



NEW YORK UNIVERSITY

# Optimization

Yann LeCun

NYU - Courant Institute & Center for Data Science

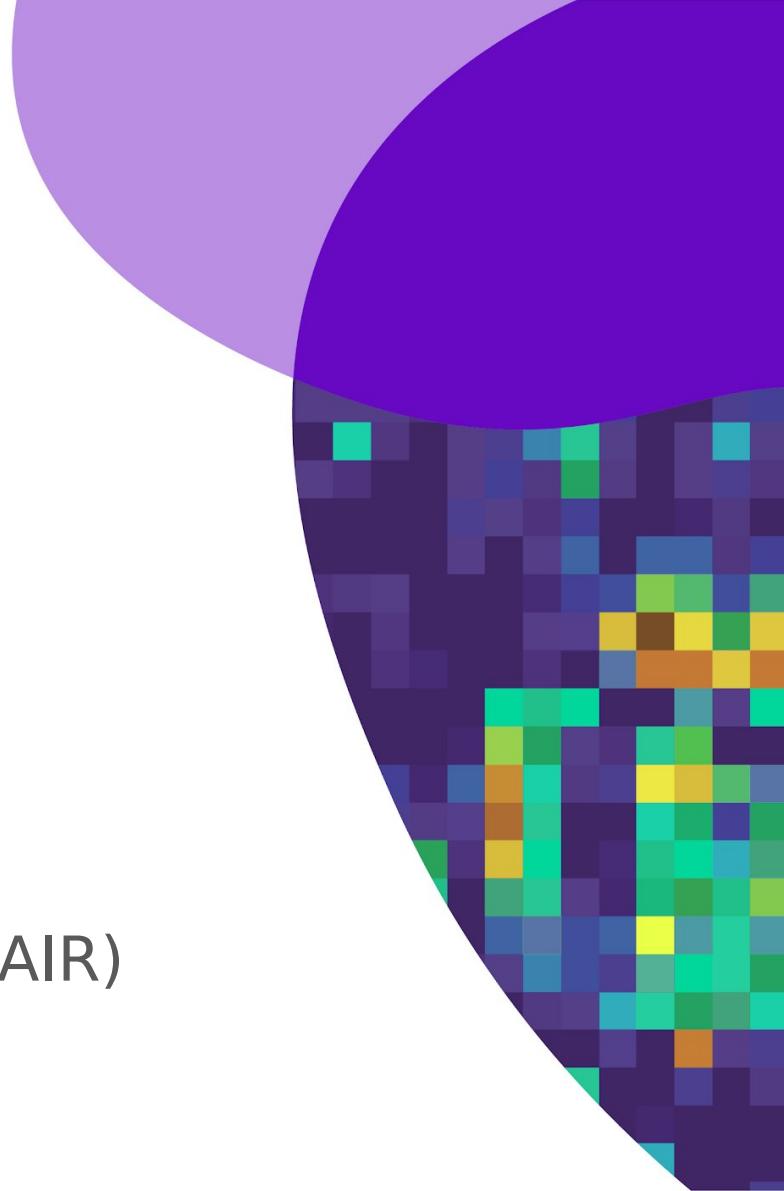
Facebook AI Research

<http://yann.lecun.com>

Deep Learning, NYU, Spring 2023

# Optimization for Deep Learning

Includes some material  
from Aaron Defazio (Meta-FAIR)



# The Convergence of Gradient Descent

$$f(w) = \frac{1}{P} \sum_{i=1}^P L(x_i, y_i, w)$$

parameter  
objective

training sample  
per-sample loss

learning rate

$$w_{k+1} \leftarrow w_k - \gamma_k \frac{\partial f(w_k)^T}{\partial w}$$

new parameter  
Gradient (transposed)

## **f(w): objective function**

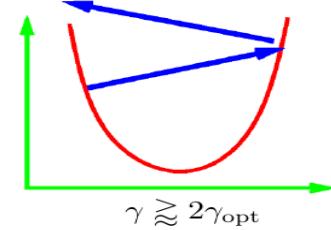
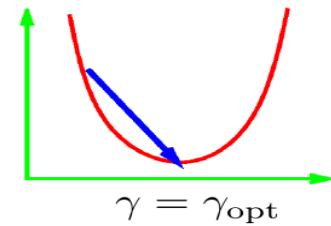
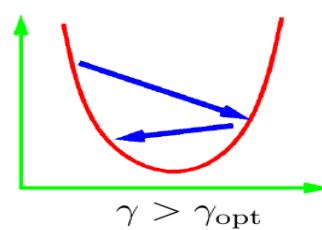
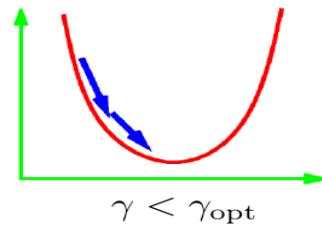
- ▶ average of many terms

## **Gradient descent**

- ▶ uses local slope information of a function to move towards a minimum.

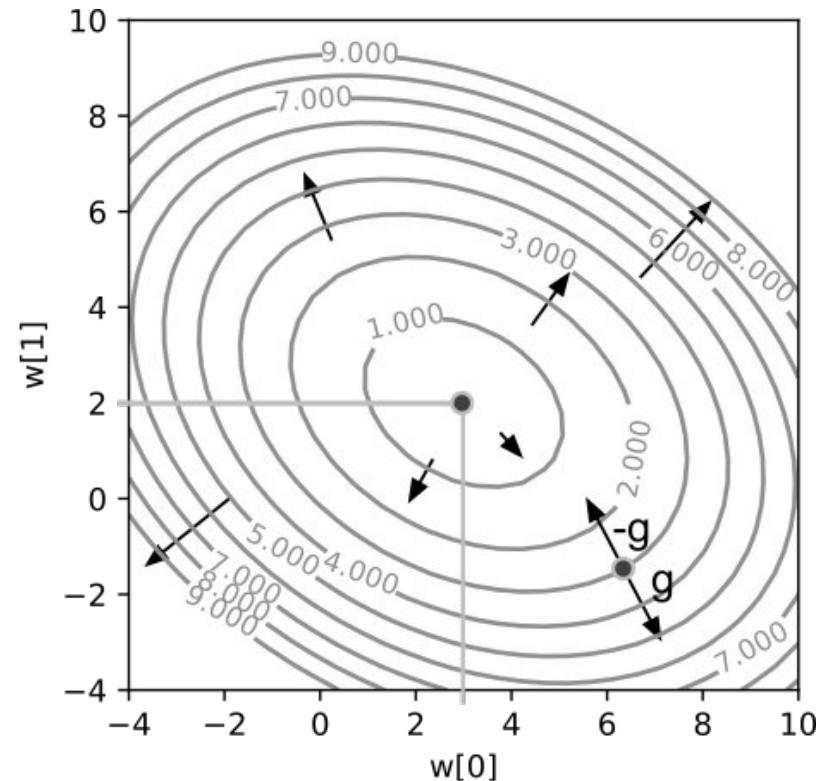
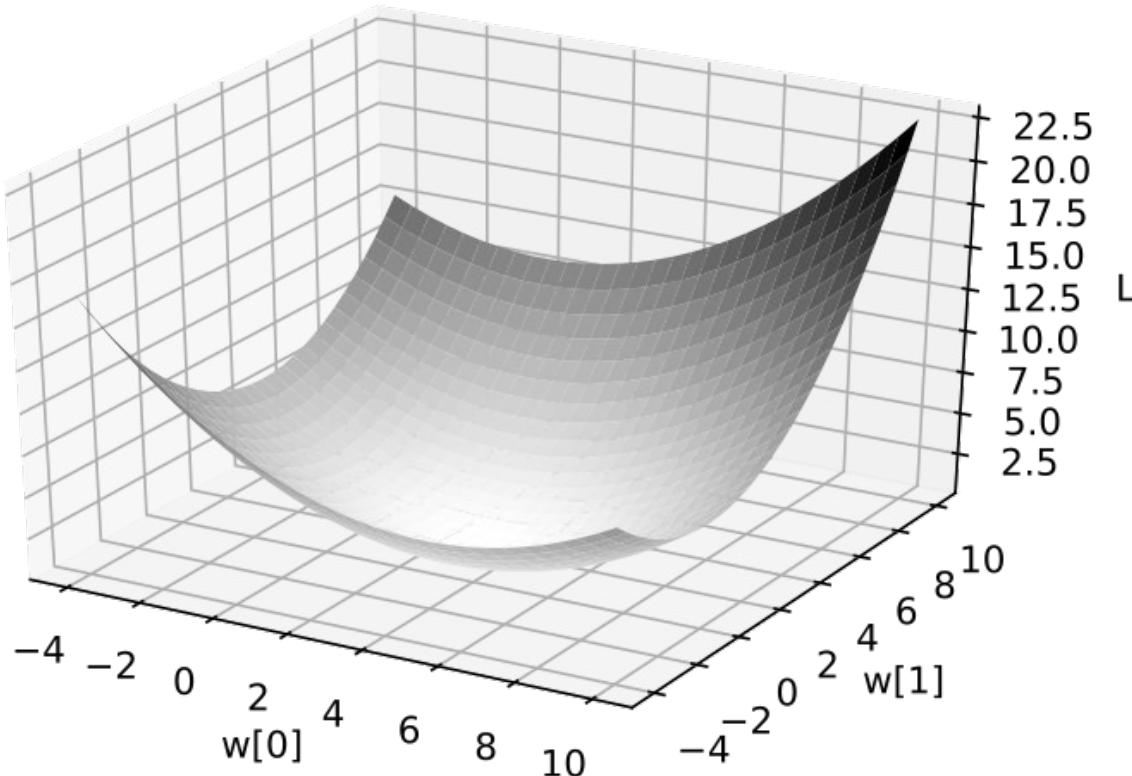
## **Requires a step size**

- ▶ aka “learning rate”



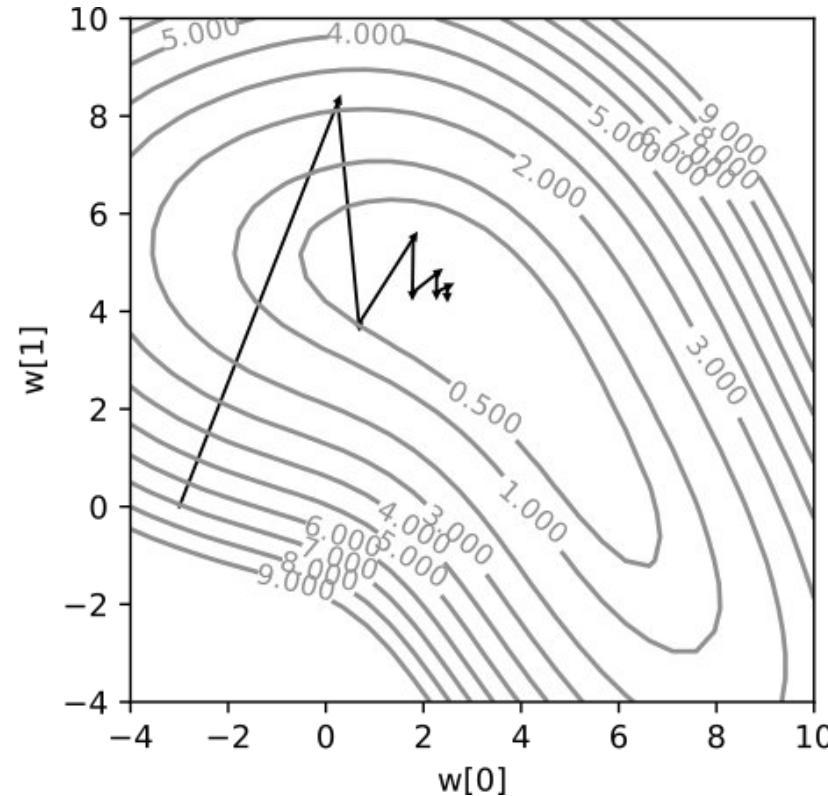
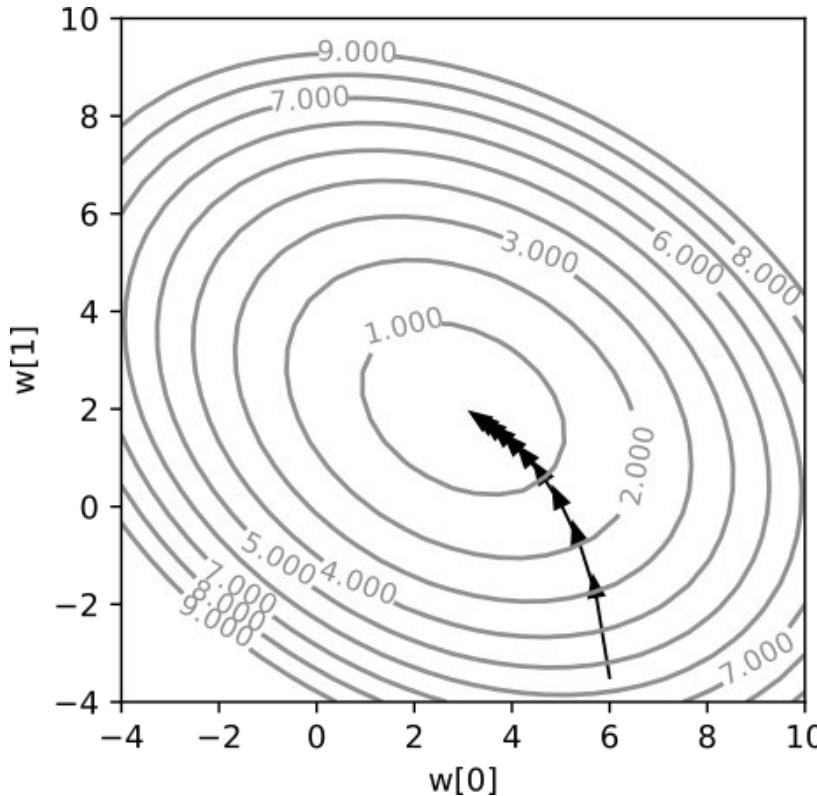
# Gradient descent

- ▶ Gradient indicates the direction of steepest ascent.
- ▶ Convex, quadratic case.



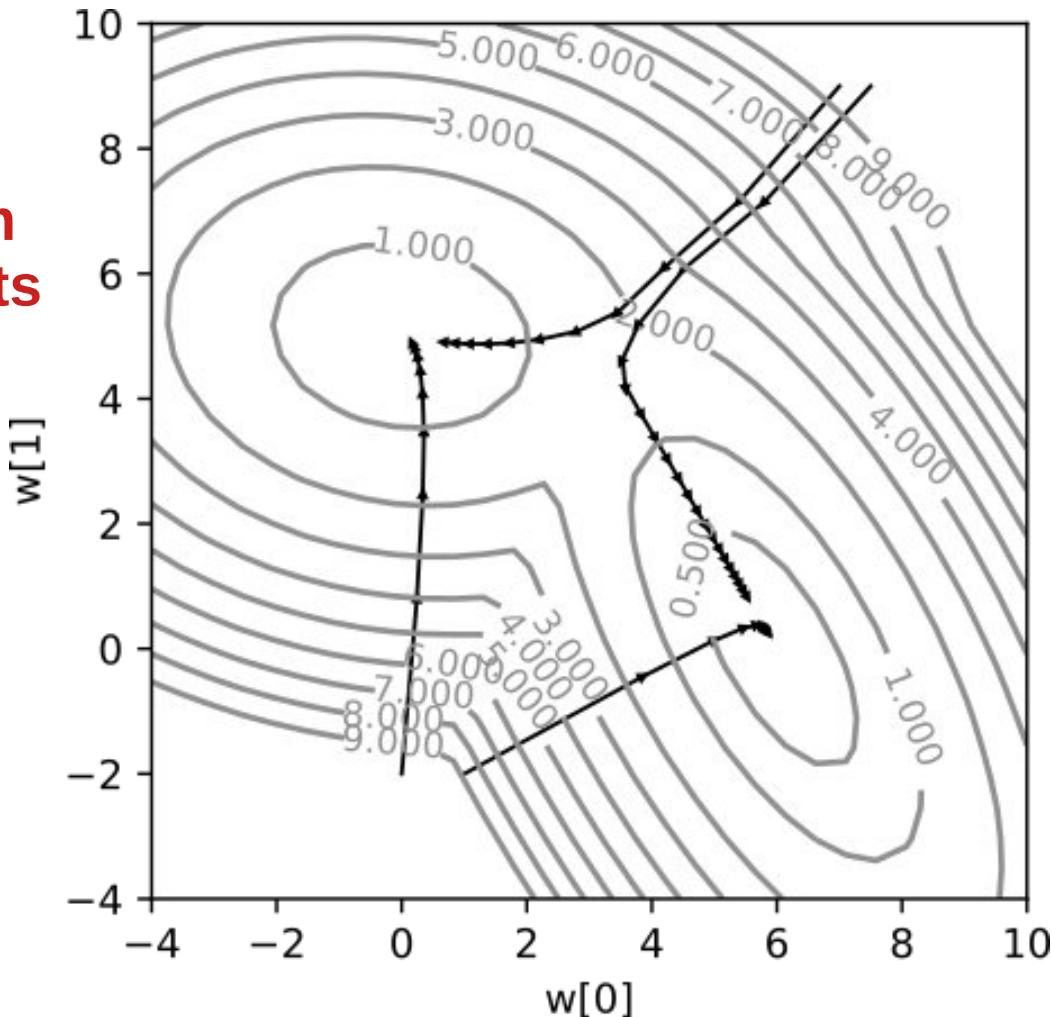
# Gradient descent

- ▶ Small learning rate, Large learning rate



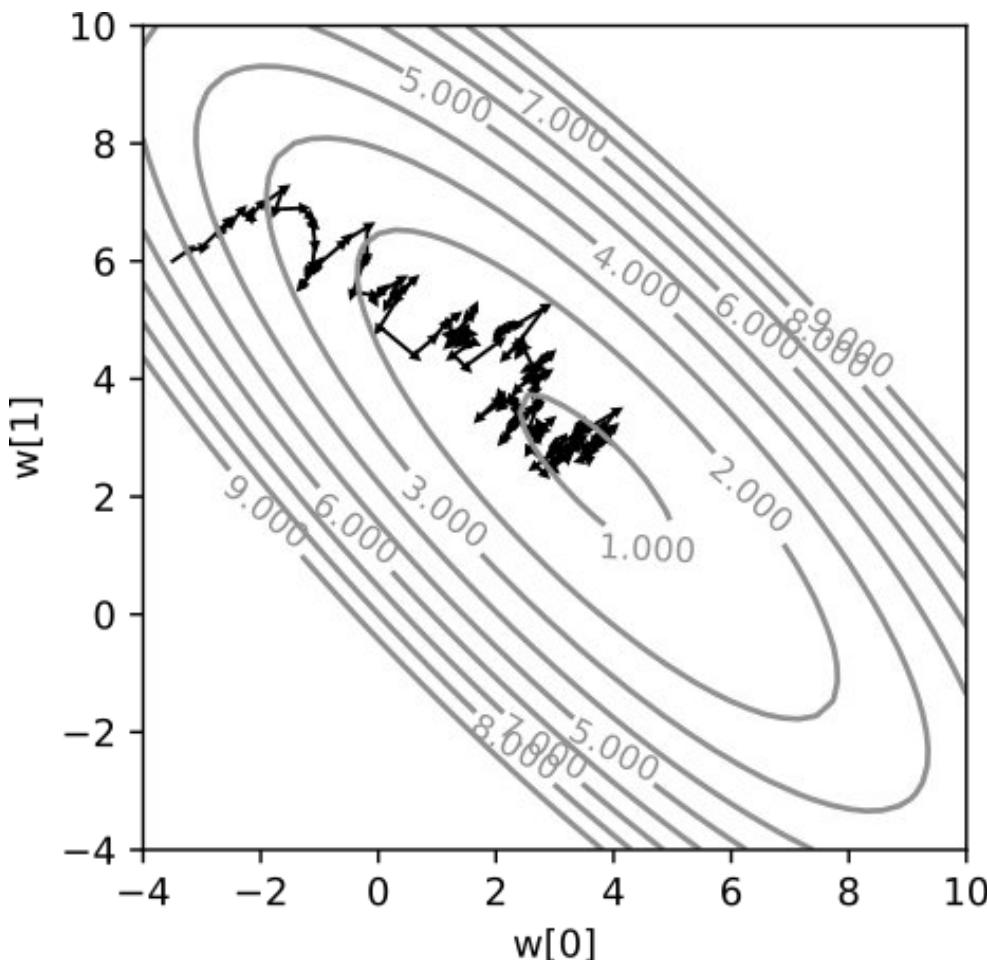
# Gradient descent

- ▶ Non convex objective
- ▶ This is not as much of a problem as you might think for neural nets
  - ▶ Because of the high dimension
  - ▶ More on this later



# Stochastic Gradient {Descent,Optimization}

- ▶ For when the objective is an average of many similar terms
- ▶ Erratic but fast convergence
- ▶ Exploits the redundancy in the data
- ▶ Difficult to prove the convergence theoretically
  - ▶ Bottou, L., Curtis, F. E., & Nocedal, J. (2018). Optimization methods for large-scale machine learning. Siam Review, 60(2), 223-311.



# The Convergence of Gradient Descent

$$w_{k+1} \leftarrow w_k - \gamma_k \frac{\partial f(w_k)}{\partial w}$$

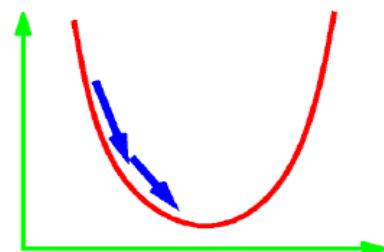
$$f(w) = \frac{1}{P} \sum_{i=1}^P L(x_i, y_i, w)$$

■ Batch Gradient

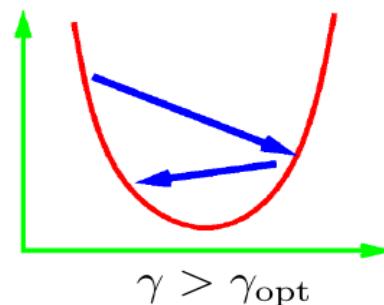
■ There is an optimal learning rate

■ Equal to inverse 2<sup>nd</sup> derivative

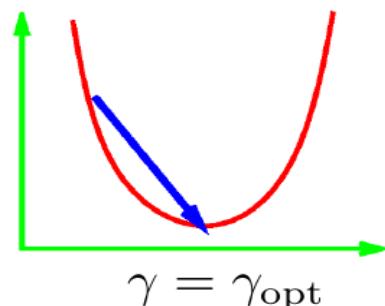
$$\gamma_{opt} = \left( \frac{\partial^2 f(w)}{\partial w^2} \right)^{-1}$$



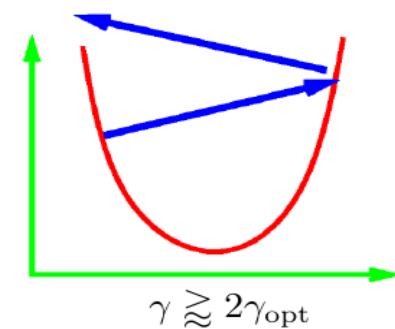
$$\gamma < \gamma_{opt}$$



$$\gamma > \gamma_{opt}$$



$$\gamma = \gamma_{opt}$$



$$\gamma \gtrsim 2\gamma_{opt}$$

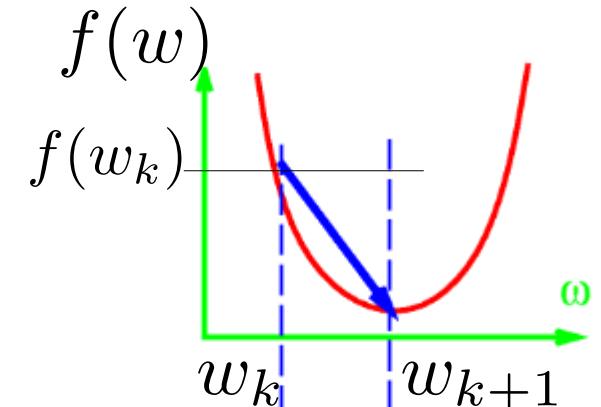
# Optimal learning rate in 1D (with formulas for multidim)

- ▶ Quadratic objective function: parabola
- ▶ Gradient (derivative): line
- ▶ Slope of gradient line: 2<sup>nd</sup> derivative
- ▶ For what value of w does the gradient line intersect the x axis (zero gradient)?

$$(w_{k+1} - w_k)^T \frac{\partial^2 f(w_k)}{\partial w^2} = \frac{\partial f(w_{k+1})}{\partial w} - \frac{\partial f(w_k)}{\partial w}$$

$$\frac{\partial f(w_{k+1})}{\partial w} - \frac{\partial f(w_k)}{\partial w}$$

$$w_{k+1} - w_k$$



$$\frac{\partial f(w)}{\partial w}$$

Slope =  $\frac{\partial^2 f(w_k)}{\partial w^2}$

# Optimal learning rate in 1D (with formulas for multidim)

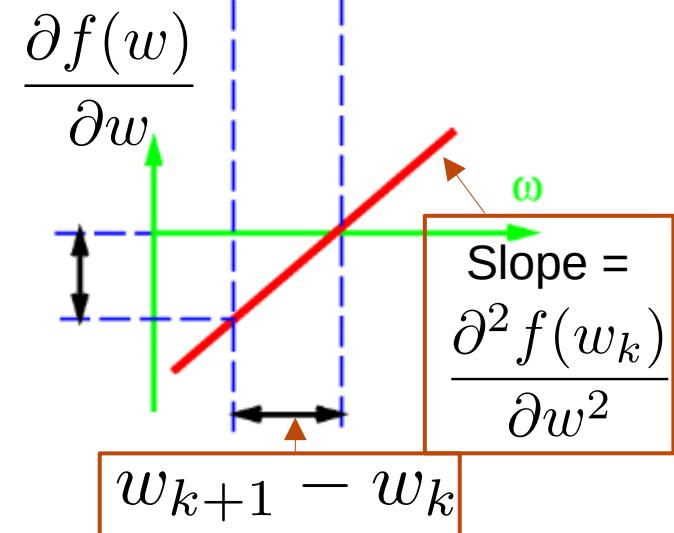
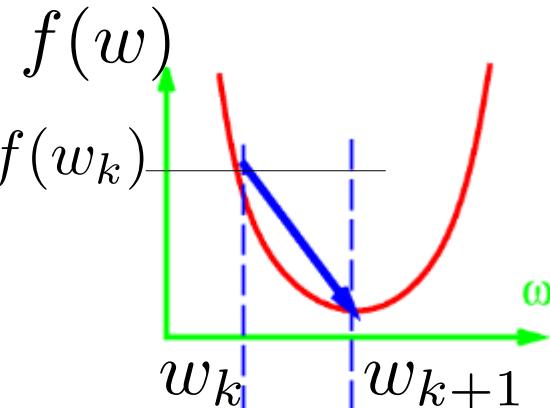
► Solve for  $w$  with gradient at  $w_{k+1} = 0$

$$\frac{\partial^2 f(w_k)}{\partial w^2}^T (w_{k+1} - w_k) = \frac{\partial f(w_{k+1})}{\partial w}^T - \frac{\partial f(w_k)}{\partial w}^T$$

$$\frac{\partial f(\check{w})}{\partial w}^T = 0 \quad \boxed{\frac{\partial^2 f(w_k)}{\partial w^2}(\check{w} - w_k) = -\frac{\partial f(w_k)}{\partial w}^T}$$

$$\check{w} = w_k - \frac{\partial^2 f(w_k)}{\partial w^2}^{-1} \frac{\partial f(w_k)}{\partial w}^T$$

$$\gamma_{opt} = \left( \frac{\partial^2 f(w)}{\partial w^2} \right)^{-1}$$



# Theory for the quadratic case

Theory in the quadratic  
(positive definite) case

$$\begin{aligned} f(w) &= \frac{1}{2}w^\top Aw - b^\top w \\ \nabla f(w) &= Aw - b \\ w_* &= A^{-1}b \end{aligned}$$

$$w_{k+1} = w_k - \gamma \nabla f(w_k)$$

$$\begin{aligned} \|w_{k+1} - w_*\| &= \|w_k - \gamma(Aw_k - b) - w_*\| \\ &= \|w_k - \gamma A(w_k - A^{-1}b) - w_*\| \\ &= \|w_k - \gamma A(w_k - w_*) - w_*\| \\ &= \|(I - \gamma A)(w_k - w_*)\| \\ &\leq \|I - \gamma A\| \|w_k - w_*\| \end{aligned}$$

Let  $\mu$  be the minimum eigenvalue of  $A$  and  $L$  the max. Then the extremal eigenvalues of  $I - \gamma A$  are  $1 - \mu\gamma$  and  $1 - L\gamma$ , so:

$$\|I - \gamma A\| = \max \{|1 - \mu\gamma|, |1 - L\gamma|\}$$

If we use two large a step size  $1 - L\gamma$  becomes negative. A standard value is  $\gamma = 1/L$  which gives:

$$\|w_{k+1} - w_*\| \leq \left(1 - \frac{\mu}{L}\right) \|w_k - w_*\|$$

# Theory for the quadratic case

The convergence of optimization methods depends on the condition number of a problem

Inverse of the condition number

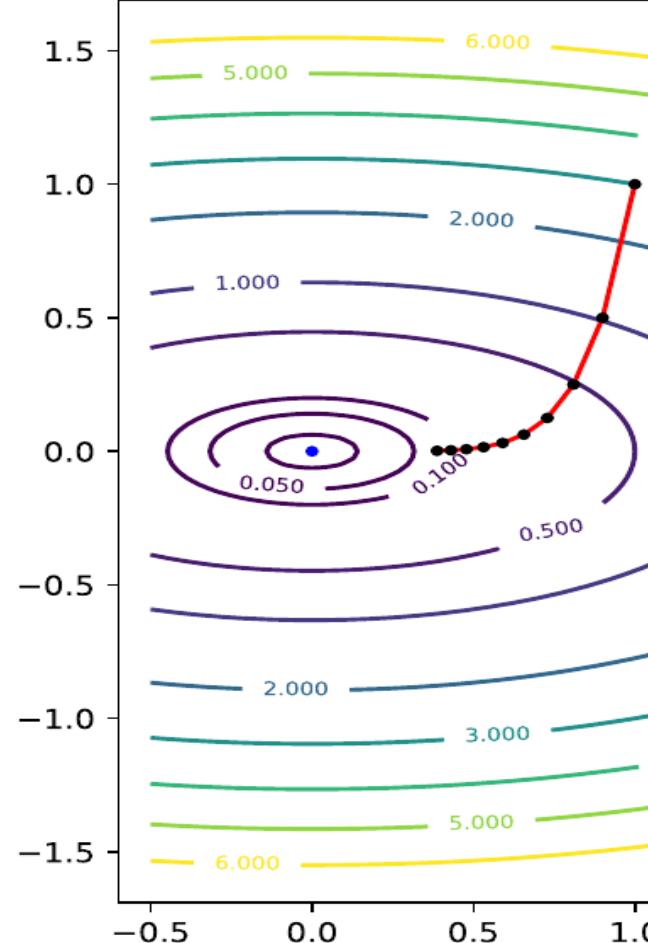
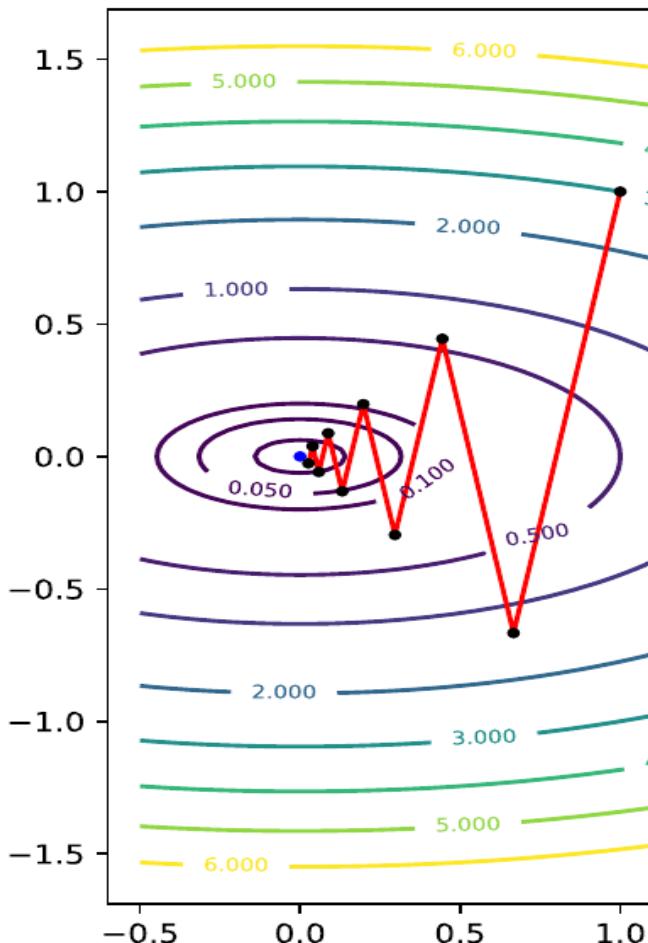


$$\|w_{k+1} - w_*\| \leq \left(1 - \frac{\mu}{L}\right) \|w_k - w_*\|$$

The condition number only formally makes sense on simple problems (“strongly convex”) But we often talk about “**poorly conditioned**” and “**well conditioned**” problems in machine learning informally

► Lecture 4 stopped here

# Step size should be as large as possible



From Aaron Defazio

# Let's Look at a single linear unit

■ Single unit, 2 inputs

■ Quadratic loss  $L(x, y, w) = (y - w^t x)^2$

■ Dataset: classification:  $y=-1$  for blue,  $+1$  for red.

■ Second derivative matrix = Hessian

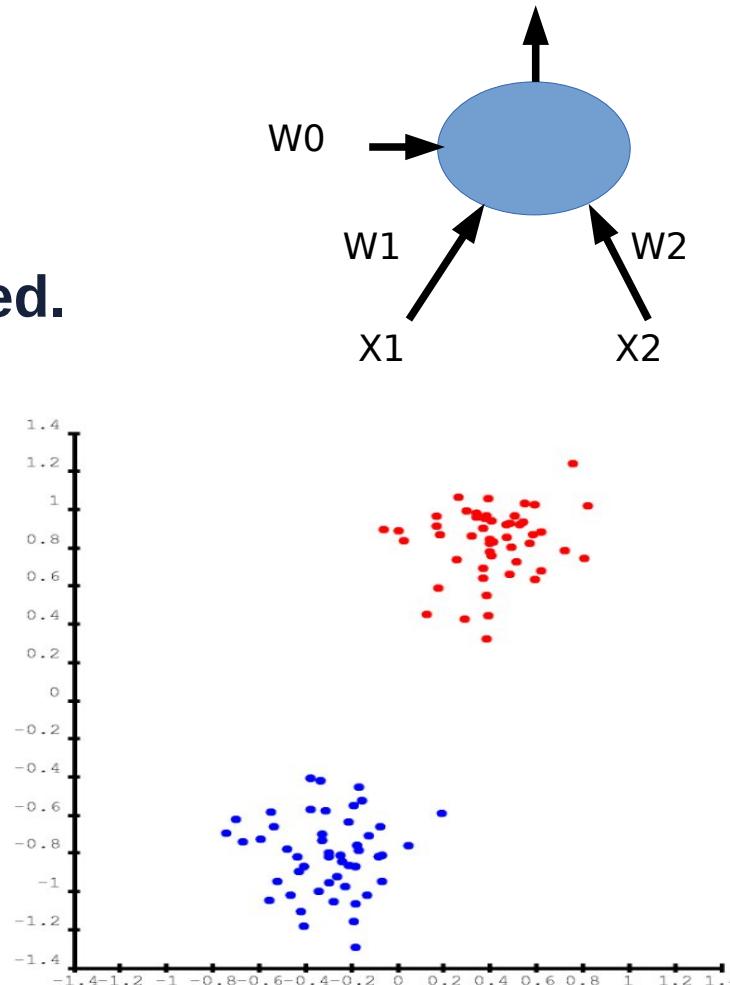
■ is covariance matrix of input vectors

$$H = \frac{\partial^2 f(w)}{\partial w} = \frac{1}{P} \sum_{i=1}^p x(i)x(i)^T$$

■ To avoid ill conditioning: **normalize the in**

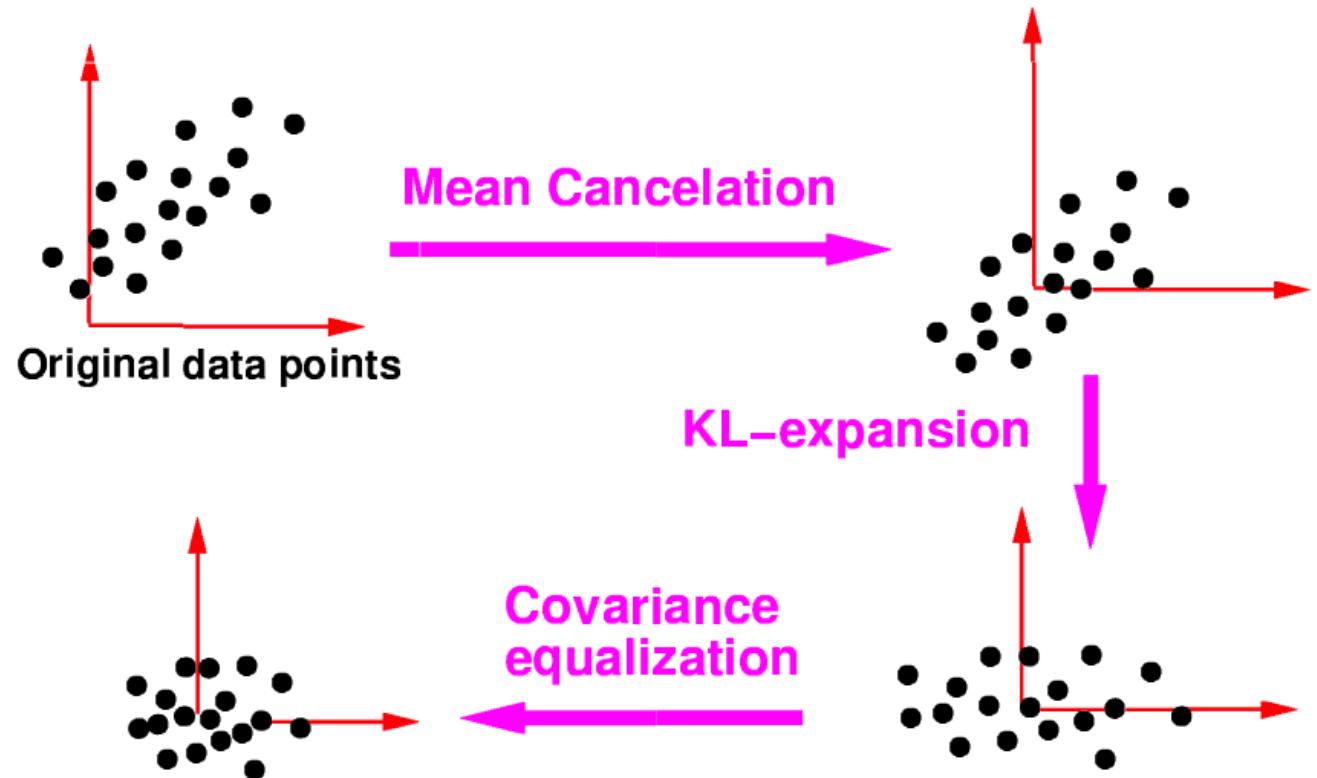
- ▶ Zero mean

- ▶ Unit variance for all variable



# Preprocessing the data

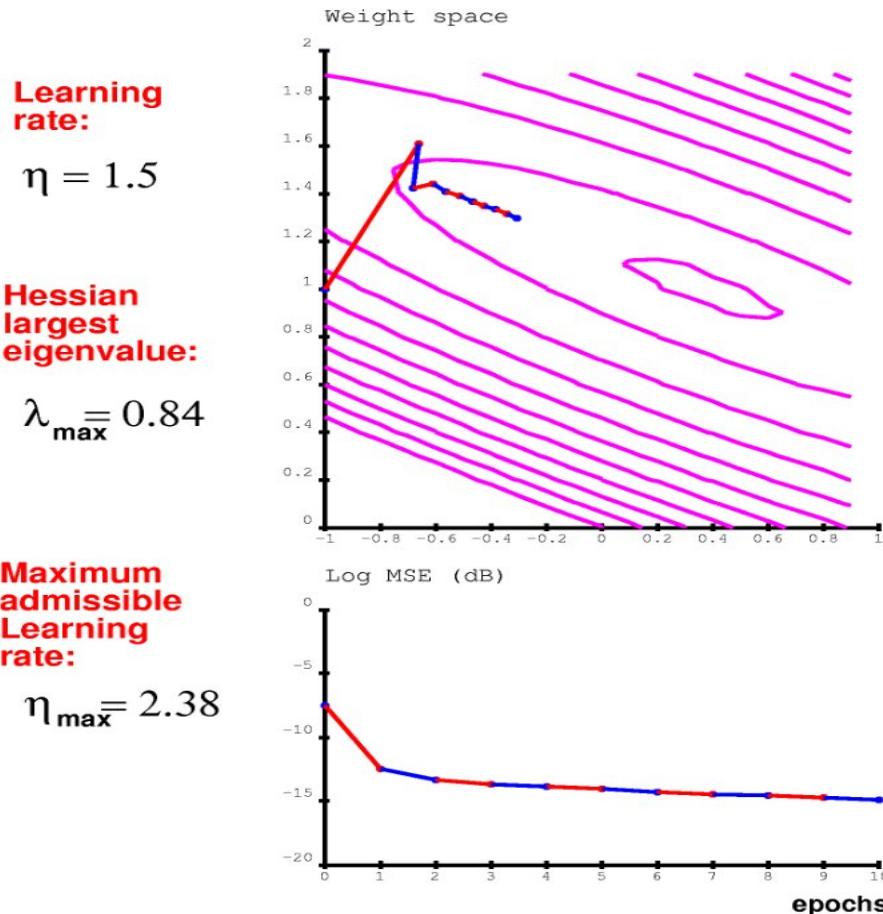
- ▶ Inputs and internal state variables should be:
  - ▶ zero mean
  - ▶ Equal variances
  - ▶ As uncorrelated with each other as possible
- ▶ BatchNorm does this inside the network!



# Convergence is Slow When Hessian has Different Eigenvalues



## Batch Gradient, small learning rate

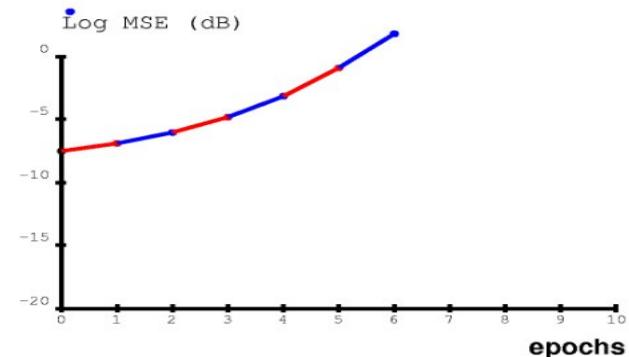
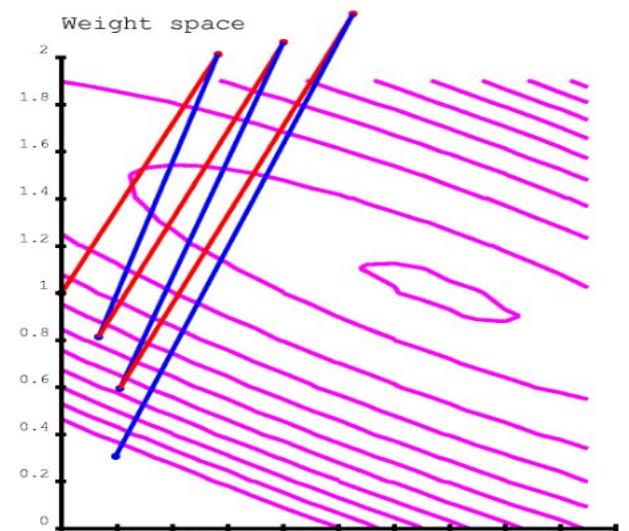


## Batch Gradient, large learning rate

Learning rate:  
 $\eta = 2.5$

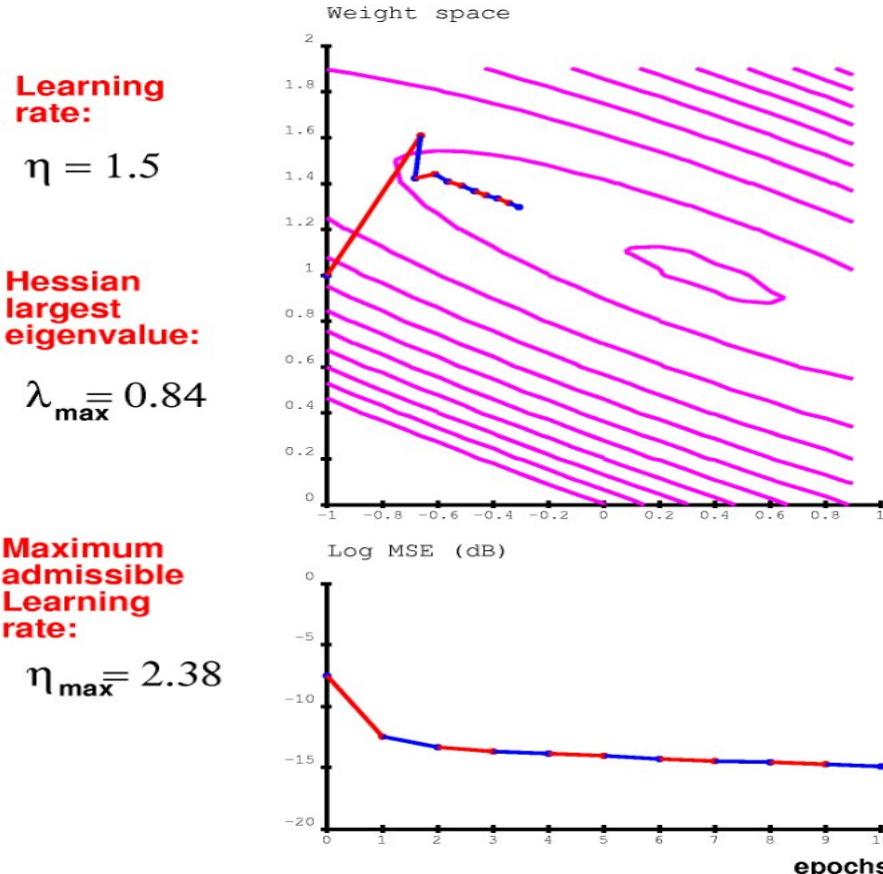
Hessian largest eigenvalue:  
 $\lambda_{\max} = 0.84$

Maximum admissible Learning rate:  
 $\eta_{\max} = 2.38$

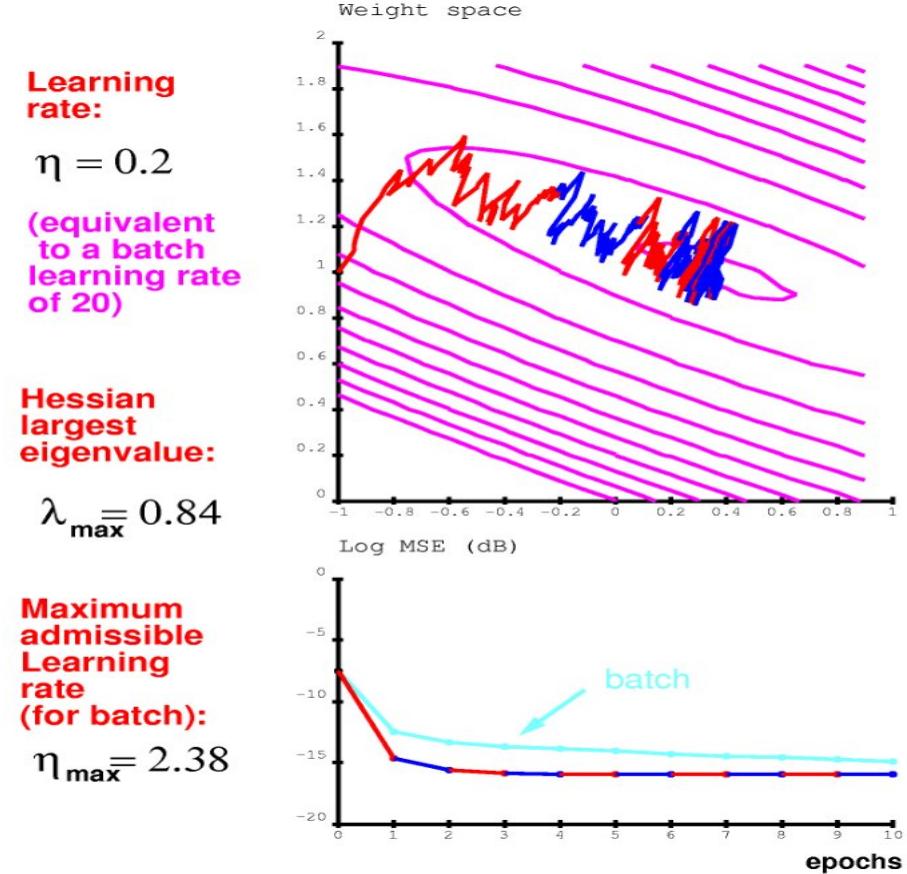


# Convergence is Slow When Hessian has Different Eigenvalues

## Batch Gradient, small learning rate



Stochastic Gradient: Much Faster  
But fluctuates near the minimum

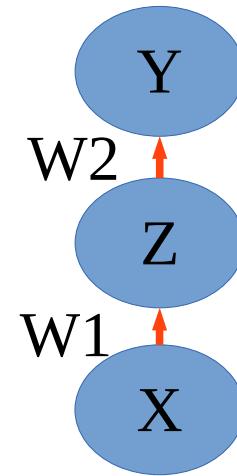
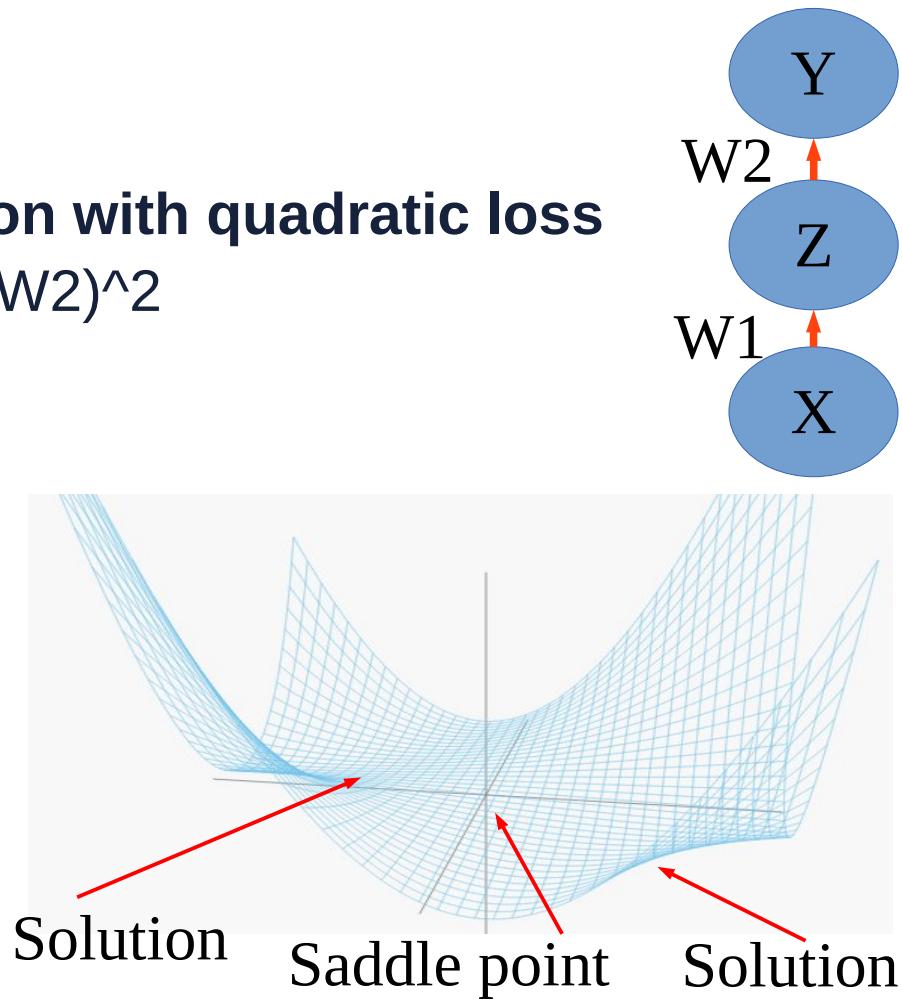
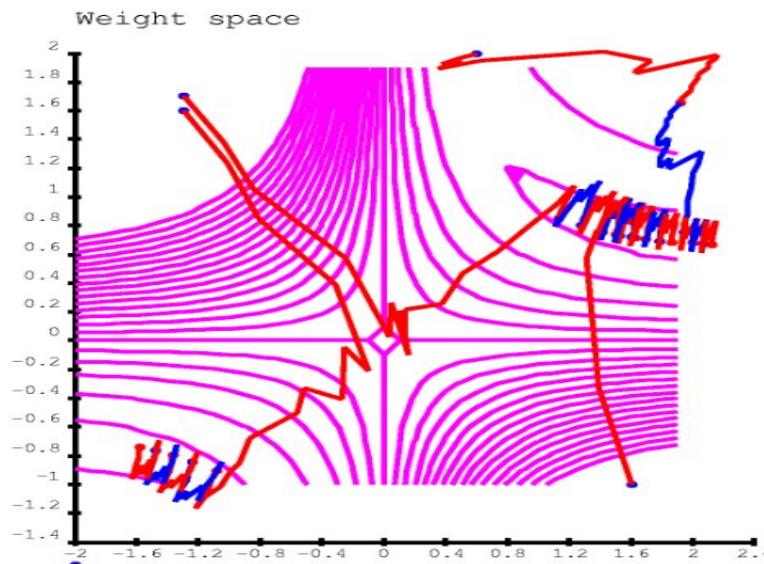


# Convergence of GD

- ▶ **Convergence speed depends on conditioning**
- ▶ **Conditioning: ratio of largest to smallest non-zero eigenvalue of the Hessian**
- ▶ **How to condition?**
  - ▶ Center all the variables that enter a weight
  - ▶ Normalize the variance of all variables that enter a weight

# Multilayer Nets Have Non-Convex Objective Functions

- ▶ **1-1-1 network**
- ▶  $Y = W_1 * W_2 * X$
- ▶ **trained to compute the identity function with quadratic loss**
- ▶ Single sample  $X=1, Y=1 \ L(W) = (1-W_1 * W_2)^2$
- ▶ **Solution:  $W_2 = 1/W_1$  hyperbola.**



# Stochastic Optimization

Stochastic optimization

$$\underset{w}{\text{minimize}} \ f(w) = \frac{1}{n} \sum_i^n f_i(w)$$

Typically each  $f_i$  is the loss of the neural network on a single instance

$$f_i(w) = \ell(x_i, y_i, w)$$

Note: the optimization community uses  $f$  by convention, so I follow that notation here

$$w_{k+1} = w_k - \gamma_k \nabla f(w_k)$$
 GD: The WORST method in virtually all situations

---

$$w_{k+1} = w_k - \gamma_k \nabla f_i(w_k)$$
 SGD: Often the BEST method available!

( $i$  chosen uniformly at random)

WHY?

# SGD is GD in expectation...

Since:

$$\mathbb{E}[\nabla f_i(w_k)] = \nabla f(w_k)$$

The SGD step's expectation is just the gradient step:

$$\mathbb{E}[w_{k+1}] = w_k - \gamma_k \nabla f(w_k)$$

It's useful to think of SGD as GD with **noise**.

# SGD exploits the redundancy in the samples

## Advantages of SGD

$$w_{k+1} = w_k - \gamma_k \nabla f_i(w_k)$$

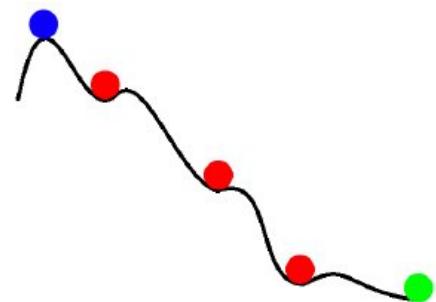
$$\underset{w}{\text{minimize}} \ f(w) = \frac{1}{n} \sum_i^n f_i(w)$$

$$f_i(w) = \ell(x_i, y_i, w)$$

- There is redundant information across instances  
(i.e. MNIST has thousands of near identical digits)  
the noise is lower when there is higher redundancy.

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

- At the early stages of optimization, the noise is small compared to the information in the gradient, so a SGD step is **virtually as good** as a GD step.
- The noise can prevent the optimizing converging to bad local minima, a phenomena called **annealing**
- Stochastic gradients are drastically cheaper to compute (proportional to your dataset size), so you can often take thousands of SGD steps for the cost of one GD step.



# Minibatching: a necessary evil due to our hardware

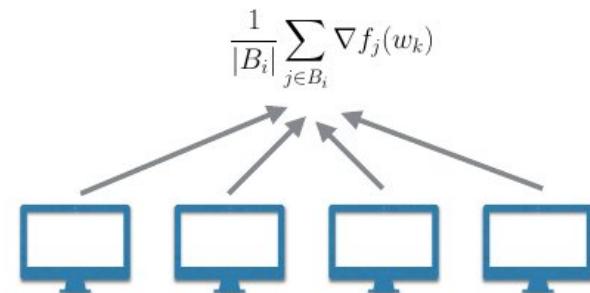
## Mini-batching

$$w_{k+1} = w_k - \gamma_k \frac{1}{|B_i|} \sum_{j \in B_i} \nabla f_j(w_k)$$

Often we are able to make better use of our hardware by using mini batches instead of single instances

For instance, modern graphics cards are poorly utilized if we try and calculate single instance batches.

The most common distributing training technique on a cluster involves splitting a large mini batch between the machines and aggregating the resulting gradient.



All methods we discuss for stochastic optimization work with mini batches without issue

# Don't use full batch, but if you must...

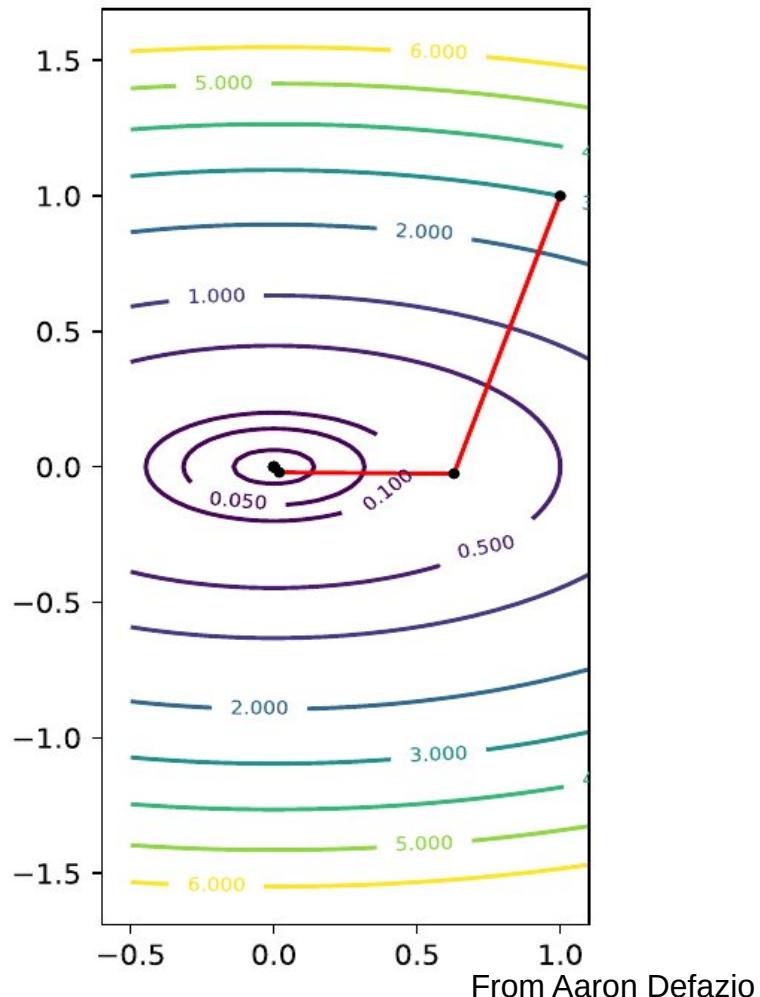
If you must use a full-batch method ...

For batch optimization, **LBFGS** & Conjugate-Gradients  
are far better than vanilla gradient descent

`torch.optim.LBFGS`      `scipy.optimize.fmin_l_bfgs_b`

**BUT** we don't know how to best combine  
these techniques with stochasticity!

Plain SGD (with momentum) is still the best  
method for training many state-of-the-art  
neural networks such as ResNet models.



# Momentum acceleration methods

The most misunderstood idea in optimization?

$$p_{k+1} = \hat{\beta}_k p_k + \nabla f_i(w_k)$$

$$w_{k+1} = w_k - \gamma_k p_{k+1}$$

Extra term on top of SGD

SGD + Momentum = Stochastic **heavy ball** method

$$w_{k+1} = w_k - \gamma_k \nabla f_i(w_k) + \beta_k (w_k - w_{k-1})$$

$$0 \leq \beta < 1$$

This is mathematically equivalent to the previous form for a particular value of beta.

**Key idea:** The next step becomes a combination of the previous step's direction and the new negative gradient

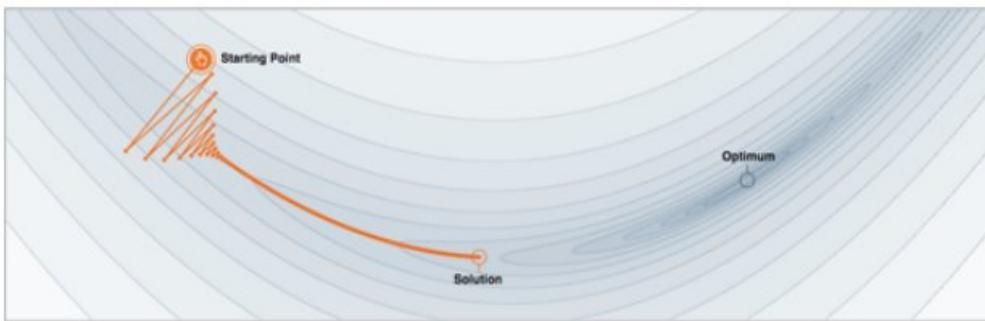
# Momentum acceleration methods

$$p_{k+1} = \hat{\beta}_k p_k + \nabla f_i(w_k)$$

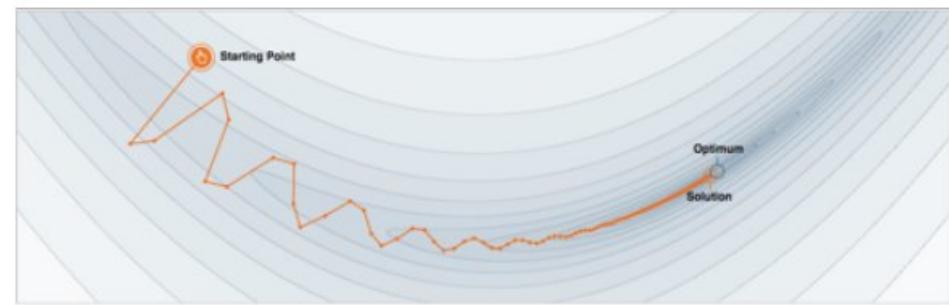
$$w_{k+1} = w_k - \gamma_k p_{k+1}$$

The optimization process resembles a **heavy ball** rolling down a hill. The ball has **momentum**, so it doesn't change direction immediately when it encounters changes to the landscape!

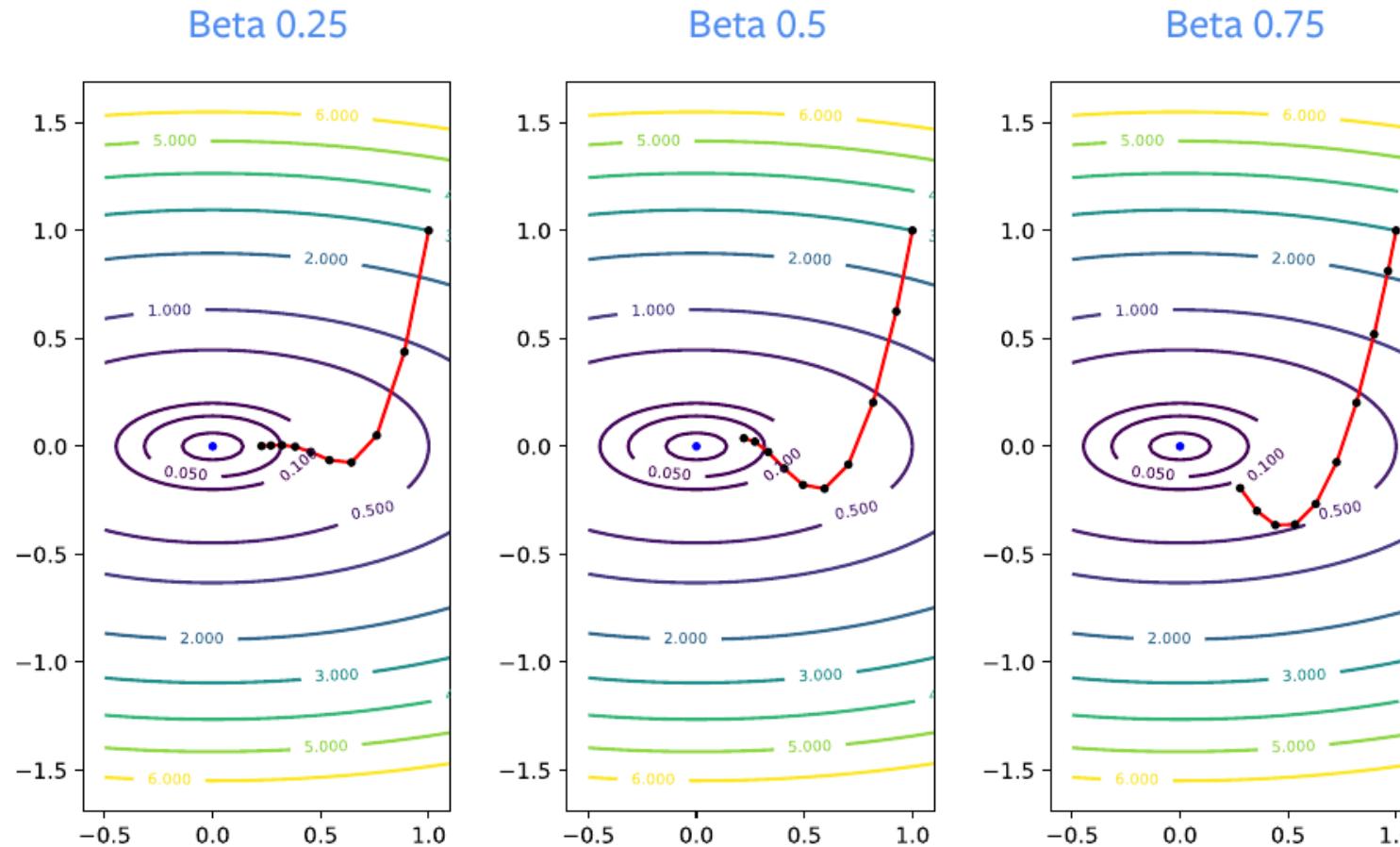
## Without momentum



## With momentum



# Larger momentum $\rightarrow$ slower reactions to changes in landscape



# Momentum in practice

## Practical Aspects of momentum

$$p_{k+1} = \hat{\beta}_k p_k + \nabla f_i(w_k)$$

$$w_{k+1} = w_k - \gamma_k p_{k+1}$$

It's basically “free lunch”, in almost all situations, SGD + momentum is better than SGD, and very rarely worse!

### Recommended Parameters:

Beta = **0.9** or **0.99** almost always works well. Sometimes slight gains can be had by tuning it.

The step size parameter usually needs to be decreased when the momentum parameter is increased to maintain convergence.

# Momentum accelerates

## Explanation one: Acceleration

The momentum method is often conflated with Nesterov's momentum

Regular momentum

$$p_{k+1} = \hat{\beta}_k p_k + \nabla f_i(w_k)$$

$$w_{k+1} = w_k - \gamma_k p_{k+1}$$

Nesterov's momentum

$$p_{k+1} = \hat{\beta}_k p_k + \nabla f_i(w_k)$$

$$w_{k+1} = w_k - \gamma_k \left( \nabla f_i(w_k) + \hat{\beta}_k p_{k+1} \right)$$

`torch.optim.SGD(..., nesterov=True)`

Nesterov's momentum, when using **VERY** carefully chosen constants, provably  
“**accelerates**” convergence on problems with simple structure (convex)

Regular momentum is also accelerated: but only on **quadratics!**

There is no theory to suggest this acceleration occurs when training neural networks,  
although a practical speedup is often observed.

**Surprisingly**, Nesterov's momentum usually performs the same as regular momentum  
when training neural networks.

Acceleration alone does not explain momentum!

From Aaron Defazio

# But momentum also smoothes the gradient noise

In optimization, we usually take the last  $w$  as our estimate of the best parameters at the end.

When using **SGD**, this is suboptimal! We should actually take an **average** over past time steps (with weights ideally depending on the problem structure)

$$\bar{w}_K = \frac{1}{K} \sum_{k=1}^K w_k$$

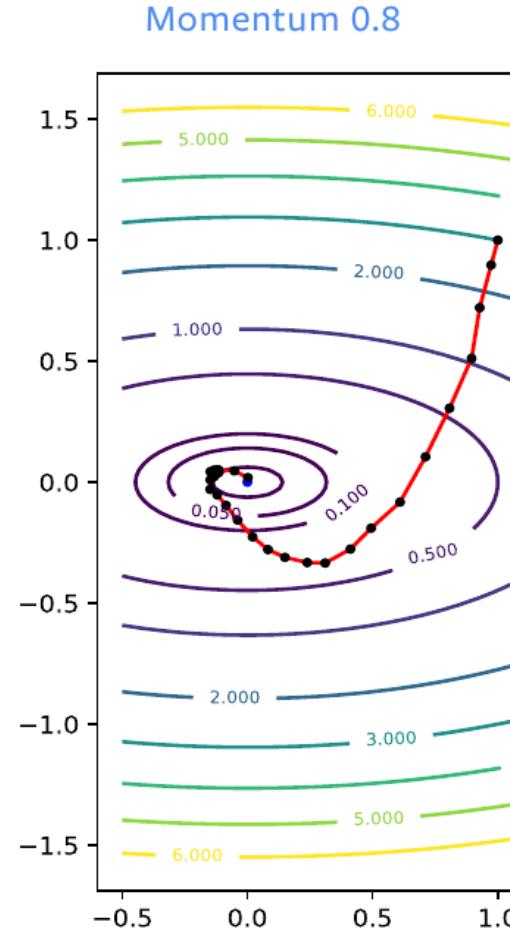
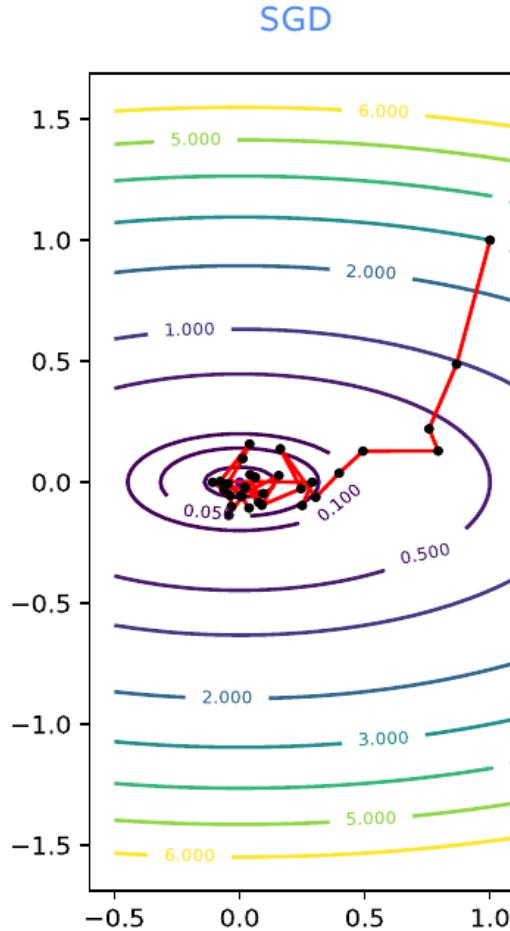
In contrast, theory suggests that SGD + Momentum requires **no averaging**, the **last value** may directly be returned!

Momentum **smooths** the noise from the stochastic gradients

Both noise smoothing and acceleration contribute to the high performance of momentum

# Is reducing the noise good or bad?

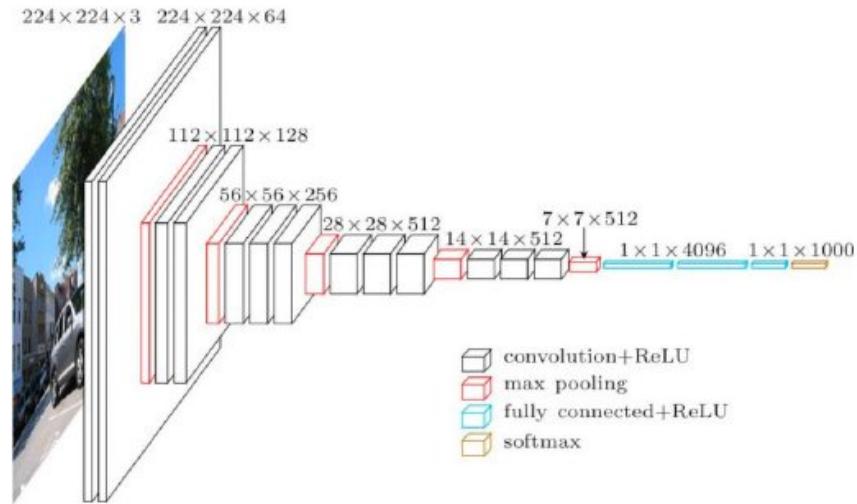
Quadratic  
with STD  
1.0 noise  
injected  
into b



- ▶ Noise can accelerate learning
- ▶ Noise can improve generalization
- ▶ Noise can drive the parameter to “flat” regions of the loss

# Automatic learning rate adjustment

The magnitude of the gradients often varies highly between layers due to, so a global learning rate may not work well.

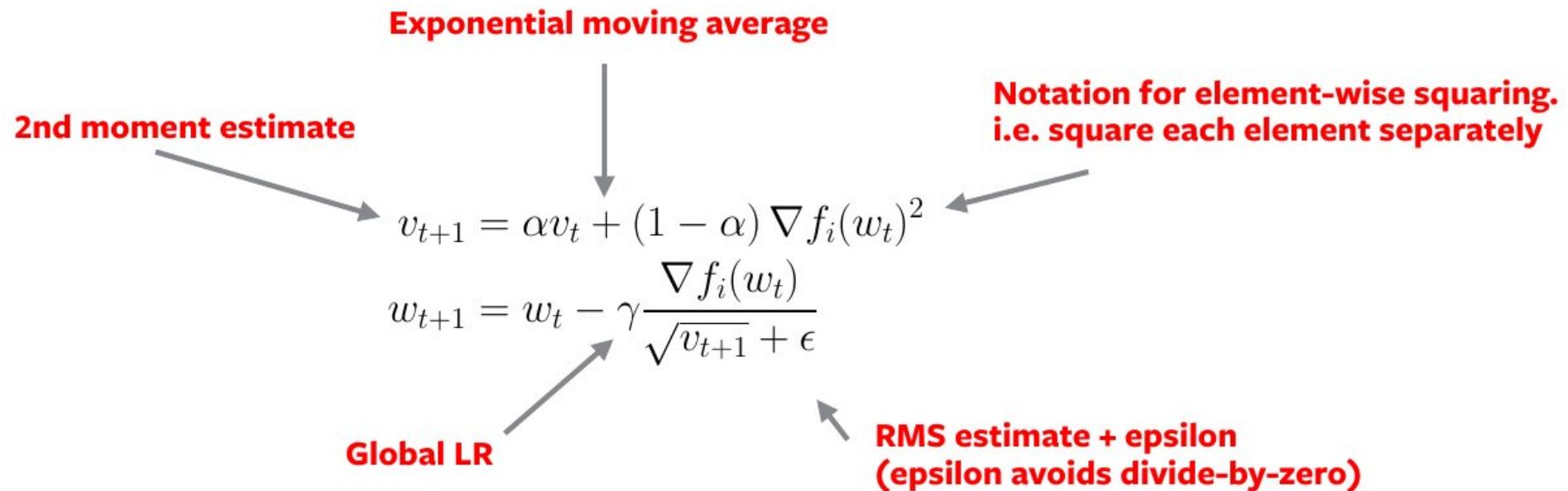


General **IDEA**: Instead of using the same learning rate for every weight in our network, **maintain an estimate of a better rate separately for each weight**.

The exact way of adapting to the learning rates varies between algorithms, But most methods either **adapt to the variance** of the weights, or to the **local curvature** of the problem.

# RMSprop

Key **IDEA**: normalize by the **root-mean-square** of the gradient



Most adaptive methods use a moving average, it's a simple way to estimate a noisy quantity that changes over time.

If the expected value of gradient is small, this is similar to dividing by the standard deviation (i.e. whitening)

From Aaron Defazio

# Adam

## Adam: RMSprop with a kind of momentum

“Adaptive Moment Estimation”

Presented here without **bias-correction** (i.e. steady state Adam)

### Momentum (Exponential moving average)

**2nd moment estimate  
(same as RMSProp)**

$$m_{t+1} = \beta v_t + (1 - \beta) \nabla f_i(w_t)$$

$$v_{t+1} = \alpha v_t + (1 - \alpha) \nabla f_i(w_t)^2$$

$$w_{t+1} = w_t - \gamma \frac{m_t}{\sqrt{v_{t+1}} + \epsilon}$$

### RMSProp

$$v_{t+1} = \alpha v_t + (1 - \alpha) \nabla f_i(w_t)^2$$

$$w_{t+1} = w_t - \gamma \frac{\nabla f_i(w_t)}{\sqrt{v_{t+1}} + \epsilon}$$

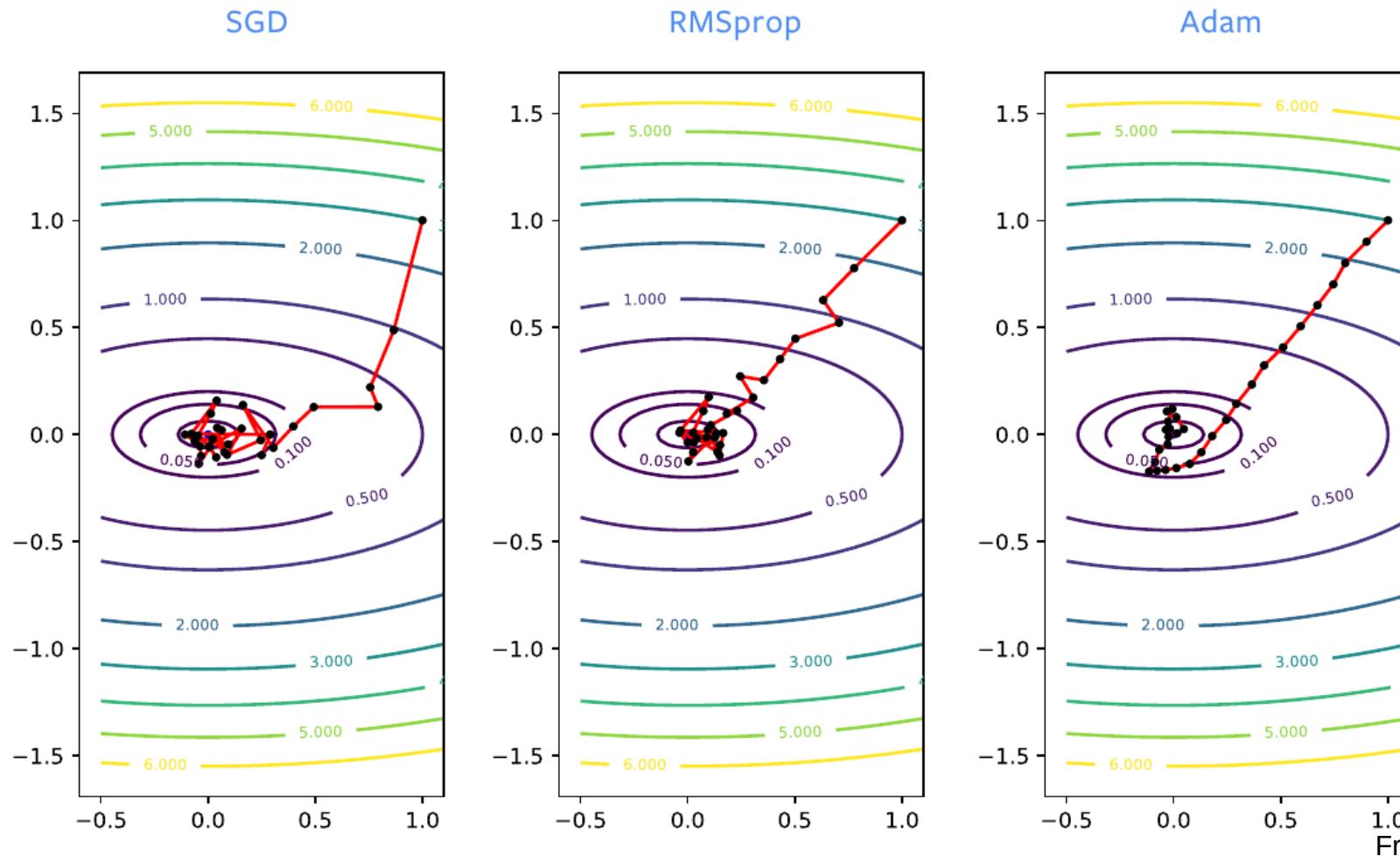
**Use  $m$  instead of the gradient**

Just as momentum improves SGD, it improves RMSProp as well.

The exponential-moving-average method of updating momentum is equivalent to the standard form under rescaling. Nothing mysterious.

The full version of Adam has **bias-correction** as well, which just keeps the moving averages **unbiased** during early iterations. The algorithm quickly approaches the above steady state form.

# Pure SGD vs RMSprop vs Adam



From Aaron Defazio

# Practical considerations

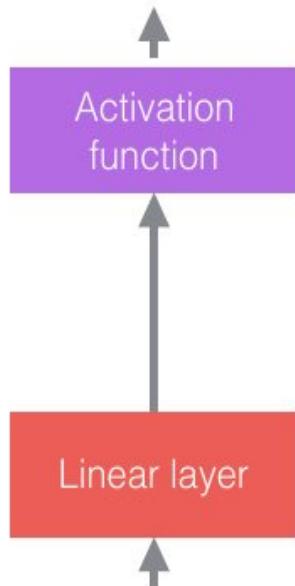
For poorly conditioned problems, Adam is often **much** better than SGD.  
I recommend using Adam over RMSprop due to the clear advantages of momentum

BUT, Adam is poorly understood theoretically, and has known disadvantages:

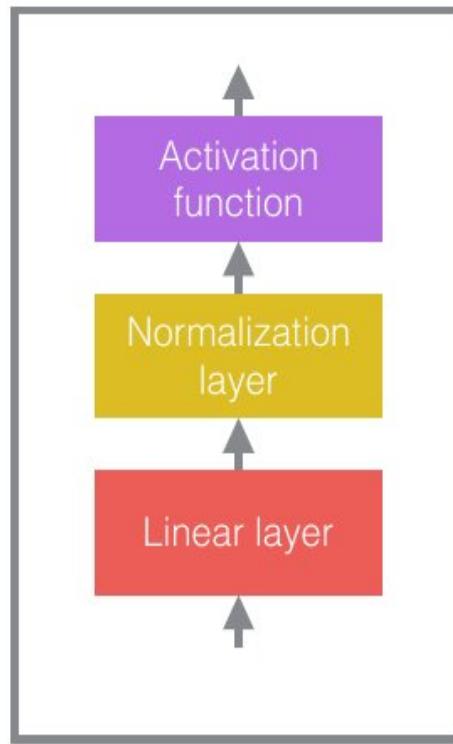
- Does not converge at all on some simple example problems!
- Gives worse generalization error on many computer vision problems (i.e. ImageNet)
- Requires more memory than SGD
- Has 2 momentum parameters, so some tuning may be needed

It's a good idea to try both SGD + momentum and Adam with a sweep of different learning rates, and use whichever works better on held out data

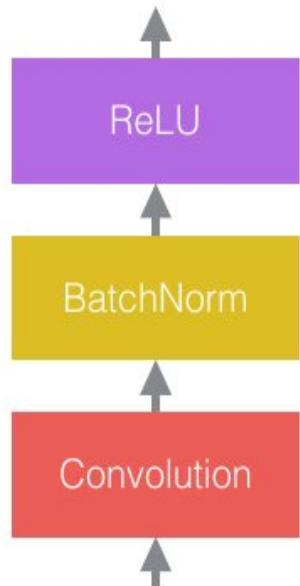
# Normalization layers



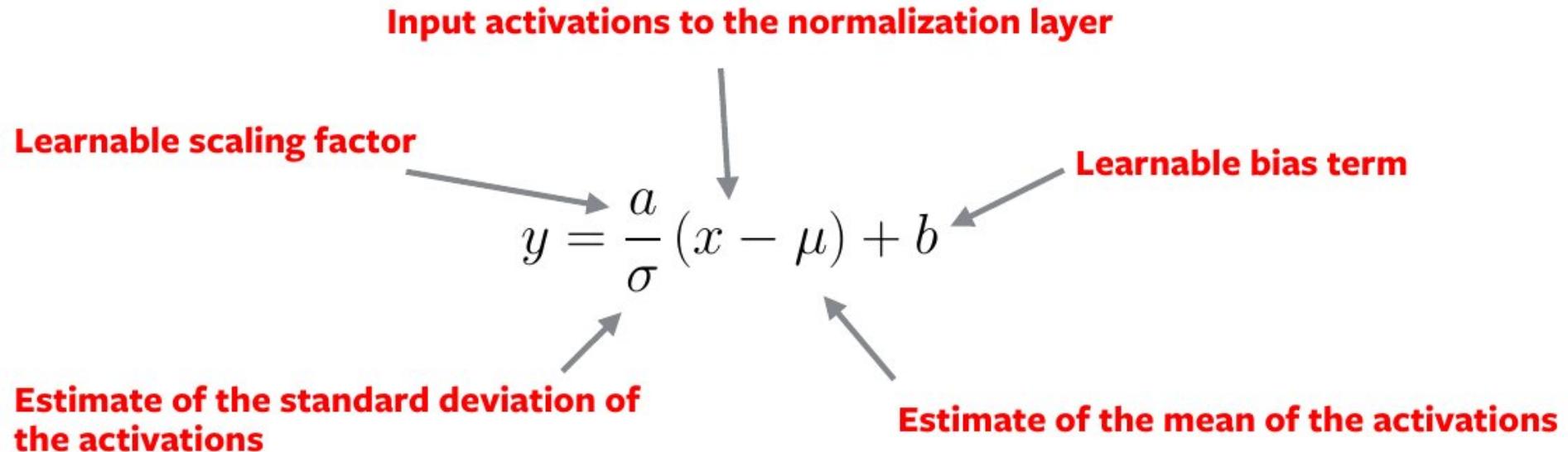
They are added in-between existing layers of a neural network



Typical image classification structure



# Normalization layers whiten the activations



The extra  $a$  &  $b$  parameters keep the representation power of the network the same

But how do we estimate the mean and standard deviation? Does it differ across channels, spatial location, instances?

# Normalization tricks

- ▶ **N=batch, C=channels, H,W space**
- ▶ **Batch norm: N,H,W**
- ▶ **Layer norm: C,H,W**
- ▶ **Instance norm: H,W**
- ▶ **Group norm: N, C subset**
  
- ▶ **Before non-linearity**
  - ▶ Needs scale and shift
- ▶ **After non-linearity**
  - ▶ Scale and shift may hurt

**Input:** Values of  $x$  over a mini-batch:  $\mathcal{B} = \{x_{1\dots m}\}$ ;  
 Parameters to be learned:  $\gamma, \beta$

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

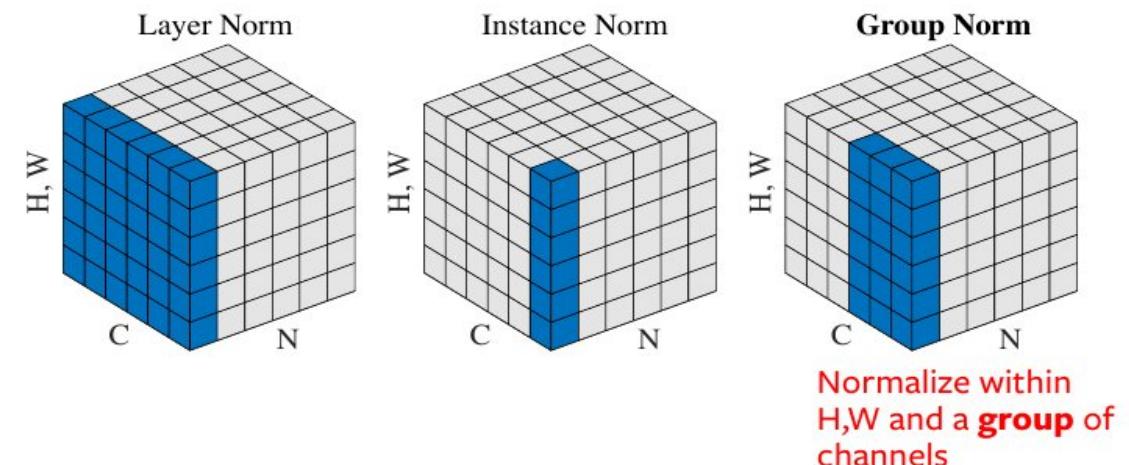
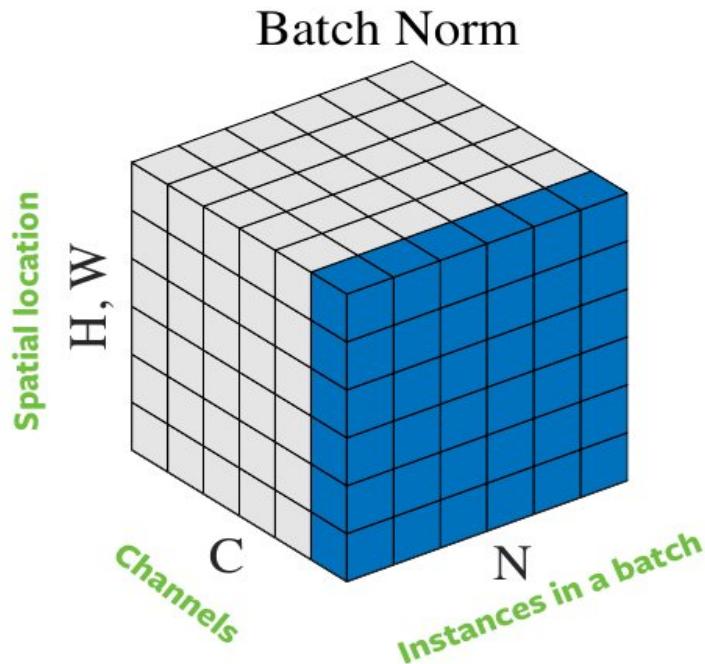
$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m} \sum_{i=1}^m x_i \quad // \text{mini-batch mean}$$

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

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

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

# Types of normalizations

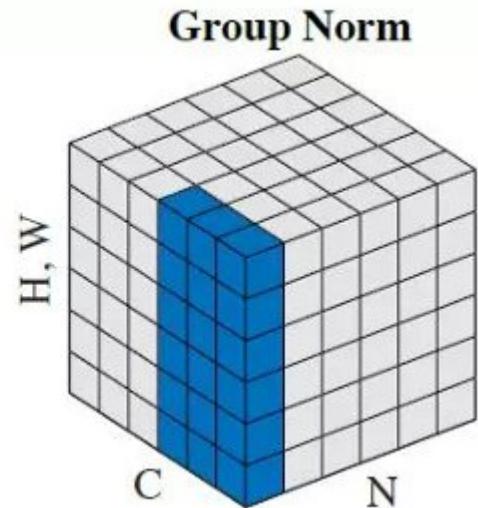
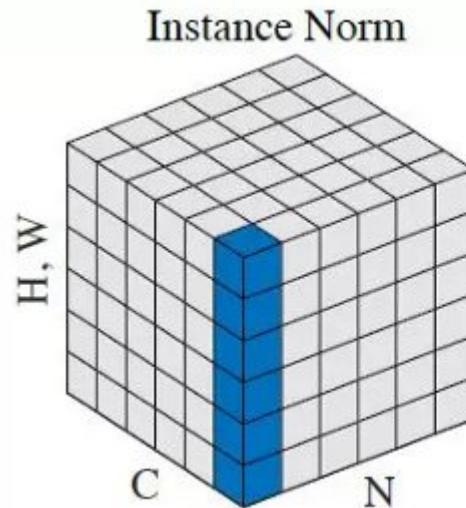
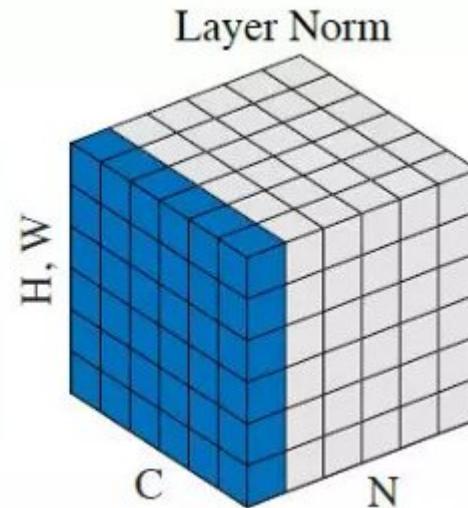
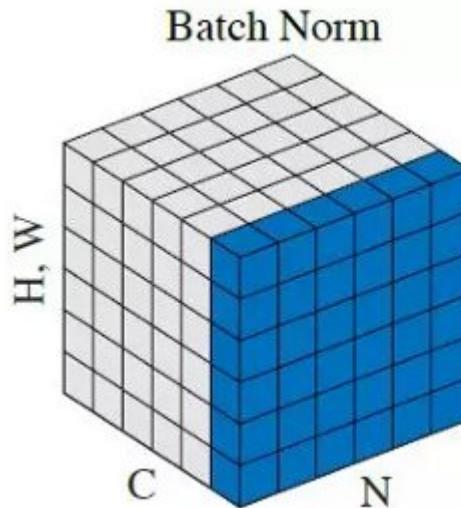


In practice group/batch norm work well for computer vision problems, and instance/layer norm are heavily used for language problems.

IMG SOURCE: <https://arxiv.org/pdf/1803.08494.pdf>

From Aaron Defazio

# Normalization tricks



- ▶ **N=batch, C=channels, H,W space**
- ▶ **Batch norm: N,H,W**
- ▶ **Layer norm: C,H,W**
- ▶ **Instance norm: H,W**
- ▶ **Group norm: N, C subset**

# Why does normalization help?

This is still disputed

The original paper said that it “reduces internal covariate shift” whatever that means

As usual, we are using something we don’t fully understand.

But, it’s clearly a combination of a number of factors:

- Networks with normalization layers are easier to optimize, allowing for the use of larger learning rates.  
(Normalization has a **optimization** effect)
- The mean/std estimates are noisy, this extra “noise” results in better generalization in some cases  
(Normalization has an **regularization** effect)
- Normalization reduces sensitivity to weight initialization

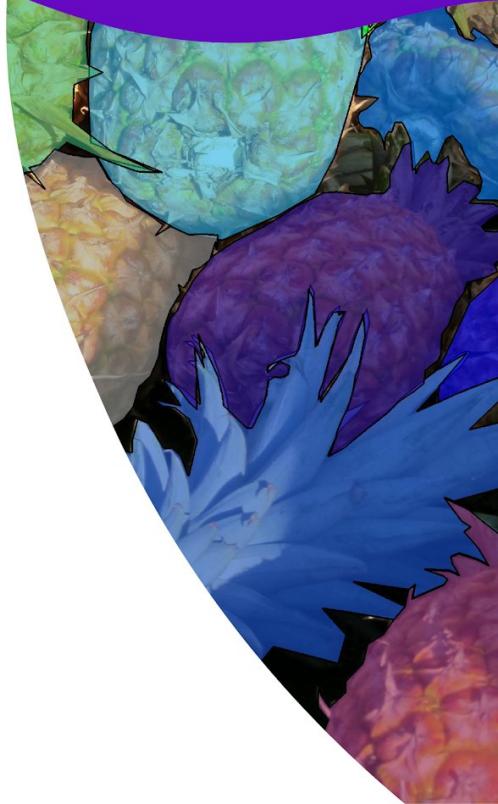
Normalization lets you be more “careless”, you can combine almost any neural network building blocks together and have a good chance of training it without having to consider how poorly conditioned it might be.

**For more info:** <https://myrtle.ai/how-to-train-your-resnet-7-batch-norm>

From Aaron Defazio

# Backprop as Lagrangian Optimization

Optimization under constraints



# Reformulating Deep Learning

## ► Loss

$$L(x, y, w) = C(z_K, y) \text{ such that } z_{k+1} = g_k(z_k, w_k), \quad z_0 = x$$

## ► Lagrangian for optimization under constraints

$$L(x, y, z, \lambda, w) = C(z_K, y) + \sum_{k=0}^{K-1} \lambda_{k+1}^T (z_{k+1} - g_k(z_k, w_k))$$

## ► Optimality conditions:

$$\frac{\partial L(x, y, z, \lambda, w)}{\partial z_k} = 0, \quad \frac{\partial L(x, y, z, \lambda, w)}{\partial \lambda_{k+1}} = 0, \quad \frac{\partial L(x, y, z, \lambda, w)}{\partial w_k} = 0$$

# Reformulating Deep Learning

$$\frac{\partial L(x, y, z, \lambda, w)}{\partial \lambda_{k+1}} = z_{k+1} - g_k(z_k, w_k) = 0 \implies z_{k+1} = g_k(z_k, w_k)$$

$$\frac{\partial L(x, y, z, \lambda, w)}{\partial z_k} = \lambda_k^T - \lambda_{k+1}^T \frac{\partial g_k(z_k, w_k)}{\partial z_k} = 0 \implies$$

► **Backprop!**

► Lambda is the gradient

$$\lambda_k = \frac{\partial g_{k-1}(z_{k-1}, w_{k-1})}{\partial z_k}^T \lambda_{k+1}$$

$$\frac{\partial L(x, y, z, \lambda, w)}{\partial w_k} = \lambda_{k+1}^T \frac{\partial g_k(z_k, w_k)}{\partial w_k}$$

# Sparse Modeling

Using L1 regularization of latent variable  
to limit its information content.

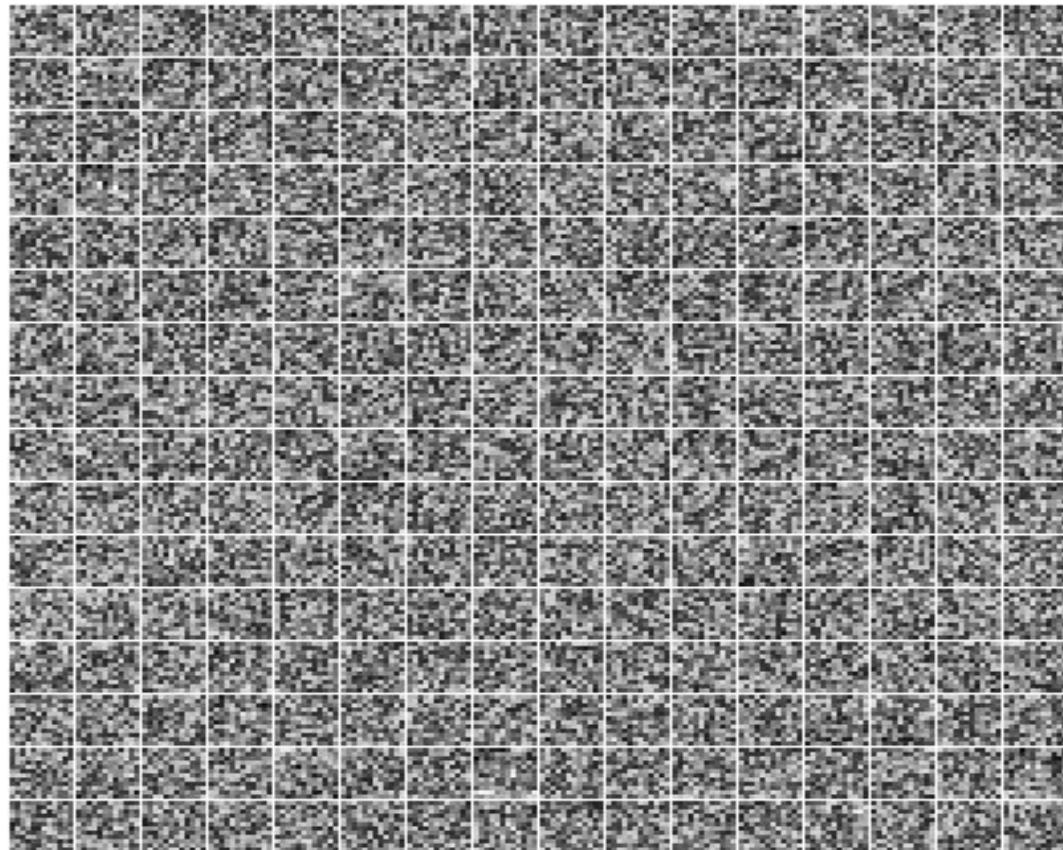
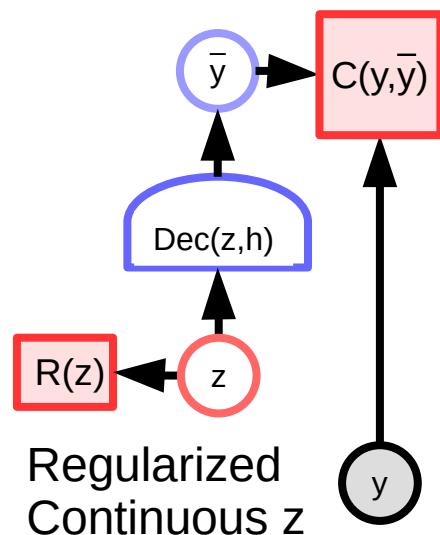
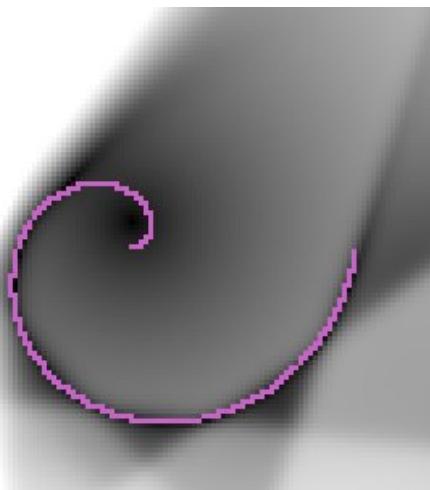


# Regularized Latent Variable: Sparse Coding

- A2: regularize the volume of the low energy regions

$$E(y, z) = \|y - wz\|^2 + \lambda|z|_{L1}$$

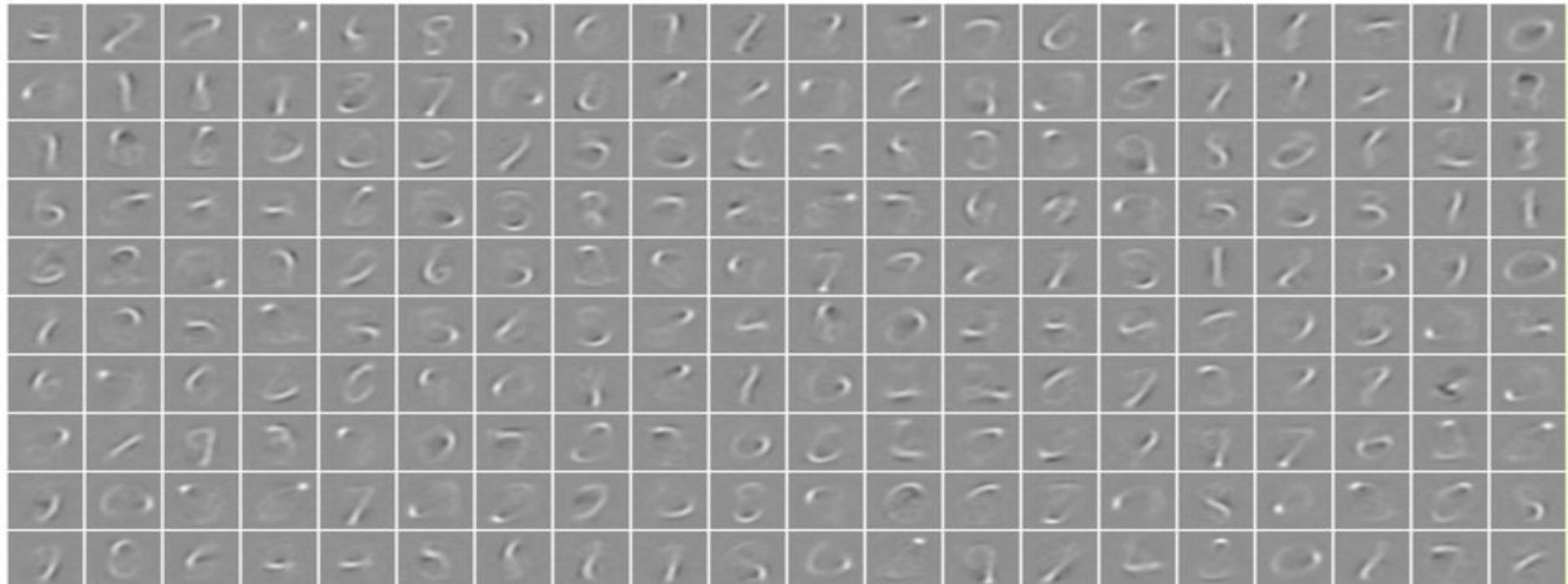
$$F(y) = \min_z E(y, z)$$



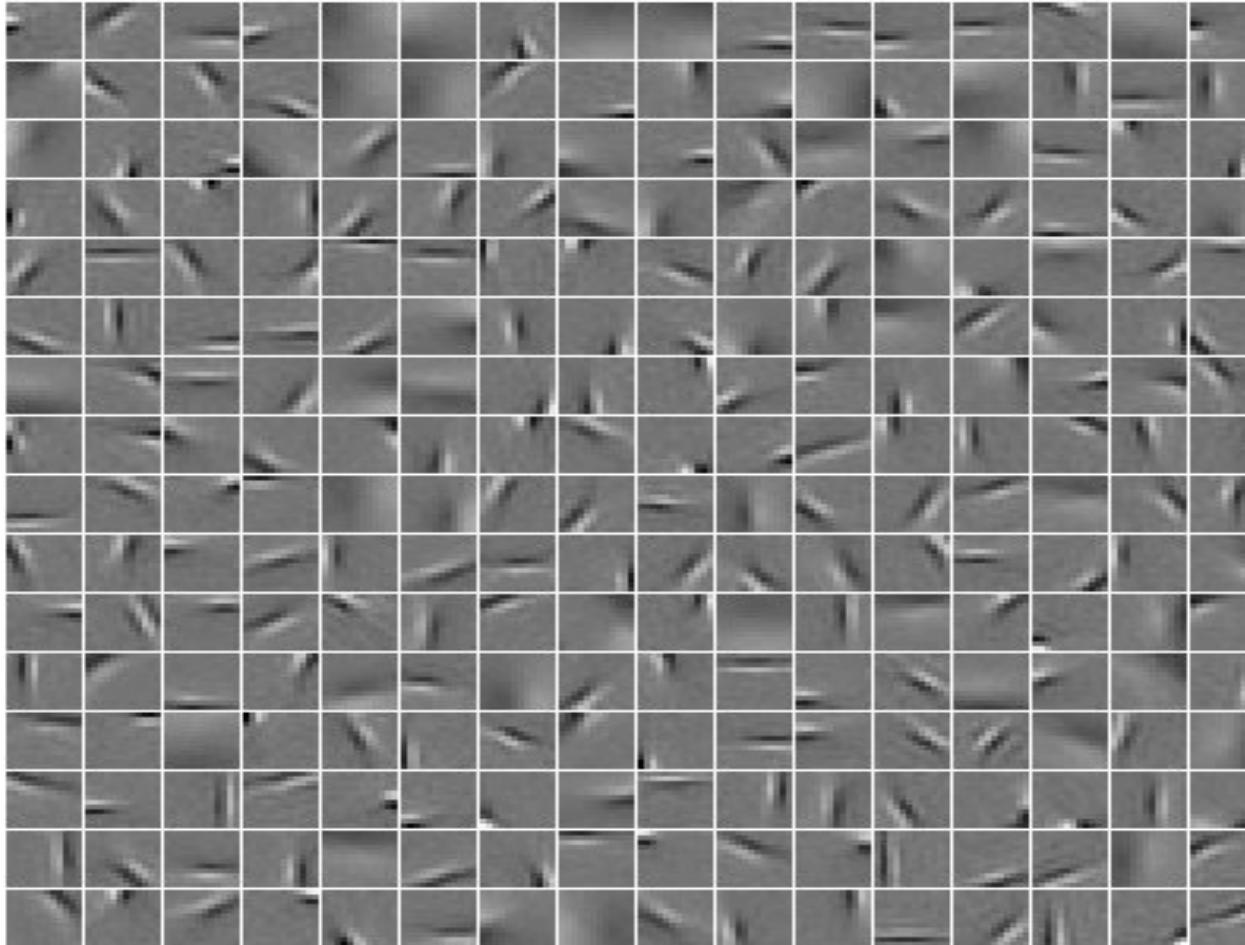
iteration no 0

# Sparse Modeling on handwritten digits (MNIST)

- Basis functions (columns of decoder matrix) are digit parts
- All digits are a linear combination of a small number of these



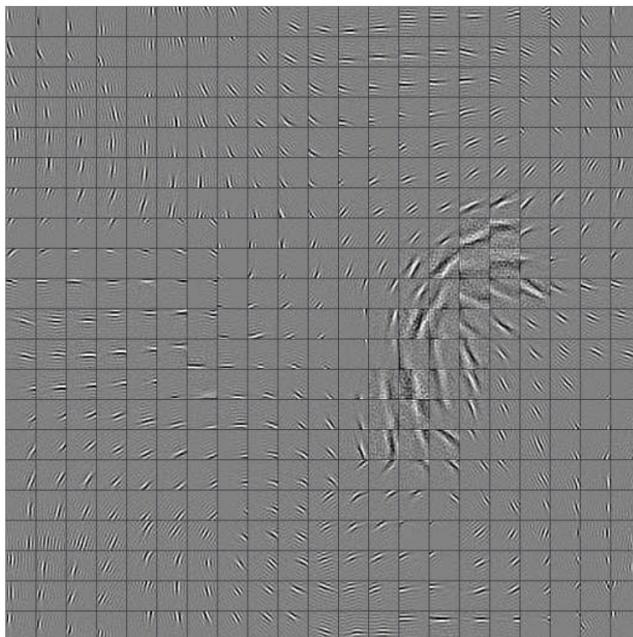
# Learned Features on natural patches: V1-like receptive fields



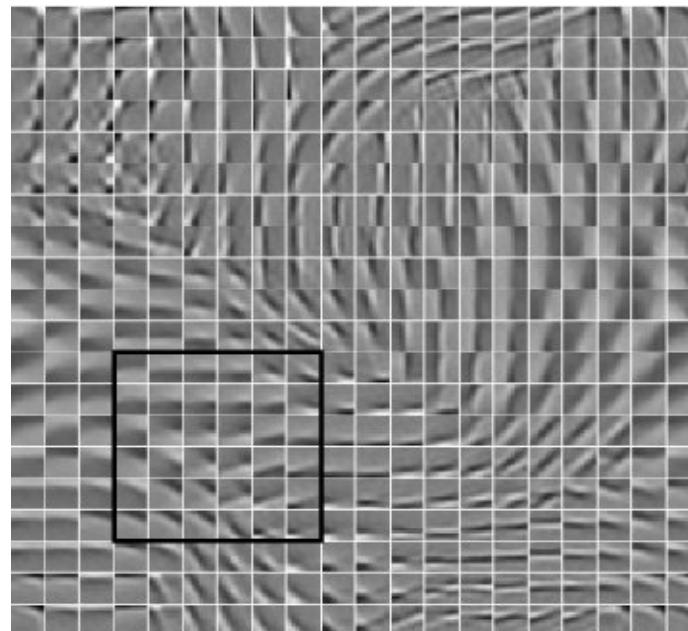
# Learning invariant features

- ▶ **Sparsity over pooling units → group sparsity**
- ▶ [Hyvarinen & Hoyer 2001], [Osindero et al. Neural Comp. 2006], [Kavukcuoglu et al. CVPR 2009], [Gregor & LeCun arXiv:1006.044], [Mairal et al. 2011].

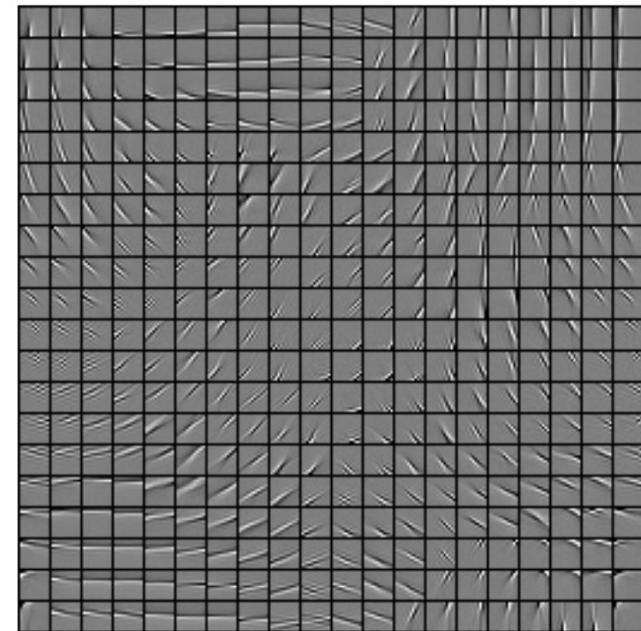
[Osindero 2006]



[Kavukcuoglu 2009]

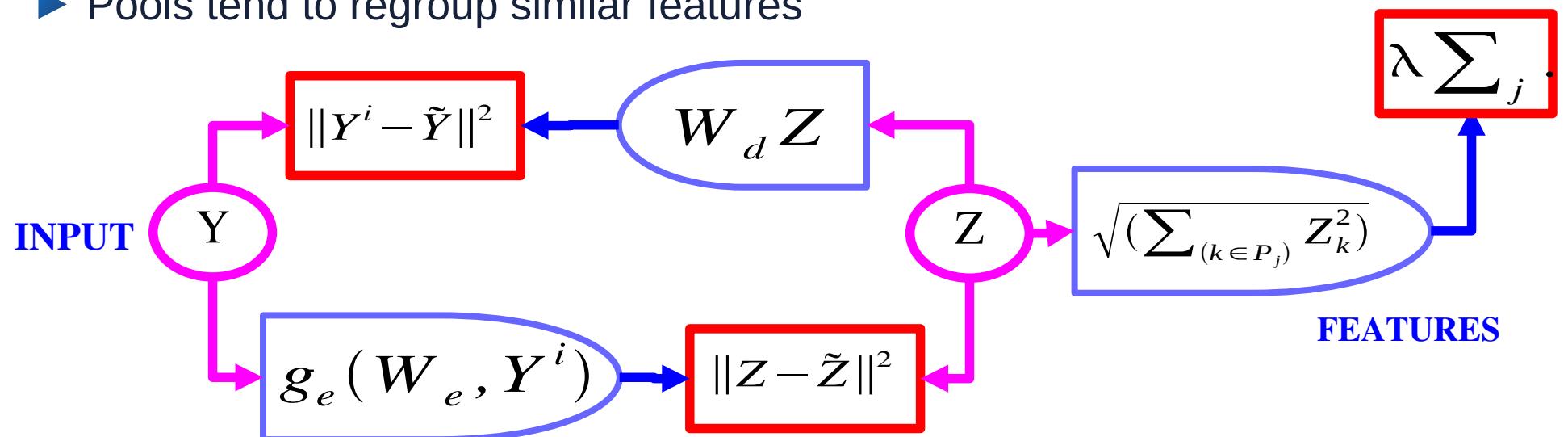


[Mairal 2011]



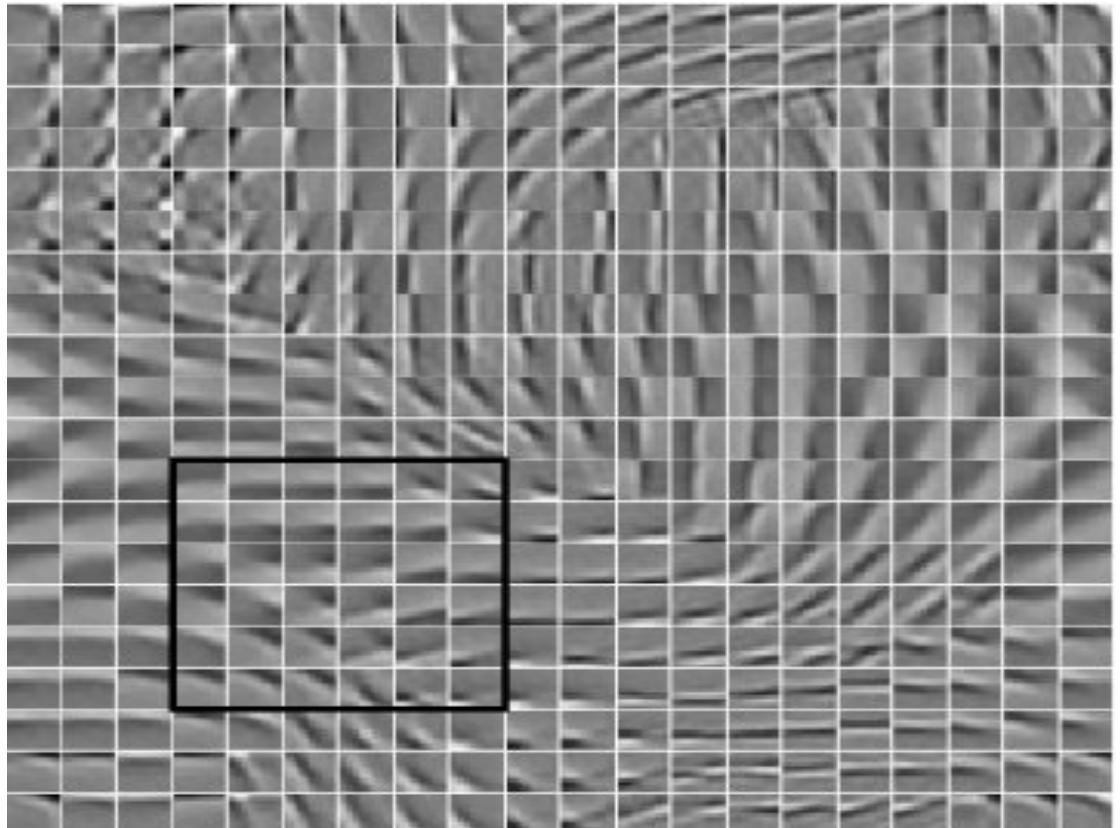
# AE with Group Sparsity [Kavukcuoglu et al. CVPR 2009]

- ▶ Trains a sparse AE to produce invariant features
- ▶ Could we devise a similar method that learns the pooling layer as well?
- ▶ **Group sparsity on pools of features**
  - ▶ Minimum number of pools must be non-zero
  - ▶ Number of features that are on within a pool doesn't matter
  - ▶ Pools tend to regroup similar features



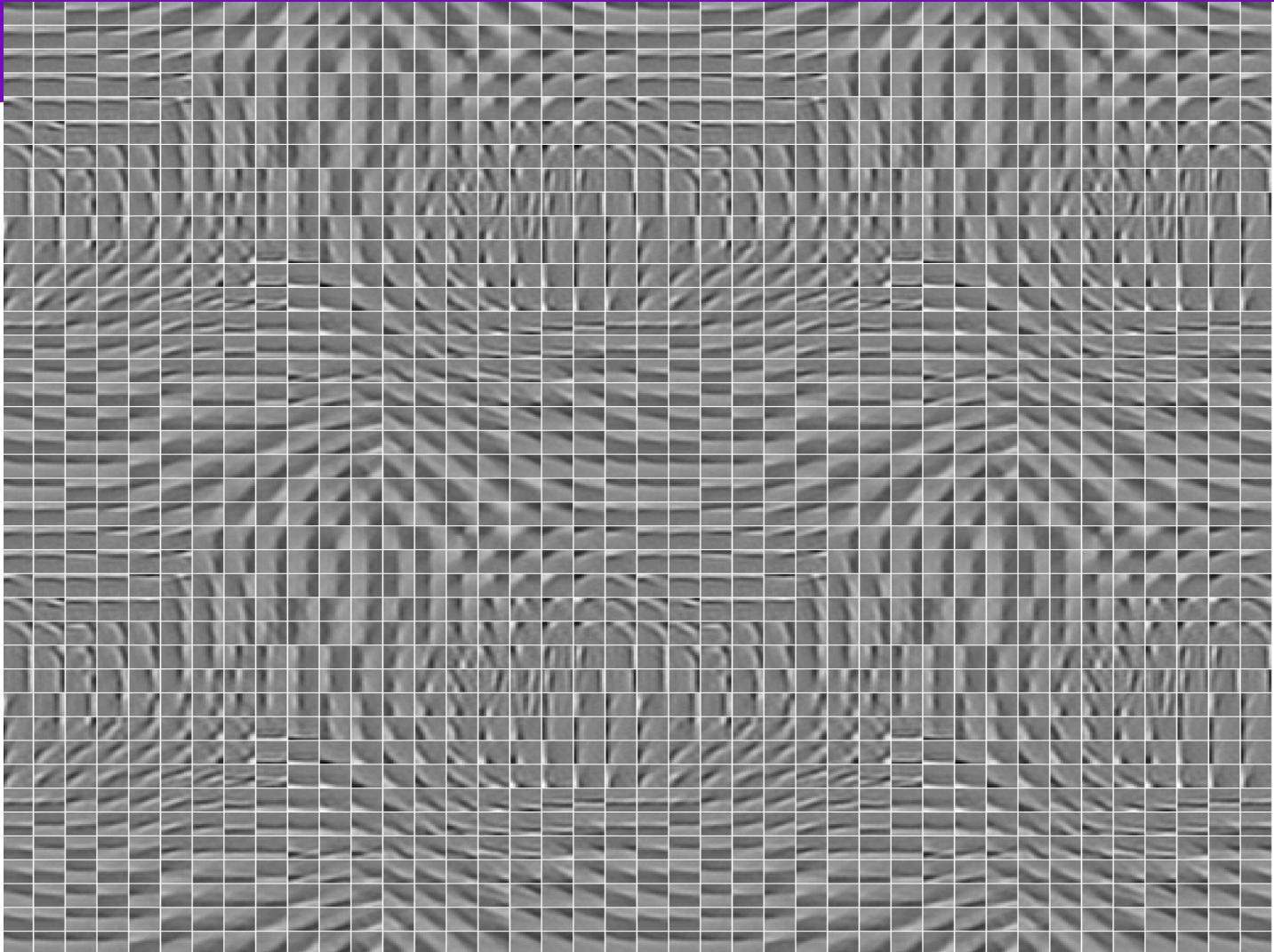
# Pooling over features in a topographic map.

- ▶ The pools are 6x6 blocks of features arranged in a 2D torus
- ▶ While training, the filters arrange themselves spontaneously so that similar filters enter the same pool.
- ▶ The pooling units can be seen as complex cells
- ▶ They are invariant to local transformations of the input
- ▶ For some it's translations, for others rotations, or other transformations.



# Pinwheels!

- ▶ The so-called “pinwheel” pattern is observed in the primary visual cortex.



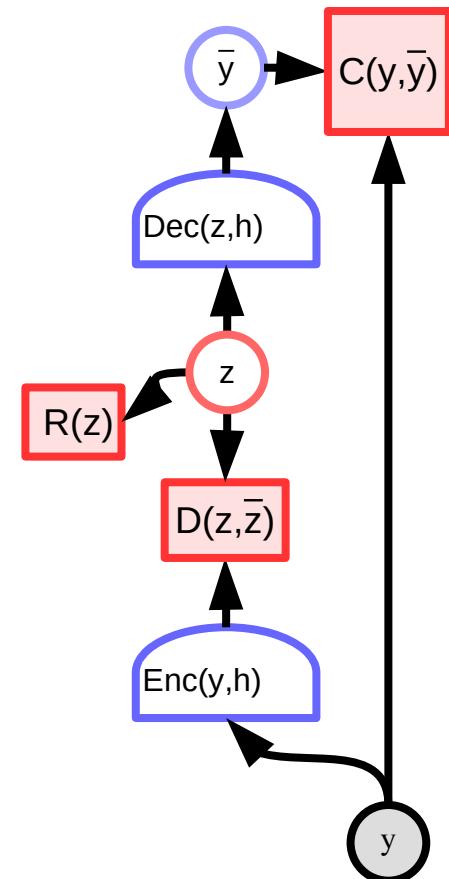
# Amortized Inference

- ▶ Training an encoder to give an approximate solution to the inference optimization problem

- ▶ Regularized Auto-Encoder, Sparse AE, LISTA

$$E(y, z) = C(y, \text{Dec}((z))) + D(z, \text{Enc}(y)) + \lambda R(z)$$

$$F(y) = \min_z E(y, z)$$



# Unconditional LVGEBM with Amortized Inference

## ► Latent-Variable Regularized Auto-Encoders

- Variational AE
- Predictive Sparse Coding

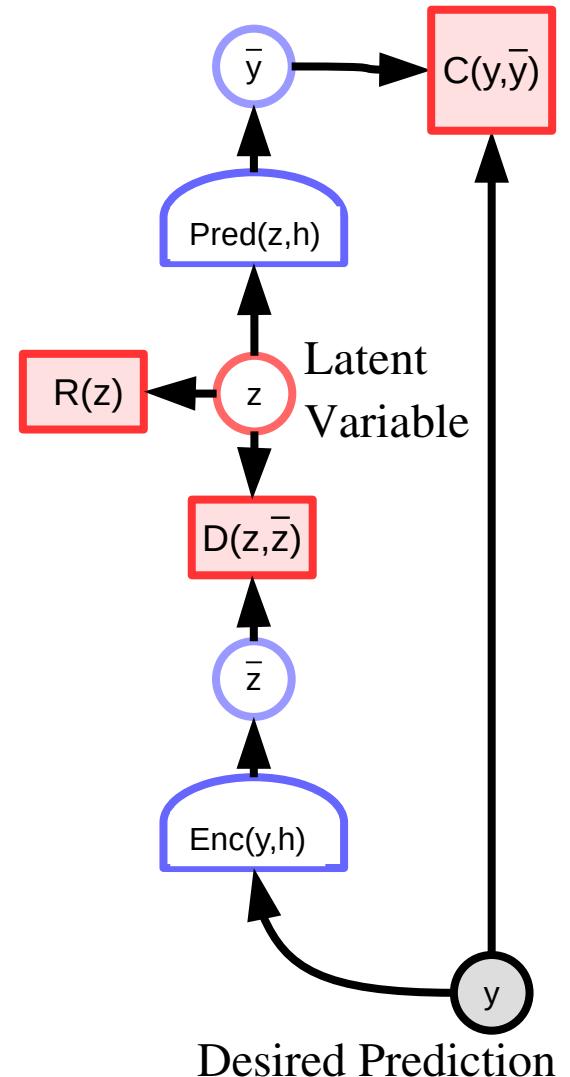
► ....

## ► Energy has three terms:

- Reconstruction  $C(y, \bar{y})$
- latent regularization  $R(z)$
- latent prediction  $D(z, \bar{z})$

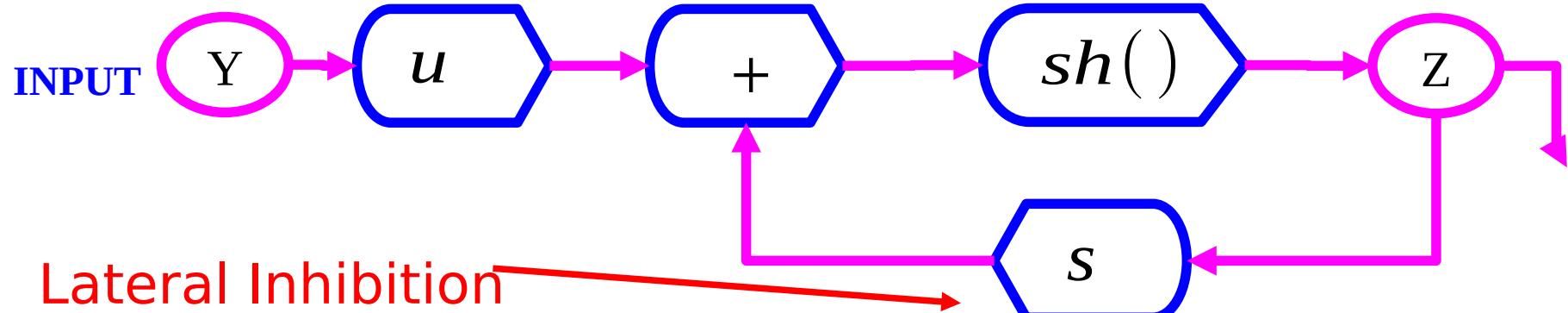
## ► Inference:

- Initialize  $z$  to  $\bar{z}$
- Find  $z$  that minimizes  $E(y, z) = C(y, \bar{y}) + R(z) + D(z, \bar{z})$



# Giving the “right” structure to the encoder

- ISTA/FISTA: iterative algorithm that converges to optimal sparse code



$$z(t+1) = \text{Shrink}_{\alpha\eta} [z(t) - \eta w^t (wz(t) - y)]$$

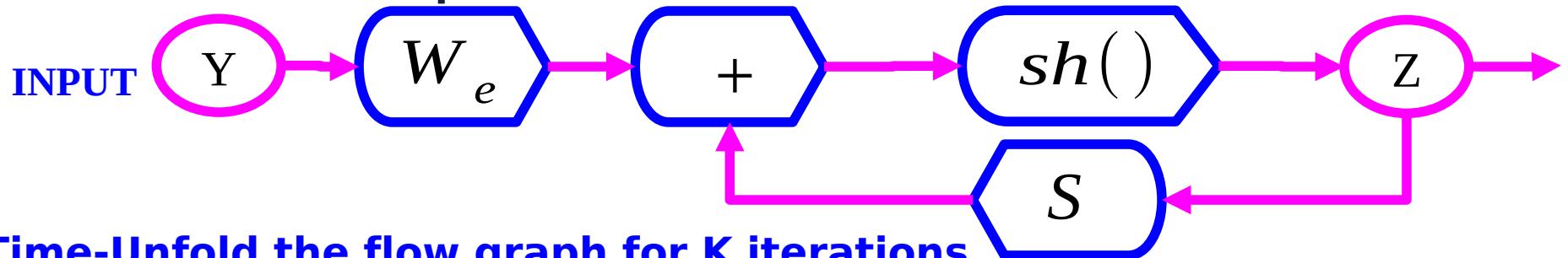
- ISTA/FastISTA reparameterized:

$$z(t+1) = \text{Shrink}_{\alpha\eta} [sz(t) + uy]; \quad u = \eta w; \quad s = I - \eta w^T w$$

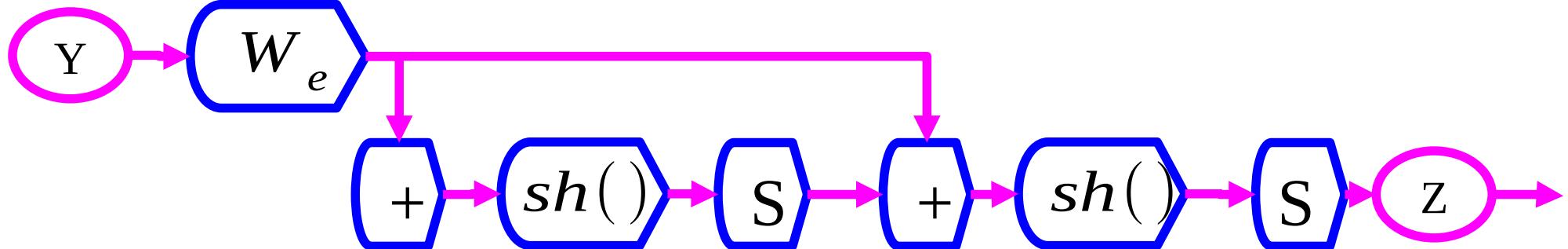
- LISTA (Learned ISTA): learn the  $W$  and  $S$  matrices to get fast solutions

LISTA: Train We and S matrices  
to give a good approximation quickly

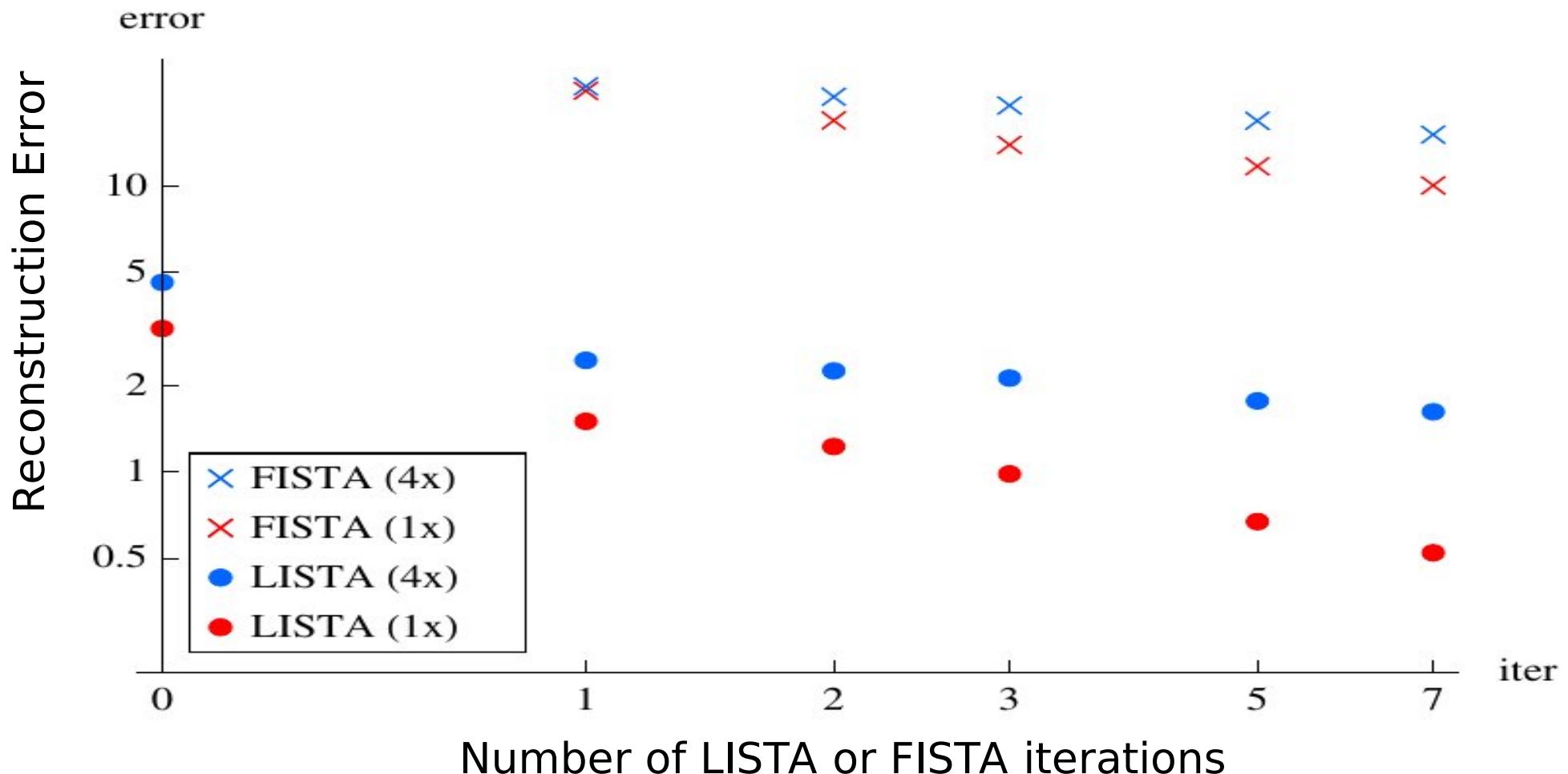
- Think of the FISTA flow graph as a recurrent neural net where We and S are trainable parameters



- Time-Unfold the flow graph for K iterations
- Learn the We and S matrices with “backprop-through-time”
- Get the best approximate solution within K iterations

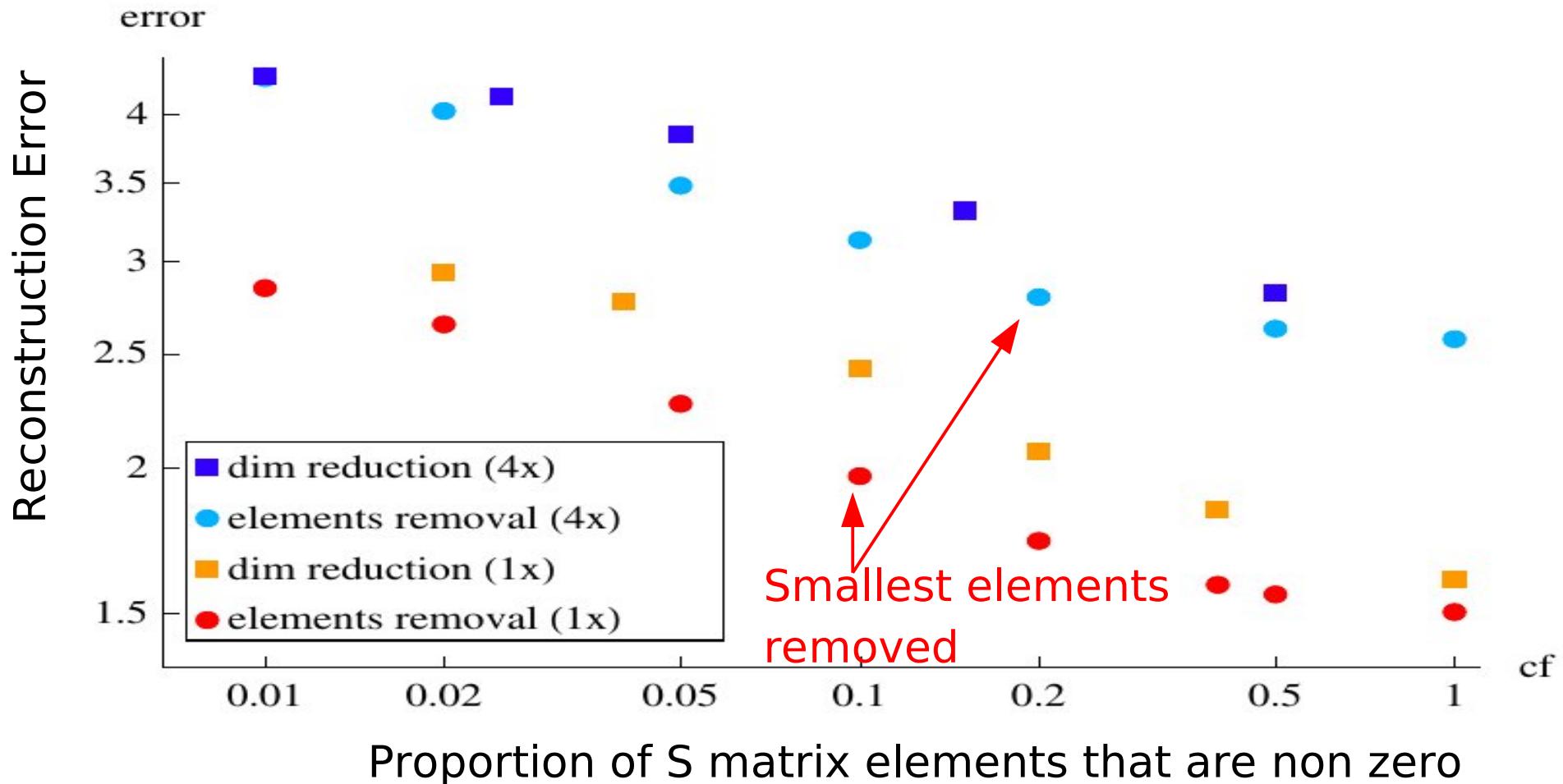


# Learning ISTA (LISTA) vs ISTA/FISTA

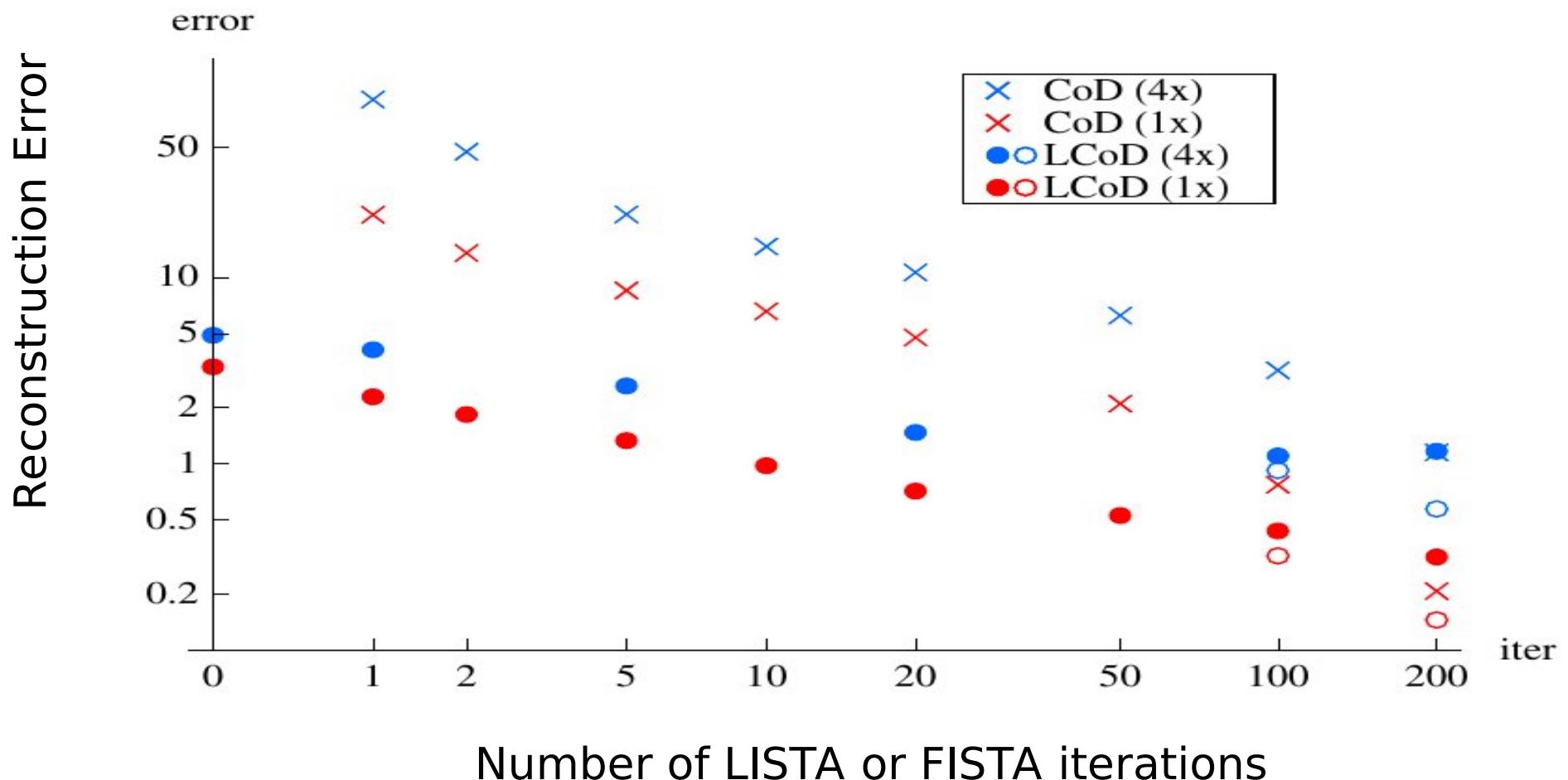


# LISTA with partial mutual inhibition matrix

Y. LeCun

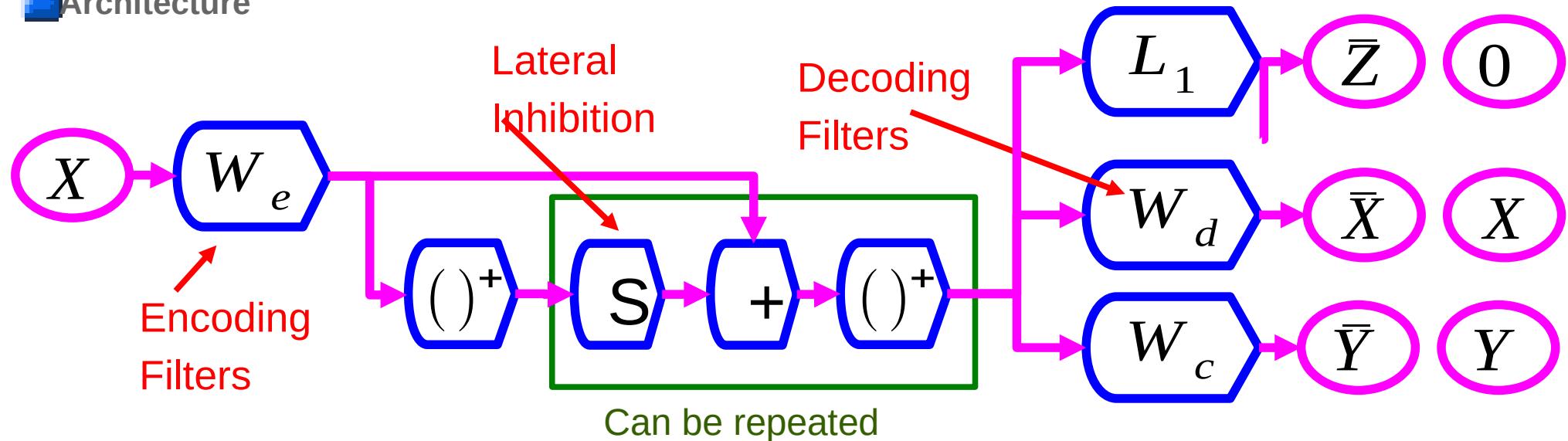


# Learning Coordinate Descent (LcoD): faster than LISTA



# Discriminative Recurrent Sparse Auto-Encoder (DrSAE)

## Architecture



Rectified linear units

Classification loss: cross-entropy

Reconstruction loss: squared error

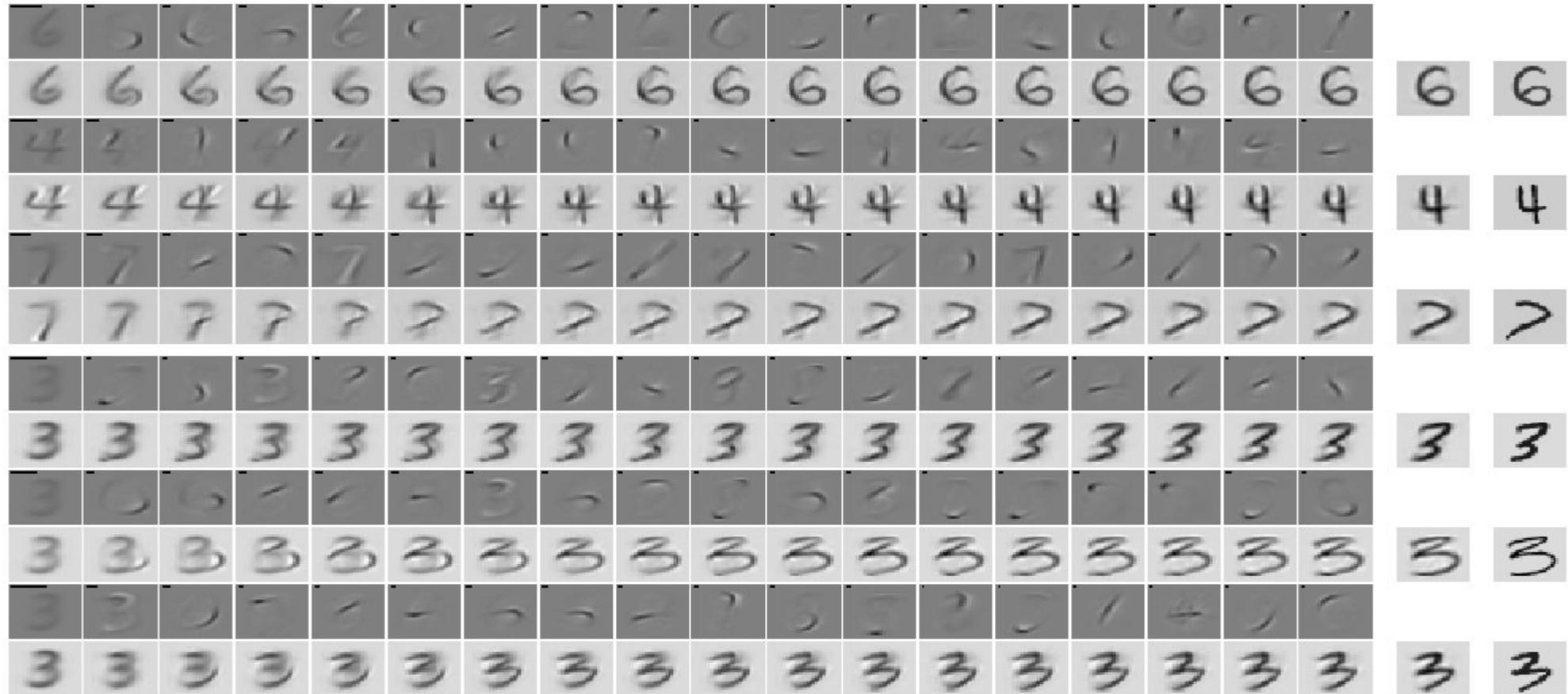
Sparsity penalty: L1 norm of last hidden layer

Rows of  $W_d$  and columns of  $W_e$  constrained in unit sphere

[Rolle & LeCun ICLR 2013]

# DrSAE Discovers manifold structure of handwritten digits

💡 Image = prototype + sparse sum of “parts” (to move around the



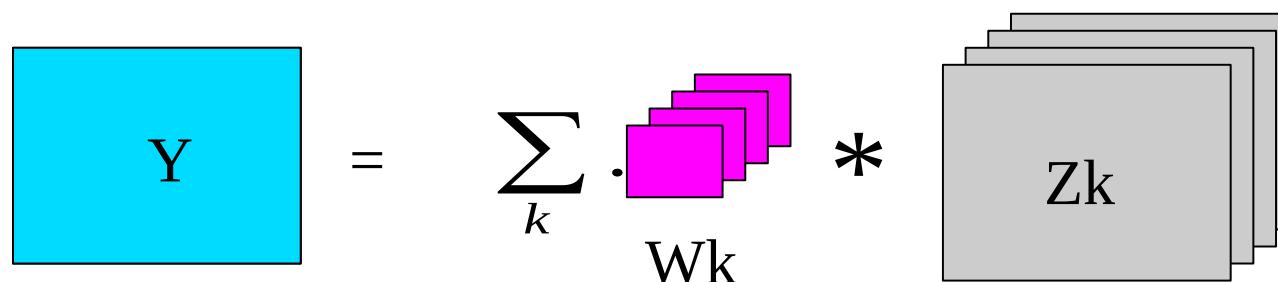
# Convolutional Sparse Coding

- Replace the dot products with dictionary element by convolutions.

- Input  $Y$  is a full image
- Each code component  $Z_k$  is a feature map (an image)
- Each dictionary element is a convolution kernel

- Regular sparse coding**  $E(Y, Z) = \sum_k |Y - W_k Z_k|^2 + \alpha \sum_k |Z_k|$

- Convolutional S.C.**  $E(Y, Z) = \sum_k |Y - W_k * Z_k|^2 + \alpha \sum_k |Z_k|$



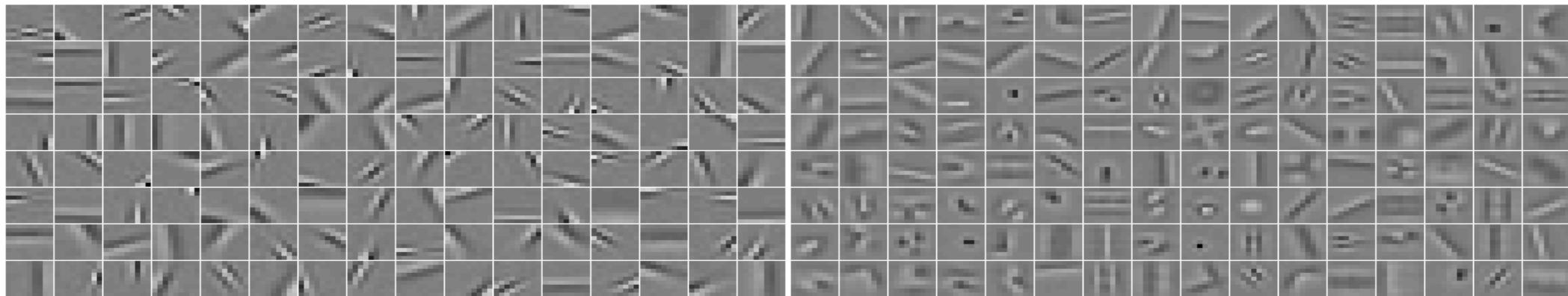
“deconvolutional networks” [Zeiler, Taylor, Fergus CVPR 2010]

# Convolutional Sparse AE: Encoder with a soft sh() Function

## Convolutional Formulation

► Extend sparse coding from **PATCH** to **IMAGE**

$$\mathcal{L}(x, z, \mathcal{D}) = \frac{1}{2} \|x - \sum_{k=1}^K \mathcal{D}_k * z_k\|_2^2 + \sum_{k=1}^K \|z_k - f(W^k * x)\|_2^2 + |z|_1$$



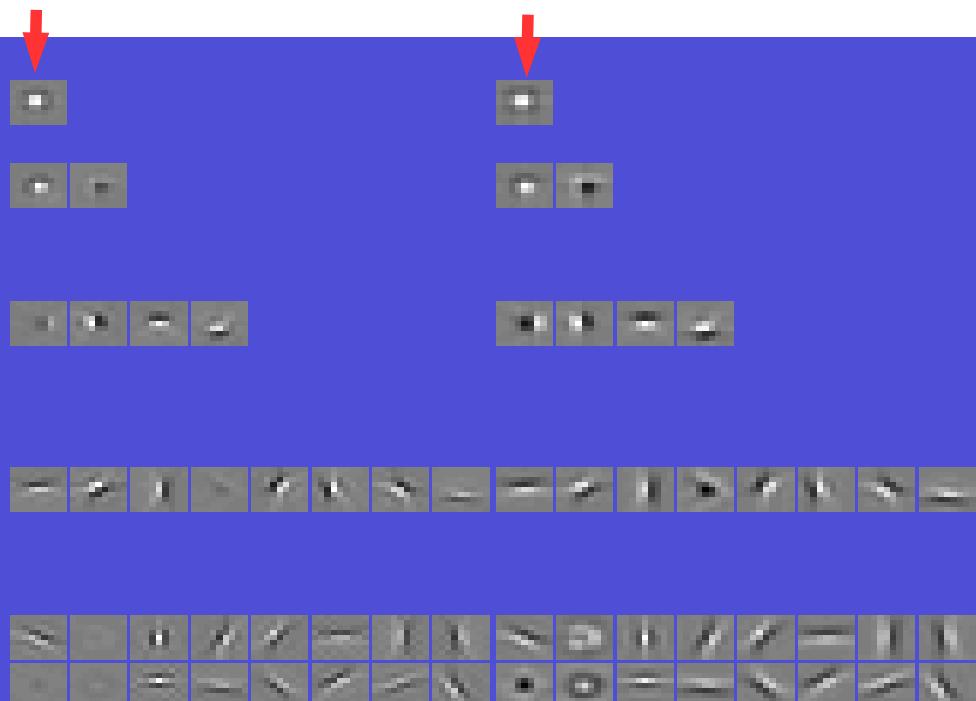
► **PATCH** based learning

► **CONVOLUTIONAL** learning

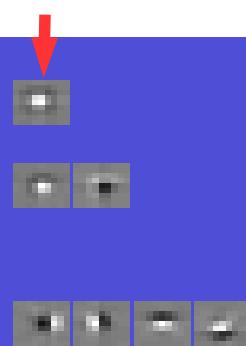
# Convolutional Sparse Auto-Encoder on Natural Images

- ▶ Filters and Basis Functions obtained. Linear decoder (conv)
  - ▶ with 1, 2, 4, 8, 16, 32, and 64 filters [Kavukcuoglu NIPS 2010]

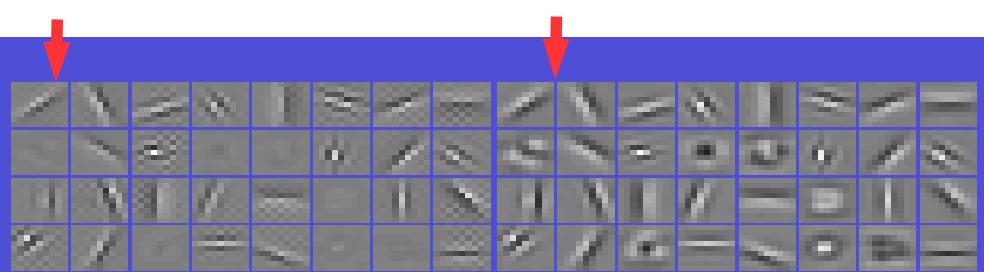
Encoder Filters



Decoder Filters



Encoder Filters



Decoder Filters

