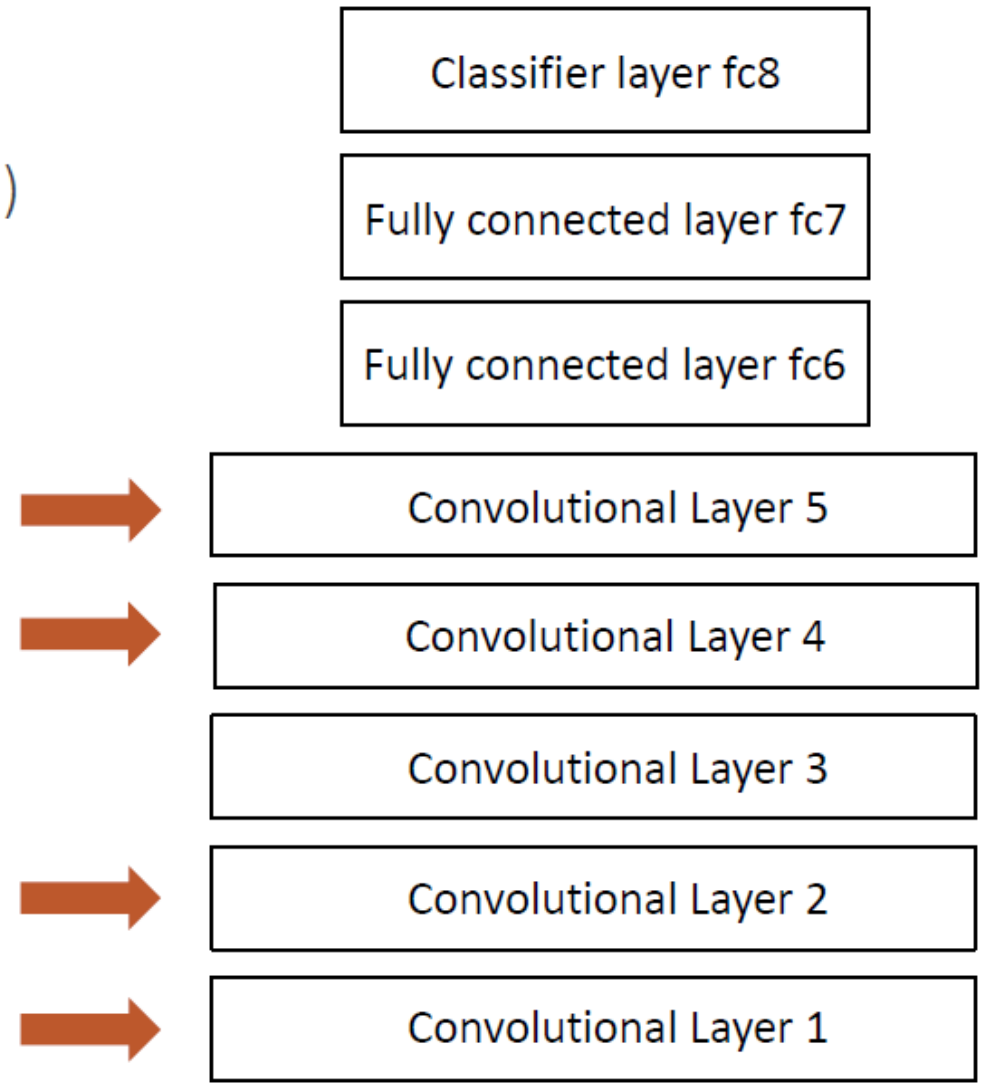
Initializing h_T with h_S

- Classifier layer to loss
  - The loss layer essentially is the "classifier"
  - Same labels → keep the weights from h_S
  - Different labels → delete the layer and start over
  - When too few data, fine-tune only this layer
- Fully connected layers
  - Very important for fine-tuning
  - Sometimes you need to completely delete the last before the classification layer if datasets are very different
  - Capture more semantic, "specific" information
  - Always try first when fine-tuning
  - If you have more data, fine-tune also these layers



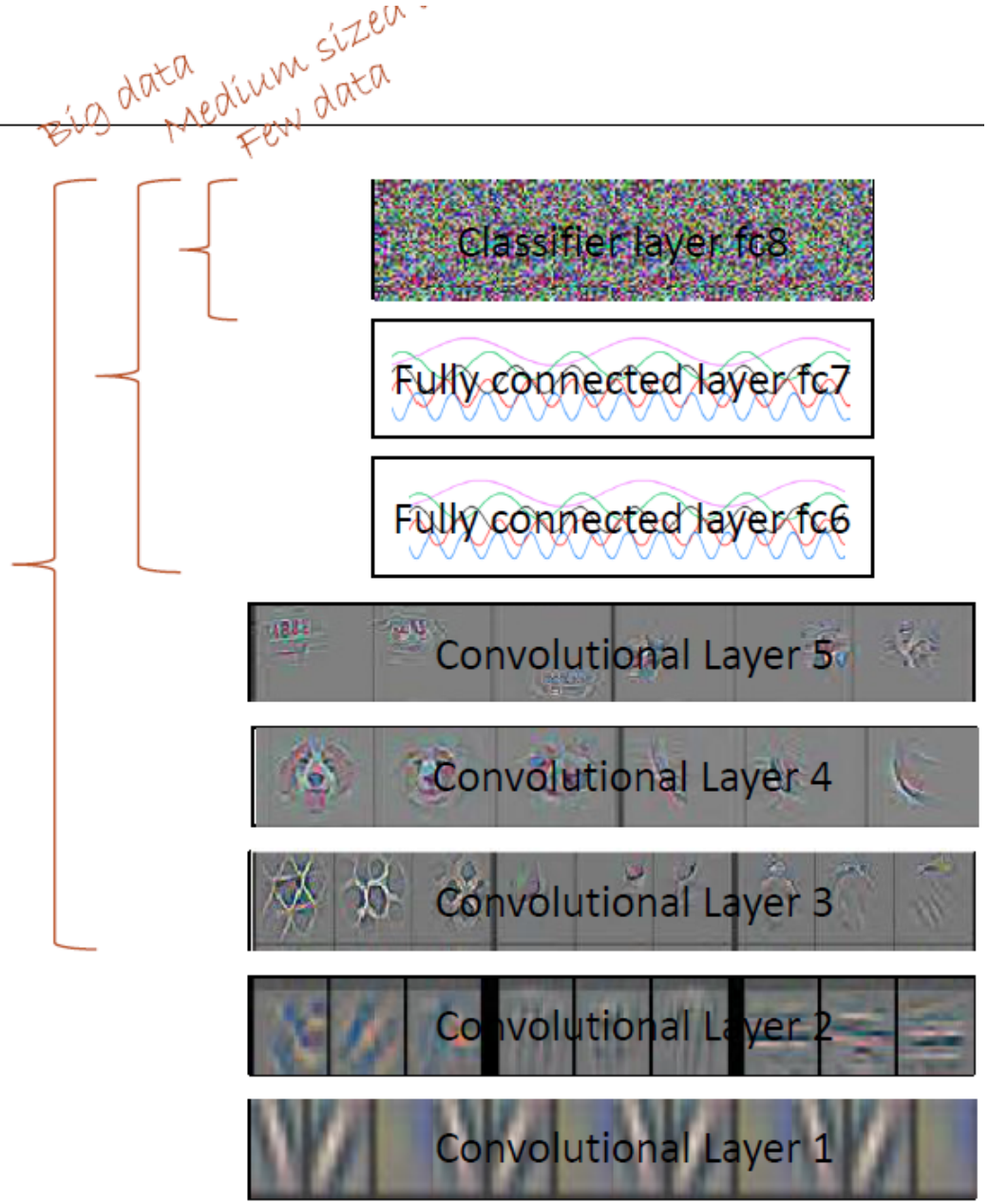
Initializing h_T with h_S

- Upper convolutional layers (conv4, conv5)
  - Mid-level spatial features (face, wheel detectors …)
  - Can be different from dataset to dataset
  - Capture more generic information
  - Fine-tuning pays off
  - Fine-tune if dataset is big enough
- Lower convolutional layers (conv1, conv2)
  - They capture low level information
  - This information does not change usually
  - Probably, no need to fine-tune but no harm trying



How to fine-tune?

- For layers initialized from h_S use a mild learning rate
  - Remember: your network is already close to a near optimum
  - If too aggressive, learning might diverge
  - A learning rate of 0.001 is a good starting choice (assuming 0.01 was the original learning rate)
- For completely new layers (e.g. loss) use aggressive learning rate
  - If too small, the training will converge very slowly
  - Remember: the rest of the network is near a solution, this layer is very far from one
  - A learning rate of 0.01 is a good starting choice
- If datasets are very similar, fine-tune only fully connected layers
- If datasets are different and you have enough data, fine-tune all layers



Solution II: Use h_s as a feature extractor for h_T

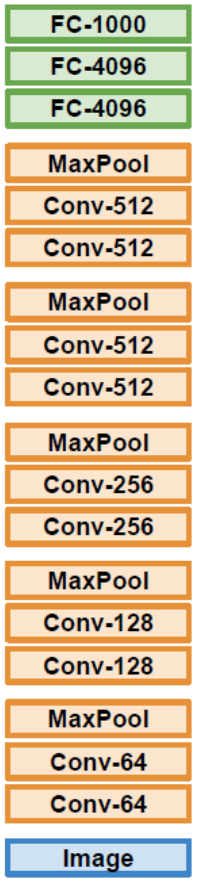
- Essentially similar to a case of solution I
  - but train only the loss layer
- Essentially use the network as a pretrained feature extractor
- Use when the target dataset T is small
  - Any fine-tuning of layer might cause overfitting
  - Or when we don't have the resources to train a deep net
  - Or when we don't care for the best possible accuracy

transfer learning

Tansfer Learning with CNNs

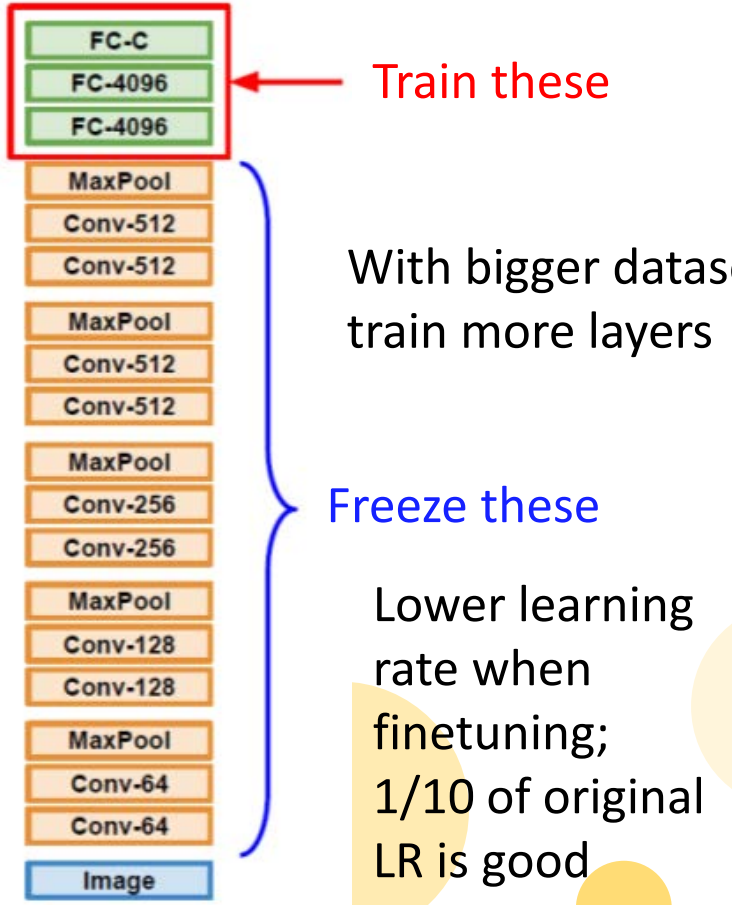
Donahue et al, "DeCAF: A Deep Convolutional Activation Feature for Generic Visual Recognition", ICML 2014
Razavian et al, "CNN Features Off-the-Shelf: An Astounding Baseline for Recognition", CVPR Workshops 2014

1.Train on Imagenet



2.Small Dataset (C classes)

3.Bigger dataset



Lower learning rate when finetuning; 1/10 of original LR is good starting point

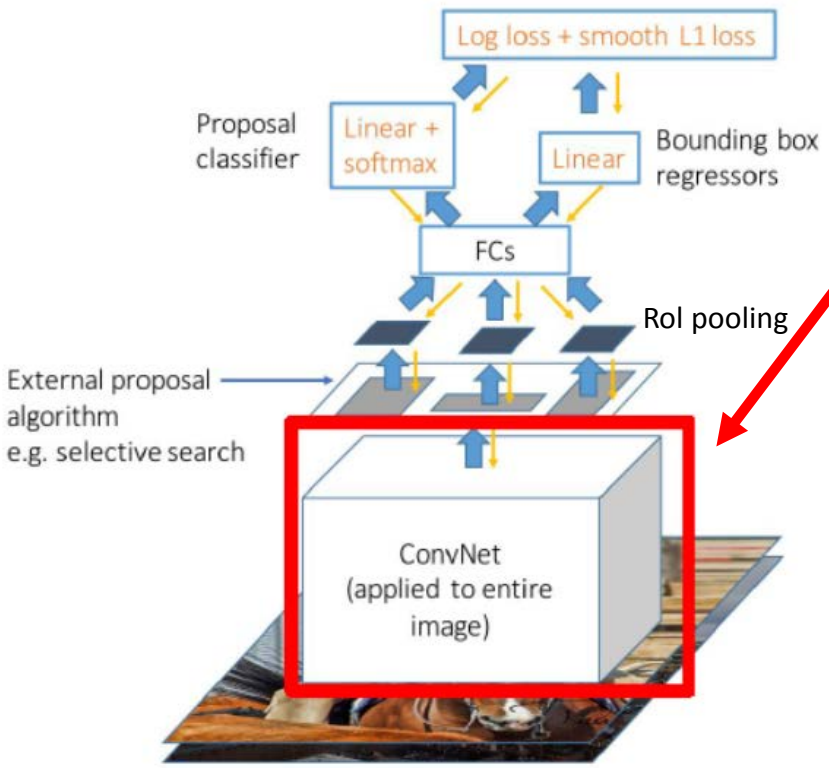
Transfer learning with CNNs is pervasive...

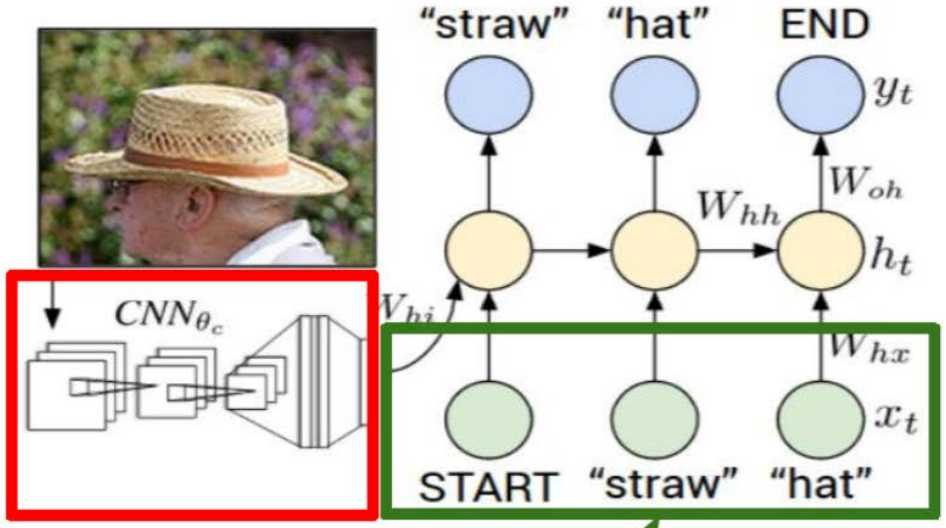
(It's the norm, not an exception)

Object Detection
(Fast R-CNN)

CNN pretrained
on ImageNet

Image Captioning: CNN + RNN



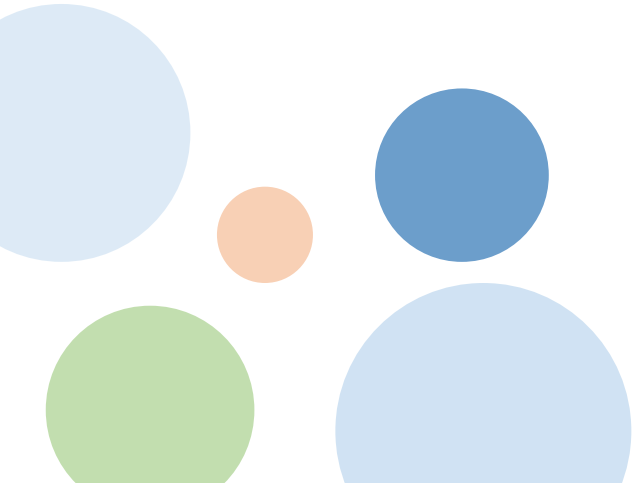


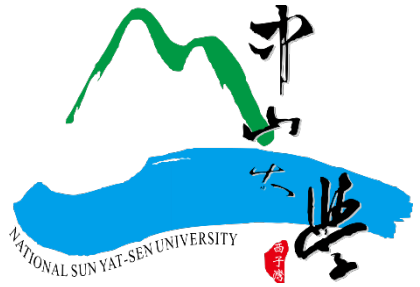
Word vectors pretrained
with word2vec

Girshick, "Fast R-CNN", ICCV 2015
Figure copyright Ross Girshick, 2015.Reproduced with permission.

Karpathy and Fei-Fei, 'Beep Visual-Semantic Alignments for Generating Image Descriptions", CVPR 2015
Figure copyright IEEE, 2015. Reproduced for educational purposes.

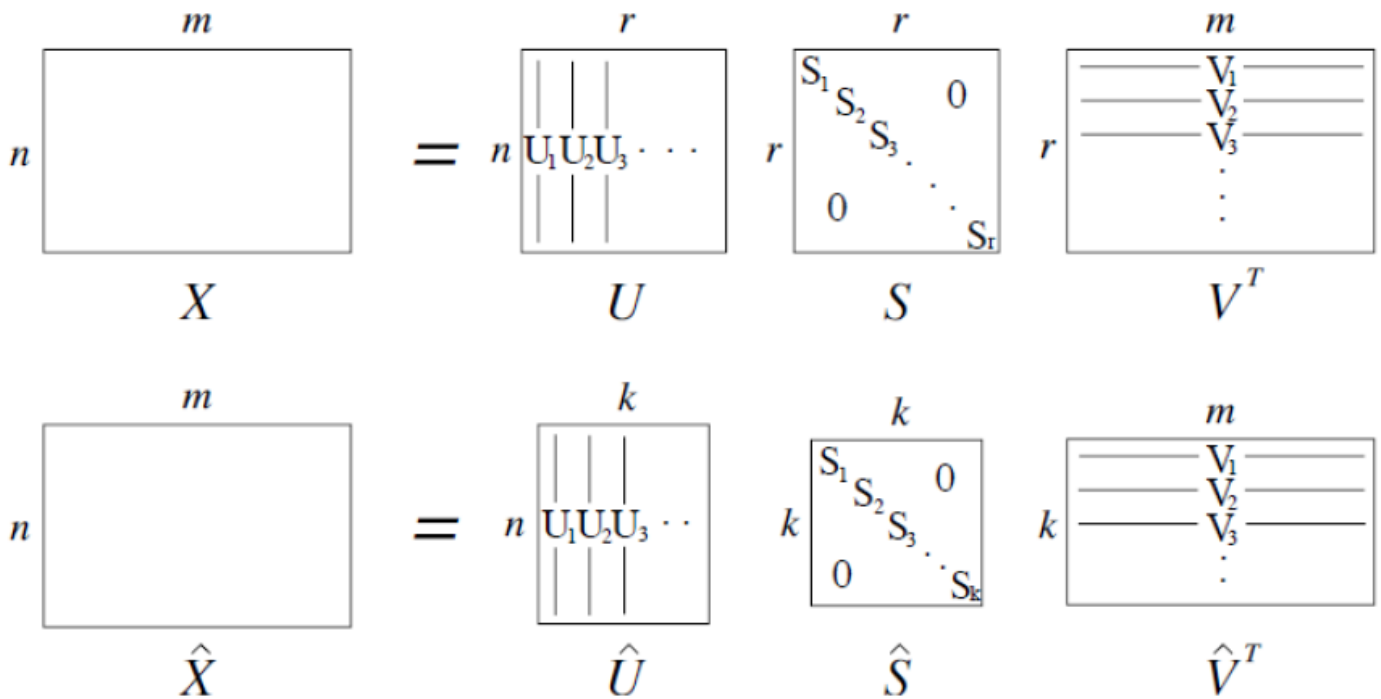
Model Compression and Quantization





[http://cs224d.stanford.edu/lectures/CS224d-Lecture2.pdf]

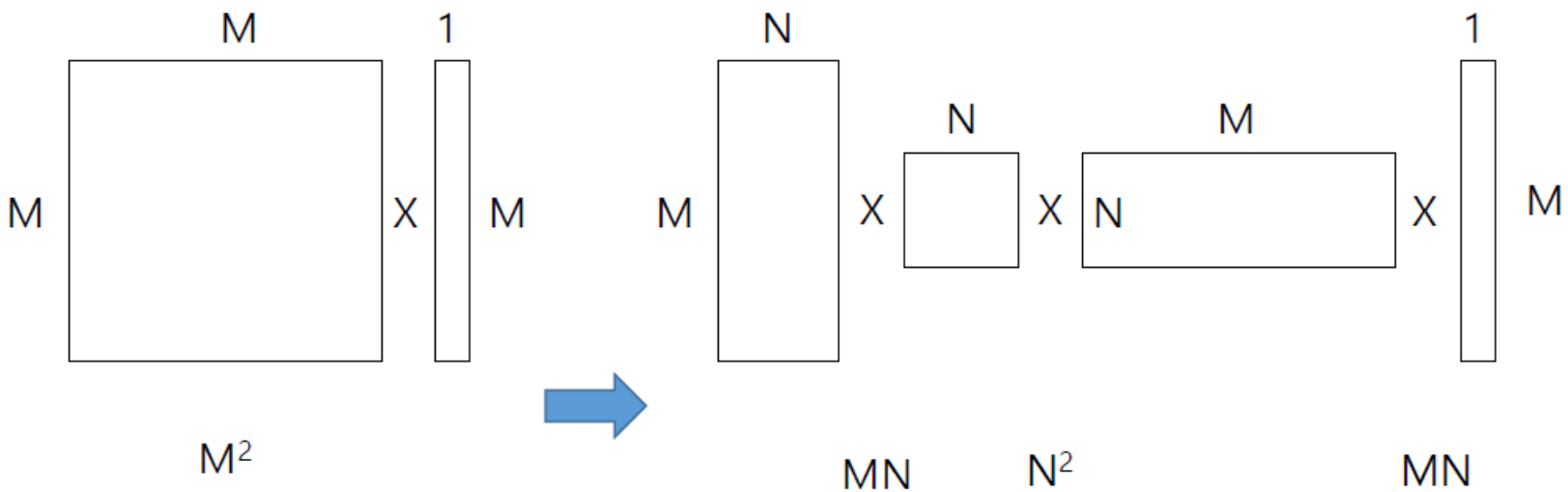
A Low-Rank Approximation: Truncated Singular Value Decomposition (SVD)



$\hat{X}$ is the best rank k approximation to X, in terms of least squares.

Truncated SVD can Reduce Computation

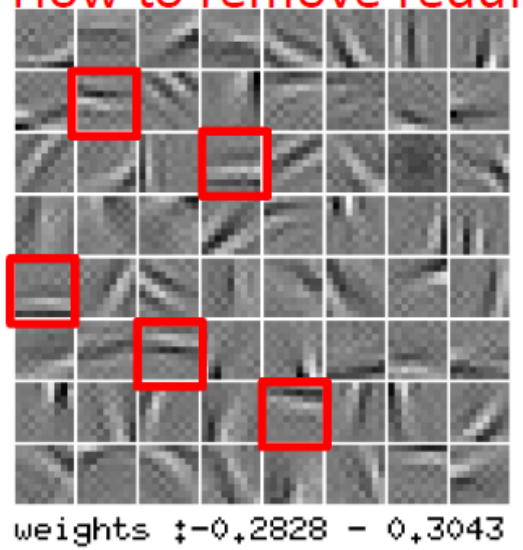
- MM x M → MN x NN x NM x M
- # multiplications: M^2 → $2MN + N^2$
  - M=100, N=20
  - 10000 → 4400 (56% reduction)



Tucker Decomposition to Resolve Redundancy Problem: Reducing # Feature Maps

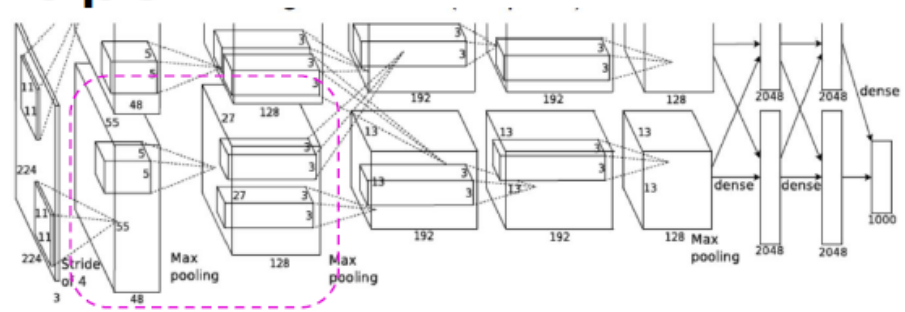
- **Problem:**
  - With patch-level training, the learning algorithm must reconstruct the entire patch with a single feature vector
  - But when the filters are used convolutionally, neighboring feature vectors will be highly redundant

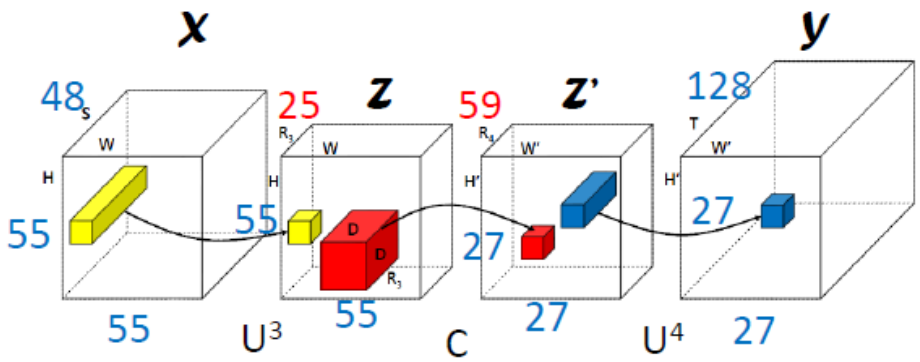
How to remove redundant feature maps?



Yann LeCun

Patch-level training produces lots of filters that are shifted versions of each other.



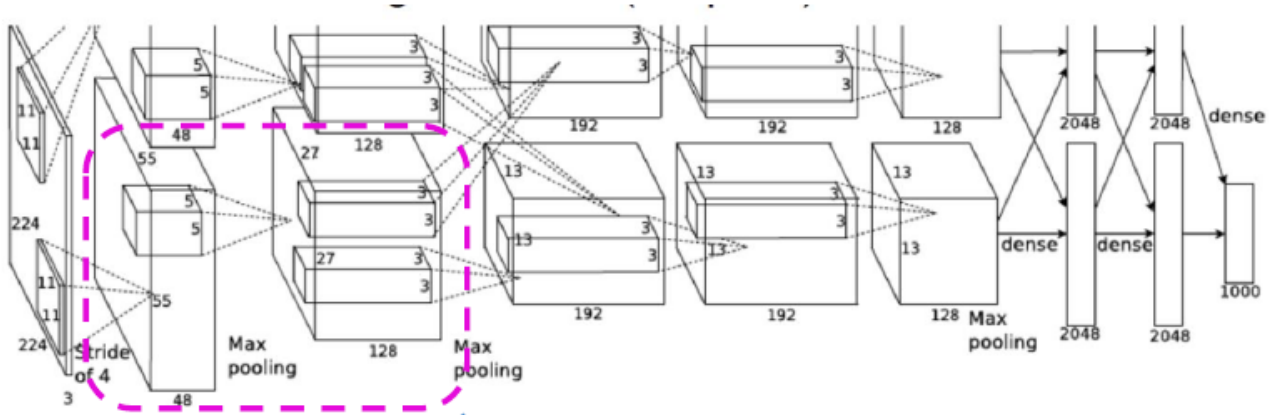


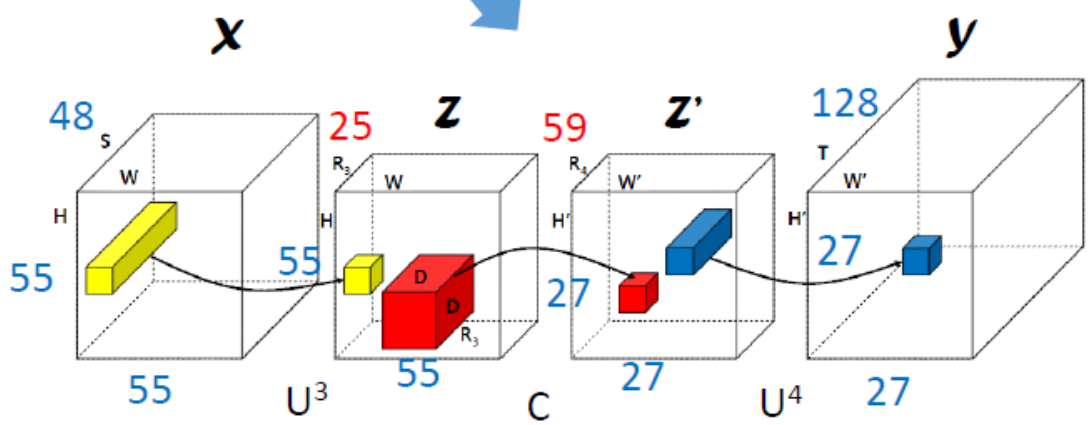
feature maps is reduced at input (48→25 in $\boldsymbol{z}$) and output (128→59 in $\boldsymbol{z}$')

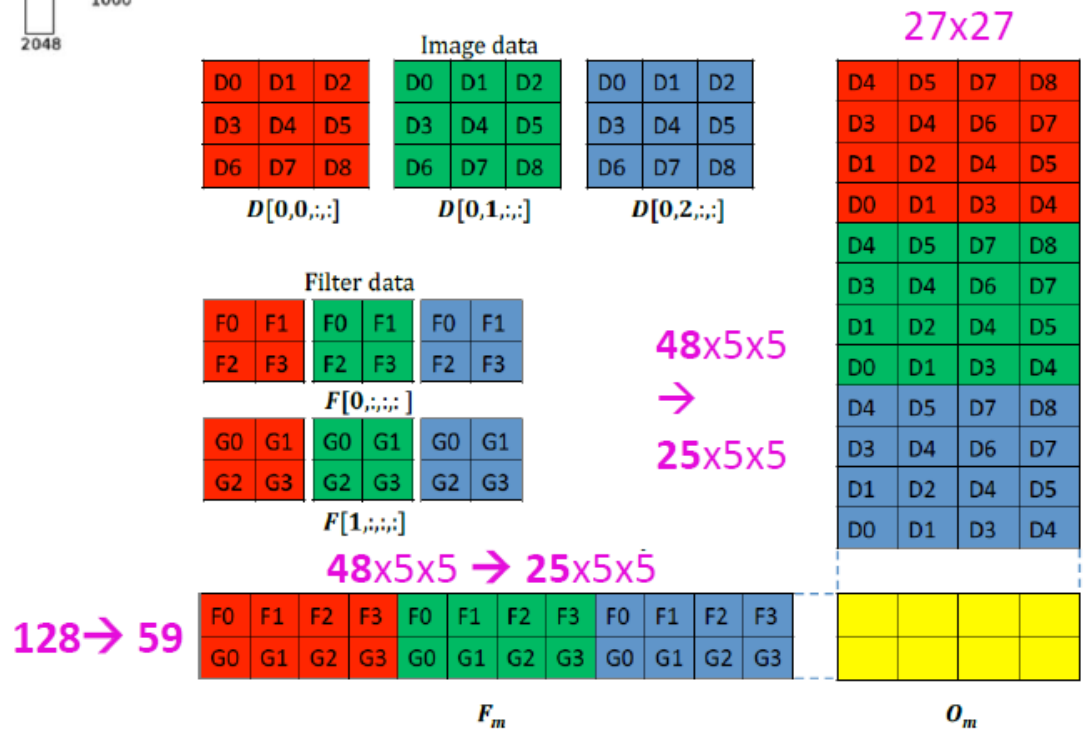
1x1 convolutions are used at both input and output to match with the original layers

[Kim, ICLR 2016]

Effect of Low-rank Approximation





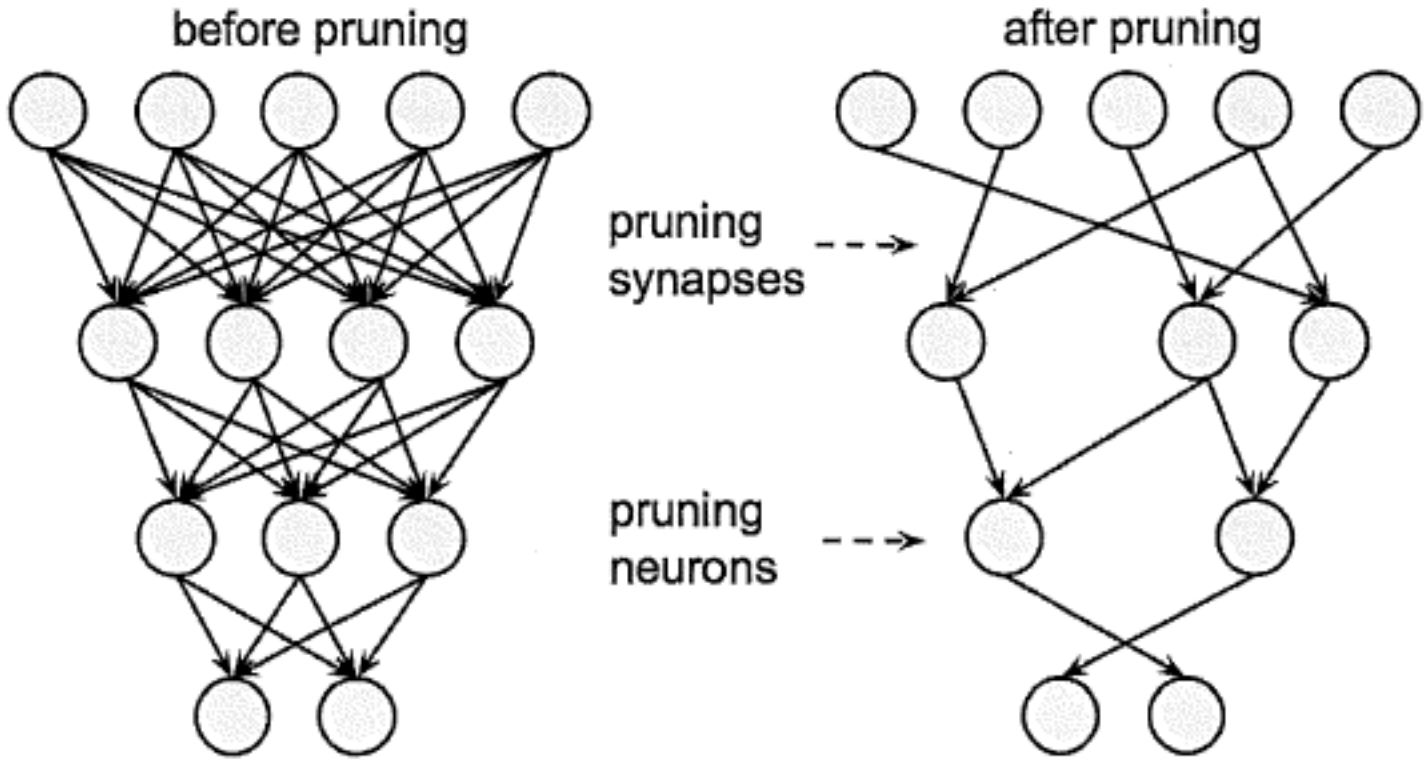


Network Pruning

- Song Han, et al., “Learning both Weights and Connections for Efficient Neural Networks,” 2015 NIPS.

Network Pruning

We can take advantage of the redundancy in large-scale DNNs by zero-out some unimportant weights or filters

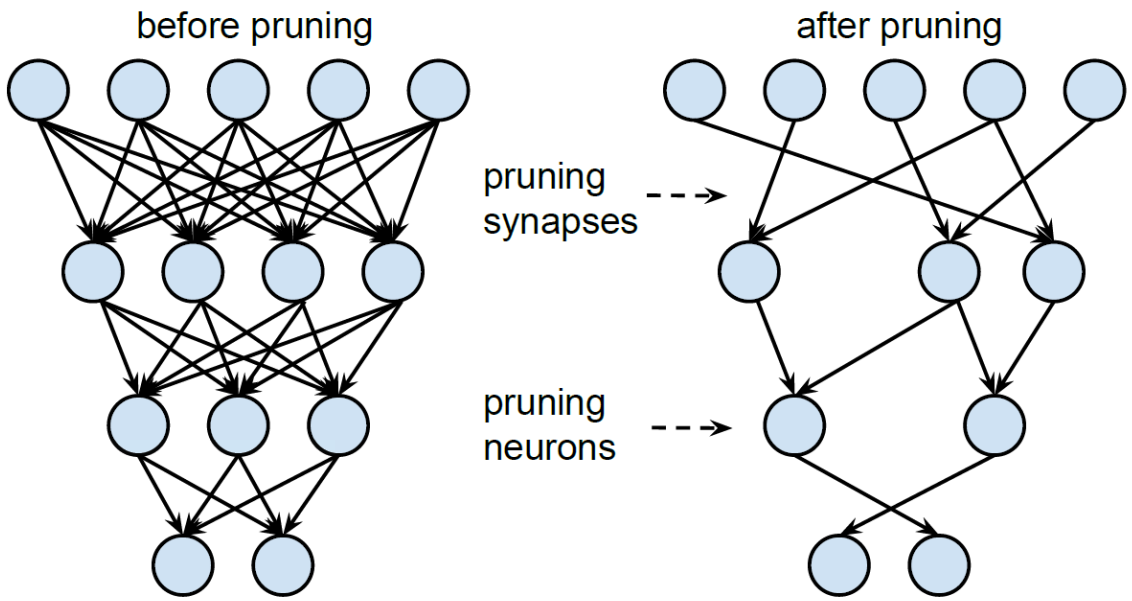


Synapses and neurons before and after pruning(fine tuning)

Pruning Neural Network

- removing weights of small magnitudes recursively
  - remove connections with weight below a per-layer adjustable threshold
    - threshold chosen as a quantity parameter multiplied by standard deviation of a layer's weights
  - retrain dropout ratio should be smaller
  - L2 regularization is better than L1

$$J(\boldsymbol{w}) = \mathrm{MSE}_{\mathrm{train}} + \lambda \boldsymbol{w}^{\top} \boldsymbol{w},$$



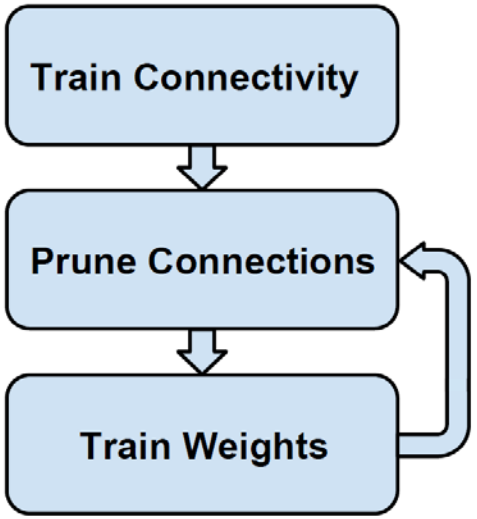
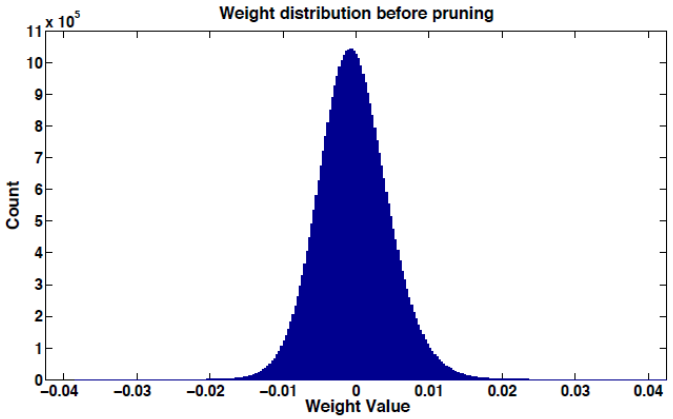
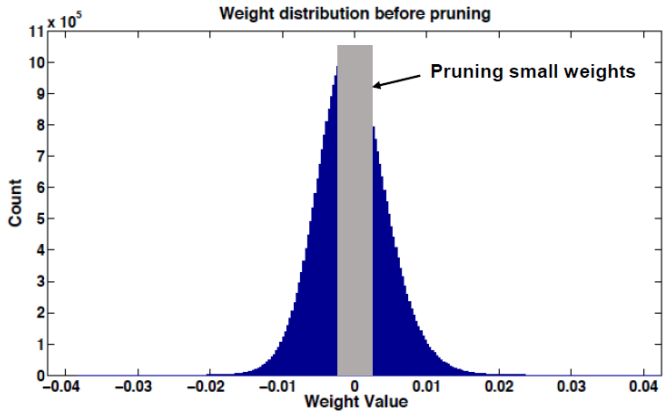


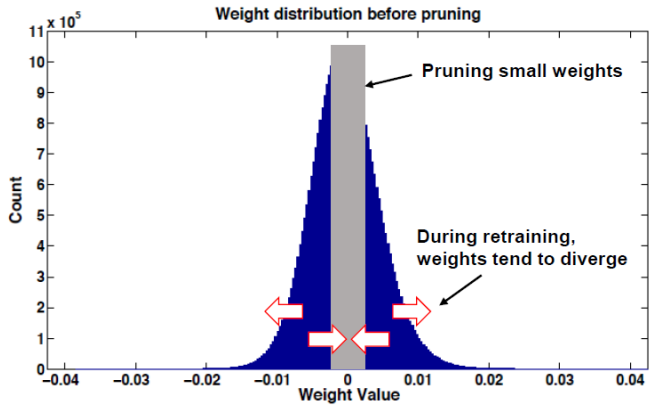
Illustration of Iterative Pruning-Retraining



weight distribution before pruning



small weights are set to zero (pruned)



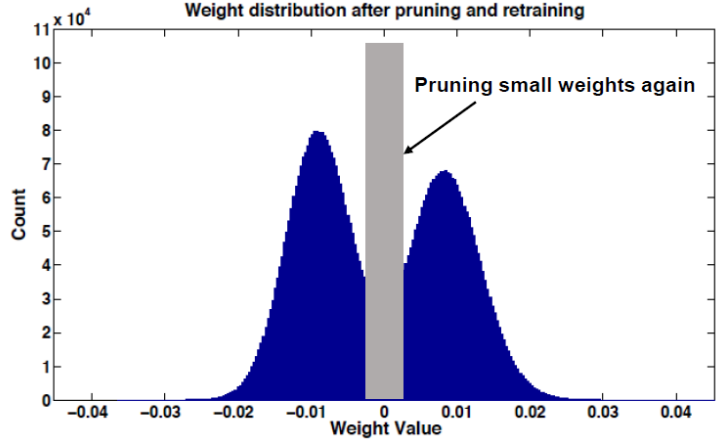
during training, weights are re-distributed

[Han, 2015]

Accuracy, Compression Ratio and # Bits (Quantization Levels)

Network	Top-1 Error	Top-5 Error	Parameters	Compress Rate
LeNet-300-100 Ref	1.64%	-	1070 KB	
LeNet-300-100 Compressed	1.58%	-	**27 KB**	40×
LeNet-5 Ref	0.80%	-	1720 KB	
LeNet-5 Compressed	0.74%	-	**44 KB**	39×
AlexNet Ref	42.78%	19.73%	240 MB	
AlexNet Compressed	42.78%	19.70%	**6.9 MB**	35×
VGG-16 Ref	31.50%	11.32%	552 MB	
VGG-16 Compressed	31.17%	10.91%	**11.3 MB**	49×

	#CONV bits / #FC bits	Top-1 Error	Top-5 Error	Top-1 Error Increase	Top-5 Error Increase
AlexNet	32bits / 32bits	42.78%	19.73%	-	-
	8 bits / 5 bits	42.78%	19.70%	0.00%	-0.03%
	8 bits / 4 bits	42.79%	19.73%	0.01%	0.00%
	4 bits / 2 bits	44.77%	22.33%	1.99%	2.60%



10X smaller scale

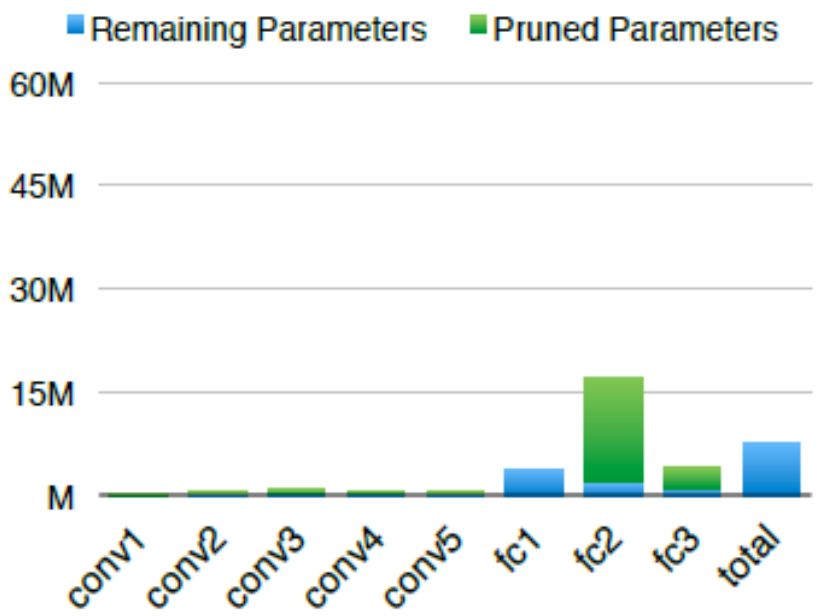
after a step of training, new small weights are obtained

AlexNet Pruning

- # of parameters from 61M to 6.7M (=61*11%)
- # of operations from 1.5Gop to 0.45Gop (1.5*30%)
- more FC-layer weights removed cp. Conv layers
- retrain with 1/100 of the original network's original learning rate
  - cp. with 1/10 learning rate in LeNet

Layer	Weights	FLOP	Act%	Weights%	FLOP%
conv1	35K	211M	88%	84%	84%
conv2	307K	448M	52%	38%	33%
conv3	885K	299M	37%	35%	18%
conv4	663K	224M	40%	37%	14%
conv5	442K	150M	34%	37%	14%
fc1	38M	75M	36%	9%	3%
fc2	17M	34M	40%	9%	3%
fc3	4M	8M	100%	25%	10%
Total	61M	1.5B	54%	**11%**	**30%**

some colors missing below



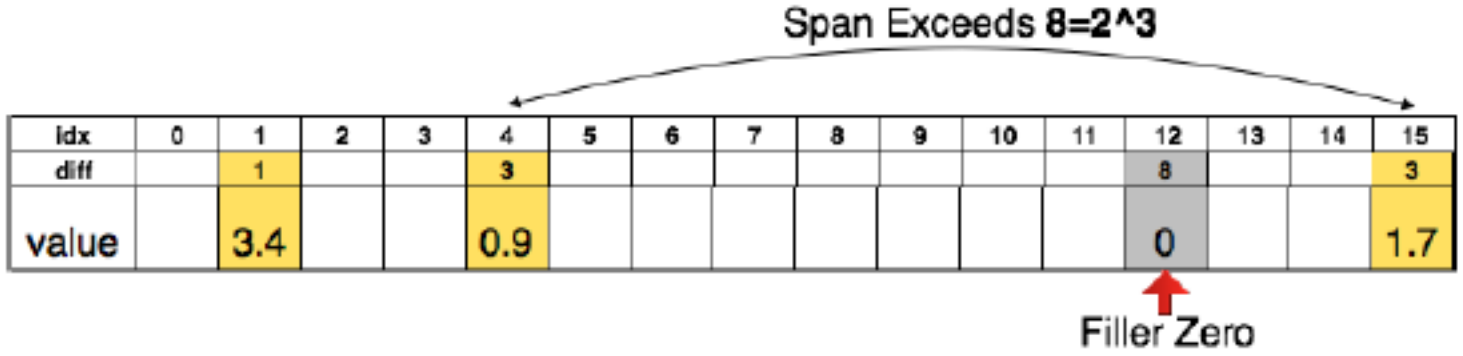
VGG-16 Pruning

- # of parameters from 138M to 10.35M (=138*7.5%)
- operation count from 30.9Gop to 6.5 (=30.9*21%)
- after pruning of AlexNet and VGG, weights can be stored on chip

Layer	Weights	FLOP	Act%	Weights%	FLOP%
conv1_1	2K	0.2B	53%	58%	58%
conv1_2	37K	3.7B	89%	22%	12%
conv2_1	74K	1.8B	80%	34%	30%
conv2_2	148K	3.7B	81%	36%	29%
conv3_1	295K	1.8B	68%	53%	43%
conv3_2	590K	3.7B	70%	24%	16%
conv3_3	590K	3.7B	64%	42%	29%
conv4_1	1M	1.8B	51%	32%	21%
conv4_2	2M	3.7B	45%	27%	14%
conv4_3	2M	3.7B	34%	34%	15%
conv5_1	2M	925M	32%	35%	12%
conv5_2	2M	925M	29%	29%	9%
conv5_3	2M	925M	19%	36%	11%
fc6	103M	206M	38%	4%	1%
fc7	17M	34M	42%	4%	2%
fc8	4M	8M	100%	23%	9%
total	138M	30.9B	64%	**7.5%**	**21%**

Discussions

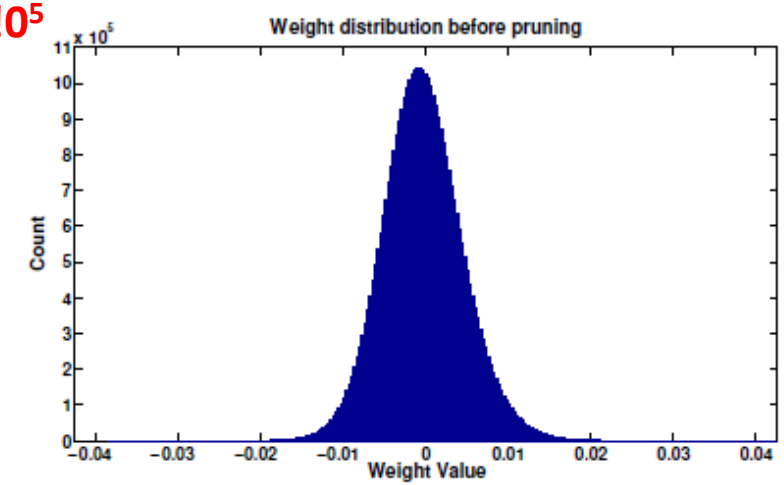
- 1st Conv. layer is more sensitive to pruning
  - due to only 3 channels with less redundancy than other layers
  - smallest threshold for the most sensitive layer (1st Conv. layer)
- Storing pruned layers as sparse matrices using relative rather than absolute indices
  - only 5-bit indices for FC layers and 8-bit indices for Conv layers



Comparison (AlexNet)

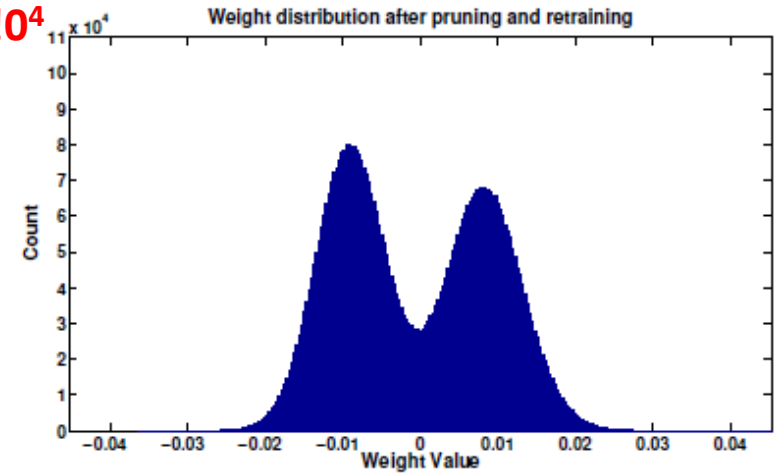
Network	Top-1 Error	Top-5 Error	Parameters	Compression Rate
Baseline Caffemodel [26]	42.78%	19.73%	61.0M	1×
Data-free pruning [28]	44.40%	-	39.6M	1.5×
Fastfood-32-AD [29]	41.93%	-	32.8M	2×
Fastfood-16-AD [29]	42.90%	-	16.4M	3.7×
Collins & Kohli [30]	44.40%	-	15.2M	4×
Naive Cut	47.18%	23.23%	13.8M	4.4×
SVD [12]	44.02%	20.56%	11.9M	5×
Network Pruning	**42.77%**	**19.67%**	**6.7M**	**9×**

x!0^5



most in [-0.015, 0.015]

x!0^4

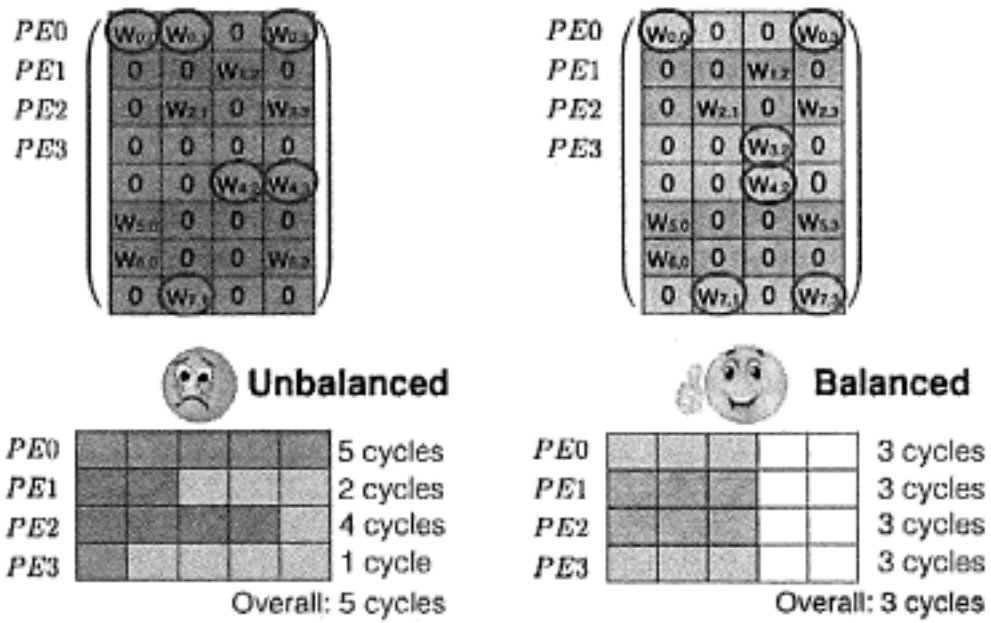


most in [-0.025, 0.025]

Figure 7: Weight distribution before and after parameter pruning. The right figure has 10× smaller scale.

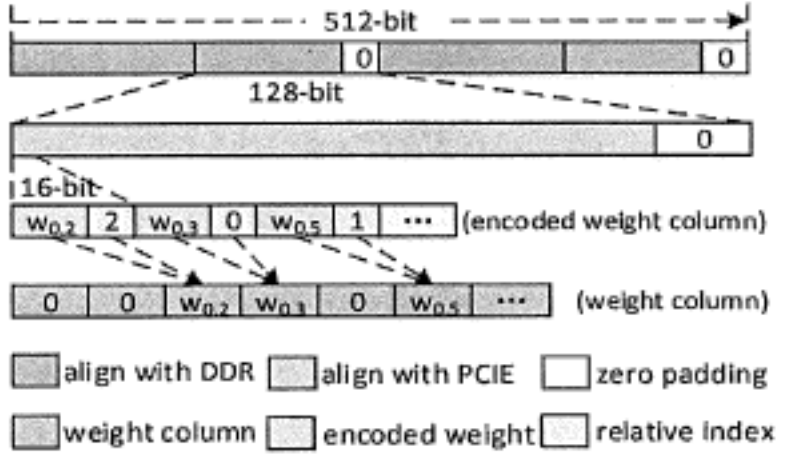
Model Compression

Load Balance-Aware Pruning



12-bit fixed point data in CSC format

Networks	WER
32bit floating original network	20.3%
32bit floating pruned network	20.7%
16bit fixed pruned network	20.7%
12bit fixed pruned network	20.7%
8bit fixed pruned network	84.5%



Weight Sharing

- Song Han, et al., “Deep Compression, Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding,” ICLR 2016.

Pruning + Weight-Sharing + Huffman-Coding

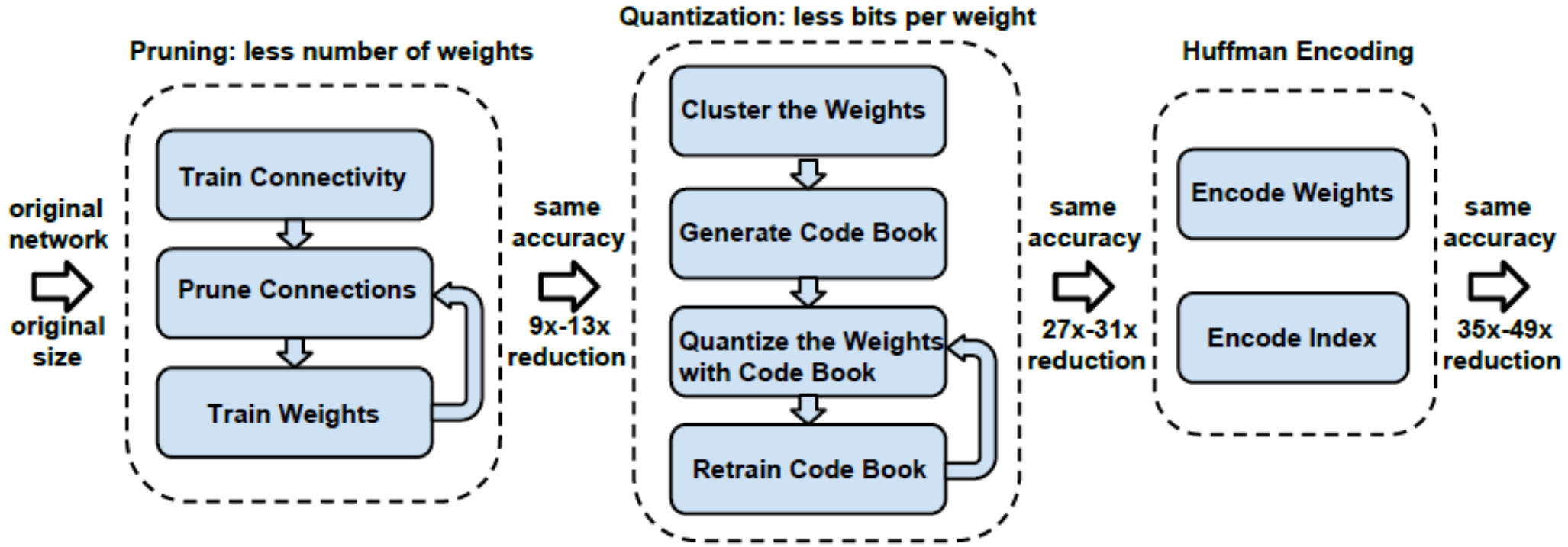


Figure 1: The three stage compression pipeline: pruning, quantization and Huffman coding. Pruning reduces the number of weights by 10×, while quantization further improves the compression rate: between 27× and 31×. Huffman coding gives more compression: between 35× and 49×. The compression rate already included the meta-data for sparse representation. The compression scheme doesn't incur any accuracy loss.

Weight Sharing with Fine-Tuned Centroids

- three methods of centroid initialization
  - foggy (random), density-based, and linear
- compress rate = n*b / (n*$\log_2$k + kb), n: #weights, b: #bits/weight, k: #shared weights (#clusters)

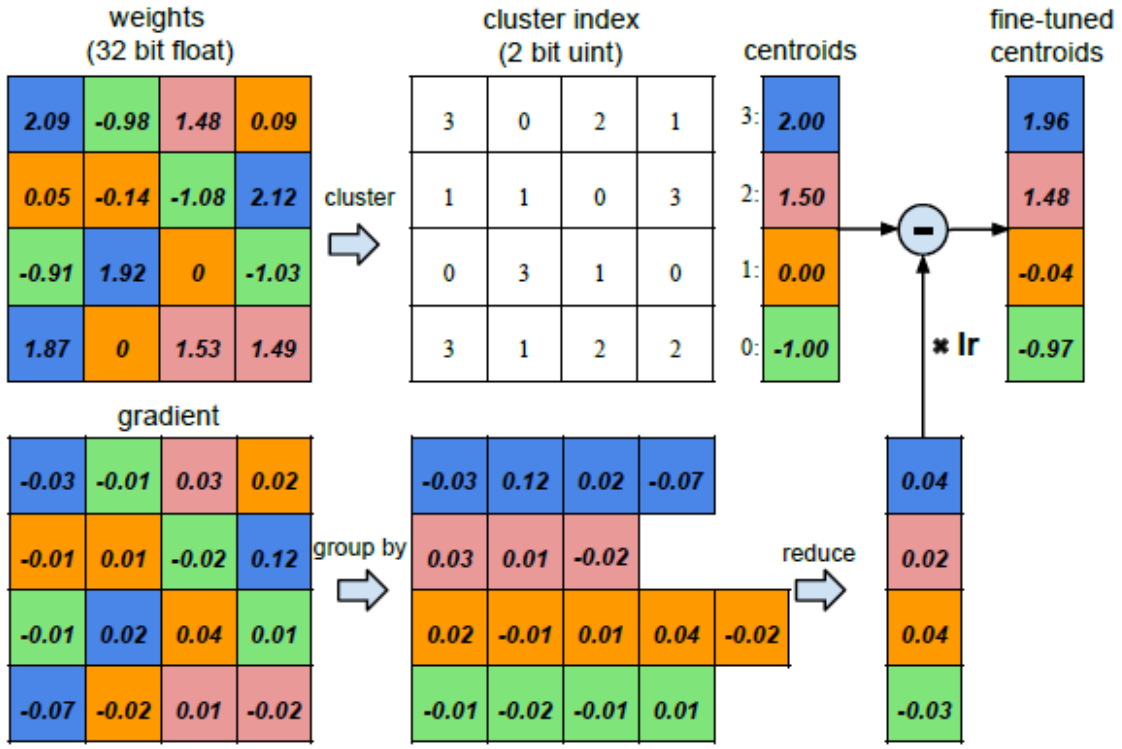
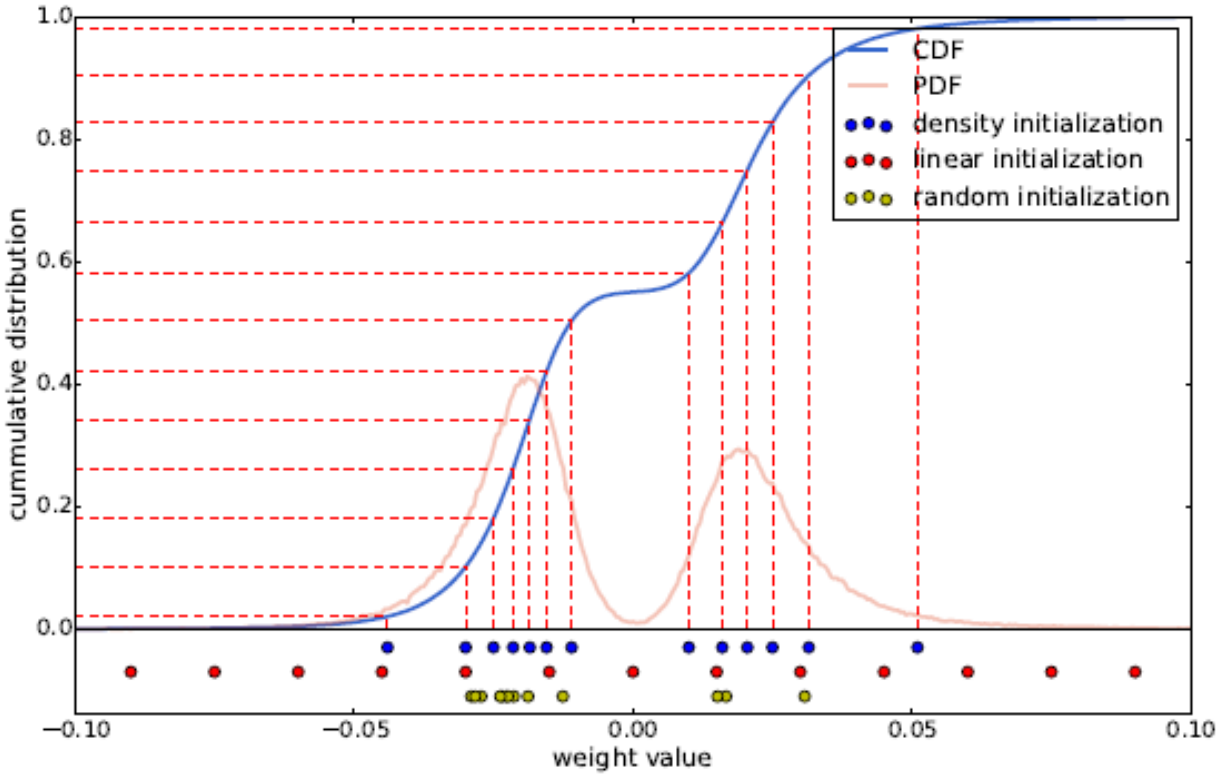
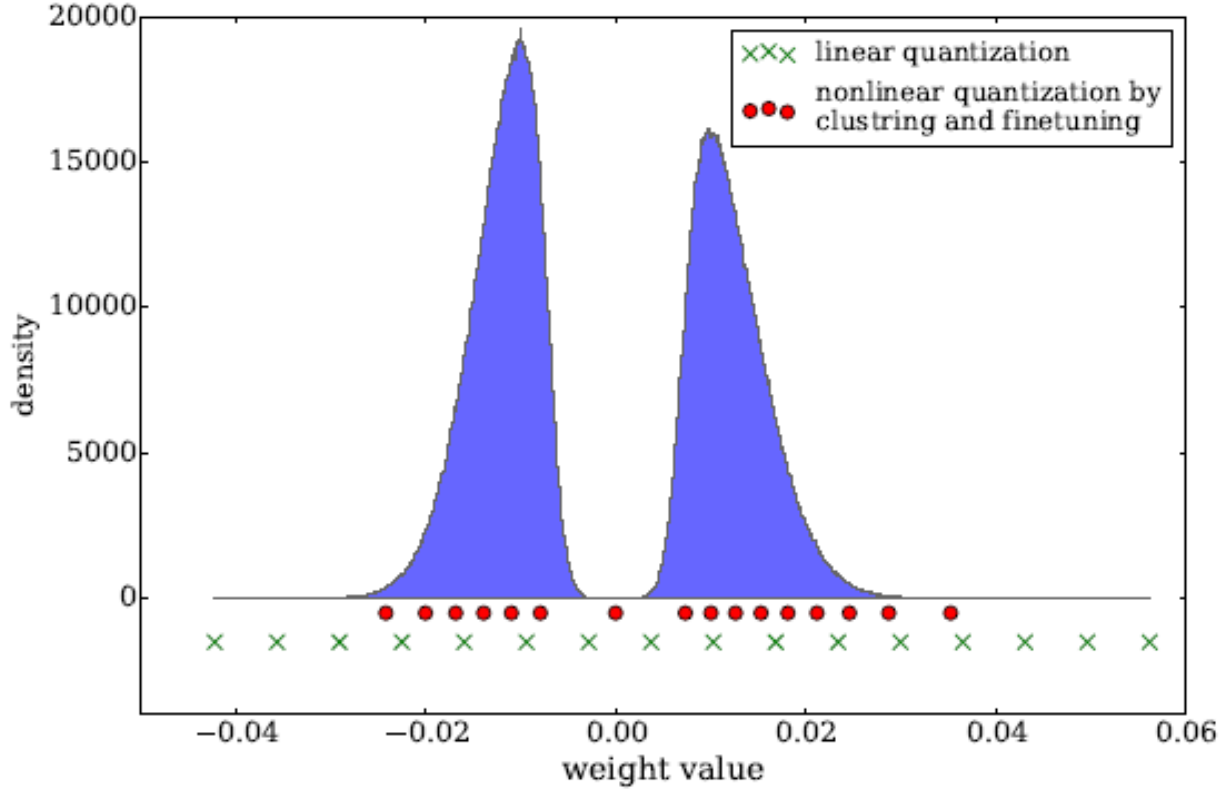


Figure 3: Weight sharing by scalar quantization (top) and centroids fine-tuning (bottom).

Initialization of Shared Weights

- foggy (random): yellow dots
- density-based: red dots
- linear: blue dots (recommended)
  - larger weights (although less frequent) are more important





Results

- # of parameters (original -> pruning -> compression)
  - AlexNet: 60M -> 6.7M -> 1.7M
  - VGG-16: 138M -> 10.35M -> 2.5M

Network	Top-1 Error	Top-5 Error	Parameters	Compress Rate
LeNet-300-100 Ref	1.64%	-	1070 KB	
LeNet-300-100 Compressed	1.58%	-	**27 KB**	**40**×
LeNet-5 Ref	0.80%	-	1720 KB	
LeNet-5 Compressed	0.74%	-	**44 KB**	**39**×
AlexNet Ref	42.78%	19.73%	240 MB	
AlexNet Compressed	42.78%	19.70%	**6.9 MB**	**35**×
VGG-16 Ref	31.50%	11.32%	552 MB	
VGG-16 Compressed	31.17%	10.91%	**11.3 MB**	**49**×

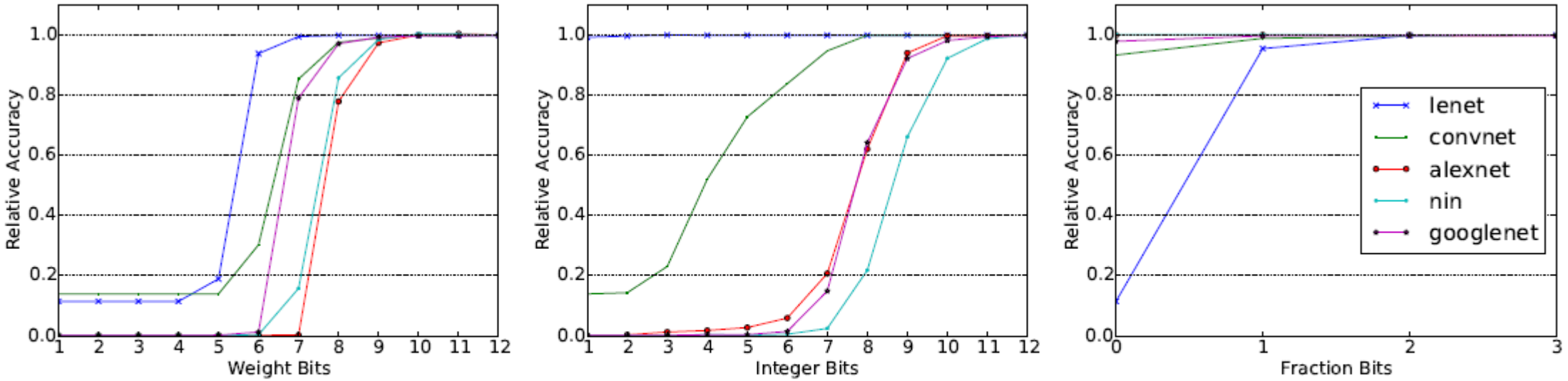
*1 parameter = 4 Bytes

Bit-Width Optimization

- P. Judd, et al., “Reduced-Precision Stategies for Bounded Memory in Deep Neural Nets,” 2016 ICLR
- P. Judd, et al., “Stripes: Bit-Serial Deep Neural Network Computing,” 2016 MICRO

Same Bit-Width for All Layers

- modify data read and write calls in Caffe framework
- use pre-trained models
- run 100 batches of 50-100 input images from respective validation dataset
- weights: Q1.9 (1 integer bit and 9 fractional bits) fixed-point
- data: Q12.2
- combined: Q12.9, a total of 21 bits

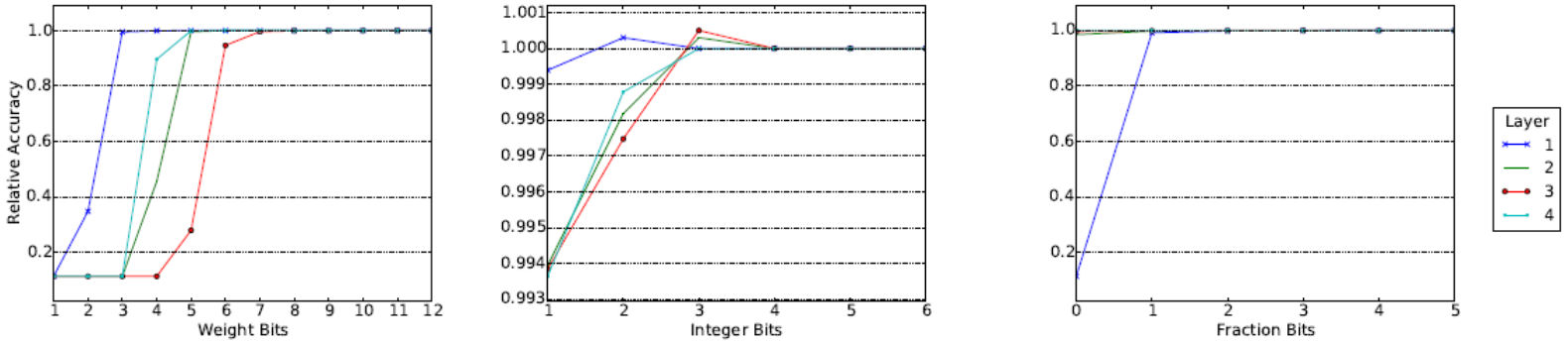


(a) Weights (b) Data: Integer (c) Data: Fraction

Figure 2: Accuracy relative to the baseline when the bit width is uniform for all layers.

Bit-Width Per Layer

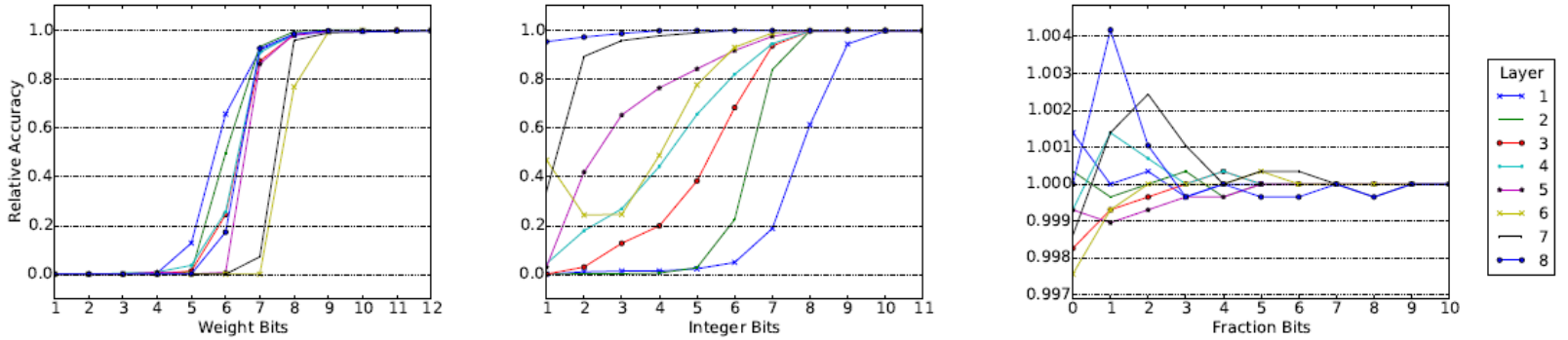
- LeNet: weights (fractional), data (integer), data (fractional)



(W, I, F)

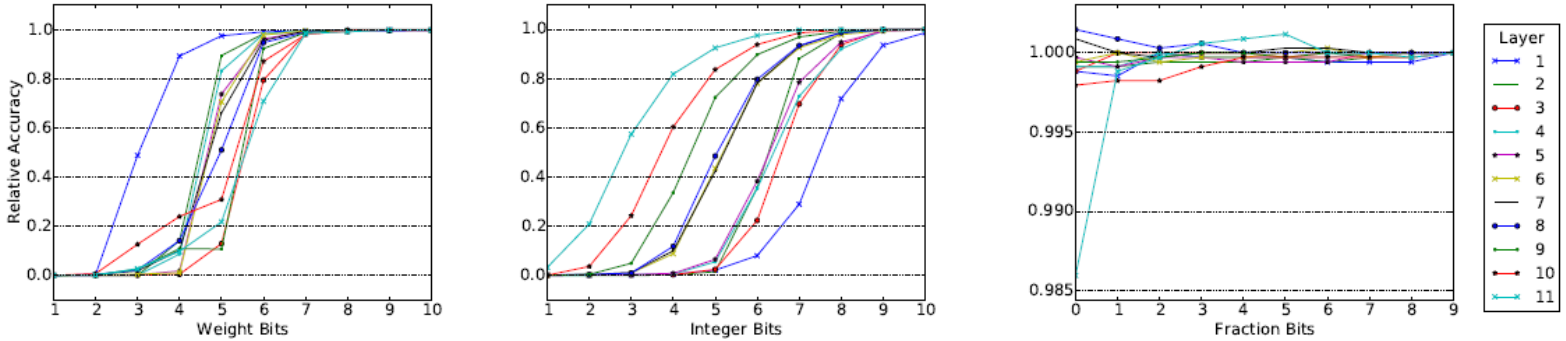
(7, 3, 1)

- AlexNet: weights (fractional), data (integer), data (fractional)



(9, 10, 0)

- GoogLeNet: weights (fractional), data (integer), data (fractional)



(7, 10, 2)

Data Bit-Width vs. Accuracy

- 2016 ICLR

Tolerance	Bits per layer in I.F	TR	Tolerance	Bits per layer (I+F)	TR
LeNet			**AlexNet** (F=0)		
1%	1.1-3.1-3.0-3.0	0.08	1%	10-8-8-8-8-8-6-4	0.28
2%	1.1-2.0-3.0-2.0	0.06	2%	10-8-8-8-8-8-5-4	0.28
5%	1.1-1.0-2.0-2.0	0.05	5%	10-8-8-8-7-7-5-3	0.28
10%	1.0-2.1-3.0-2.0	0.05	10%	9-8-8-8-7-7-5-3	0.26
Convnet			**NiN** (F=0)		
1%	8.0-7.0-7.0-5.0-5.0	0.24	1%	10-10-9-12-12-11-11-11-10-10-10-9	0.32
2%	7.0-8.0-6.0-4.0-4.0	0.22	2%	10-10-9-12-12-11-11-11-10-10-10-9	0.32
5%	7.0-8.0-5.0-3.0-3.0	0.22	5%	10-10-10-11-10-11-11-11-10-9-9-8	0.32
10%	7.0-8.0-5.0-3.0-3.0	0.22	10%	9-10-9-11-11-10-10-10-9-9-9-8	0.30
			GoogLeNet (F=2)		
			1%	14-10-12-12-12-12-11-11-11-10-9	0.36
			2%	13-11-11-10-12-11-11-11-11-10-9	0.35
			5%	12-11-11-11-11-11-10-10-10-9-9	0.34
			10%	12-9-11-11-11-10-10-10-10-9-9	0.32

Table 2: Minimum bandwidth for mixed precision for error tolerance between 1% and 10%. TR reports the traffic ratio over the 32-bit baseline. LeNet and Convnet report the integer bits and fractional bits as $I.F$. Fractional bits are fixed for AlexNet, NiN and GoogLeNet and the total bit width is reported.

- 2016 MICRO

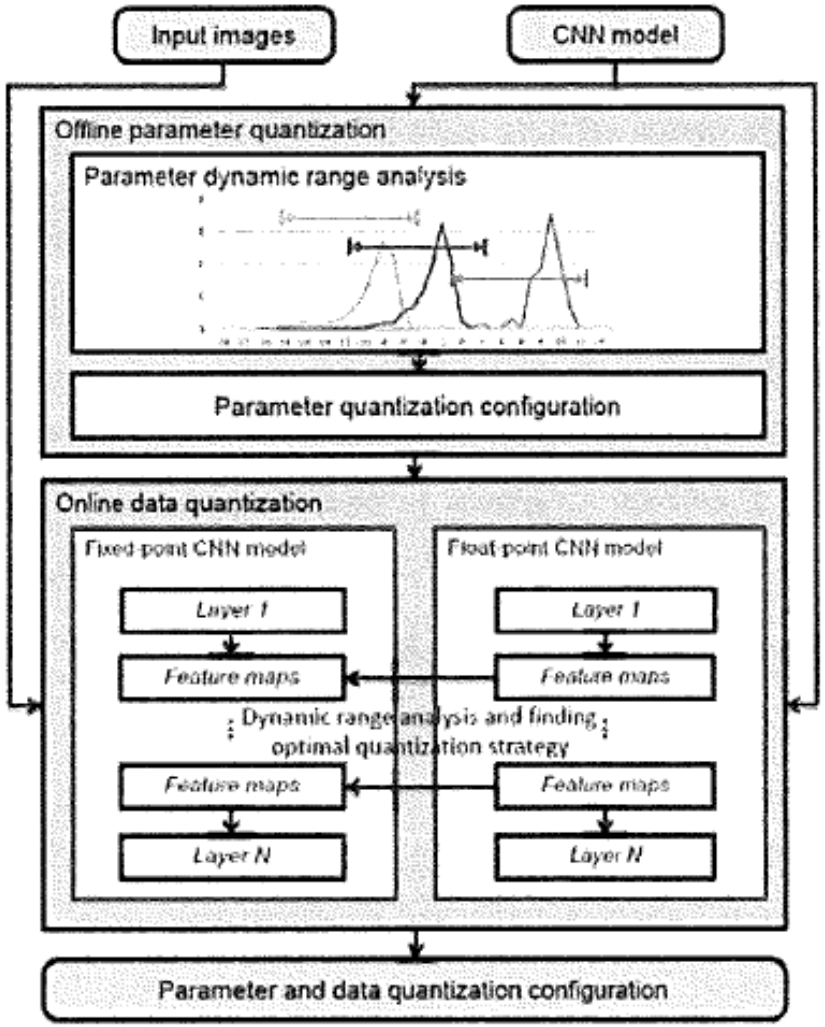
Network	Relative Accuracy 100% Per Layer Neuron Precision in Bits	100% Ideal Speedup	99% Per Layer Neuron Precision in Bits	99% Ideal Speedup
LeNet	3-3	5.33	2-3	7.33
Convnet	4-8-8	2.89	4-5-7	3.53
AlexNet	9-8-5-5-7	2.38	9-7-4-5-7	2.58
NiN	8-8-8-9-7-8-8-9-9-8-8-8	1.91	8-8-7-9-7-8-8-9-9-8-7-8	1.93
GoogLeNet	10-8-10-9-8-10-9-8-9-10-7	1.76	10-8-9-8-8-9-10-8-9-10-8	1.80
VGG_M	7-7-7-8-7	2.23	6-8-7-7-7	2.34
VGG_S	7-8-9-7-9	2.04	7-8-9-7-9	2.04
VGG_19	12-12-12-11-12-10-11-11-13-12-13-13-13-13-13-13	1.35	9-9-9-8-12-10-10-12-13-11-12-13-13-13-13-13	1.57

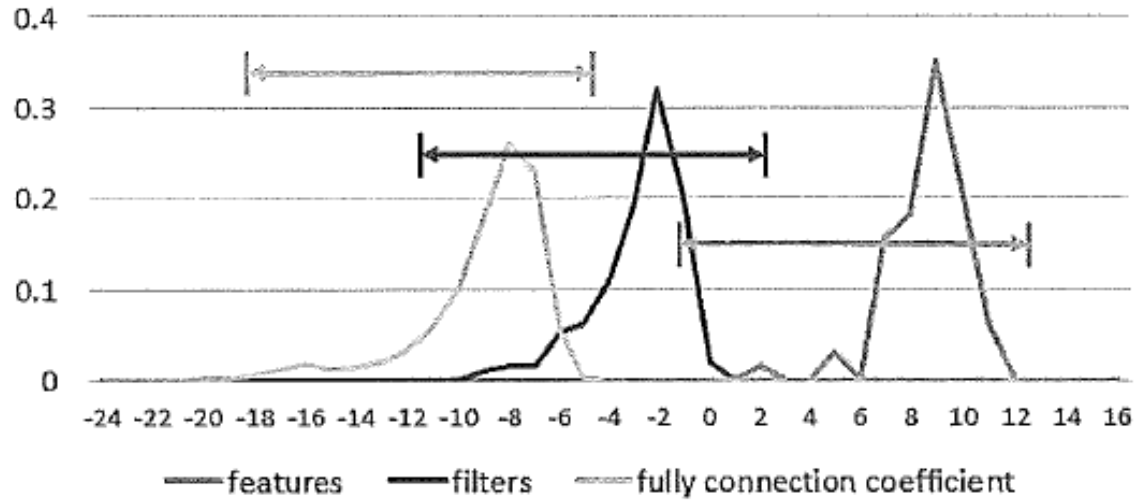
TABLE I: Per Convolutional layer neuron precision profiles needed to maintain the same accuracy as in the baseline (100%) and to reduce it within 1% of the baseline (99%). *Ideal*: Potential speedup with *Stripes* over a 16-bit baseline.

Quantization with Narrow Accumulator

- B. de Briunm, Z. Zivkovic and Henk Corporaal, Quantization of Constrained Processor Data Paths Applied to Convolutional Neural Networks," DSD 2018.

Model Compression with Data Quantization





data and parameter distribution

- Fixed precision in each layer
- Dynamic precision for different layers
- 8/4 bit dynamic fixed-point format: 0.4% accuracy loss for VGG-16 model

Value Range

- weight
- data

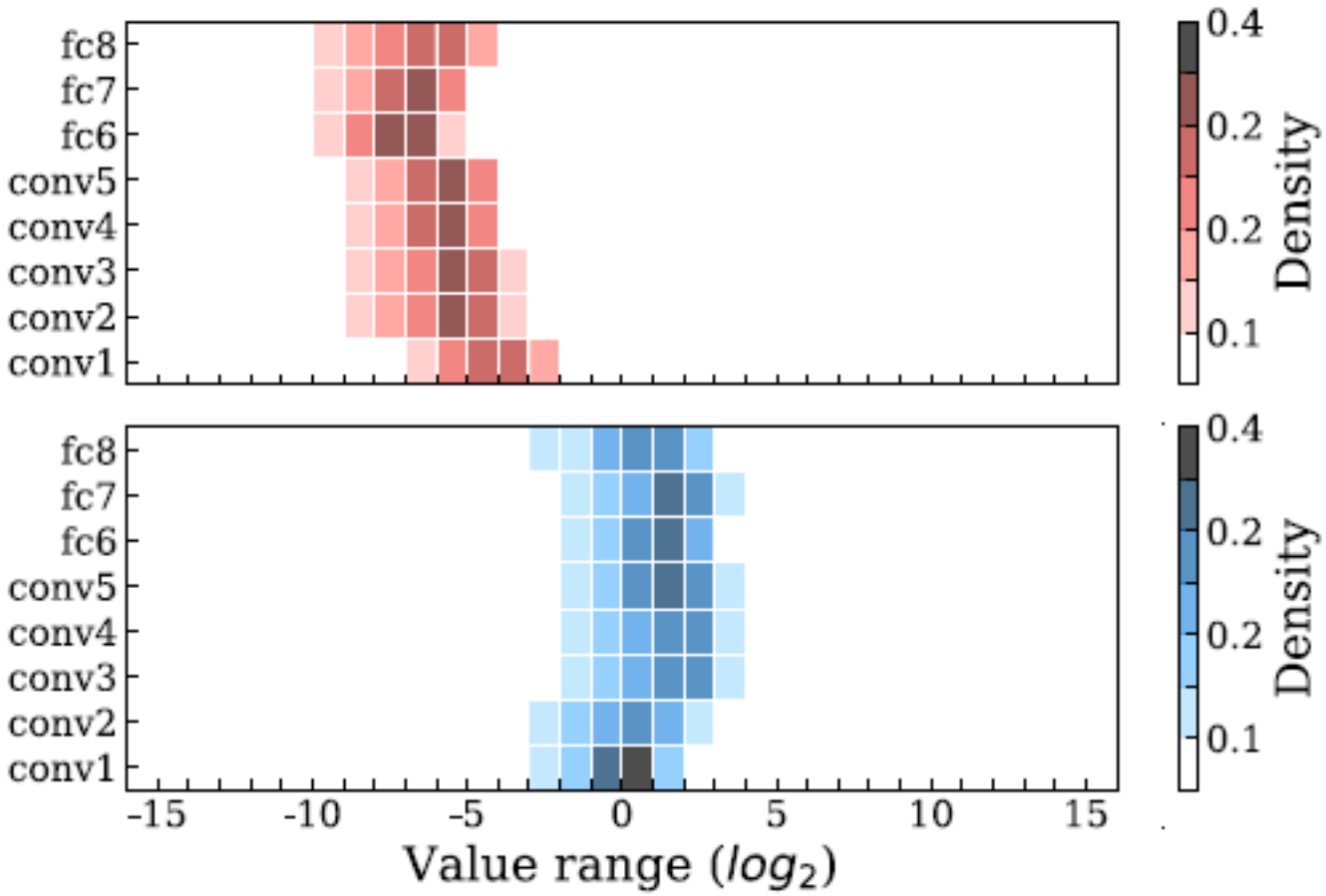


Fig. 3. Value range of weights (top) and input data (bottom) of AlexNet's convolutional and fully-connected layers (conv3 corresponds to Fig. 2).

Constraints with Given Accumulator Bitwidth

BW_{acc}=16
K=5x5x16

- WC_w/WC_d: worse-case weight/data values

$$y = \sum_{i=1}^{K} w_i d_i$$

$$BW_{\boldsymbol{w}} + BW_{\boldsymbol{d}} = BW_{acc} + 1 - \lceil \log_2 K \rceil$$

- ACT_w/WC_d:

$$R_{acc} = \sum_{i=1}^{K} w_i d_i \leq \sum_{i=1}^{K} |w_i||d_i| \qquad R_{kernel} = \max_{kernels \in layer} \left(\sum_{i=1}^{K} |\hat{w}_i| \right)$$

$$BW_{\boldsymbol{w}} + BW_{\boldsymbol{d}} = BW_{acc} - \lfloor \log_2 (R_{kernel}) \rfloor + IL_{\boldsymbol{w}}$$

- ACT_y:

$$BW_{acc} = FL_{\boldsymbol{w}} + FL_{\boldsymbol{d}} + IL_{\boldsymbol{y}} + 1$$

$$BW_{\boldsymbol{w}} + BW_{\boldsymbol{d}} = BW_{acc} + 1 - \max(0, IL_{\boldsymbol{y}} - (IL_{\boldsymbol{w}} + IL_{\boldsymbol{d}}))$$

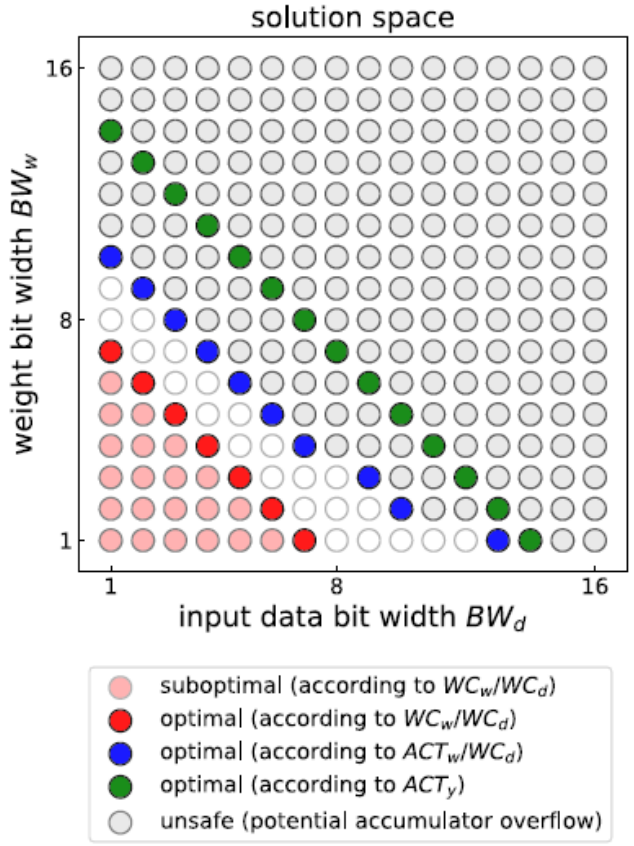


Fig. 4. Illustration of the solution space of a convolutional or fully-connected layer. Red dots are solutions that utilize all accumulator bits for the worst-case constraint. Light red dots are suboptimal as these do not utilize all accumulator bits. Blue dots represent optimal solutions for the ACT_w/WC_d constraint. For small bit widths most weights are quantized to zero, which explains the distortion at $BW_w < 4$. Green represents the optimal solutions according to constraint ACT_y. However, these solutions may result in accumulator overflow.

Heuristic Layer-Wise Optimization

- in each layer, choose from the configurations that minimizes $\sum_{i=1}^{N} |y_i - \hat{y}_i|$

N: # of output samples

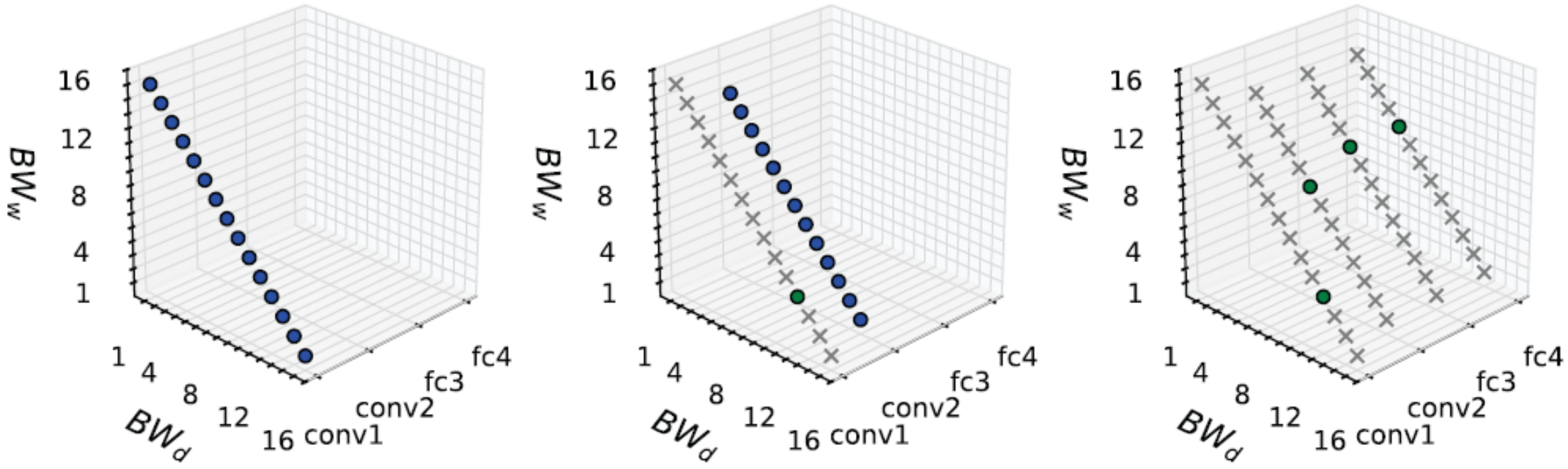


Fig. 5. Quantization procedure for a 4-layer network. Left: Analyse the chosen accumulator constraint for the first layer and test every eligible solution (blue dots). Middle: Set the previous layer to best-performing solution (green dot) and continue quantization of the next layer. Right: Repeat the procedure until all layers are quantized.

Comparison of Required Accumulator Range (IL_{acc})

- three constraint methods to avoid overflow
- two methods allowing overflow: wrap-around and clip
- LB_{wrap} and LB_{clip} are minimum integer length of the accumulator with > 99% of normalized classification accuracy

Layer	WC_w/WC_d	ACT_w/WC_d	ACT_y	LB_{wrap}	LB_{clip}
conv1	11	8	6	5	5
conv2	19	13	8	8	5
conv3	18	14	8	7	5
conv4	18	14	7	6	4
conv5	17	13	7	6	4
fc6	17	13	7	6	4
fc7	16	13	7	6	4
fc8	17	13	6	6	5

Binary/Ternary Neural Network

- [1] M. Courbariaux and Y. Bengio, “BinaryConnect: Training Deep Neural Networks with Binary Weights during Propagation,” NIPS 2015.
- [2] M. Courbariaux, I. Hubara, et al., “Binarized Neural Networks: Training Neural Networks with Weights and Activations Constrained to +1 and -1,” NIPS 2016.
- [3] I. Hubara, M. Courbariaux, et al., “Quantized Neural Networks: Training Neural Networks with Low Precision Weights and Activations,” arXiv: 1609.07061v1, 2016.
- [4] M. Rastegari, et al., “XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks,” ECCV 2016.
- [5] S. Zhou, et al., “DoReFa-Net: Training Low Bitwidth Convolutional Neural Networks with Low Bitwidth Gradients,” arXiv:1606.06160, 2016.
- [6] F. Li, B. Zhang, and B. Lin, “Ternary Weight Networks,” arXiv:1605.04711v2, 2016.
- [7] C. Zhu, S. Han, et al., “Trained Ternary Quantization,” ICLR 2017

binary/ternary neural networks

- BinaryConnect [1]
  - binary weights in {+1, -1}, using either deterministic or stochastic binarization
  - applied to MNIST (error rate=1.29%, det.), CIFAR-10 (9.9%, det.), and SVHN
- Binary Neural Network (BNN) [2]
  - binary weights and activations in {+1, -1}
  - MINST (1.40%), CIFAR-10 (10.15%), SVHN (2.53%)
- Quantized Neural Network (QNN) [3]
  - applied to AlextNet ImageNet with top-1 and top-5 (cp. original) accuracy of 41.8% (56.6%) and 67.1% (80.2%)
- Binary Weight Network (BWN), XNOR-Net [4]
  - BWN: binary weights in $\{+A_{lk}, -A_{lk}\}$ (l-th layer, k-th filter for k-th output channel)
  - XNOR-Net: binary weights and activations with ***channel-wise*** scaling factors
  - appliedto AlexNet: top-5 acc. (%): full/BWN/XNOR-Net=80.2/79.4/69.5, top-1 acc. (%)=56.6/56.8/44.2
- DoReFa-Net, [5]
  - binary weights with **layer-wise** scaling factors
- Ternary Weight Network (TWN), [6]
  - ternary weights in $\{+w_l, 0, -w_l\}$ with **layer-wise** scaling and threshold
- Trained Ternary Quantization (TTQ), [7]
  - ternary weights in $\{+w_{p,l}, 0, -w_{n,l}\}$ with different per-layer scaling factors for positive/negative values
  - AlexNet top1 (top-5) error%: full/DoreFa/TWN/TTQ: 42.8/46.1/45.5/42.5 (19.7/23.7/23.2/20.3)
  - ResNet-18 top1 (top5) error%: full/BWN/TWN/TTQ: 30.4/39.2/34.7/33.4 (10.8/17.0/13.8/12.8)

1. M. Courbariaux and Y. Bengio, "BinaryConnect: Training Deep Neural Networks with Binary Weights during Propagation," NIPS 2015.
2. M. Courbariaux, I. Hubara, et al., "Binarized Neural Networks: Training Neural Networks with Weights and Activations Constrained to +1 and -1," NIPS 2016.
3. I. Hubara, M. Courbariaux, et al., "Quantized Neural Networks: Training Neural Networks with Low Precision Weights and Activations," arXiv: 1609.07061v1, 2016.
4. M. Rastegari, et al., "XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks," ECCV 2016.
5. S. Zhou, et al., "DoReFa-Net: Training Low Bitwidth Convolutional Neural Networks with Low Bitwidth Gradients," arXiv:1606.06160, 2016.
6. F. Li, B. Zhang, and B. Lin, "Ternary Weight Networks," arXiv:1605.04711v2, 2016.
7. C. Zhu, S. Han, et al., "Trained Ternary Quantization," ICLR 2017

binary/ternary neural networks

- BinaryConnect [1]
  - binary weights in {+1, -1}, using either deterministic or stochastic binarization
  - applied to MNIST (error rate=1.29%, det.), CIFAR-10 (9.9%, det.), and SVHN
- Binary Neural Network (BNN) [2]
  - binary weights and activations in {+1, -1}
  - MINST (1.40%), CIFAR-10 (10.15%), SVHN (2.53%)
- Quantized Neural Network (QNN) [3]
  - applied to AlextNet ImageNet with top-1 and top-5 (cp. original) accuracy of 41.8% (56.6%) and 67.1% (80.2%)
- Binary Weight Network (BWN), XNOR-Net [4]
  - BWN: binary weights in $\{+A_{lk}, -A_{lk}\}$ (l-th layer, k-th filter for k-th output channel)
  - XNOR-Net: binary weights and activations with ***channel-wise*** scaling factors
  - appliedto AlexNet: top-5 acc. (%): full/BWN/XNOR-Net=80.2/79.4/69.5, top-1 acc. (%)=56.6/56.8/44.2
- DoReFa-Net, [5]
  - binary weights with **layer-wise** scaling factors
- Ternary Weight Network (TWN), [6]
  - ternary weights in $\{+w_l, 0, -w_l\}$ with **layer-wise** scaling and threshold
- Trained Ternary Quantization (TTQ), [7]
  - ternary weights in $\{+w_{p,l}, 0, -w_{n,l}\}$ with different per-layer scaling factors for positive/negative values
  - AlexNet top1 (top-5) error%: full/DoreFa/TWN/TTQ: 42.8/46.1/45.5/42.5 (19.7/23.7/23.2/20.3)
  - ResNet-18 top1 (top5) error%: full/BWN/TWN/TTQ: 30.4/39.2/34.7/33.4 (10.8/17.0/13.8/12.8)

1. M. Courbariaux and Y. Bengio, "BinaryConnect: Training Deep Neural Networks with Binary Weights during Propagation," NIPS 2015.
2. M. Courbariaux, I. Hubara, et al., "Binarized Neural Networks: Training Neural Networks with Weights and Activations Constrained to +1 and -1," NIPS 2016.
3. I. Hubara, M. Courbariaux, et al., "Quantized Neural Networks: Training Neural Networks with Low Precision Weights and Activations," arXiv: 1609.07061v1, 2016.
4. M. Rastegari, et al., "XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks," ECCV 2016.
5. S. Zhou, et al., "DoReFa-Net: Training Low Bitwidth Convolutional Neural Networks with Low Bitwidth Gradients," arXiv:1606.06160, 2016.
6. F. Li, B. Zhang, and B. Lin, "Ternary Weight Networks," arXiv:1605.04711v2, 2016.
7. C. Zhu, S. Han, et al., "Trained Ternary Quantization," ICLR 2017

BinaryConnect[1]

- binaryization
  - deterministic
  - stochastic
- negligible impact of accuracy on small dataset
-

$$w_b = \begin{cases} +1 & \text{if } w \geq 0, \\ -1 & \text{otherwise.} \end{cases}$$

$$w_b = \begin{cases} +1 & \text{with probability } p = \sigma(w), \\ -1 & \text{with probability } 1 - p. \end{cases}$$

$$\sigma(x) = \text{clip}(\frac{x+1}{2}, 0, 1) = \max(0, \min(1, \frac{x+1}{2}))$$

Method	MNIST	CIFAR-10	SVHN
No regularizer	$1.30 \pm 0.04\%$	10.64%	2.44%
BinaryConnect (det.)	$1.29 \pm 0.08\%$	9.90%	2.30%
BinaryConnect (stoch.)	$1.18 \pm 0.04\%$	**8.27%**	2.15%
50% Dropout	$1.01 \pm 0.04\%$		
Maxout Networks [29]	0.94%	11.68%	2.47%
Deep L2-SVM [30]	**0.87%**		
Network in Network [31]		10.41%	2.35%
DropConnect [21]			1.94%
Deeply-Supervised Nets [32]		9.78%	**1.92%**

Table 2: Test error rates of DNNs trained on the MNIST (no convolution and no unsupervised pretraining), CIFAR-10 (no data augmentation) and SVHN, depending on the method. We see that in spite of using only a single bit per weight during propagation, performance is not worse than ordinary (no regularizer) DNNs, it is actually better, especially with the stochastic version, suggesting that BinaryConnect acts as a regularizer.

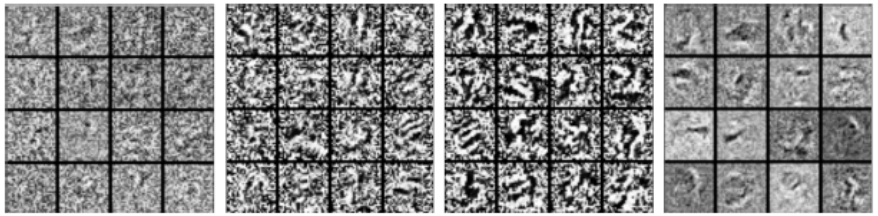


Figure 1: Features of the first layer of an MLP trained on MNIST depending on the regularizer. From left to right: no regularizer, deterministic BinaryConnect, stochastic BinaryConnect and Dropout.

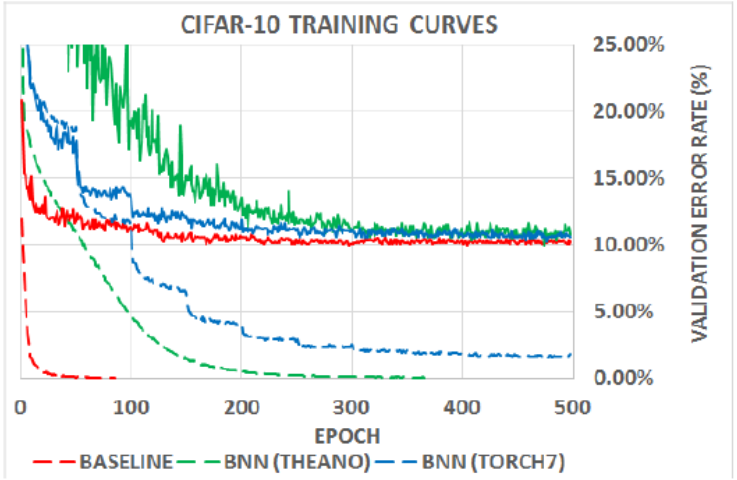
[1] M. Courbariaux and Y. Bengio, "BinaryConnect: Training Deep Neural Networks with Binary Weights during Propagation," NIPS 2015.

Binarized Neural Network [2]

- binarize both weights and activation data
- negligible accuracy loss for small datasets

Data set	MNIST	SVHN	CIFAR-10
Binarized activations+weights, during training and test			
BNN (Torch7)	1.40%	2.53%	10.15%
BNN (Theano)	0.96%	2.80%	11.40%
Committee Machines' Array (Baldassi et al., 2015)	1.35%	-	-
Binarized weights, during training and test			
BinaryConnect (Courbariaux et al., 2015)	1.29± 0.08%	2.30%	9.90%
Binarized activations+weights, during test			
EBP (Cheng et al., 2015)	2.2± 0.1%	-	-
Bitwise DNNs (Kim & Smaragdis, 2016)	1.33%	-	-
Ternary weights, binary activations, during test			
(Hwang & Sung, 2014)	1.45%	-	-
No binarization (standard results)			
Maxout Networks (Goodfellow et al.)	0.94%	2.47%	11.68%
Network in Network (Lin et al.)	-	2.35%	10.41%
Gated pooling (Lee et al., 2015)	-	1.69%	7.62%

Figure 1. Training curves of a ConvNet on CIFAR-10 depending on the method. The dotted lines represent the training costs (square hinge losses) and the continuous lines the corresponding validation error rates. Although BNNs are slower to train, they are nearly as accurate as 32-bit float DNNs.



[2] M. Courbariaux, I. Hubara, et al., "Binarized Neural Networks: Training Neural Networks with Weights and Activations Constrained to +1 and -1," NIPS 2016.

Quantized Neural Network [3]

- quantize both weights and activation to n-bit
  - n=8 for 1st layer
  - n=1 for the rest of layers
- applied to large dataset

Algorithm 4 Running a BNN with L layers.

Require: 8-bit input vector a_0, binary weights W^b, and BatchNorm parameters θ.
Ensure: the MLP output a_L.

{1. First layer:}
$a_1 \leftarrow 0$
for $n = 1$ to 8 **do**
  $a_1 \leftarrow a_1 + 2^{n-1} \times \text{XnorDotProduct}(\text{a}_0^{\text{n}}, \text{W}_1^{\text{b}})$
end for
$a_1^b \leftarrow \text{Sign}(\text{BatchNorm}(a_1, \theta_1))$
{2. Remaining hidden layers:}
for $k = 2$ to $L - 1$ **do**
  $a_k \leftarrow \text{XnorDotProduct}(a_{k-1}^b, W_k^b)$
  $a_k^b \leftarrow \text{Sign}(\text{BatchNorm}(a_k, \theta_k))$
end for
{3. Output layer:}
$a_L \leftarrow \text{XnorDotProduct}(a_{L-1}^b, W_L^b)$
$a_L \leftarrow \text{BatchNorm}(a_L, \theta_L)$

Table 2: Classification test error rates of the AlexNet model trained on the ImageNet 1000 classification task. No unsupervised pre-training or data augmentation was used.

Model	Top-1	Top-5
Binarized activations+weights, during training and test		
BNN	41.8%	67.1%
Xnor-Nets[4] (Rastegari et al., 2016)	44.2%	69.2%
Binary weights and Quantize activations during training and test		
QNN 2-bit activation	51.03%	73.67%
DoReFaNet 2-bit activation[4] (Zhou et al., 2016)	50.7%	72.57%
Quantize weights, during test		
Deep Compression 4/2-bit (conv/FC layer) (Han et al., 2015a)	55.34%	77.67%
(Gysel et al., 2016) - 2-bit	0.01%	-%
No Quantization (standard results)		
AlexNet - our implementation	56.6%	80.2%

Model	Top-1	Top-5
Binarized activations+weights, during training and test		
BNN	47.1%	69.1%
Quantize weights and activations during training and test		
QNN 4-bit	66.5%	83.4%
Quantize activation,weights and gradients during training and test		
QNN 6-bit	66.4%	83.1%
No Quantization (standard results)		
GoogleNet - our implementation	71.6%	91.2%

[3] Hubara, M. Courbariaux, et al., "Quantized Neural Networks: Training Neural Networks with Low Precision Weights and Activations," arXiv: 1609.07061v1, 2016.

BWN, XNOR-Net[4]

- two approaches
  - binary weight network (TWN) with per-channel scaling
  - binary weight and activation with scaling factor
- on large data set

Algorithm 1. Training an L-layers CNN with binary weights:

Input: A minibatch of inputs and targets $(\mathbf{I}, \mathbf{Y})$, cost function $C(\mathbf{Y}, \hat{\mathbf{Y}})$, current weight $\mathcal{W}^t$ and current learning rate η^t.

Output: updated weight $\mathcal{W}^{t+1}$ and updated learning rate η^{t+1}.

1: Binarizing weight filters:
2: **for** $l = 1$ to L **do**
3: **for** k^{th} filter in l^{th} layer **do**
4: $\mathcal{A}_{lk} = \frac{1}{n}\|\mathcal{W}^t_{lk}\|_{\ell 1}$
5: $\mathcal{B}_{lk} = \text{sign}(\mathcal{W}^t_{lk})$
6: $\widetilde{\mathcal{W}}_{lk} = \mathcal{A}_{lk}\mathcal{B}_{lk}$
7: $\hat{\mathbf{Y}} =$ **BinaryForward**$(\mathbf{I}, \mathcal{B}, \mathcal{A})$ // standard forward propagation except that convolutions are computed using Eq. 1 or 11
8: $\frac{\partial C}{\partial \widetilde{\mathcal{W}}} =$ **BinaryBackward**$(\frac{\partial C}{\partial \hat{\mathbf{Y}}}, \widetilde{\mathcal{W}})$ // standard backward propagation except that gradients are computed using $\widetilde{\mathcal{W}}$ instead of $\mathcal{W}^t$
9: $\mathcal{W}^{t+1} =$ **UpdateParameters**$(\mathcal{W}^t, \frac{\partial C}{\partial \widetilde{\mathcal{W}}}, \eta_t)$ // Any update rules (*e.g.*, SGD or ADAM)
10: $\eta^{t+1} =$ **UpdateLearningrate**(η^t, t) // Any learning rate scheduling function

Classification accuracy (%)									
Binary-weight				Binary-input-binary-weight				Full-precision	
BWN		BC [11]		XNOR-Net		BNN [11]		AlexNet [1]	
Top-1	Top-5	Top-1	Top-5	Top-1	Top-5	Top-1	Top-5	Top-1	Top-5
56.8	**79.4**	35.4	61.0	**44.2**	**69.2**	27.9	50.42	56.6	80.2

	ResNet-18		GoogLenet	
Network variations	Top-1	Top-5	Top-1	Top-5
Binary-weight-network	60.8	83.0	65.5	86.1
XNOR-network	51.2	73.2	N/A	N/A
Full-precision-network	69.3	89.2	71.3	90.0

[4] M. Rastegari, et al., "XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks," ECCV 2016.

Ternary Weight Network (TWN) [6]

- ternary weight with per-layer scaling

$$\begin{cases} \alpha^*, \mathbf{W}^{t*} = \arg\min_{\alpha, \mathbf{W}^t} J(\alpha, \mathbf{W}^t) = ||\mathbf{W} - \alpha\mathbf{W}^t||_2^2 \\ \text{s.t.} \quad \alpha \geq 0, \mathrm{W}_i^t \in \{-1, 0, 1\}, i = 1, 2, \ldots \end{cases}$$

Here n is the size of the filter. With the approximation $\mathbf{W} \approx \alpha\mathbf{W}^t$, a

Table 2: Validation accuracies (%). Results on ImageNet are with ResNet-18 / ResNet-18B.

	MNIST	CIFAR-10	ImageNet (top-1)	ImageNet (top-5)
TWNs	**99.35**	**92.56**	**61.8 / 65.3**	**84.2 / 86.2**
BPWNs	99.05	90.18	57.5 / 61.6	81.2 / 83.9
FPWNs	99.41	92.88	65.4 / 67.6	86.76 / 88.0
BinaryConnect	98.82	91.73	-	-
Binarized Neural Networks	88.6	89.85	-	-
Binary Weight Networks	-	-	60.8	83.0
XNOR-Net	-	-	51.2	73.2

F. Li, B. Zhang, and B. Lin, "Ternary Weight Networks," arXiv:1605.04711v2, 2016.

Trained Ternary Quantization

- different scaling factors for +/-

$$w_l^t = \begin{cases} W_l : \tilde{w}_l > \Delta_l \\ 0 : |\tilde{w}_l| \leq \Delta_l \\ -W_l : \tilde{w}_l < -\Delta_l \end{cases}$$

Model	Full resolution	Ternary (Ours)	Improvement
ResNet-20	8.23	**8.87**	**-0.64**
ResNet-32	7.67	**7.63**	**0.04**
ResNet-44	7.18	**7.02**	**0.16**
ResNet-56	6.80	**6.44**	**0.36**

Table 1: Error rates of full-precision and ternary ResNets on Cifar-10

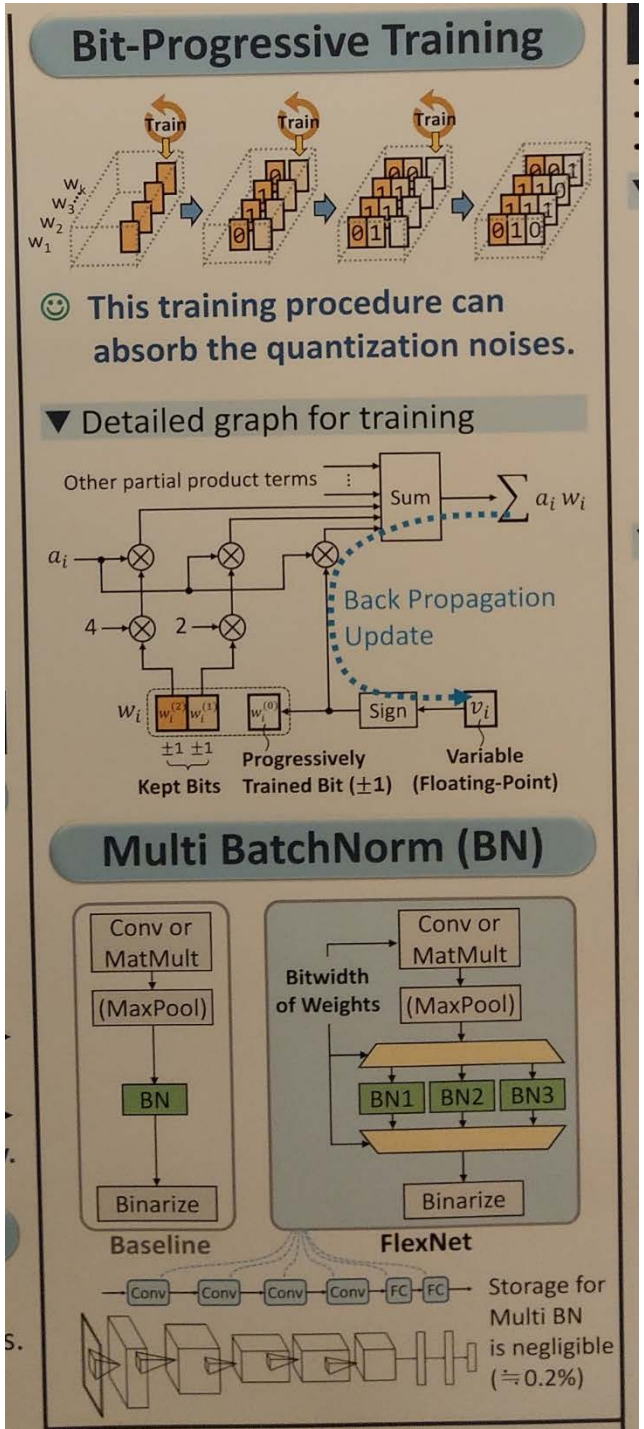
Error	Full precision	1-bit (DoReFa)	2-bit (TWN)	2-bit (Ours)
Top1	42.8%	46.1%	45.5%	**42.5%**
Top5	19.7%	23.7%	23.2%	**20.3%**

Table 2: Top1 and Top5 error rate of AlexNet on ImageNet

Error	Full precision	1-bit (BWN)	2-bit (TWN)	2-bit (Ours)
Top1	30.4%	39.2%	34.7%	**33.4%**
Top5	10.8%	17.0%	13.8%	**12.8%**

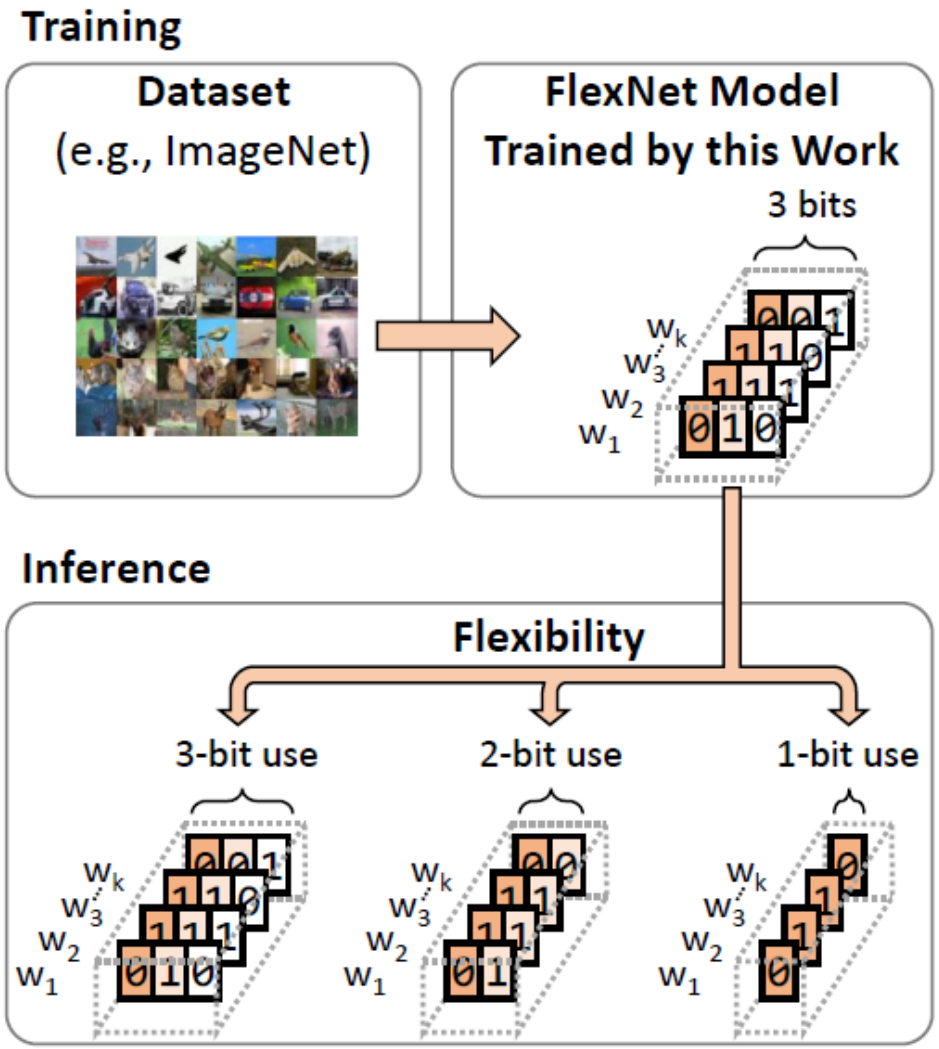
Table 3: Top1 and Top5 error rate of ResNet-18 on ImageNet

C. Zhu, S. Han, et al., "Trained Ternary Quantization," ICLR 2017



FlexNet

bit-by-bit progressive training



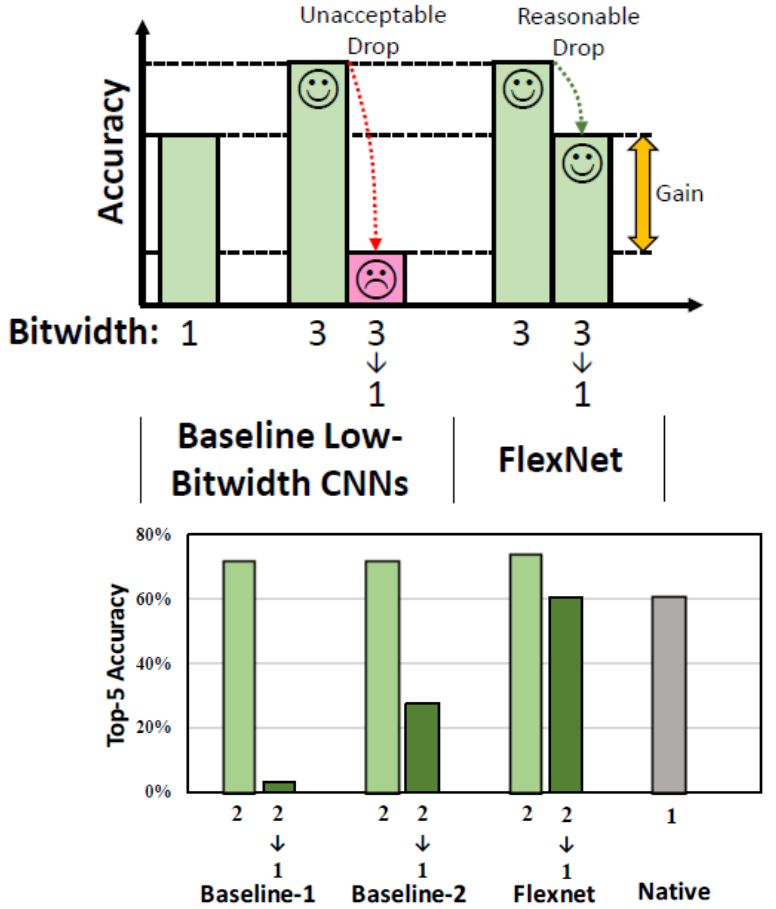


Fig. 2. Top-5 accuracy of 2-bit baselines and FlexNet

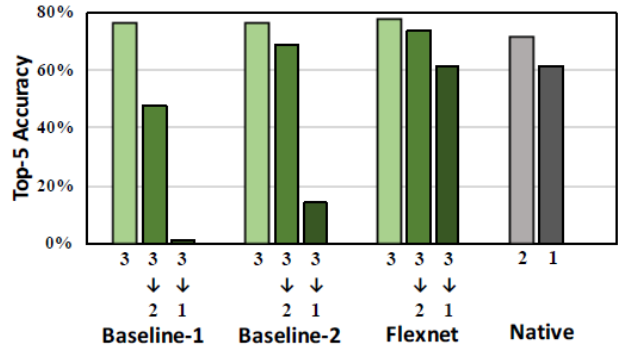


Fig. 3. Top-5 accuracy of 3-bit baselines and FlexNet