# یادگیری عمیق

## جلسه ۱۸

# ملاحظاتی در آموزش شبکه‌های عصبی عمیق

# Considerations on Deep Neural Networks Training


کاظم فولادی قلعه

دانشکده مهندسی، دانشکدگان فارابی

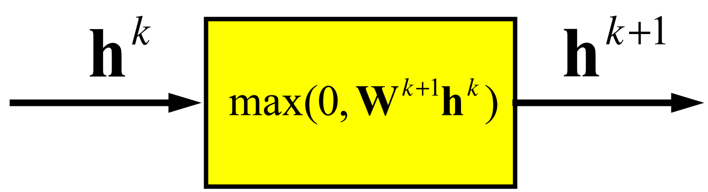
دانشگاه تهران

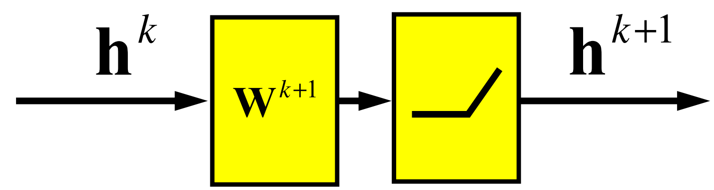

http://courses.fouladi.ir/deep

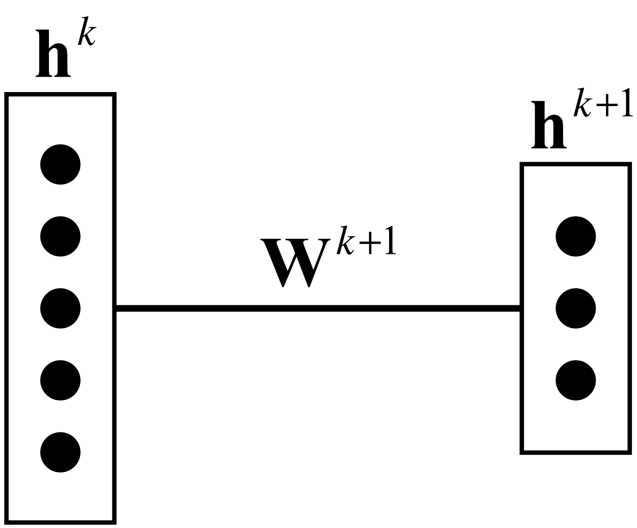
# نمادگذاری‌های معادل

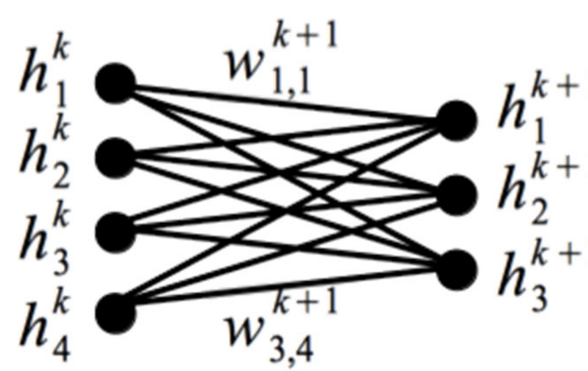
## نمایش یک دو لایه و اتصالات آنها

## EQUIVALENT REPRESENTATIONS



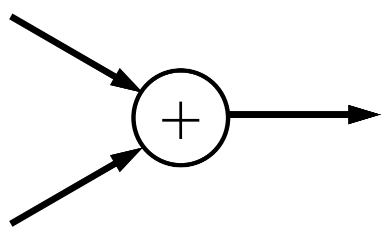
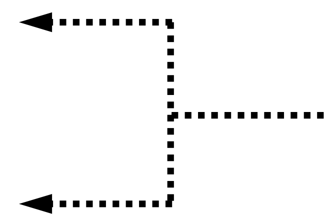
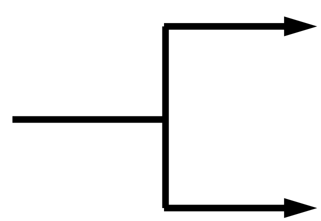
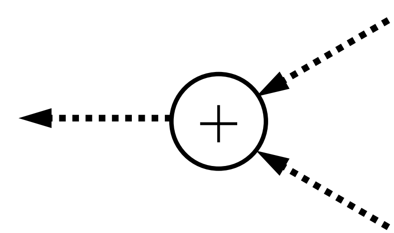



# دوگانی پیش‌انتشار و پس‌انتشار

## Duality of FPROP & BPROP

| پیش‌انتشار *Forward Propagation* **(FPROP)** | | پس‌انتشار *Back Propagation* **(BPROP)** | |
|---|---|---|---|
| جمع *SUM* |  |  | کپی *COPY* |
| کپی *COPY* |  |  | جمع *SUM* |

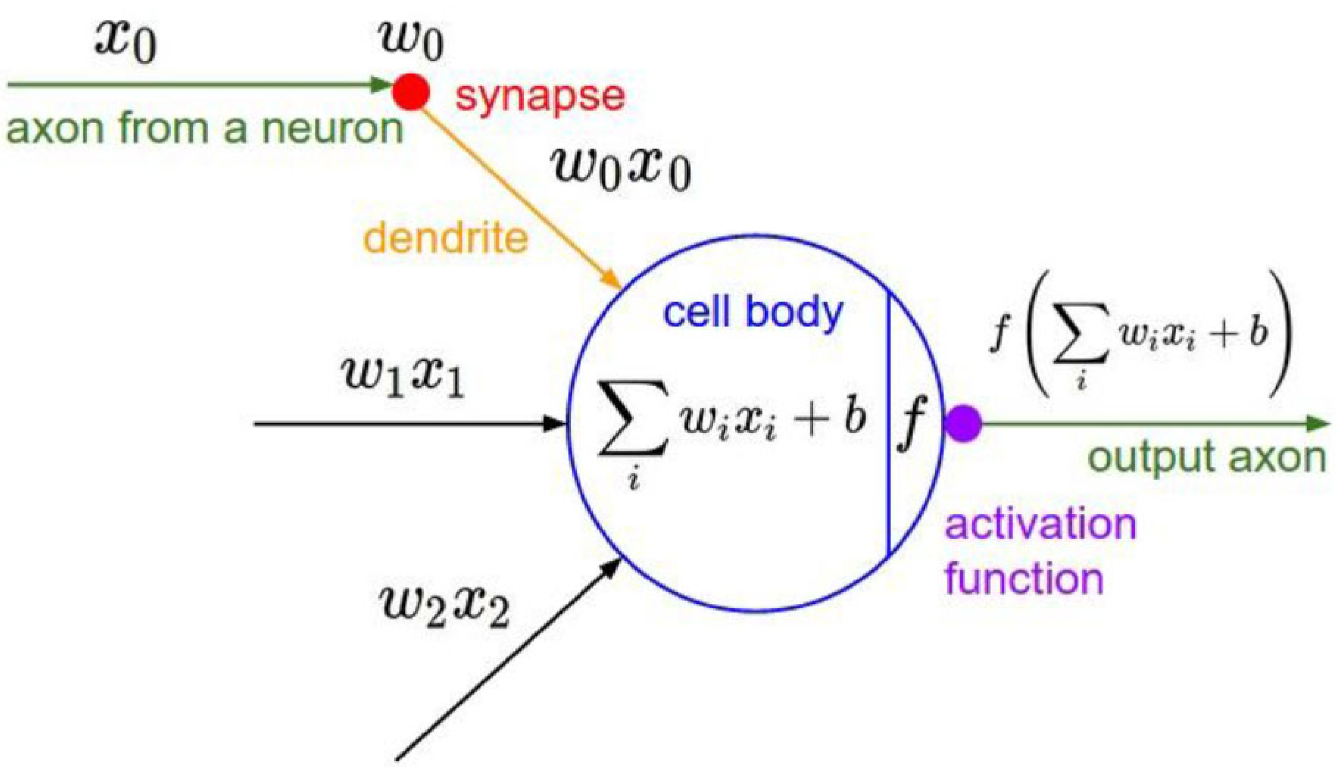
ملاحظاتی در
آموزش شبکه‌های عصبی عمیق

# ۱

# توابع فعالیت

# ساختار نرون

## NEURON STRUCTURE






Ref: http://cs231n.stanford.edu/

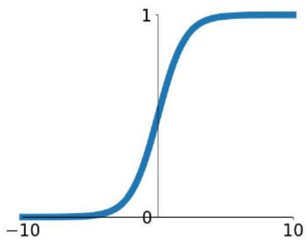
# توابع فعالیت

(توابع انتقال)

## Activation (Transfer) Functions

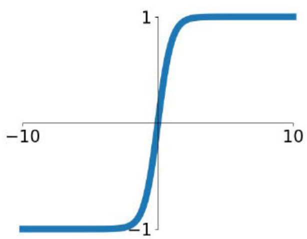
**Sigmoid**

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



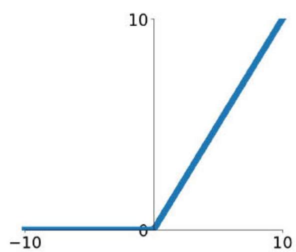

**tanh**

$\tanh(x)$



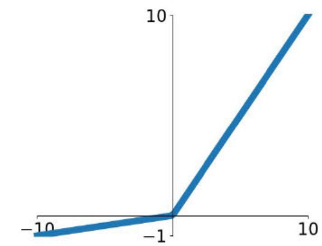

**ReLU**

$\max(0, x)$




**Leaky ReLU**

$\max(0.1x, x)$

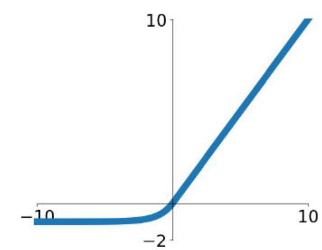



**Maxout**

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

**ELU**

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





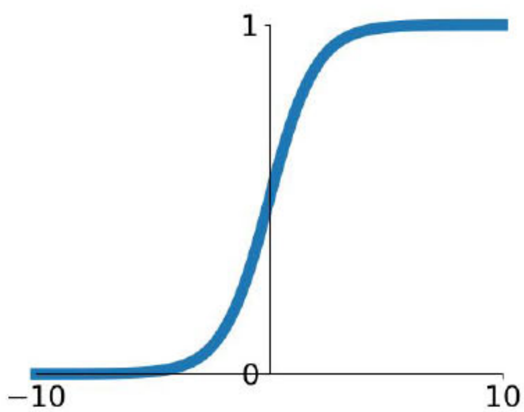

Ref: http://cs231n.stanford.edu/

# توابع فعالیت

## تابع سیگموئید

## Activation Functions

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



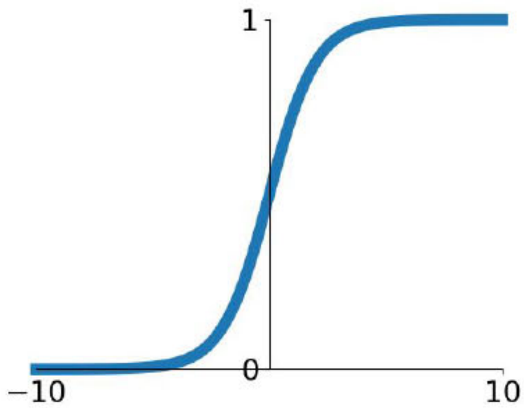

**Sigmoid**

- Squashes numbers to range [0,1]
- Historically popular since they have nice interpretation as a saturating “firing rate” of a neuron

# توابع فعالیت

تابع سیگموئید : مشکلات

## Activation Functions




**Sigmoid**

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

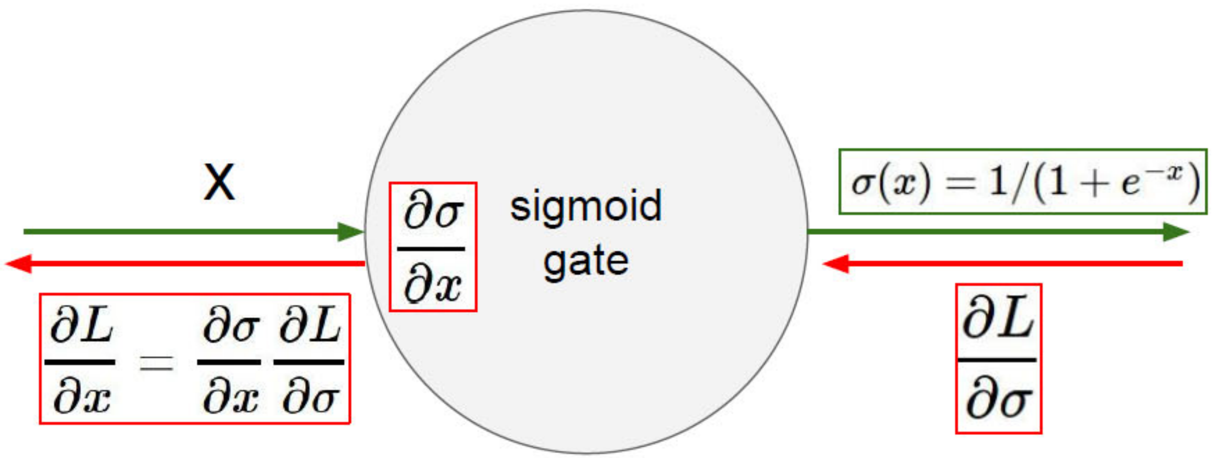
- Squashes numbers to range [0,1]
- Historically popular since they have nice interpretation as a saturating "firing rate" of a neuron

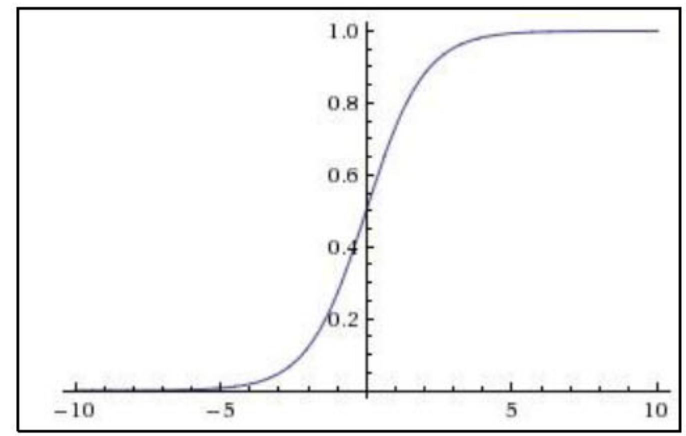
3 problems:

1. Saturated neurons "kill" the gradients

# توابع فعالیت

## تابع سیگموئید : مشکلات



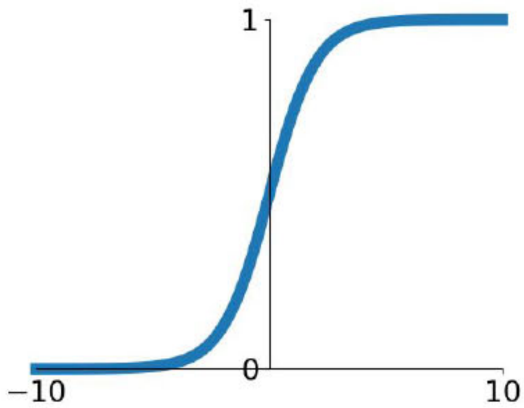

What happens when x = -10?
What happens when x = 0?
What happens when x = 10?

# توابع فعالیت

تابع سیگموئید : مشکلات

## Activation Functions

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




**Sigmoid**

- Squashes numbers to range [0,1]
- Historically popular since they have nice interpretation as a saturating “firing rate” of a neuron

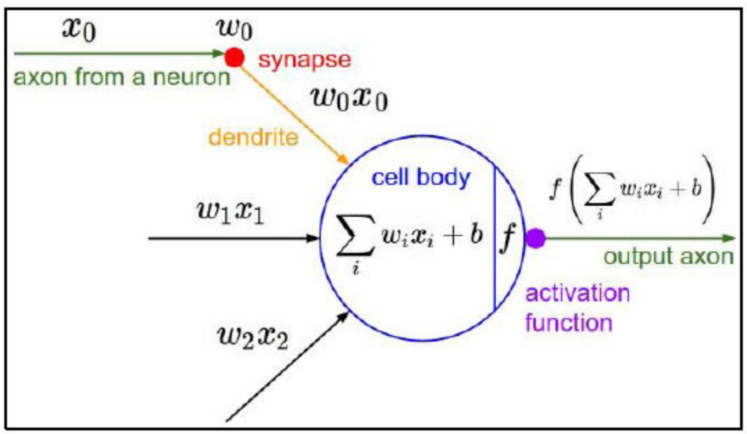
3 problems:

1. Saturated neurons “kill” the gradients
2. Sigmoid outputs are not zero-centered

# توابع فعالیت

تابع سیگموئید: مشکلات: اگر ورودی‌های نرون همیشه مثبت باشد …

Consider what happens when the input to a neuron (x) is always positive:




$$f\left(\sum_i w_i x_i + b\right)$$

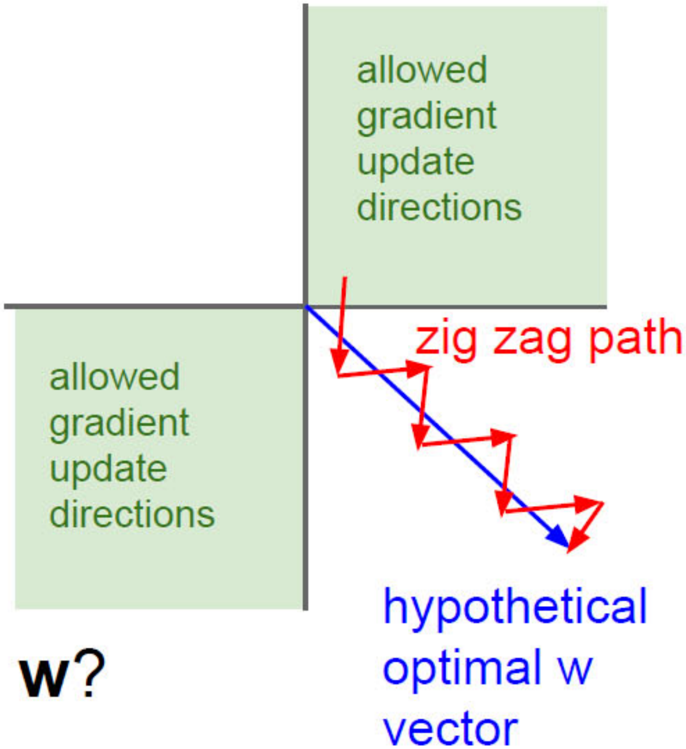
What can we say about the gradients on **w**?

# توابع فعالیت

تابع سیگموئید: مشکلات: اگر ورودی‌های نرون همیشه مثبت باشد ...

Consider what happens when the input to a neuron is always positive...

$$f\left(\sum_i w_i x_i + b\right)$$




What can we say about the gradients on **w**?

Always all positive or all negative :(

(this is also why you want zero-mean data!)



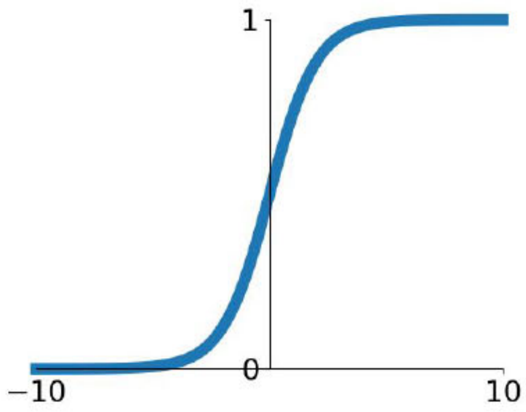
Ref: http://cs231n.stanford.edu/

# توابع فعالیت

تابع سیگموئید : مشکلات

## Activation Functions

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




**Sigmoid**

- Squashes numbers to range [0,1]
- Historically popular since they have nice interpretation as a saturating "firing rate" of a neuron

3 problems:

1. Saturated neurons "kill" the gradients
2. Sigmoid outputs are not zero-centered
3. exp() is a bit compute expensive

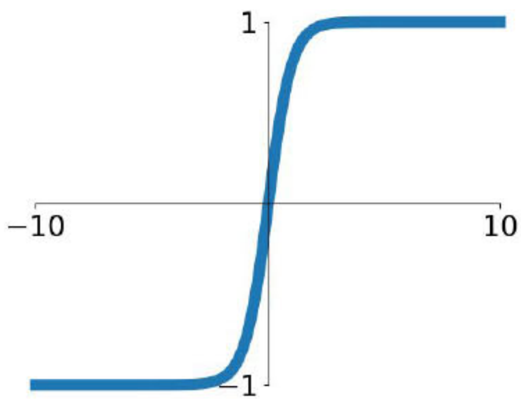


Ref: http://cs231n.stanford.edu/

# توابع فعالیت

تابع تانژانت هایپربولیک

# Activation Functions




**tanh(x)**

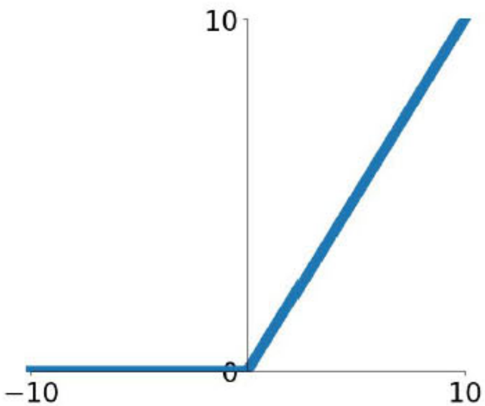
- Squashes numbers to range [-1,1]
- zero centered (nice)
- still kills gradients when saturated :(

[LeCun et al., 1991]

# توابع فعالیت

## تابع واحد خطی یکسوشده

## Activation Functions




**ReLU**
(Rectified Linear Unit)

- Computes **f(x) = max(0,x)**

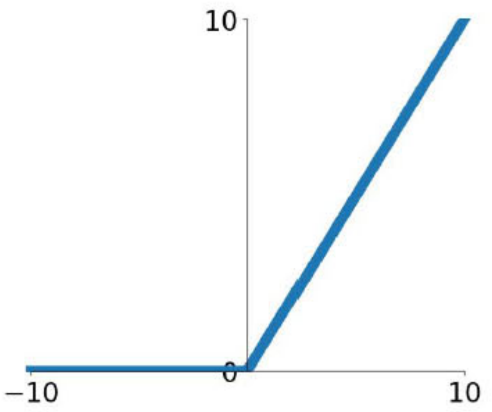
- Does not saturate (in +region)
- Very computationally efficient
- Converges much faster than sigmoid/tanh in practice (e.g. 6x)
- Actually more biologically plausible than sigmoid

[Krizhevsky et al., 2012]

# توابع فعالیت

## تابع واحد خطی یکسوشده : مشکلات

# Activation Functions




**ReLU**
(Rectified Linear Unit)

- Computes **f(x) = max(0,x)**

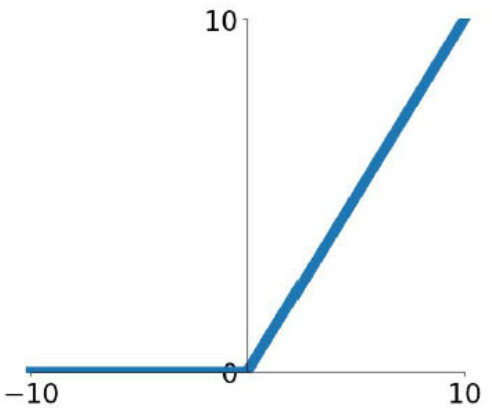
- Does not saturate (in +region)
- Very computationally efficient
- Converges much faster than sigmoid/tanh in practice (e.g. 6x)
- Actually more biologically plausible than sigmoid

- Not zero-centered output

## توابع فعالیت

تابع واحد خطی یکسوشده : مشکلات

# Activation Functions




**ReLU**
(Rectified Linear Unit)

- Computes **f(x) = max(0,x)**

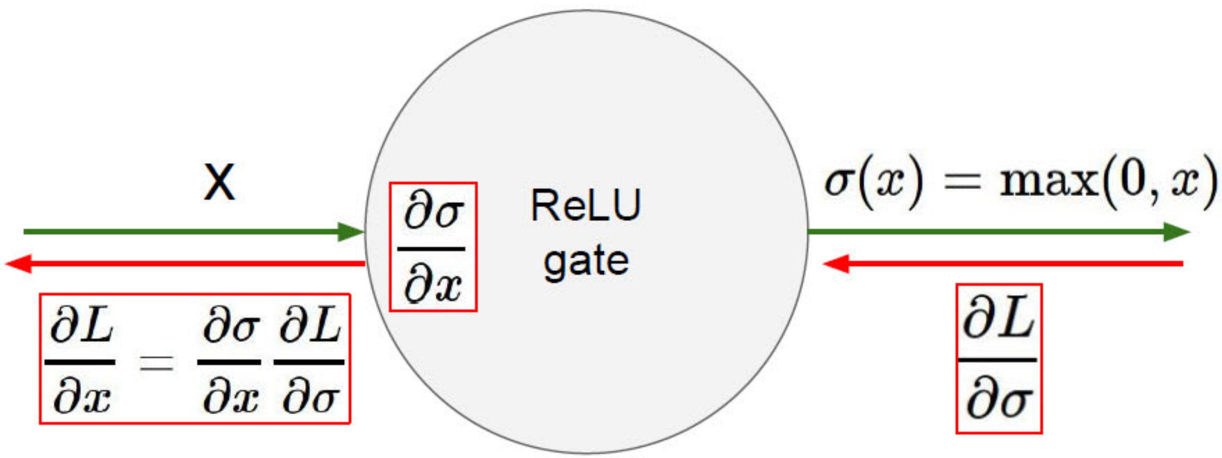
- Does not saturate (in +region)
- Very computationally efficient
- Converges much faster than sigmoid/tanh in practice (e.g. 6x)
- Actually more biologically plausible than sigmoid

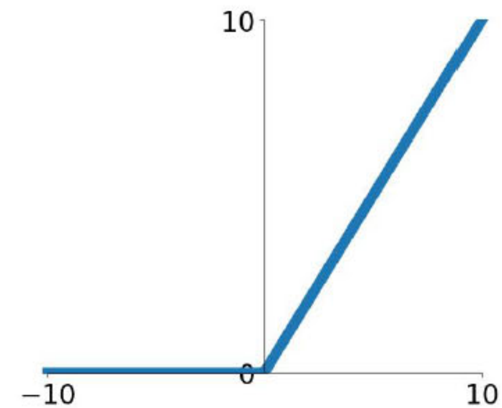
- Not zero-centered output
- An annoyance:

hint: what is the gradient when x < 0?

# توابع فعالیت

## تابع واحد خطی یکسوشده : مشکلات

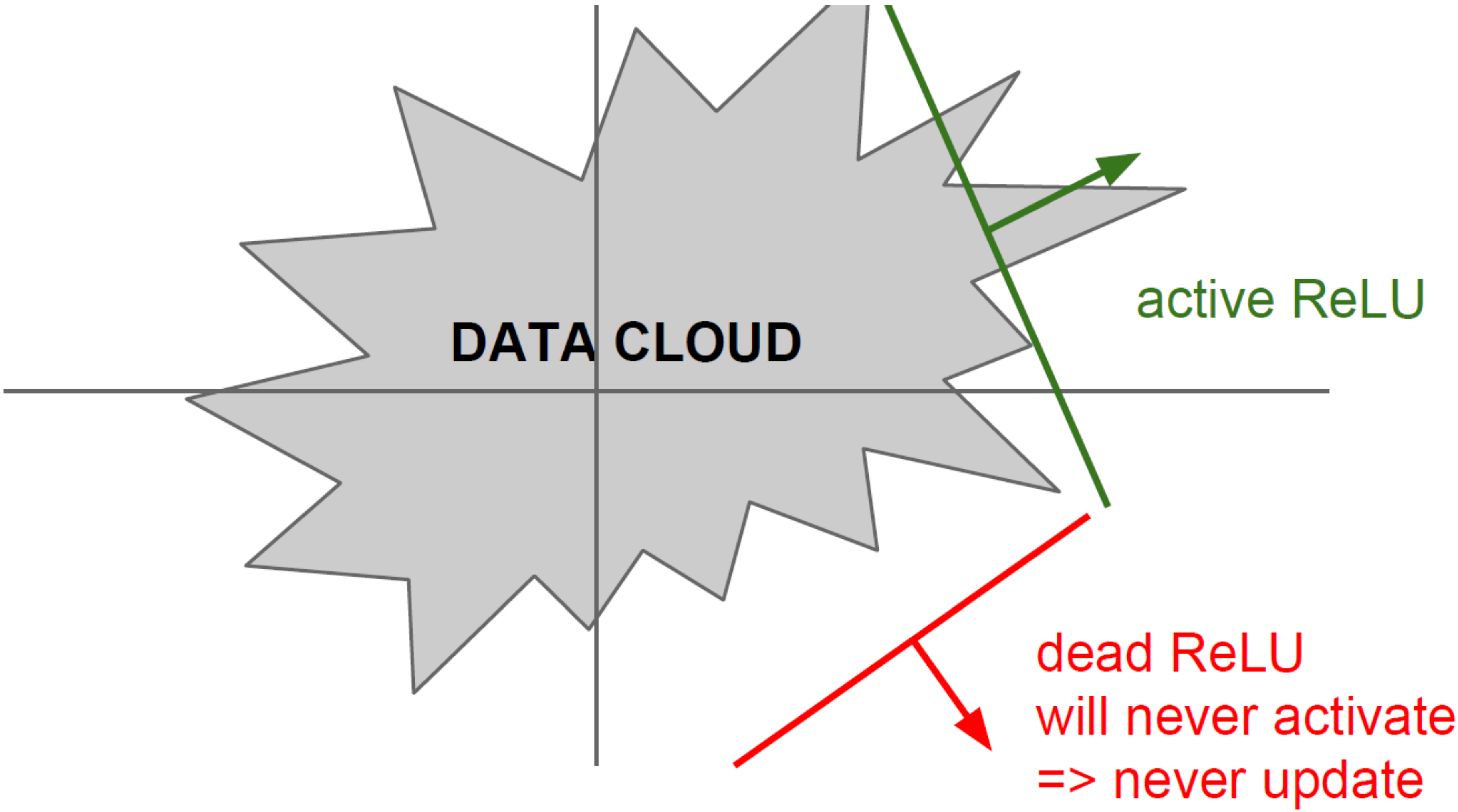



What happens when x = -10?
What happens when x = 0?
What happens when x = 10?

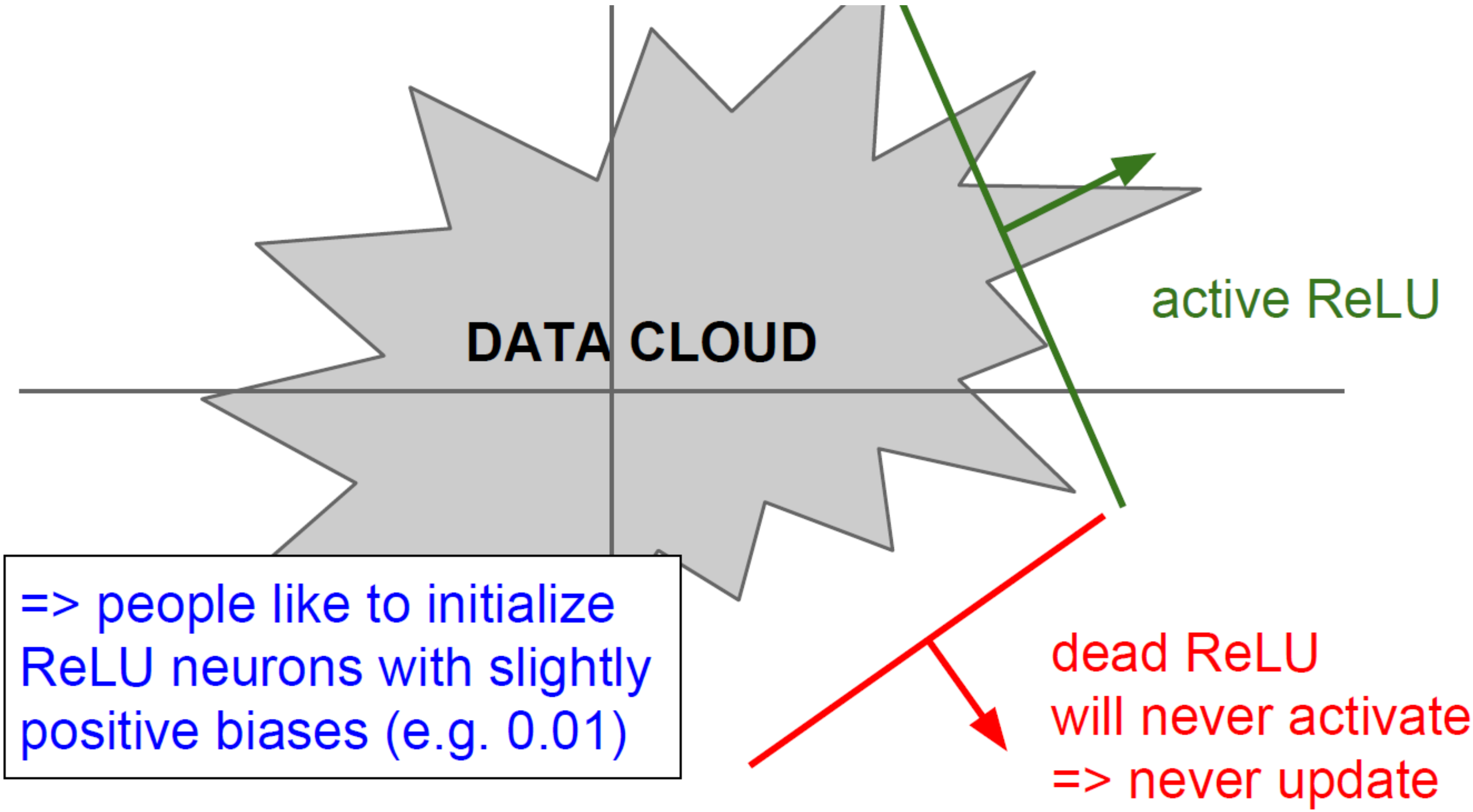
# توابع فعالیت

تابع واحد خطی یکسوشده : مشکلات



# توابع فعالیت

تابع واحد خطی یکسوشده : مشکلات



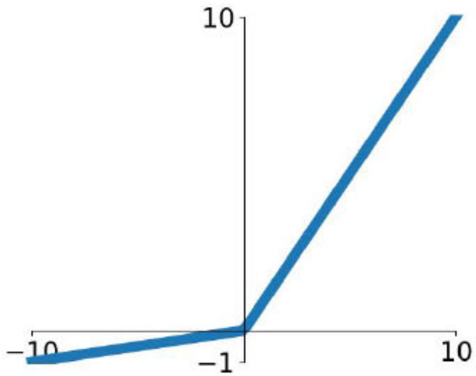

=> people like to initialize ReLU neurons with slightly positive biases (e.g. 0.01)

# توابع فعالیت

تابع واحد خطی یکسوشده‌ی نشتی‌دار

## Activation Functions

[Mass et al., 2013]
[He et al., 2015]




- Does not saturate
- Computationally efficient
- Converges much faster than sigmoid/tanh in practice! (e.g. 6x)
- **will not “die”.**

**Leaky ReLU**

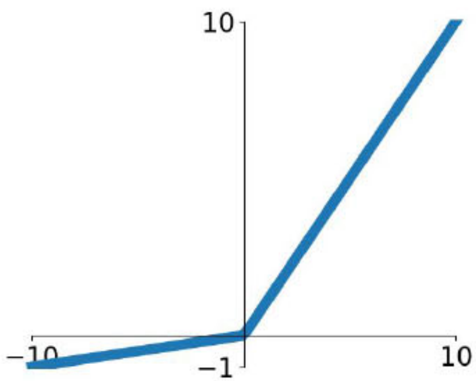
$$f(x) = \max(0.01x, x)$$

# توابع فعالیت

تابع واحد خطی یکسوشده‌ی پارامتری

## Activation Functions

[Mass et al., 2013]
[He et al., 2015]




- Does not saturate
- Computationally efficient
- Converges much faster than sigmoid/tanh in practice! (e.g. 6x)
- **will not "die".**

**Leaky ReLU**

$$f(x) = \max(0.01x, x)$$

**Parametric Rectifier (PReLU)**

$$f(x) = \max(\alpha x, x)$$

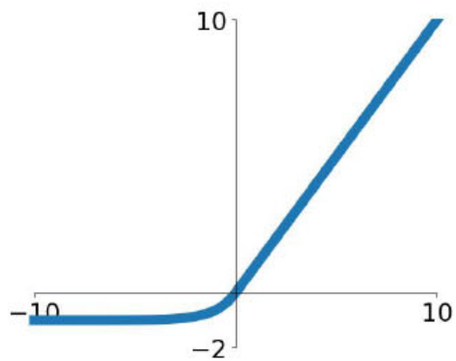
backprop into \alpha (parameter)

تابع واحد خطی نمایی

# Activation Functions

[Clevert et al., 2015]

## Exponential Linear Units (ELU)




- All benefits of ReLU
- Closer to zero mean outputs
- Negative saturation regime compared with Leaky ReLU adds some robustness to noise

$$f(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha\,(\exp(x) - 1) & \text{if } x \le 0 \end{cases}$$

- Computation requires exp()

## Maxout "Neuron"

[Goodfellow et al., 2013]

- Does not have the basic form of dot product -> nonlinearity
- Generalizes ReLU and Leaky ReLU
- Linear Regime! Does not saturate! Does not die!

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

Problem: doubles the number of parameters/neuron :(

# توابع فعالیت

توصیه‌هایی برای انتخاب تابع فعالیت

- Use ReLU. Be careful with your learning rates
- Try out Leaky ReLU / Maxout / ELU
- Try out tanh but don’t expect much
- Don’t use sigmoid

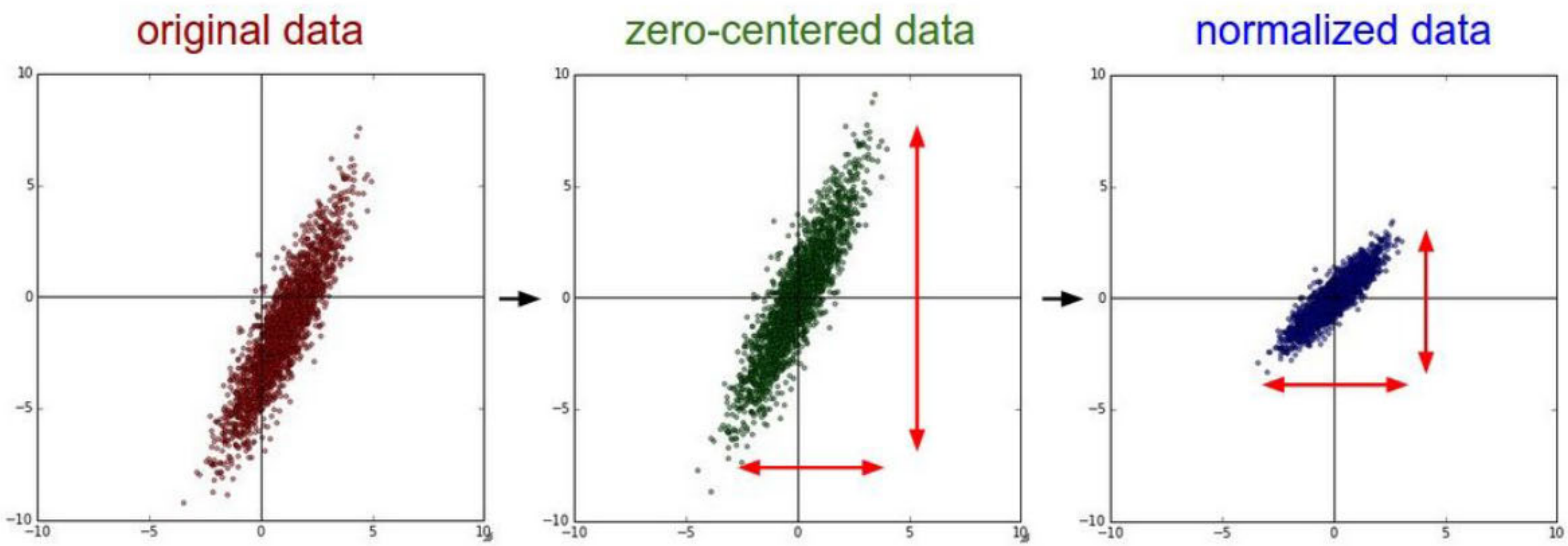

Ref: http://cs231n.stanford.edu/

ملاحظاتی در
آموزش شبکه‌های عصبی عمیق

# ۲

# پیش‌پردازش داده‌ها

# پیش‌پردازش داده‌ها




```
X -= np.mean(X, axis = 0)
```

```
X /= np.std(X, axis = 0)
```

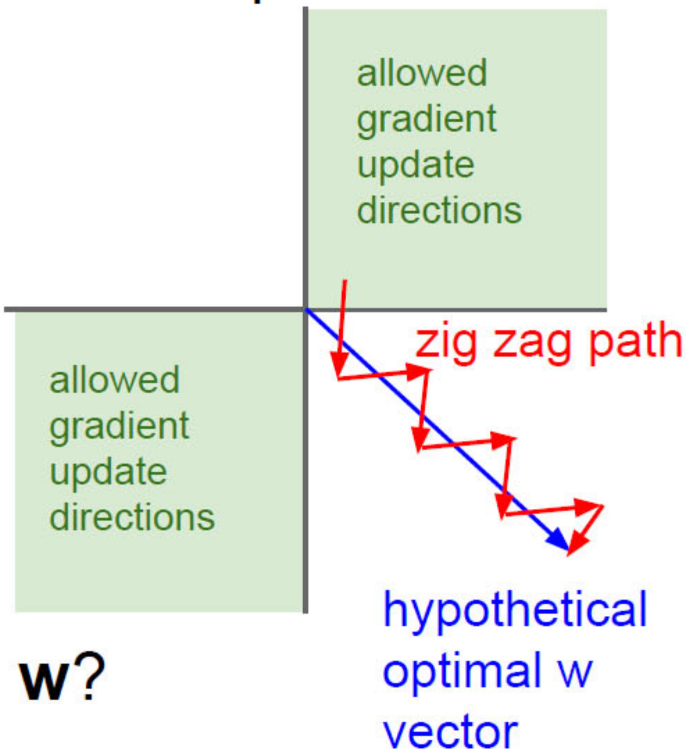
(Assume X [NxD] is data matrix, each example in a row)

# پیش‌پردازش داده‌ها

## ضرروت نرمال‌سازی/ استانداردسازی داده‌ها

Remember: Consider what happens when the input to a neuron is always positive...

$$f\left(\sum_i w_i x_i + b\right)$$




What can we say about the gradients on **w**?

Always all positive or all negative :(

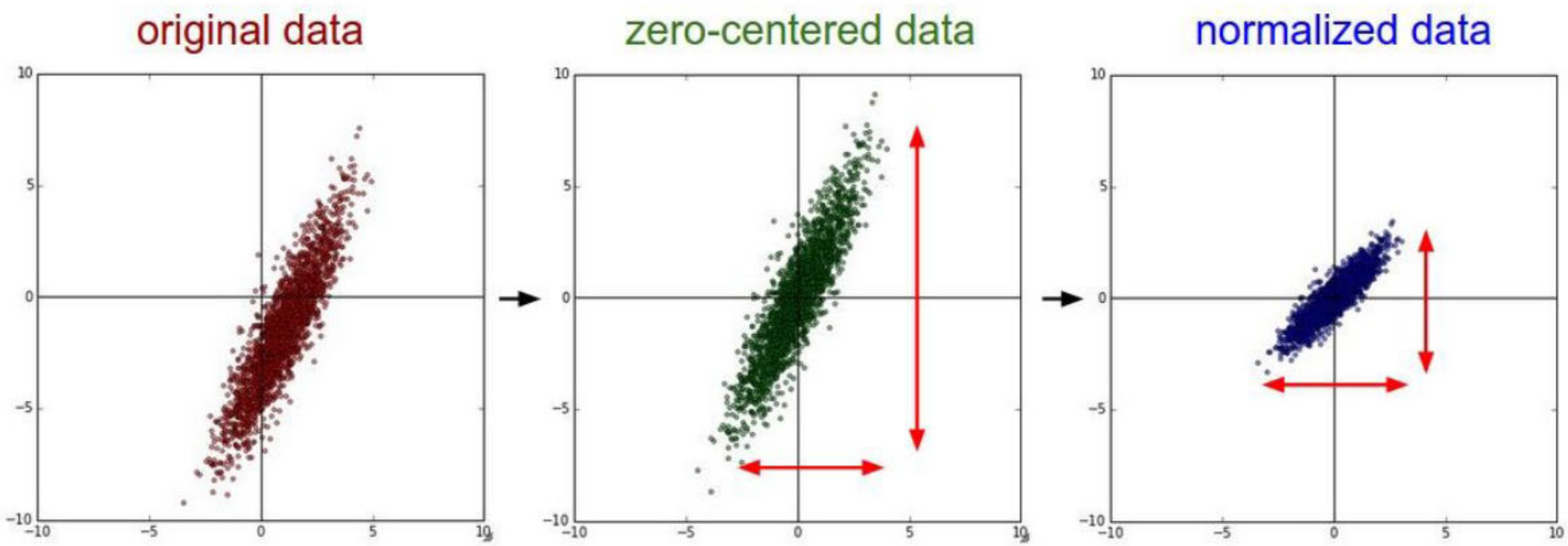
(this is also why you want zero-mean data!)



Ref: http://cs231n.stanford.edu/

# پیش‌پردازش داده‌ها

داده‌های صفر-مرکزشده / داده‌های نرمال‌شده




```
X -= np.mean(X, axis = 0)
```

```
X /= np.std(X, axis = 0)
```

(Assume X [NxD] is data matrix,
each example in a row)

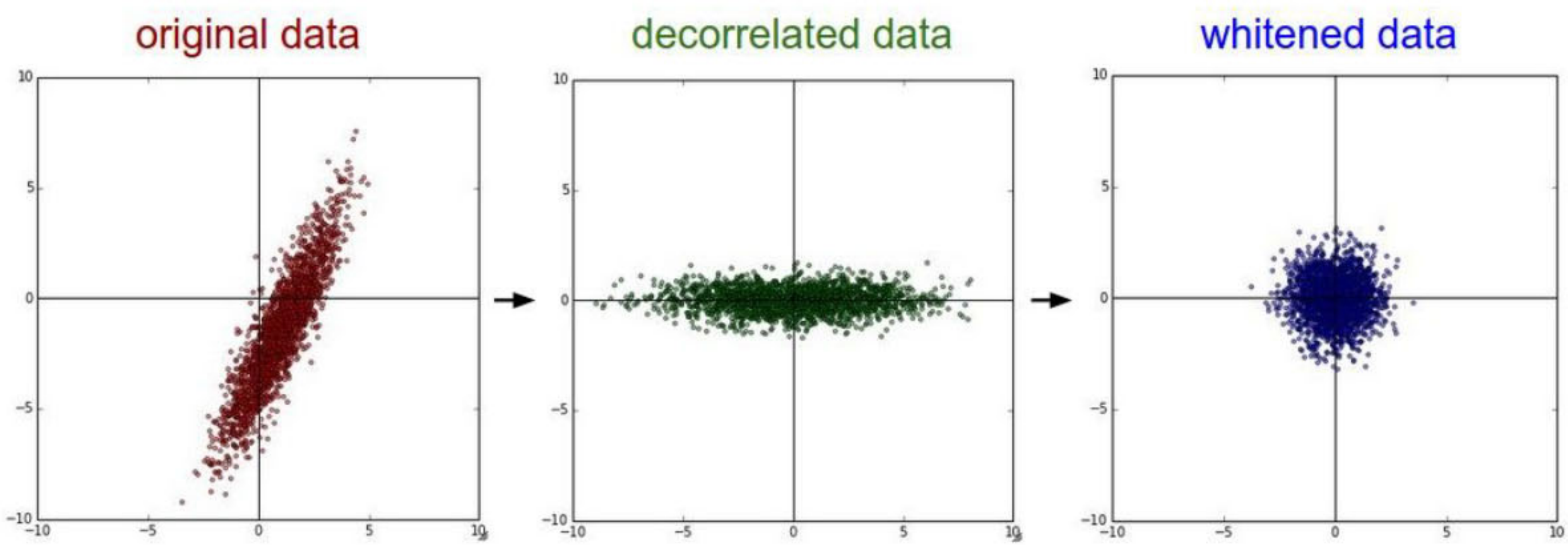


Ref: http://cs231n.stanford.edu/

# پیش‌پردازش داده‌ها

## داده‌های غیرهمبسته‌شده / داده‌های سفید شده

In practice, you may also see **PCA** and **Whitening** of the data




(data has diagonal covariance matrix)

(covariance matrix is the identity matrix)



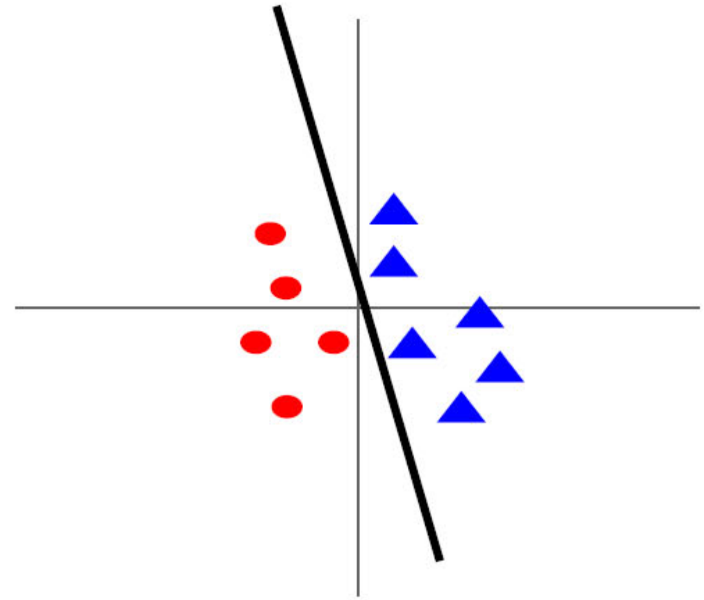


Ref: http://cs231n.stanford.edu/

# پیش‌پردازش داده‌ها

## نتیجه‌ی نرمال‌سازی داده‌ها

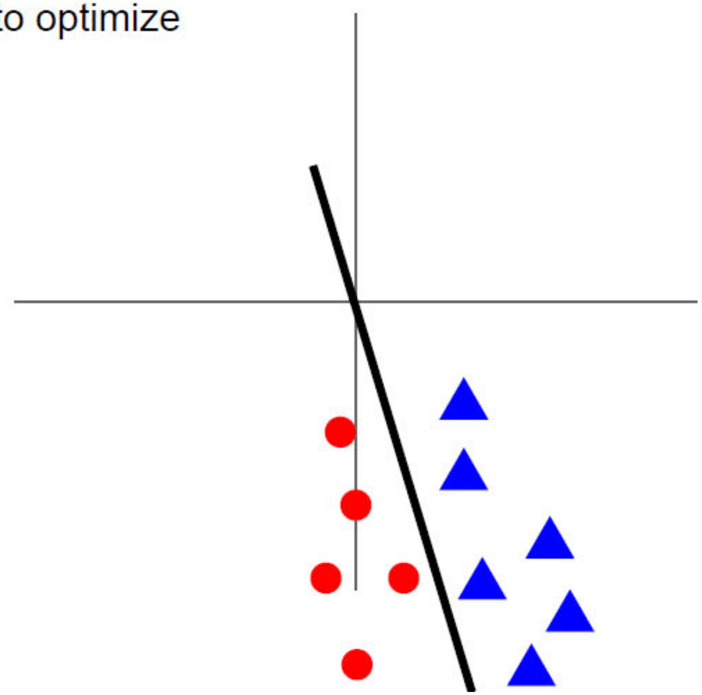
**After normalization**: less sensitive to small changes in weights; easier to optimize



**پس از نرمال‌سازی:**

میزان اتلاف طبقه‌بندی، به تغییرات در ماتریس وزن کمتر حساس است ⇐ بهینه‌سازی ساده‌تر است.

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



**پیش از نرمال‌سازی:**

میزان اتلاف طبقه‌بندی، به تغییرات در ماتریس وزن بسیار حساس است ⇐ بهینه‌سازی دشوار است.



Ref: http://cs231n.stanford.edu/

توصیه‌های کاربردی برای تصاویر

**TLDR: In practice for Images:** center only

e.g. consider CIFAR-10 example with [32,32,3] images

- Subtract the mean image (e.g. AlexNet)
  (mean image = [32,32,3] array)
- Subtract per-channel mean (e.g. VGGNet)
  (mean along each channel = 3 numbers)

Not common to normalize variance, to do PCA or whitening



Ref: http://cs231n.stanford.edu/

ملاحظاتی در
آموزش شبکه‌های عصبی عمیق

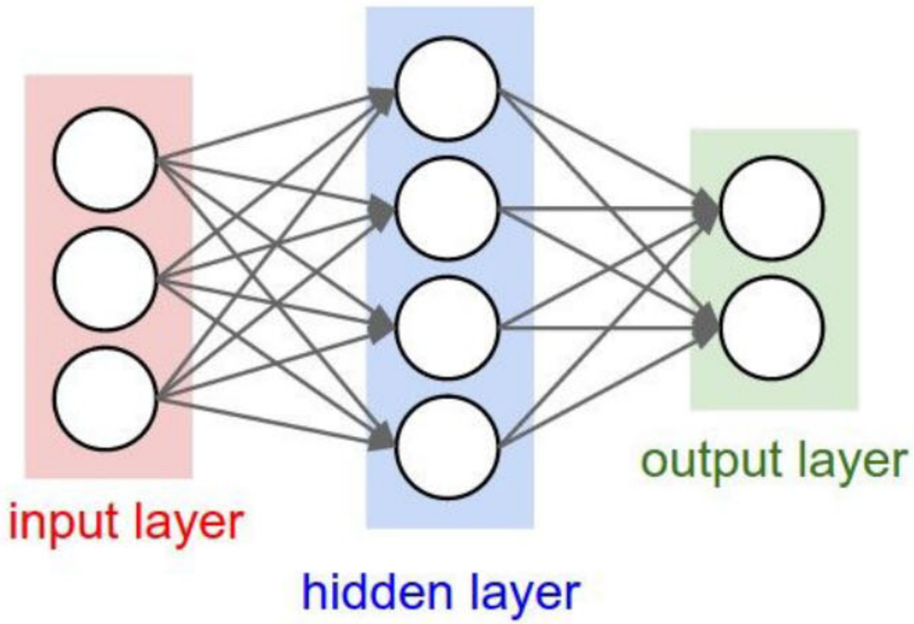
# ۳

# مقداردهی اولیه‌ی وزن‌ها

# مقداردهی اولیه‌ی وزن‌ها

اگر از وزن‌های ثابت مساوی برای مقداردهی آغازین استفاده کنیم ...

- Q: what happens when W=constant init is used?






Ref: http://cs231n.stanford.edu/

# مقداردهی اولیه‌ی وزن‌ها

ایده‌ی اول: استفاده از اعداد تصادفی کوچک

- First idea: **Small random numbers**

(gaussian with zero mean and 1e-2 standard deviation)

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

## مقداردهی اولیه‌ی وزن‌ها

ایده‌ی اول : استفاده از اعداد تصادفی کوچک

- First idea: **Small random numbers**
(gaussian with zero mean and 1e-2 standard deviation)

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

Works ~okay for small networks, but problems with deeper networks.

# مقداردهی اولیه‌ی وزن‌ها

Lets look at some activation statistics

E.g. 10-layer net with 500 neurons on each layer, using tanh non-linearities, and initializing as described in last slide.

```
# 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
```

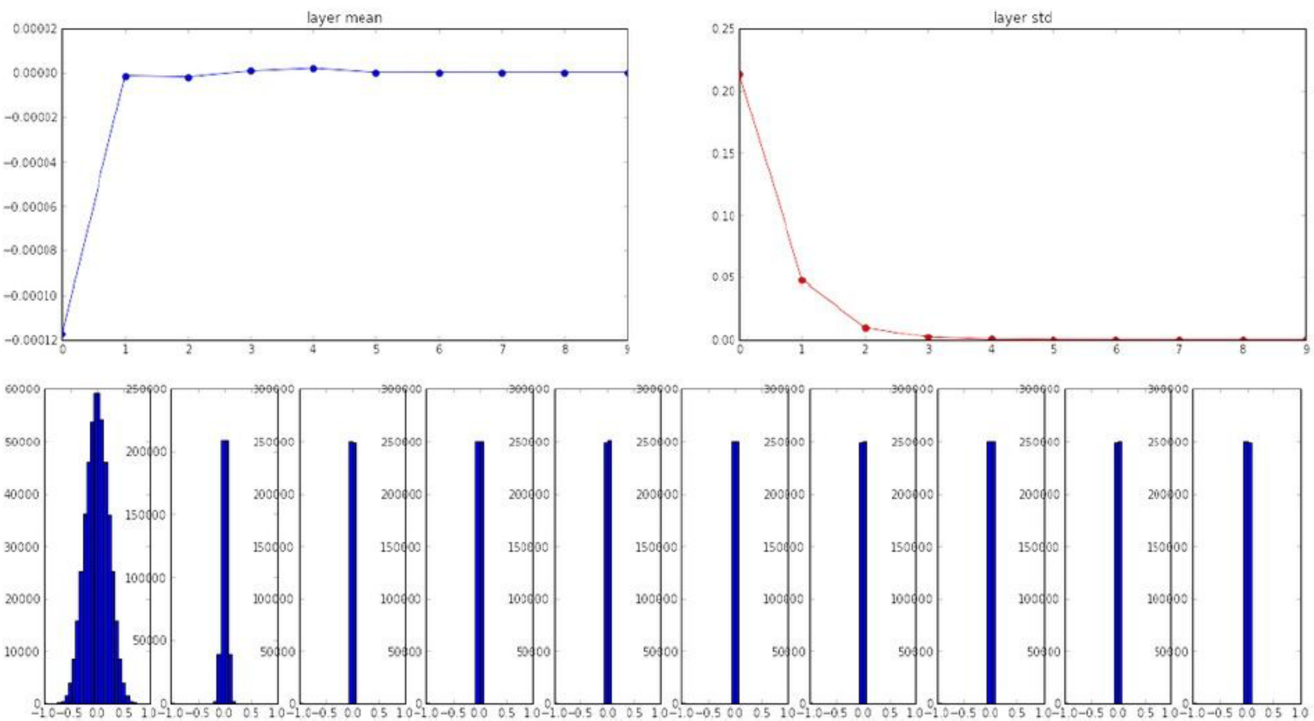
```
# look at distributions at each layer
print 'input layer had mean %f and std %f' % (np.mean(D), np.std(D))
layer_means = [np.mean(H) for i,H in Hs.iteritems()]
layer_stds = [np.std(H) for i,H in Hs.iteritems()]
for i,H in Hs.iteritems():
    print 'hidden layer %d had mean %f and std %f' % (i+1,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

# مقداردهی اولیه‌ی وزن‌ها

```
input layer had mean 0.000927 and std 0.998388
hidden layer 1 had mean -0.000117 and std 0.213081
hidden layer 2 had mean -0.000001 and std 0.047551
hidden layer 3 had mean -0.000002 and std 0.010630
hidden layer 4 had mean 0.000001 and std 0.002378
hidden layer 5 had mean 0.000002 and std 0.000532
hidden layer 6 had mean -0.000000 and std 0.000119
hidden layer 7 had mean 0.000000 and std 0.000026
hidden layer 8 had mean -0.000000 and std 0.000006
hidden layer 9 had mean 0.000000 and std 0.000001
hidden layer 10 had mean -0.000000 and std 0.000000
```

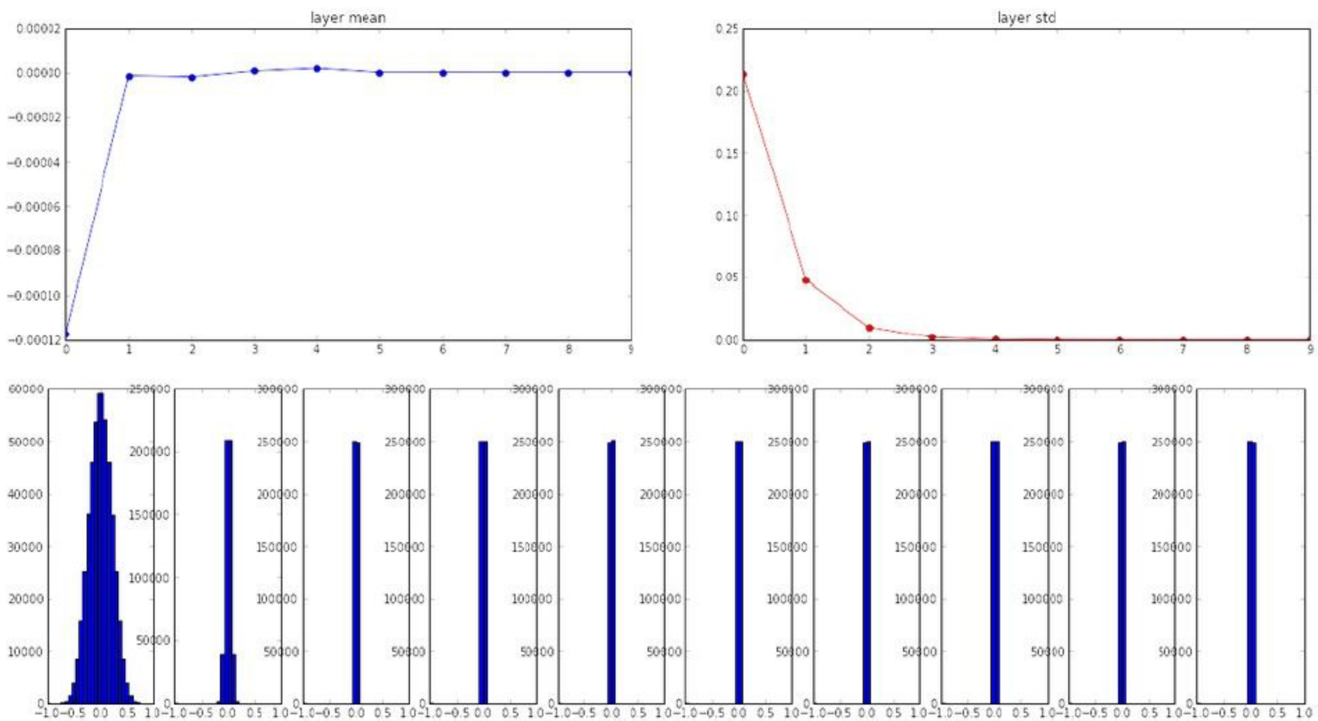





Ref: http://cs231n.stanford.edu/

# مقداردهی اولیه‌ی وزن‌ها

```
input layer had mean 0.000927 and std 0.998388
hidden layer 1 had mean -0.000117 and std 0.213081
hidden layer 2 had mean -0.000001 and std 0.047551
hidden layer 3 had mean -0.000002 and std 0.010630
hidden layer 4 had mean 0.000001 and std 0.002378
hidden layer 5 had mean 0.000002 and std 0.000532
hidden layer 6 had mean -0.000000 and std 0.000119
hidden layer 7 had mean 0.000000 and std 0.000026
hidden layer 8 had mean -0.000000 and std 0.000006
hidden layer 9 had mean 0.000000 and std 0.000001
hidden layer 10 had mean -0.000000 and std 0.000000
```




All activations become zero!

Q: think about the backward pass. What do the gradients look like?

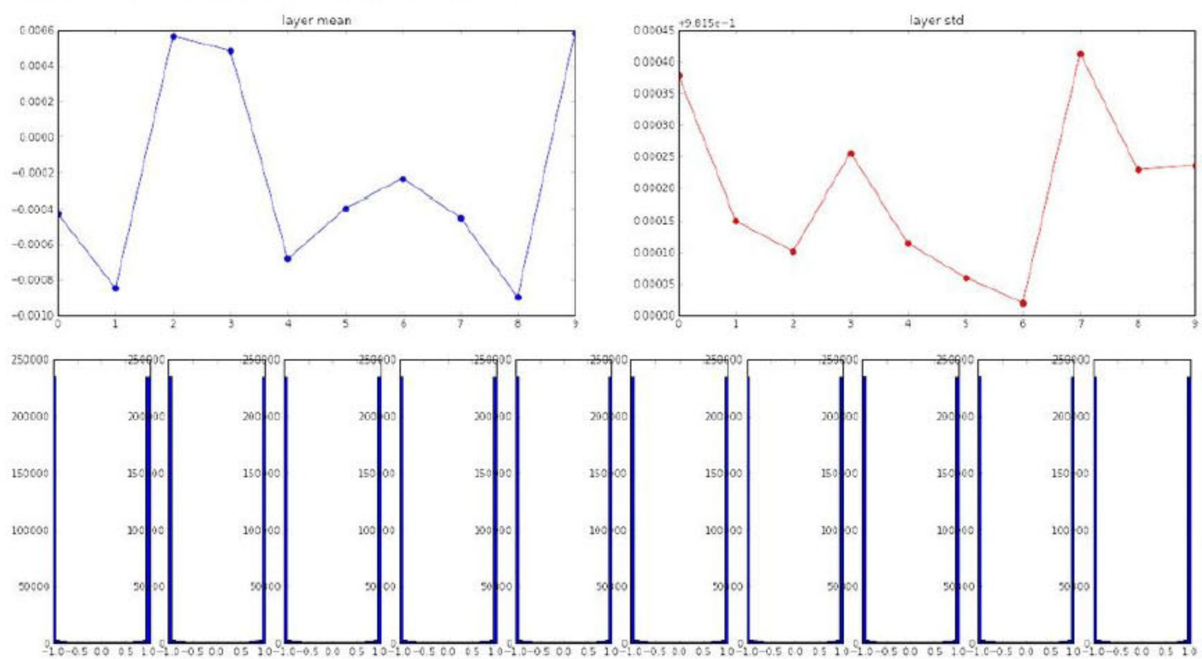
Hint: think about backward pass for a W*X gate.

# مقداردهی اولیه‌ی وزن‌ها

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

```
input layer had mean 0.001800 and std 1.001311
hidden layer 1 had mean -0.000430 and std 0.981879
hidden layer 2 had mean -0.000849 and std 0.981649
hidden layer 3 had mean 0.000566 and std 0.981601
hidden layer 4 had mean 0.000483 and std 0.981755
hidden layer 5 had mean -0.000682 and std 0.981614
hidden layer 6 had mean -0.000401 and std 0.981560
hidden layer 7 had mean -0.000237 and std 0.981520
hidden layer 8 had mean -0.000448 and std 0.981913
hidden layer 9 had mean -0.000899 and std 0.981728
hidden layer 10 had mean 0.000584 and std 0.981736
```

*1.0 instead of *0.01




Almost all neurons completely saturated, either -1 and 1. Gradients will be all zero.



Ref: http://cs231n.stanford.edu/

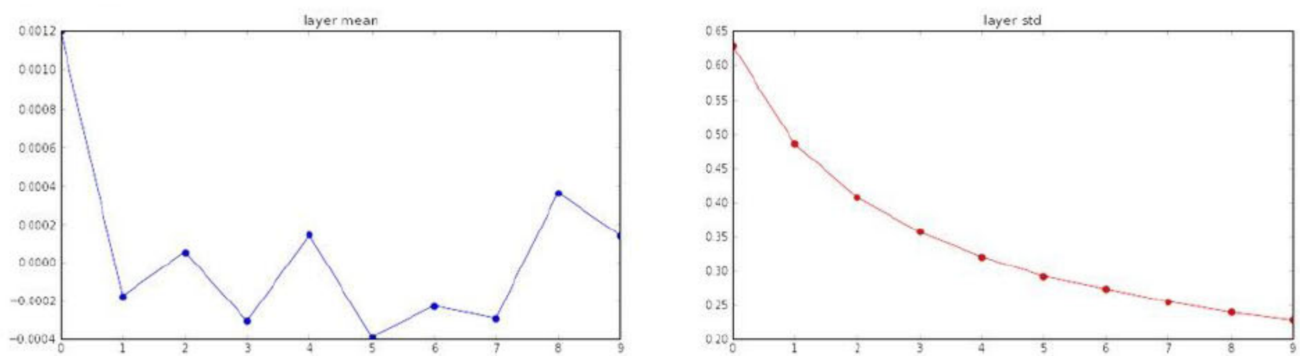
# مقداردهی اولیه‌ی وزن‌ها

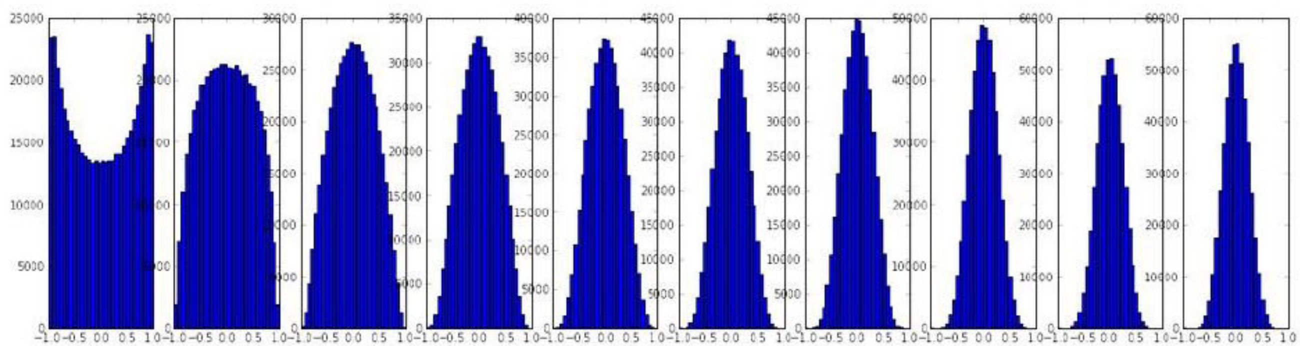
## روش مقداردهی اولیه‌ی Xavier

```
input layer had mean 0.001800 and std 1.001311
hidden layer 1 had mean 0.001198 and std 0.627953
hidden layer 2 had mean -0.000175 and std 0.486051
hidden layer 3 had mean 0.000055 and std 0.407723
hidden layer 4 had mean -0.000306 and std 0.357108
hidden layer 5 had mean 0.000142 and std 0.320917
hidden layer 6 had mean -0.000389 and std 0.292116
hidden layer 7 had mean -0.000228 and std 0.273387
hidden layer 8 had mean -0.000291 and std 0.254935
hidden layer 9 had mean 0.000361 and std 0.239266
hidden layer 10 had mean 0.000139 and std 0.228008
```

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

"Xavier initialization"
[Glorot et al., 2010]



**Reasonable initialization.**
(Mathematical derivation assumes linear activations)



Ref: http://cs231n.stanford.edu/

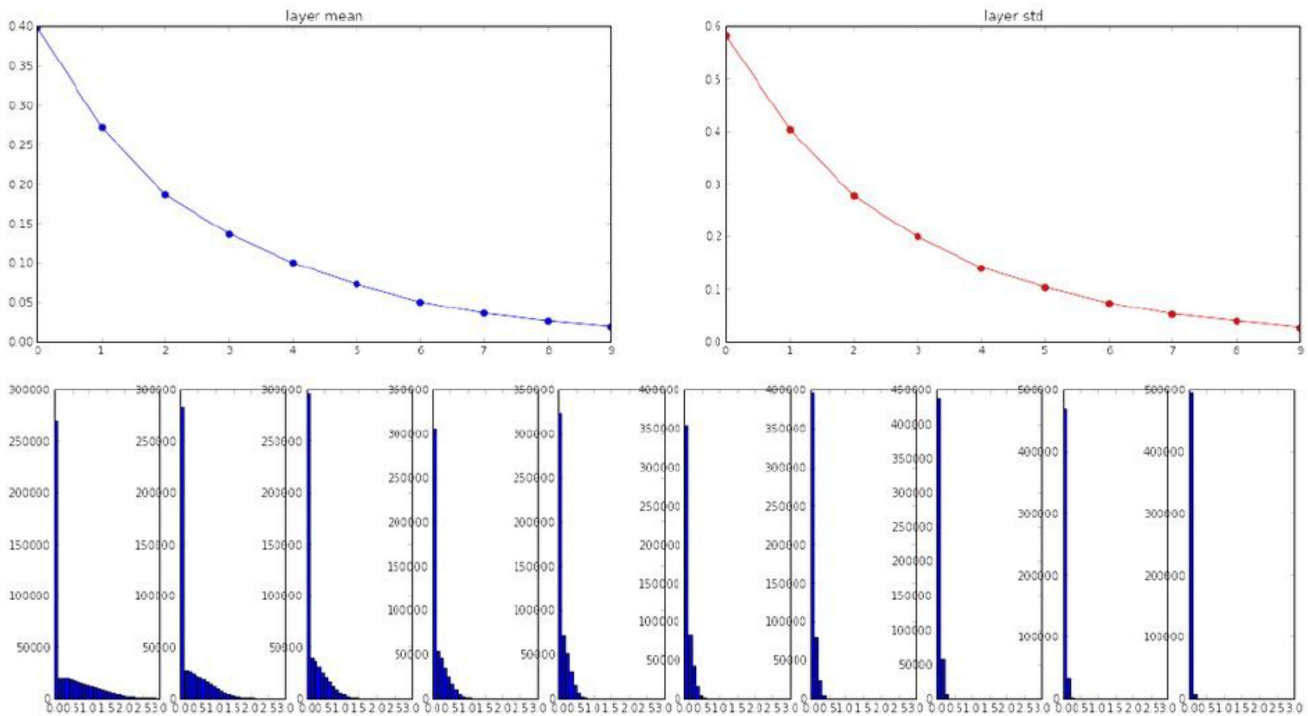
# مقداردهی اولیه‌ی وزن‌ها

## روش مقداردهی اولیه‌ی Xavier

```
input layer had mean 0.000501 and std 0.999444
hidden layer 1 had mean 0.398623 and std 0.582273
hidden layer 2 had mean 0.272352 and std 0.403795
hidden layer 3 had mean 0.186076 and std 0.276912
hidden layer 4 had mean 0.136442 and std 0.198685
hidden layer 5 had mean 0.099568 and std 0.140299
hidden layer 6 had mean 0.072234 and std 0.103280
hidden layer 7 had mean 0.049775 and std 0.072748
hidden layer 8 had mean 0.035138 and std 0.051572
hidden layer 9 had mean 0.025404 and std 0.038583
hidden layer 10 had mean 0.018408 and std 0.026076
```

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

but when using the ReLU nonlinearity it breaks.



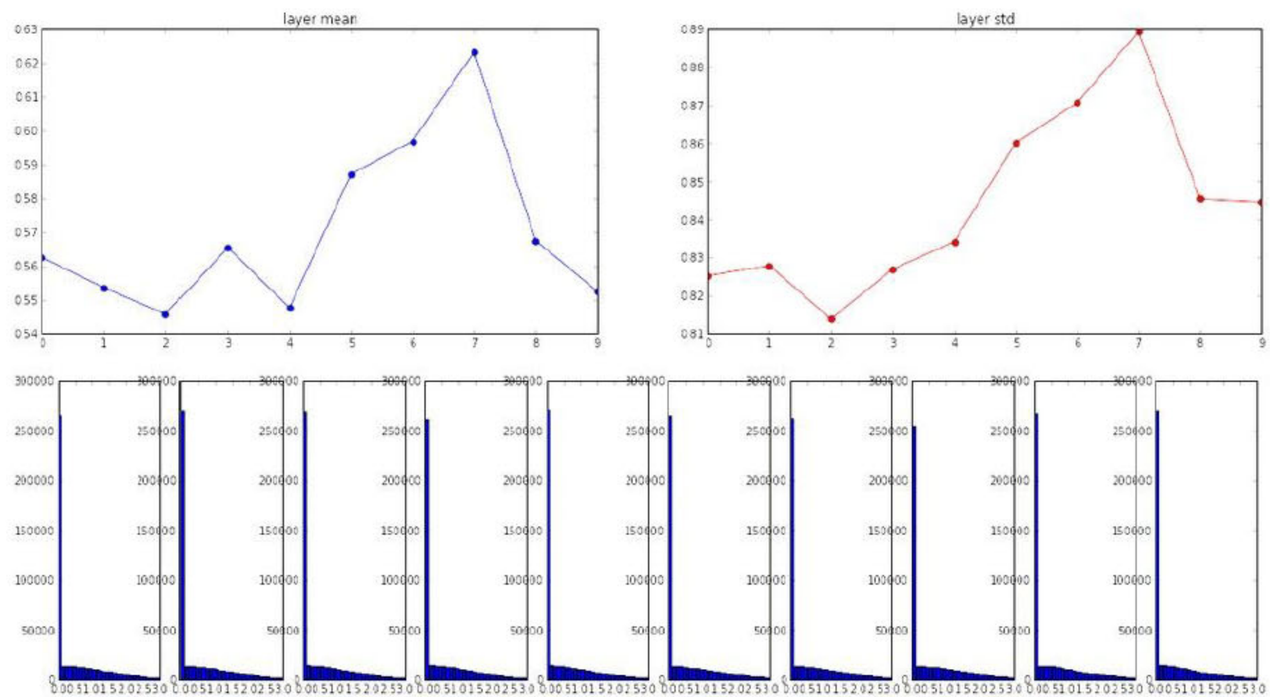
# مقداردهی اولیه‌ی وزن‌ها

روش مقداردهی اولیه‌ی He

```
input layer had mean 0.000501 and std 0.999444
hidden layer 1 had mean 0.562488 and std 0.825232
hidden layer 2 had mean 0.553614 and std 0.827835
hidden layer 3 had mean 0.545867 and std 0.813855
hidden layer 4 had mean 0.565396 and std 0.826902
hidden layer 5 had mean 0.547678 and std 0.834092
hidden layer 6 had mean 0.587103 and std 0.860035
hidden layer 7 had mean 0.596867 and std 0.870610
hidden layer 8 had mean 0.623214 and std 0.889348
hidden layer 9 had mean 0.567498 and std 0.845357
hidden layer 10 had mean 0.552531 and std 0.844523
```

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

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






Ref: http://cs231n.stanford.edu/

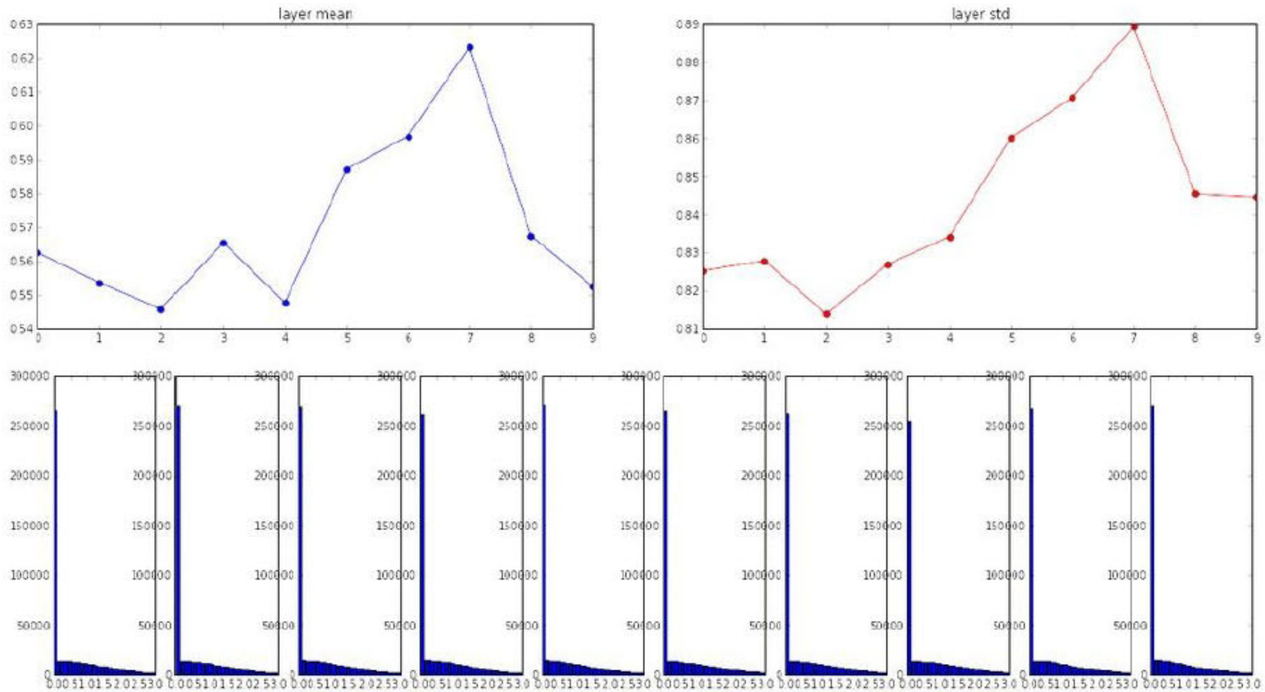
# مقداردهی اولیه‌ی وزن‌ها

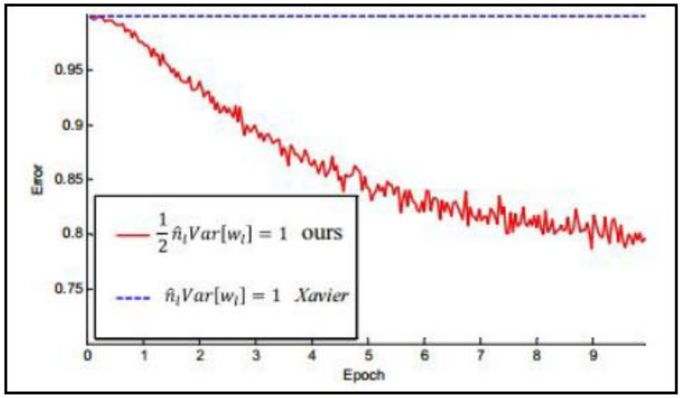
روش مقداردهی اولیه‌ی He

```
input layer had mean 0.000501 and std 0.999444
hidden layer 1 had mean 0.562488 and std 0.825232
hidden layer 2 had mean 0.553614 and std 0.827835
hidden layer 3 had mean 0.545867 and std 0.813855
hidden layer 4 had mean 0.565396 and std 0.826902
hidden layer 5 had mean 0.547678 and std 0.834092
hidden layer 6 had mean 0.587103 and std 0.860035
hidden layer 7 had mean 0.596867 and std 0.870610
hidden layer 8 had mean 0.623214 and std 0.889348
hidden layer 9 had mean 0.567498 and std 0.845357
hidden layer 10 had mean 0.552531 and std 0.844523
```

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

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






Ref: http://cs231n.stanford.edu/

# مقداردهی اولیه‌ی وزن‌ها

روش مناسب مقداردهی اولیه‌ی وزن‌ها یک حوزه‌ی پژوهشی فعال است ...

## Proper initialization is an active area of research…

***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

…



Ref: http://cs231n.stanford.edu/

## Initialization Strategies

- In convex problems with good $\epsilon$ no matter what the initialization, convergence is guaranteed
- In the non-convex regime initialization is much more important
- Some parameter initialization can be unstable, not converge
- Neural Networks are not well understood to have principled, mathematically nice initialization strategies
- What is known: Initialization should break symmetry (quiz!)
- What is known: Scale of weights is important
- Most initialization strategies are based on intuitions and heuristics

# استراتژی‌های مقداردهی اولیه

## چند هیوریستیک

## Initialization Strategies: Some Heuristics

- For a fully connected layer with $m$ inputs and $n$ outputs, sample:

$$W_{ij} \sim U(-\frac{1}{\sqrt{m}}, \frac{1}{\sqrt{m}})$$

- Xavier Initialization: Sample

$$W_{ij} \sim U(-\frac{6}{\sqrt{m+n}}, \frac{6}{\sqrt{m+n}})$$

- Xavier initialization is derived considering that the network consists of matrix multiplications with no nonlinearites
- Works well in practice!



Ref: https://ttic.uchicago.edu/~shubhendu/Pages/CMSC35246.html

# استراتژی‌های مقداردهی اولیه

چند هیوریستیک دیگر

## INITIALIZATION STRATEGIES: MORE HEURISTICS

- Saxe *et al.* 2013, recommend initialzing to random orthogonal matrices, with a carefully chosen gain $g$ that accounts for non-linearities
- If $g$ could be divined, it could solve the vanishing and exploding gradients problem (more later)
- The idea of choosing $g$ and initializing weights accordingly is that we want norm of activations to increase, and pass back strong gradients
- Martens 2010, suggested an initialization that was sparse: Each unit could only receive $k$ non-zero weights
- Motivation: Ir is a bad idea to have all initial weights to have the same standard deviation $\frac{1}{\sqrt{m}}$



Ref: https://ttic.uchicago.edu/~shubhendu/Pages/CMSC35246.html

ملاحظاتی در
آموزش شبکه‌های عصبی عمیق

# ۴

# نرمال‌سازی دسته‌ای

## نرمال‌سازی دسته‌ای

# Batch Normalization

[Ioffe and Szegedy, 2015]

"you want zero-mean unit-variance activations? just make them so."

می‌خواهیم خروجی هر لایه در همه‌ی ابعاد آن دارای میانگین صفر و واریانس یک باشد.

consider a batch of activations at some layer. To make each dimension zero-mean unit-variance, apply:

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

this is a vanilla differentiable function...

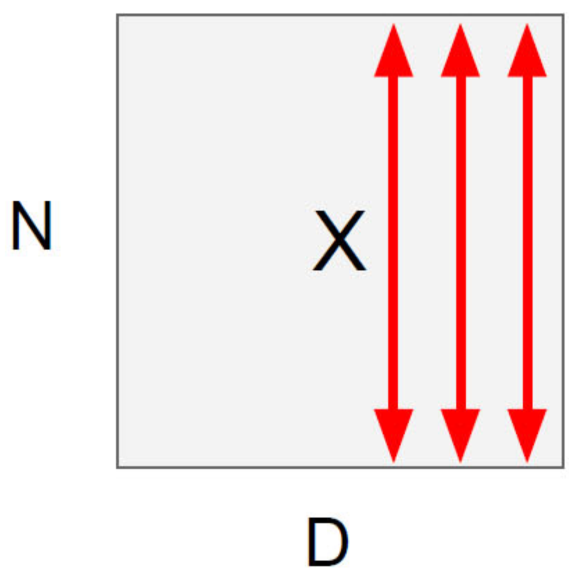
# نرمال‌سازی دسته‌ای

الگوریتم

## Batch Normalization

[Ioffe and Szegedy, 2015]

"you want zero-mean unit-variance activations? just make them so."




1. compute the empirical mean and variance independently for each dimension.

2. Normalize

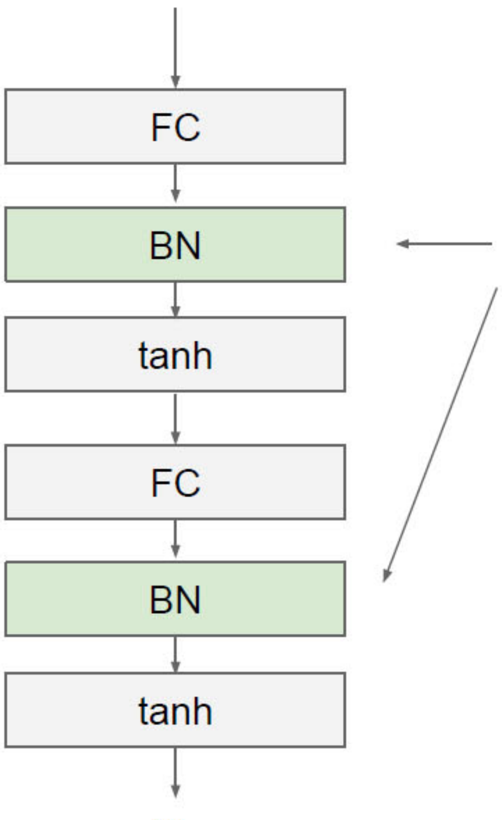
$$\widehat{x}^{(k)} = \frac{x^{(k)} - \mathrm{E}[x^{(k)}]}{\sqrt{\mathrm{Var}[x^{(k)}]}}$$

# نرمال‌سازی دسته‌ای

## لایه‌ی BN

# Batch Normalization

[Ioffe and Szegedy, 2015]




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

معمولاً پس از لایه‌های تماماً متصل یا لایه‌های کانوولوشنال، و پیش از تابع فعالیت غیرخطی قرار می‌گیرد.

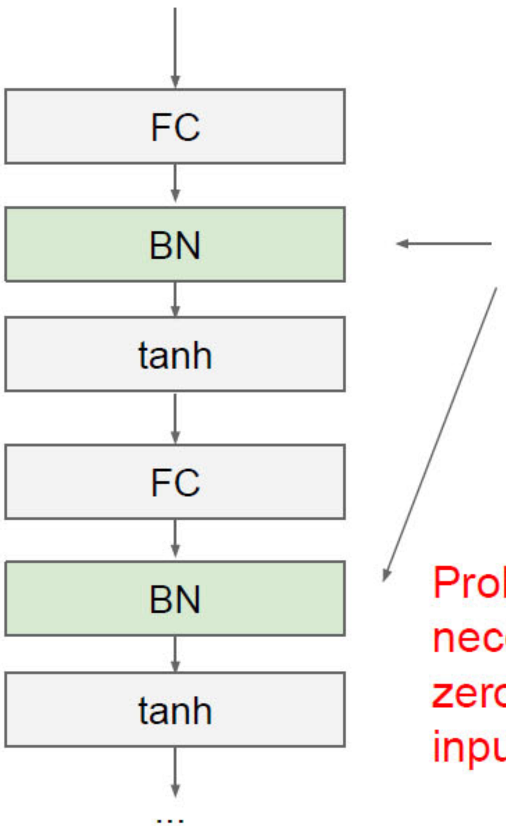
$$\widehat{x}^{(k)} = \frac{x^{(k)} - \mathrm{E}[x^{(k)}]}{\sqrt{\mathrm{Var}[x^{(k)}]}}$$

## نرمال‌سازی دسته‌ای

مسئله: لزوم نیاز به ورودی مرکز صفر و واریانس یک؟

# Batch Normalization

[Ioffe and Szegedy, 2015]




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

Problem: do we necessarily want a zero-mean unit-variance input?

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



Ref: http://cs231n.stanford.edu/

# Batch Normalization

[Ioffe and Szegedy, 2015]

Normalize:

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

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

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

Note, the network can learn:

$$\gamma^{(k)} = \sqrt{\mathrm{Var}[x^{(k)}]}$$

$$\beta^{(k)} = \mathrm{E}[x^{(k)}]$$

to recover the identity mapping.

پارامترها به شبکه اجازه می‌دهند بازه‌ی مقادیر خروجی لایه را در هنگام آموزش تعیین کنند.



Ref: http://cs231n.stanford.edu/

# نرمال‌سازی دسته‌ای

مزایا

## Batch Normalization

[Ioffe and Szegedy, 2015]

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

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

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

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

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

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

- 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

- بهبود جریان گرادیان در سرتاسر شبکه
- امکان‌دهی به نرخ‌های یادگیری بالاتر
- کاهش وابستگی شدید به مقادیر آغازین
- ایفای نقش به‌عنوان نوعی رگولاریزاسیون، به‌صورتی جالب و احتمالاً اندکی کاهش در نیاز به dropout.

Ref: http://cs231n.stanford.edu/

# نرمال‌سازی دسته‌ای

عملکرد لایه‌ی BN در هنگام آزمایش

## Batch Normalization

[Ioffe and Szegedy, 2015]

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

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

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

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

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

**Note: at test time BatchNorm layer functions differently:**

The mean/std are not computed based on the batch. Instead, a single fixed empirical mean of activations during training is used.

(e.g. can be estimated during training with running averages)

در هنگام آزمایش، mean/std بر اساس دسته (batch) محاسبه نمی‌شود.
در عوض، مقادیر میانگین تجربی ثابت فعالیت‌ها در طول آموزش، استفاده می‌شود.



کاظم فولادی

Ref: http://cs231n.stanford.edu/

خلاصه

# Batch Normalization

**Input:** $x : N \times D$

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

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

**Output:** $y : N \times D$

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

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

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

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



Ref: http://cs231n.stanford.edu/

# نرمال‌سازی دسته‌ای

در هنگام آموزش: تخمین میانگین و واریانس از روی مینی‌بچ

## Batch Normalization

Estimate mean and variance from minibatch; Can't do this at test-time

**Input**: $x : N \times D$

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

**Learnable params**:

$\gamma, \beta : D$

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

**Intermediates**: $\mu, \sigma : D$

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

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

**Output**: $y : N \times D$

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



Ref: http://cs231n.stanford.edu/

## نرمال‌سازی دسته‌ای

در هنگام آزمایش

# 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

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

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

# نرمال‌سازی دسته‌ای

برای شبکه‌های کانوولوشنال

## Batch Normalization for ConvNets

Batch Normalization for **fully-connected** networks

x: N × D

Normalize (over N)

μ,σ: 1 × D

γ,β: 1 × D

y = γ(x−μ)/σ+β

Batch Normalization for **convolutional** networks (Spatial Batchnorm, BatchNorm2D)

x: N×C×H×W

Normalize (over N, H, W)

μ,σ: 1×C×1×1

γ,β: 1×C×1×1

y = γ(x−μ)/σ+β

نرمال‌سازی لایه‌ای

# Layer Normalization

**Batch Normalization** for fully-connected networks

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

Normalize

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

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

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

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

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

Normalize

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

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

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

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

## نرمال‌سازی دسته‌ای

برای شبکه‌های کانوولوشنال: نرمال‌سازی نمونه‌ای

# Instance Normalization

**Batch Normalization** for convolutional networks

x: N×C×H×W

Normalize (arrows under N, H, W)

$\mu, \sigma$: 1×C×1×1

$\gamma, \beta$: 1×C×1×1

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

**Instance Normalization** for convolutional networks

Same behavior at train / test!

x: N×C×H×W

Normalize (arrows under H, W)

$\mu, \sigma$: N×C×1×1

$\gamma, \beta$: 1×C×1×1

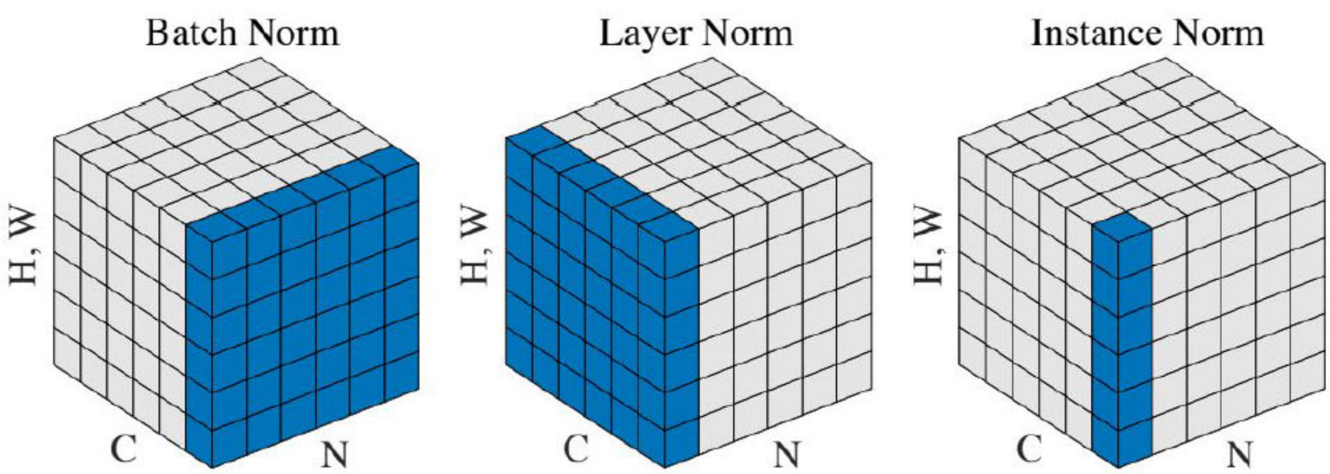
$y = \gamma(x-\mu)/\sigma+\beta$

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

# نرمال‌سازی دسته‌ای

برای شبکه‌های کانوولوشنال: مقایسه‌ی لایه‌های نرمال‌سازی

## Comparison of Normalization Layers

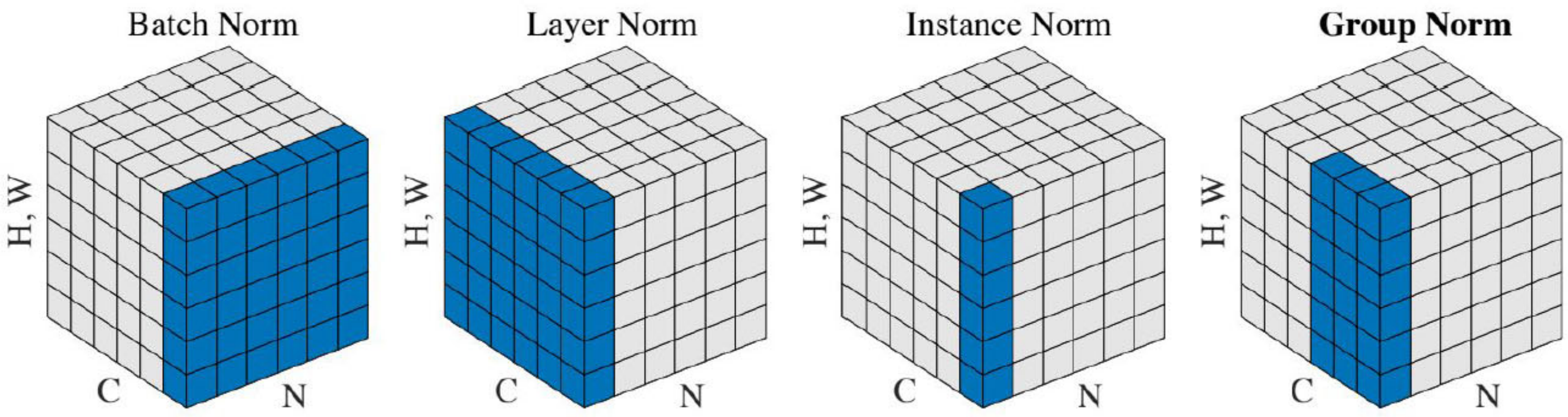



Wu and He, “Group Normalization”, arXiv 2018

# نرمال‌سازی دسته‌ای

برای شبکه‌های کانوولوشنال: نرمال‌سازی گروهی

# Group Normalization



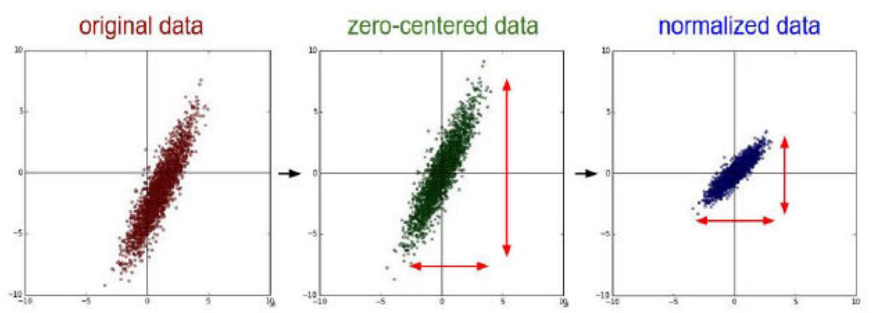

Wu and He, "Group Normalization", arXiv 2018 (Appeared 3/22/2018)

## نرمال‌سازی دسته‌ای

نرمال‌سازی دسته‌ای غیرهمبسته شده

# Decorrelated Batch Normalization

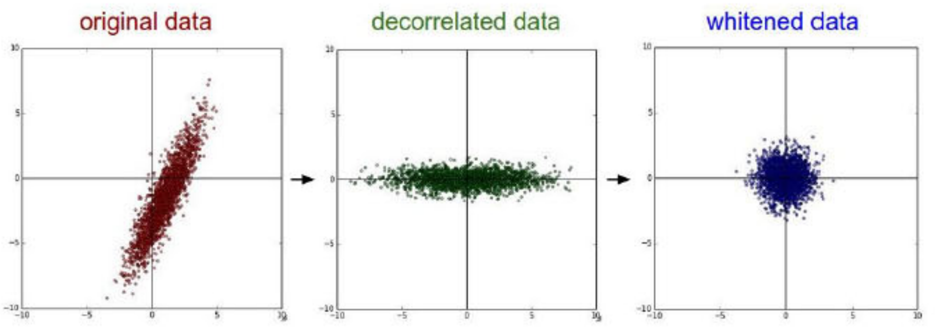
### Batch Normalization




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

BatchNorm normalizes the data, but cannot correct for correlations among the input features

### Decorrelated Batch Normalization




$$\hat{x}_i = \Sigma^{-\frac{1}{2}}(x_i - \mu)$$

DBN **whitens** the data using the full covariance matrix of the minibatch; this corrects for correlations

Huang et al, "Decorrelated Batch Normalization", arXiv 2018 (Appeared 4/23/2018)



Ref: http://cs231n.stanford.edu/

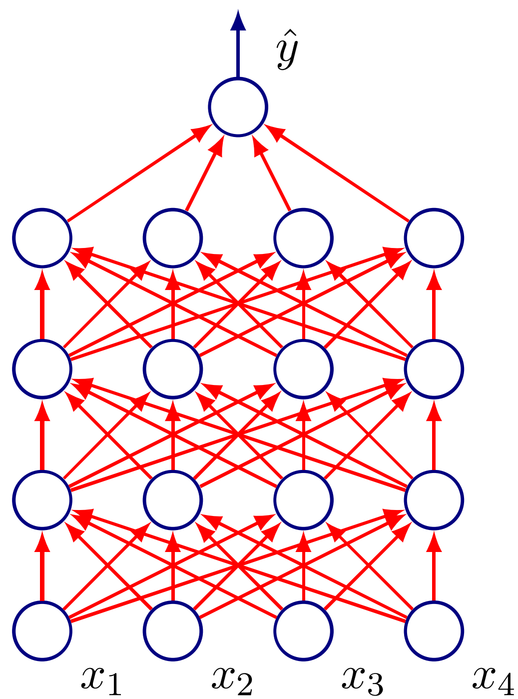
# یک دشواری در آموزش شبکه‌های عصبی عمیق

A DIFFICULTY IN TRAINING DEEP NEURAL NETWORKS

## A Difficulty in Training Deep Neural Networks

A deep model involves composition of several functions

$$\hat{y} = W_4^T(\tanh(W_3^T(\tanh(W_2^T(\tanh(W_1^T\mathbf{x} + \mathbf{b}_1) + \mathbf{b}_2) + \mathbf{b}_3))))$$






Ref: https://ttic.uchicago.edu/~shubhendu/Pages/CMSC35246.html

# یک دشواری در آموزش شبکه‌های عصبی عمیق

## A Difficulty in Training Deep Neural Networks

- We have a recipe to compute gradients (Backpropagation), and update every parameter (we saw half a dozen methods)
- Implicit Assumption: Other layers don't change i.e. other functions are fixed
- In Practice: We update all layers simultaneously
- This can give rise to unexpected difficulties
- Let's look at two illustrations

# یک دشواری در آموزش شبکه‌های عصبی عمیق

شـهود

## INTUITION

- Consider a second order approximation of our cost function (which is a function composition) around current point $\theta^{(0)}$:

$$J(\theta) \approx J(\theta^{(0)}) + (\theta - \theta^{(0)})^T \mathbf{g} + \frac{1}{2}(\theta - \theta^{(0)})^T H (\theta - \theta^{(0)})$$

- $\mathbf{g}$ is gradient and $H$ the Hessian at $\theta^{(0)}$
- If $\epsilon$ is the learning rate, the new point

$$\theta = \theta^{(0)} - \epsilon \mathbf{g}$$



Ref: https://ttic.uchicago.edu/~shubhendu/Pages/CMSC35246.html

## یک دشواری در آموزش شبکه‌های عصبی عمیق

شهود

### INTUITION

- Plugging our new point, $\theta = \theta^{(0)} - \epsilon \mathbf{g}$ into the approximation:

$$J(\theta^{(0)} - \epsilon \mathbf{g}) = J(\theta^{(0)}) - \epsilon \mathbf{g}^T \mathbf{g} + \frac{1}{2} \mathbf{g}^T H \mathbf{g}$$

- There are three terms here:
  - Value of function before update
  - Improvement using gradient (i.e. first order information)
  - Correction factor that accounts for the curvature of the function



Ref: https://ttic.uchicago.edu/~shubhendu/Pages/CMSC35246.html

# یک دشواری در آموزش شبکه‌های عصبی عمیق

شـهود

## INTUITION

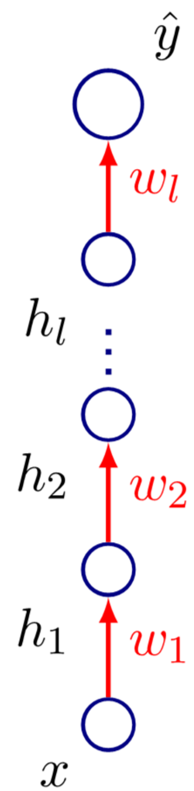
$$J(\theta^{(0)} - \epsilon \mathbf{g}) = J(\theta^{(0)}) - \epsilon \mathbf{g}^T \mathbf{g} + \frac{1}{2}\mathbf{g}^T H \mathbf{g}$$

- Observations:
  - $\mathbf{g}^T H \mathbf{g}$ too large: Gradient will start moving upwards
  - $\mathbf{g}^T H \mathbf{g} = 0$: $J$ will decrease for even large $\epsilon$
  - Optimal step size $\epsilon^* = \mathbf{g}^T \mathbf{g}$ for zero curvature, $\epsilon^* = \frac{\mathbf{g}^T \mathbf{g}}{\mathbf{g}^T H \mathbf{g}}$ to take into account curvature
- Conclusion: Just neglecting second order effects can cause problems (remedy: second order methods). What about higher order effects?

# یک دشواری در آموزش شبکه‌های عصبی عمیق

آثار مرتبه بالاتر

## Higher Order Effects: Toy Model




- Just one node per layer, no non-linearity
- $\hat{y}$ is linear in $x$ but non-linear in $w_i$

# یک دشواری در آموزش شبکه‌های عصبی عمیق

آثار مرتبه بالاتر

## HIGHER ORDER EFFECTS: TOY MODEL

- Suppose $\delta = 1$, so we want to decrease our output $\hat{y}$
- Usual strategy:
  - Using backprop find $\mathbf{g} = \nabla_{\mathbf{w}}(\hat{y} - y)^2$
  - Update weights $\mathbf{w} := \mathbf{w} - \epsilon\mathbf{g}$
- The first order Taylor approximation (in previous slide) says the cost will reduce by $\epsilon\mathbf{g}^T\mathbf{g}$
- If we need to reduce cost by 0.1, then learning rate should be $\frac{0.1}{\mathbf{g}^T\mathbf{g}}$



Ref: https://ttic.uchicago.edu/~shubhendu/Pages/CMSC35246.html

# یک دشواری در آموزش شبکه‌های عصبی عمیق

آثار مرتبه بالاتر

## Higher Order Effects: Toy Model

- The new $\hat{y}$ will however be:

$$\hat{y} = x(w_1 - \epsilon g_1)(w_2 - \epsilon g_2) \dots (w_l - \epsilon g_l)$$

- Contains terms like $\epsilon^3 g_1 g_2 g_3 w_4 w_5 \dots w_l$
- If weights $w_4, w_5, \dots, w_l$ are small, the term is negligible. But if large, it would explode
- Conclusion: Higher order terms make it very hard to choose the right learning rate
- Second Order Methods are already expensive, $n$th order methods are hopeless. Solution?



Ref: https://ttic.uchicago.edu/~shubhendu/Pages/CMSC35246.html

## Batch Normalization

- Method to reparameterize a deep network to reduce co-ordination of update across layers
- Can be applied to input layer, or any hidden layer
- Let $H$ be a design matrix having activations in any layer for $m$ examples in the mini-batch

$$H = \begin{bmatrix} h_{11} & h_{12} & h_{13} & \dots & h_{1k} \\ h_{21} & h_{22} & h_{23} & \dots & h_{2k} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ h_{m1} & h_{m2} & h_{m3} & \dots & h_{mk} \end{bmatrix}$$





Ref: https://ttic.uchicago.edu/~shubhendu/Pages/CMSC35246.html

# نرمال‌سازی دسته‌ای

## Batch Normalization

$$H = \begin{bmatrix} h_{11} & h_{12} & h_{13} & \dots & h_{1k} \\ h_{21} & h_{22} & h_{23} & \dots & h_{2k} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \\ h_{m1} & h_{m2} & h_{m3} & \dots & h_{mk} \end{bmatrix}$$

- Each row represents all the activations in layer for one example
- Idea: Replace $H$ by $H'$ such that:

$$H' = \frac{H - \mu}{\sigma}$$

- $\mu$ is mean of each unit and $\sigma$ the standard deviation



Ref: https://ttic.uchicago.edu/~shubhendu/Pages/CMSC35246.html

## Batch Normalization

- $\mu$ is a vector with $\mu_j$ the column mean
- $\sigma$ is a vector with $\sigma_j$ the column standard deviation
- $H_{i,j}$ is normalized by subtracting $\mu_j$ and dividing by $\sigma_j$



Ref: https://ttic.uchicago.edu/~shubhendu/Pages/CMSC35246.html

# نرمال‌سازی دسته‌ای

## Batch Normalization

- During training we have:

$$\mu = \frac{1}{m} \sum_j H_{:,j}$$

$$\sigma = \sqrt{\delta + \frac{1}{m} \sum_j (H - \mu)_j^2}$$

- We then operate on $H'$ as before $\implies$ we backpropagate *through* the normalized activations


کاظم فولادی
Ref: https://ttic.uchicago.edu/~shubhendu/Pages/CMSC35246.html

# نرمال‌سازی دسته‌ای

چرا خوب است؟

## Batch Normalization: Why Is This Good?

- The update will never act to only increase the mean and standard deviation of any activation
- Previous approaches added penalties to cost or per layer to encourage units to have standardized outputs
- Batch normalization makes the reparameterization easier
- At test time: Use running averages of $\mu$ and $\sigma$ collected during training, use these for evaluating new input $\mathbf{x}$



Ref: https://ttic.uchicago.edu/~shubhendu/Pages/CMSC35246.html

# نرمال‌سازی دسته‌ای

یک نوآوری

## AN INNOVATION

- Standardizing the output of a unit can limit the expressive power of the neural network
- Solution: Instead of replacing $H$ by $H'$, replace it will $\gamma H' + \beta$
- $\gamma$ and $\beta$ are also learned by backpropagation
- Normalizing for mean and standard deviation was the goal of batch normalization, why add $\gamma$ and $\beta$ again?



Ref: https://ttic.uchicago.edu/~shubhendu/Pages/CMSC35246.html

ملاحظاتی در
آموزش شبکه‌های عصبی عمیق

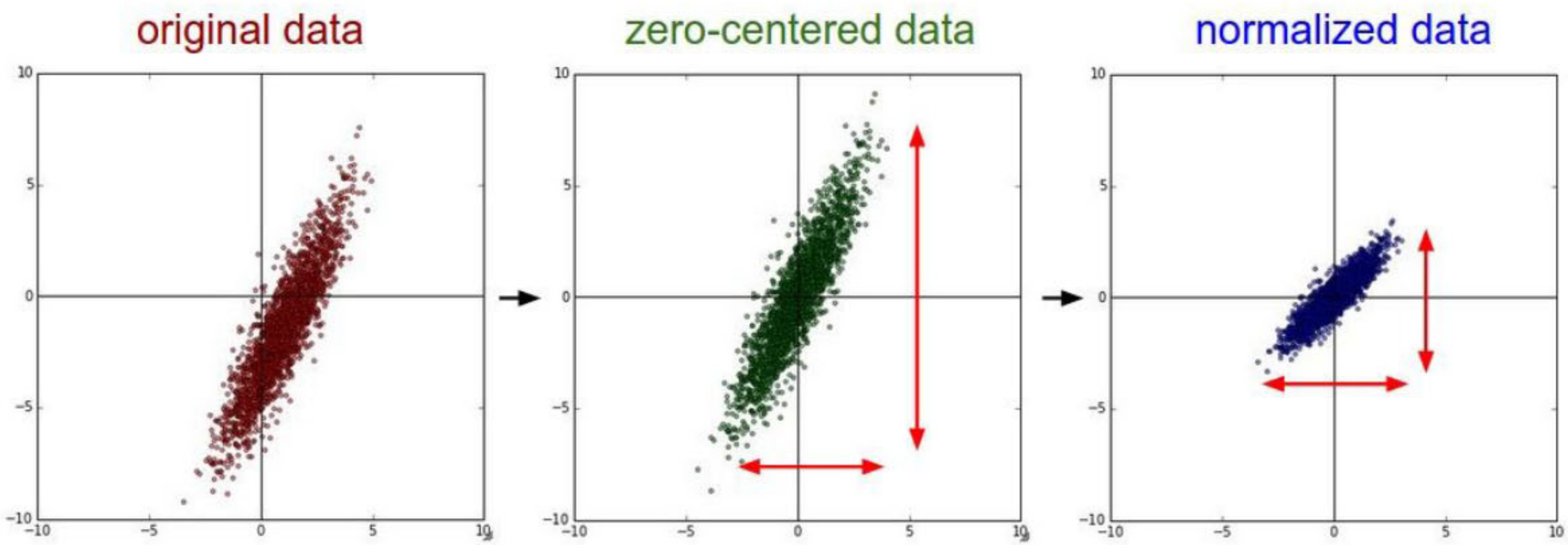
# ۵

# فرآیند یادگیری

# فرآیند یادگیری

گام ۱ : پیش‌پردازش داده‌ها

# Step 1: Preprocess the data




```
X -= np.mean(X, axis = 0)
```

```
X /= np.std(X, axis = 0)
```

(Assume X [NxD] is data matrix, each example in a row)



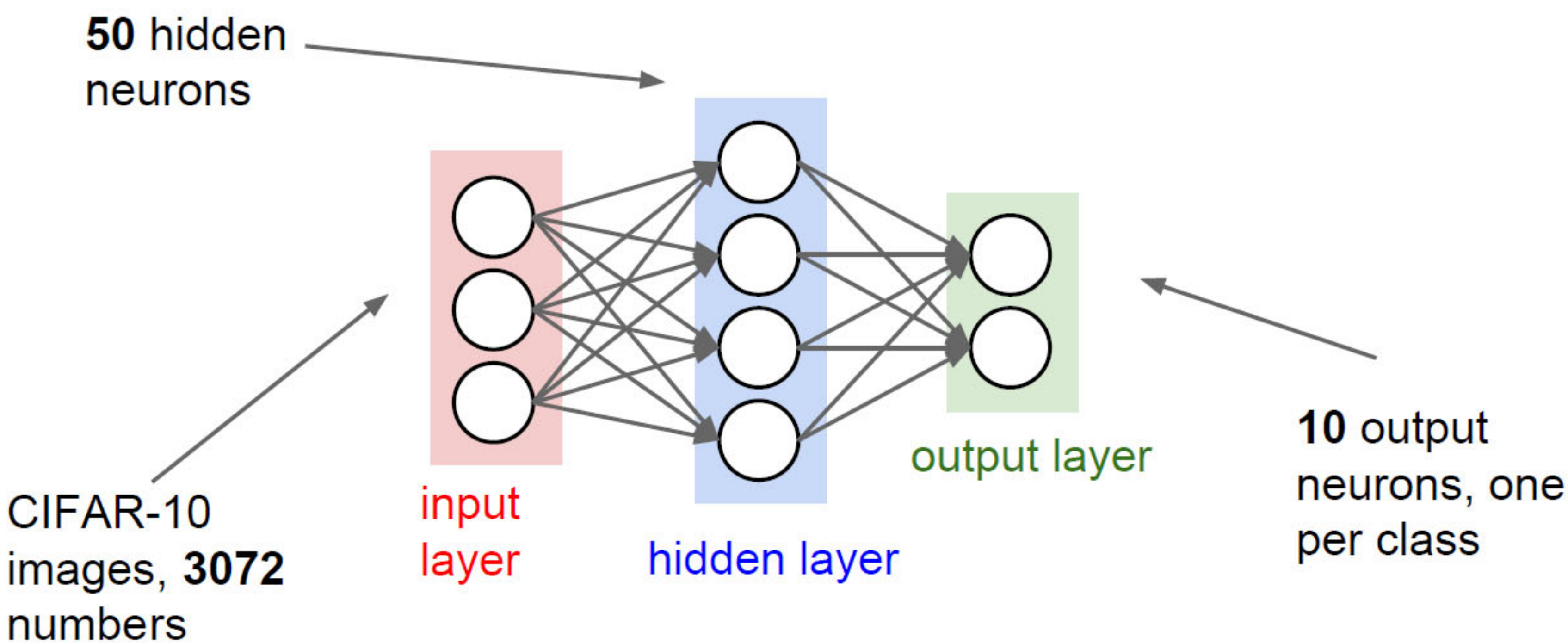
Ref: http://cs231n.stanford.edu/

# فرآیند یادگیری

گام ۲ : انتخاب معماری

**Step 2: Choose the architecture:**

say we start with one hidden layer of 50 neurons:






Ref: http://cs231n.stanford.edu/

# فرآیند یادگیری

بررسی معقول بودن مقدار اتلاف ...

# Double check that the loss is reasonable:

```
def init_two_layer_model(input_size, hidden_size, output_size):
  # initialize a model
  model = {}
  model['W1'] = 0.0001 * np.random.randn(input_size, hidden_size)
  model['b1'] = np.zeros(hidden_size)
  model['W2'] = 0.0001 * np.random.randn(hidden_size, output_size)
  model['b2'] = np.zeros(output_size)
  return model
```

```
model = init_two_layer_model(32*32*3, 50, 10) # input_size, hidden_size, number of classes
loss, grad = two_layer_net(X_train, model, y_train, 0.0)
print loss
```

```
2.30261216167
```

disable regularization

loss ~2.3.
"correct" for
10 classes

returns the loss and the gradient for all parameters

## فرآیند یادگیری

بررسی معقول بودن مقدار اتلاف ...

# Double check that the loss is reasonable:

```
def init_two_layer_model(input_size, hidden_size, output_size):
  # initialize a model
  model = {}
  model['W1'] = 0.0001 * np.random.randn(input_size, hidden_size)
  model['b1'] = np.zeros(hidden_size)
  model['W2'] = 0.0001 * np.random.randn(hidden_size, output_size)
  model['b2'] = np.zeros(output_size)
  return model
```

```
model = init_two_layer_model(32*32*3, 50, 10) # input_size, hidden size, number of classes
loss, grad = two_layer_net(X_train, model, y_train, 1e3)
print loss
```

```
3.06859716482
```

crank up regularization

loss went up, good. (sanity check)

# فرآیند یادگیری

شروع آموزش

Lets try to train now…

**Tip**: Make sure that you can overfit very small portion of the training data

```
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
trainer = ClassifierTrainer()
X_tiny = X_train[:20] # take 20 examples
y_tiny = y_train[:20]
best_model, stats = trainer.train(X_tiny, y_tiny, X_tiny, y_tiny,
                                  model, two_layer_net,
                                  num_epochs=200, reg=0.0,
                                  update='sgd', learning_rate_decay=1,
                                  sample_batches = False,
                                  learning_rate=1e-3, verbose=True)
```

The above code:
- take the first 20 examples from CIFAR-10
- turn off regularization (reg = 0.0)
- use simple vanilla ‘sgd’

# فرآیند یادگیری

## شروع آموزش

Lets try to train now…

**Tip**: Make sure that you can overfit very small portion of the training data

```
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
trainer = ClassifierTrainer()
X_tiny = X_train[:20] # take 20 examples
y_tiny = y_train[:20]
best_model, stats = trainer.train(X_tiny, y_tiny, X_tiny, y_tiny,
                                  model, two_layer_net,
                                  num_epochs=200, reg=0.0,
                                  update='sgd', learning_rate_decay=1,
                                  sample_batches = False,
                                  learning_rate=1e-3, verbose=True)
```

```
Finished epoch 1 / 200: cost 2.302603, train: 0.400000, val 0.400000, lr 1.000000e-03
Finished epoch 2 / 200: cost 2.302258, train: 0.450000, val 0.450000, lr 1.000000e-03
Finished epoch 3 / 200: cost 2.301849, train: 0.600000, val 0.600000, lr 1.000000e-03
Finished epoch 4 / 200: cost 2.301196, train: 0.650000, val 0.650000, lr 1.000000e-03
Finished epoch 5 / 200: cost 2.300044, train: 0.650000, val 0.650000, lr 1.000000e-03
Finished epoch 6 / 200: cost 2.297864, train: 0.550000, val 0.550000, lr 1.000000e-03
Finished epoch 7 / 200: cost 2.293595, train: 0.600000, val 0.600000, lr 1.000000e-03
Finished epoch 8 / 200: cost 2.285096, train: 0.550000, val 0.550000, lr 1.000000e-03
Finished epoch 9 / 200: cost 2.268094, train: 0.550000, val 0.550000, lr 1.000000e-03
Finished epoch 10 / 200: cost 2.234787, train: 0.500000, val 0.500000, lr 1.000000e-03
Finished epoch 11 / 200: cost 2.173187, train: 0.500000, val 0.500000, lr 1.000000e-03
Finished epoch 12 / 200: cost 2.076862, train: 0.500000, val 0.500000, lr 1.000000e-03
Finished epoch 13 / 200: cost 1.974090, train: 0.400000, val 0.400000, lr 1.000000e-03
Finished epoch 14 / 200: cost 1.895885, train: 0.400000, val 0.400000, lr 1.000000e-03
Finished epoch 15 / 200: cost 1.820876, train: 0.450000, val 0.450000, lr 1.000000e-03
Finished epoch 16 / 200: cost 1.737430, train: 0.450000, val 0.450000, lr 1.000000e-03
Finished epoch 17 / 200: cost 1.642356, train: 0.500000, val 0.500000, lr 1.000000e-03
Finished epoch 18 / 200: cost 1.535239, train: 0.600000, val 0.600000, lr 1.000000e-03
Finished epoch 19 / 200: cost 1.421527, train: 0.600000, val 0.600000, lr 1.000000e-03
```

Very small loss, train accuracy 1.00, nice!

```
Finished epoch 195 / 200: cost 0.002694, train: 1.000000, val 1.000000, lr 1.000000e-03
Finished epoch 196 / 200: cost 0.002674, train: 1.000000, val 1.000000, lr 1.000000e-03
Finished epoch 197 / 200: cost 0.002655, train: 1.000000, val 1.000000, lr 1.000000e-03
Finished epoch 198 / 200: cost 0.002635, train: 1.000000, val 1.000000, lr 1.000000e-03
Finished epoch 199 / 200: cost 0.002617, train: 1.000000, val 1.000000, lr 1.000000e-03
Finished epoch 200 / 200: cost 0.002597, train: 1.000000, val 1.000000, lr 1.000000e-03
finished optimization. best validation accuracy: 1.000000
```



Ref: http://cs231n.stanford.edu/

# فرآیند یادگیری

شروع با ضریب رگولاریزاسیون کوچک و یافتن نرخ یادگیری مناسب

Lets try to train now…

Start with small regularization and find learning rate that makes the loss go down.

```
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
trainer = ClassifierTrainer()
best_model, stats = trainer.train(X_train, y_train, X_val, y_val,
                                  model, two_layer_net,
                                  num_epochs=10, reg=0.000001,
                                  update='sgd', learning_rate_decay=1,
                                  sample_batches = True,
                                  learning_rate=1e-6, verbose=True)
```

# فرآیند یادگیری

## شروع با ضریب رگولاریزاسیون کوچک و یافتن نرخ یادگیری مناسب

Lets try to train now…

Start with small regularization and find learning rate that makes the loss go down.

```
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
trainer = ClassifierTrainer()
best_model, stats = trainer.train(X_train, y_train, X_val, y_val,
                                  model, two_layer_net,
                                  num_epochs=10, reg=0.000001,
                                  update='sgd', learning_rate_decay=1,
                                  sample_batches = True,
                                  learning_rate=1e-6, verbose=True)
```

```
Finished epoch 1 / 10: cost 2.302576, train: 0.080000, val 0.103000, lr 1.000000e-06
Finished epoch 2 / 10: cost 2.302582, train: 0.121000, val 0.124000, lr 1.000000e-06
Finished epoch 3 / 10: cost 2.302558, train: 0.119000, val 0.138000, lr 1.000000e-06
Finished epoch 4 / 10: cost 2.302519, train: 0.127000, val 0.151000, lr 1.000000e-06
Finished epoch 5 / 10: cost 2.302517, train: 0.158000, val 0.171000, lr 1.000000e-06
Finished epoch 6 / 10: cost 2.302518, train: 0.179000, val 0.172000, lr 1.000000e-06
Finished epoch 7 / 10: cost 2.302466, train: 0.180000, val 0.176000, lr 1.000000e-06
Finished epoch 8 / 10: cost 2.302452, train: 0.175000, val 0.185000, lr 1.000000e-06
Finished epoch 9 / 10: cost 2.302459, train: 0.206000, val 0.192000, lr 1.000000e-06
Finished epoch 10 / 10: cost 2.302420, train: 0.190000, val 0.192000, lr 1.000000e-06
finished optimization. best validation accuracy: 0.192000
```

Loss barely changing

Ref: http://cs231n.stanford.edu/

# فرآیند یادگیری

شروع با ضریب رگولاریزاسیون کوچک و یافتن نرخ یادگیری مناسب

Lets try to train now…

Start with small regularization and find learning rate that makes the loss go down.

**loss not going down:**
learning rate too low

```
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
trainer = ClassifierTrainer()
best_model, stats = trainer.train(X_train, y_train, X_val, y_val,
                                  model, two_layer_net,
                                  num_epochs=10, reg=0.000001,
                                  update='sgd', learning_rate_decay=1,
                                  sample_batches = True,
                                  learning_rate=1e-6, verbose=True)
```

```
Finished epoch 1 / 10: cost 2.302576, train: 0.080000, val 0.103000, lr 1.000000e-06
Finished epoch 2 / 10: cost 2.302582, train: 0.121000, val 0.124000, lr 1.000000e-06
Finished epoch 3 / 10: cost 2.302558, train: 0.119000, val 0.138000, lr 1.000000e-06
Finished epoch 4 / 10: cost 2.302519, train: 0.127000, val 0.151000, lr 1.000000e-06
Finished epoch 5 / 10: cost 2.302517, train: 0.158000, val 0.171000, lr 1.000000e-06
Finished epoch 6 / 10: cost 2.302518, train: 0.179000, val 0.172000, lr 1.000000e-06
Finished epoch 7 / 10: cost 2.302466, train: 0.180000, val 0.176000, lr 1.000000e-06
Finished epoch 8 / 10: cost 2.302452, train: 0.175000, val 0.185000, lr 1.000000e-06
Finished epoch 9 / 10: cost 2.302459, train: 0.206000, val 0.192000, lr 1.000000e-06
Finished epoch 10 / 10: cost 2.302420, train: 0.190000, val 0.192000, lr 1.000000e-06
finished optimization. best validation accuracy: 0.192000
```

Loss barely changing: Learning rate is probably too low



Ref: http://cs231n.stanford.edu/

# فرآیند یادگیری

شروع با ضریب رگولاریزاسیون کوچک و یافتن نرخ یادگیری مناسب

Lets try to train now…

Start with small regularization and find learning rate that makes the loss go down.

**loss not going down:**
learning rate too low

```
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
trainer = ClassifierTrainer()
best_model, stats = trainer.train(X_train, y_train, X_val, y_val,
                                  model, two_layer_net,
                                  num_epochs=10, reg=0.000001,
                                  update='sgd', learning_rate_decay=1,
                                  sample_batches = True,
                                  learning_rate=1e-6, verbose=True)
Finished epoch 1 / 10: cost 2.302576, train: 0.080000, val 0.103000, lr 1.000000e-06
Finished epoch 2 / 10: cost 2.302582, train: 0.121000, val 0.124000, lr 1.000000e-06
Finished epoch 3 / 10: cost 2.302558, train: 0.119000, val 0.138000, lr 1.000000e-06
Finished epoch 4 / 10: cost 2.302519, train: 0.127000, val 0.151000, lr 1.000000e-06
Finished epoch 5 / 10: cost 2.302517, train: 0.158000, val 0.171000, lr 1.000000e-06
Finished epoch 6 / 10: cost 2.302518, train: 0.179000, val 0.172000, lr 1.000000e-06
Finished epoch 7 / 10: cost 2.302466, train: 0.180000, val 0.176000, lr 1.000000e-06
Finished epoch 8 / 10: cost 2.302452, train: 0.175000, val 0.185000, lr 1.000000e-06
Finished epoch 9 / 10: cost 2.302459, train: 0.206000, val 0.192000, lr 1.000000e-06
Finished epoch 10 / 10: cost 2.302420, train: 0.190000, val 0.192000, lr 1.000000e-06
finished optimization. best validation accuracy: 0.192000
```

Loss barely changing: Learning rate is probably too low

Notice train/val accuracy goes to 20% though, what's up with that? (remember this is softmax)

# فرآیند یادگیری

شروع با ضریب رگولاریزاسیون کوچک و یافتن نرخ یادگیری مناسب

Lets try to train now…

Start with small regularization and find learning rate that makes the loss go down.

**loss not going down:**
learning rate too low

```
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
trainer = ClassifierTrainer()
best_model, stats = trainer.train(X_train, y_train, X_val, y_val,
                                  model, two_layer_net,
                                  num_epochs=10, reg=0.000001,
                                  update='sgd', learning_rate_decay=1,
                                  sample_batches = True,
```

Now let's try learning rate 1e6.

# فرآیند یادگیری

## شروع با ضریب رگولاریزاسیون کوچک و یافتن نرخ یادگیری مناسب

Lets try to train now…

Start with small regularization and find learning rate that makes the loss go down.

**loss not going down:**
learning rate too low
**loss exploding:**
learning rate too high

```
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
trainer = ClassifierTrainer()
best_model, stats = trainer.train(X_train, y_train, X_val, y_val,
                                  model, two_layer_net,
                                  num_epochs=10, reg=0.000001,
                                  update='sgd', learning_rate_decay=1,
                                  sample_batches = True,
                                  learning_rate=1e6, verbose=True)
```

```
/home/karpathy/cs231n/code/cs231n/classifiers/neural_net.py:50: RuntimeWarning: divide by zero en
countered in log
  data_loss = -np.sum(np.log(probs[range(N), y])) / N
/home/karpathy/cs231n/code/cs231n/classifiers/neural_net.py:48: RuntimeWarning: invalid value enc
ountered in subtract
  probs = np.exp(scores - np.max(scores, axis=1, keepdims=True))
Finished epoch 1 / 10: cost nan, train: 0.091000, val 0.087000, lr 1.000000e+06
Finished epoch 2 / 10: cost nan, train: 0.095000, val 0.087000, lr 1.000000e+06
Finished epoch 3 / 10: cost nan, train: 0.100000, val 0.087000, lr 1.000000e+06
```

cost: NaN almost always means high learning rate...



Ref: http://cs231n.stanford.edu/

# فرآیند یادگیری

شروع با ضریب رگولاریزاسیون کوچک و یافتن نرخ یادگیری مناسب

Lets try to train now…

Start with small regularization and find learning rate that makes the loss go down.

**loss not going down:**
learning rate too low
**loss exploding:**
learning rate too high

```
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
trainer = ClassifierTrainer()
best_model, stats = trainer.train(X_train, y_train, X_val, y_val,
                                  model, two_layer_net,
                                  num_epochs=10, reg=0.000001,
                                  update='sgd', learning_rate_decay=1,
                                  sample_batches = True,
                                  learning_rate=3e-3, verbose=True)
```

```
Finished epoch 1 / 10: cost 2.186654, train: 0.308000, val 0.306000, lr 3.000000e-03
Finished epoch 2 / 10: cost 2.176230, train: 0.330000, val 0.350000, lr 3.000000e-03
Finished epoch 3 / 10: cost 1.942257, train: 0.376000, val 0.352000, lr 3.000000e-03
Finished epoch 4 / 10: cost 1.827868, train: 0.329000, val 0.310000, lr 3.000000e-03
Finished epoch 5 / 10: cost inf, train: 0.128000, val 0.128000, lr 3.000000e-03
Finished epoch 6 / 10: cost inf, train: 0.144000, val 0.147000, lr 3.000000e-03
```

3e-3 is still too high. Cost explodes….

=> Rough range for learning rate we should be cross-validating is somewhere [1e-3 … 1e-5]



کاظم فولادی

Ref: http://cs231n.stanford.edu/

ملاحظاتی در
آموزش شبکه‌های عصبی عمیق

# ۶

# بهینه‌سازی هایپر-پارامترها

استراتژی اعتبارسنجی تقاطعی

# Cross-validation strategy

## **coarse -> fine** cross-validation in stages

**First stage**: only a few epochs to get rough idea of what params work
**Second stage**: longer running time, finer search
… (repeat as necessary)

Tip for detecting explosions in the solver:
If the cost is ever > 3 * original cost, break out early

# بهینه‌سازی هایپر-پارامترها

استراتژی اعتبارسنجی تقاطعی: مثال (۱ از ۳)

For example: run coarse search  for 5 epochs

```
max_count = 100
for count in xrange(max_count):
      reg = 10**uniform(-5, 5)
      lr = 10**uniform(-3, -6)

      trainer = ClassifierTrainer()
      model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
      trainer = ClassifierTrainer()
      best_model_local, stats = trainer.train(X_train, y_train, X_val, y_val,
                                   model, two_layer_net,
                                   num_epochs=5, reg=reg,
                                   update='momentum', learning_rate_decay=0.9,
                                   sample_batches = True, batch_size = 100,
                                   learning_rate=lr, verbose=False)
```

note it's best to optimize in log space!

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

nice →





Ref: http://cs231n.stanford.edu/

# بهینه‌سازی هایپر-پارامترها

## استراتژی اعتبارسنجی تقاطعی: مثال (۲ از ۳)

# Now run finer search...

```
max_count = 100
for count in xrange(max_count):
      reg = 10**uniform(-5, 5)
      lr = 10**uniform(-3, -6)
```

adjust range →

```
max_count = 100
for count in xrange(max_count):
      reg = 10**uniform(-4, 0)
      lr = 10**uniform(-3, -4)
```

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

**53%** - relatively good for a 2-layer neural net with 50 hidden neurons.

# بهینه‌سازی هایپر-پارامترها

## استراتژی اعتبارسنجی تقاطعی: مثال (۳ از ۳)

# Now run finer search...

```
max_count = 100
for count in xrange(max_count):
      reg = 10**uniform(-5, 5)
      lr = 10**uniform(-3, -6)
```

adjust range →

```
max_count = 100
for count in xrange(max_count):
      reg = 10**uniform(-4, 0)
      lr = 10**uniform(-3, -4)
```

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

**53%** - relatively good for a 2-layer neural net with 50 hidden neurons.

But this best cross-validation result is worrying. Why?

# بهینه‌سازی هایپر-پارامترها

جستجوی تصادفی در برابر جستجوی توری برای یافتن هایپر-پارامترهای بهینه

## Random Search vs. Grid Search

*Random Search for Hyper-Parameter Optimization*
Bergstra and Bengio, 2012

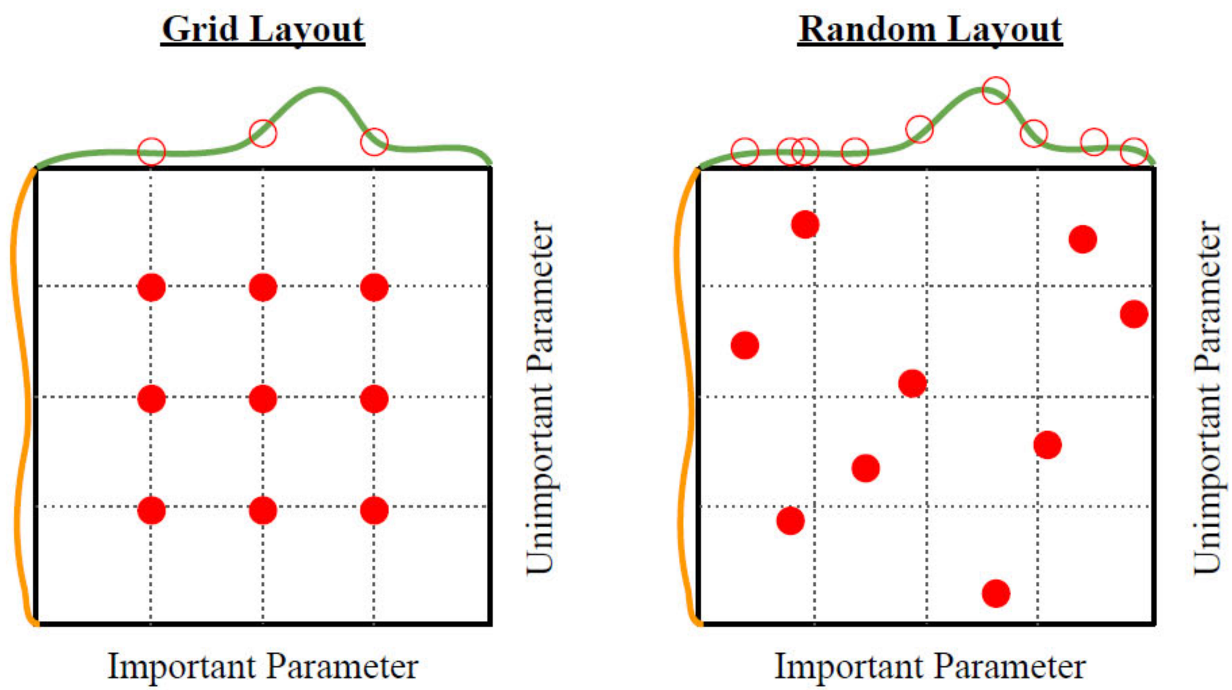



Illustration of Bergstra et al., 2012 by Shayne Longpre, copyright CS231n 2017



Ref: http://cs231n.stanford.edu/

# بهینه‌سازی هایپر-پارامترها

**Hyperparameters to play with:**

- network architecture
- learning rate, its decay schedule, update type
- regularization (L2/Dropout strength)

neural networks practitioner
music = loss function

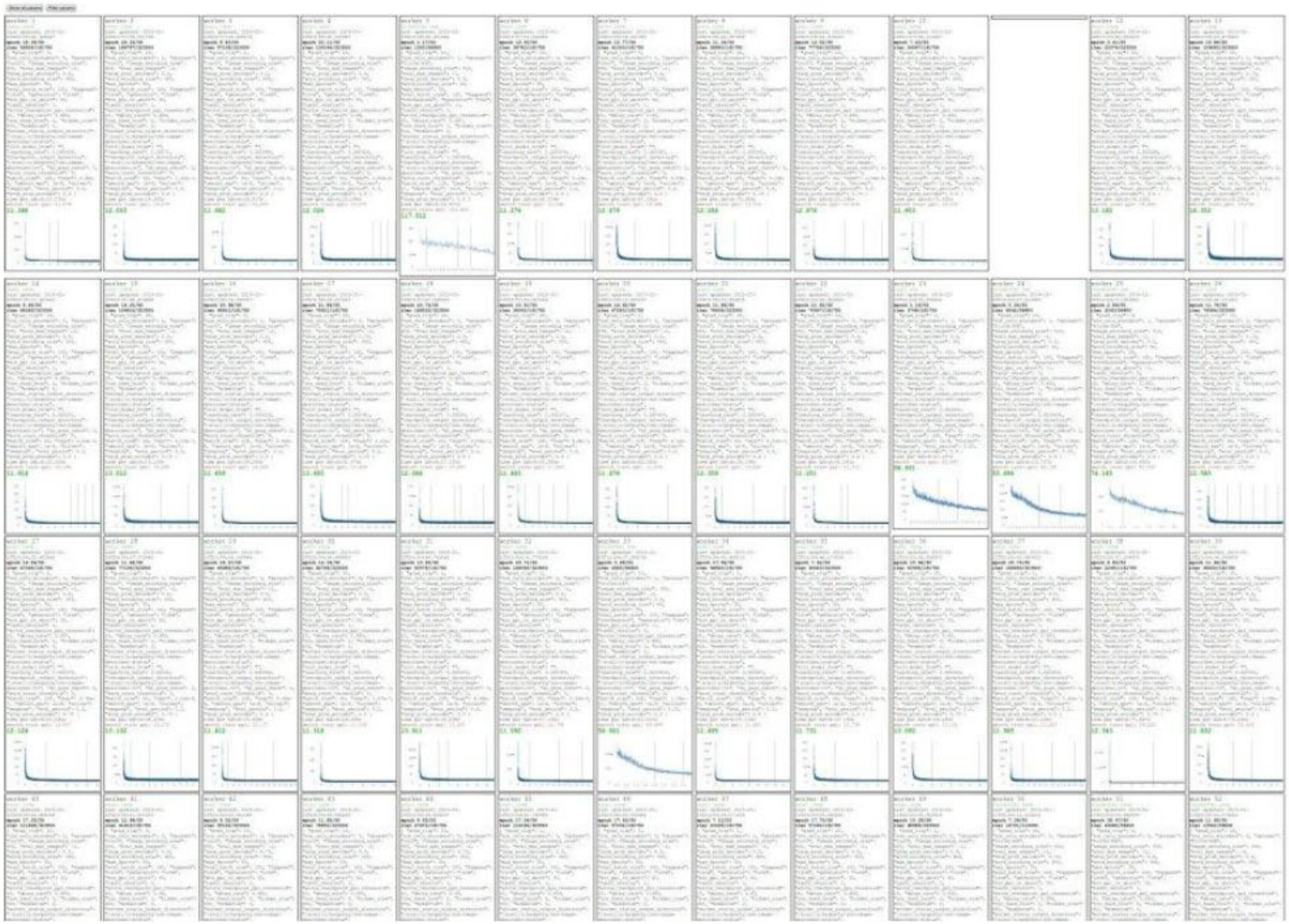




Ref: http://cs231n.stanford.edu/

# بهینه‌سازی هایپر-پارامترها

Cross-validation
“command center”



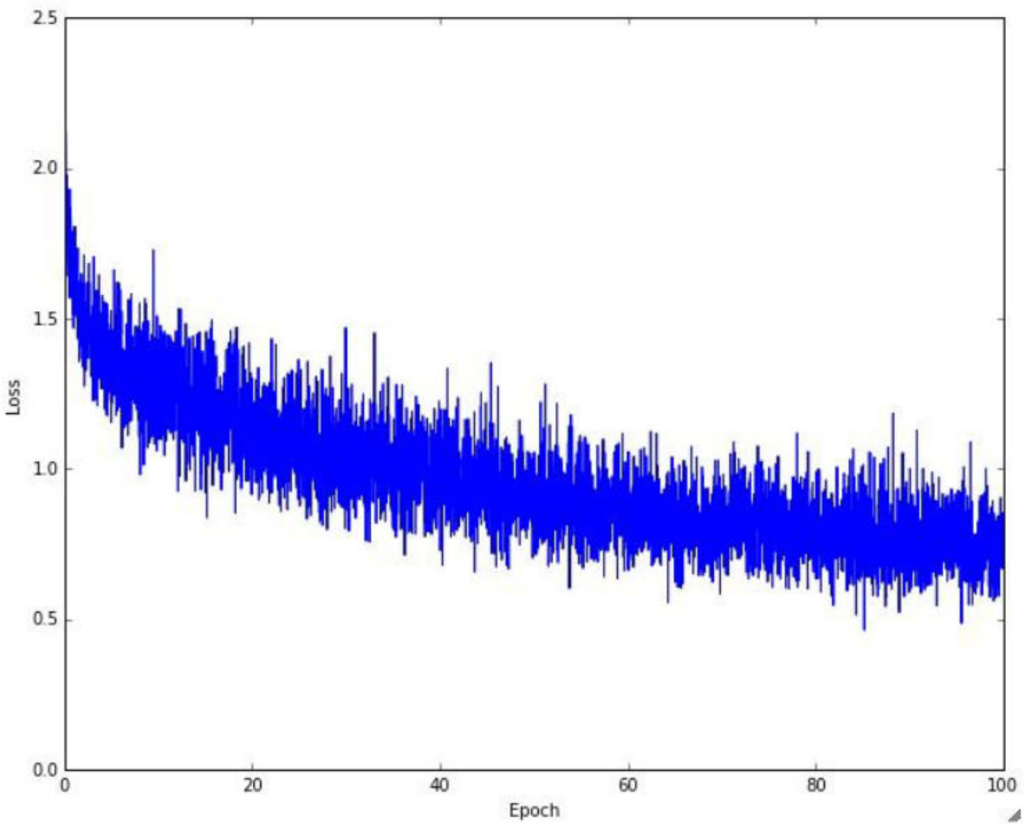


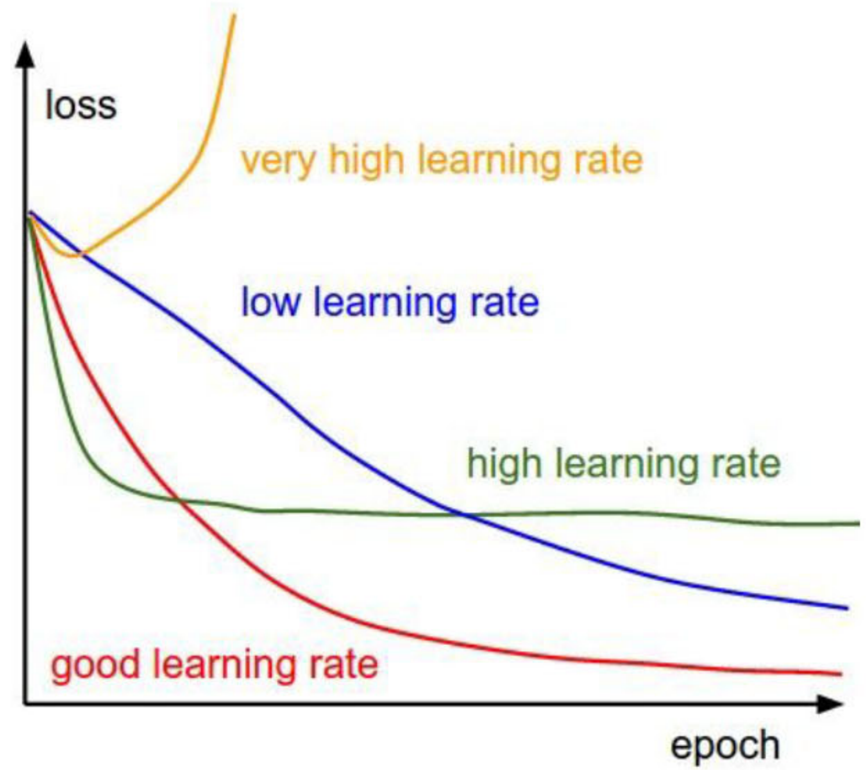
Ref: http://cs231n.stanford.edu/

# بهینه‌سازی هایپر-پارامترها

## نظارت بر منحنی اتلاف

## Monitor and visualize the loss curve



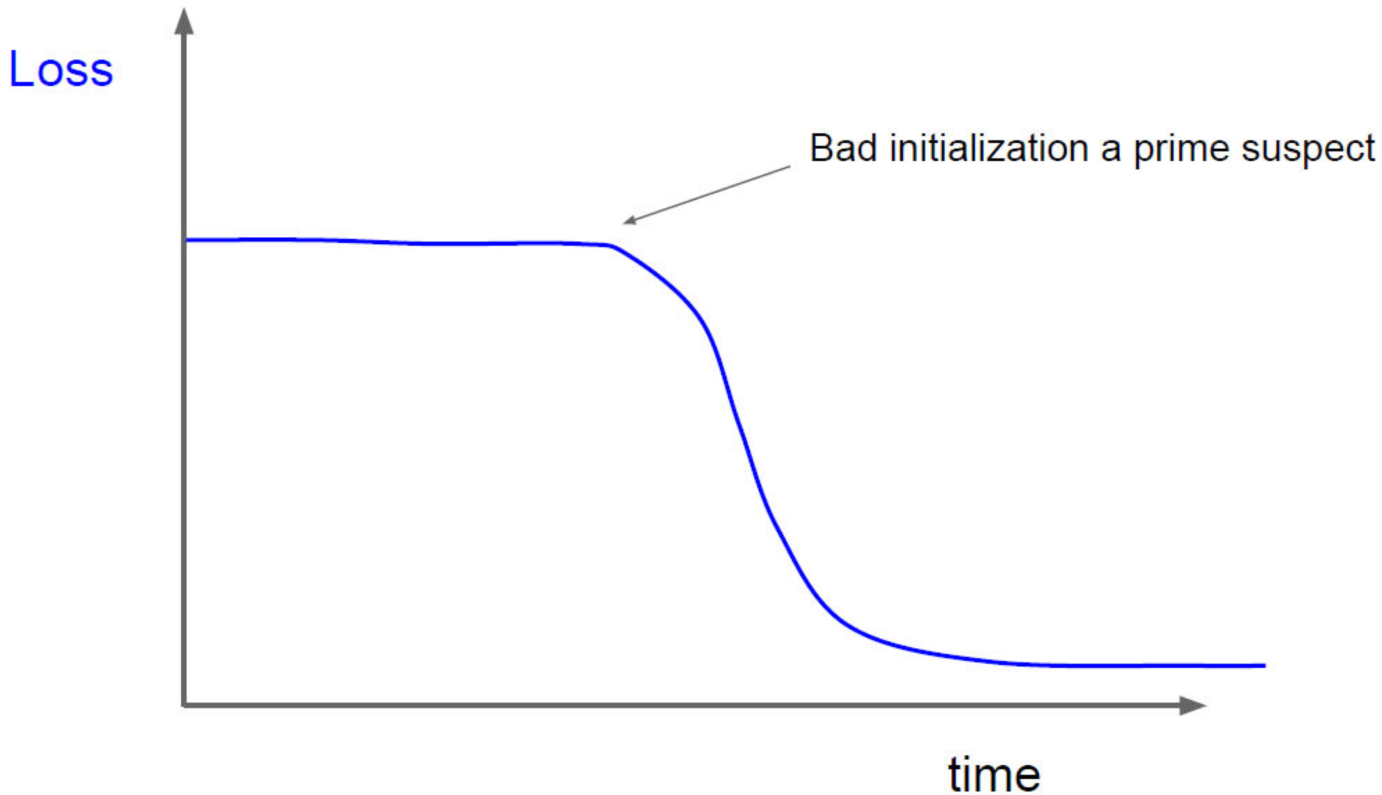






Ref: http://cs231n.stanford.edu/

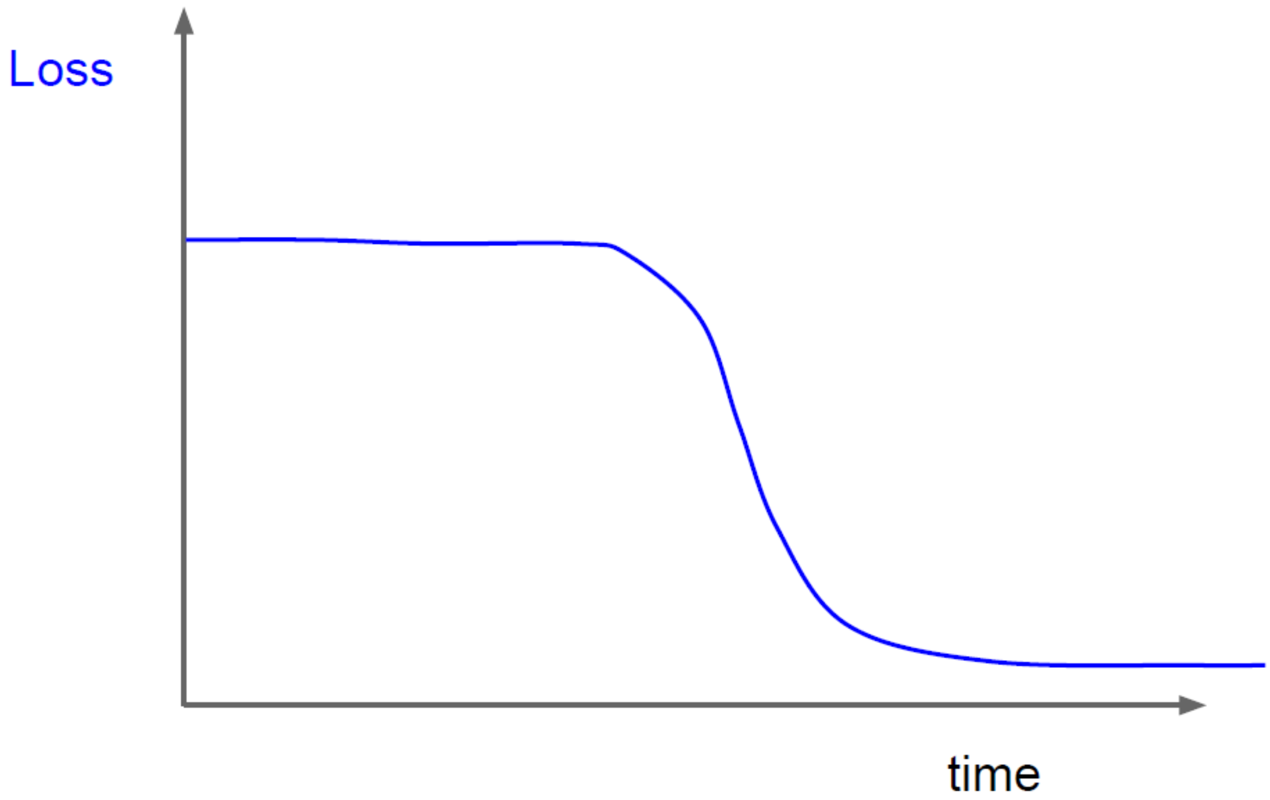
# بهینه‌سازی هایپر-پارامترها

نظارت بر منحنی اتلاف



# بهینه‌سازی هایپر-پارامترها

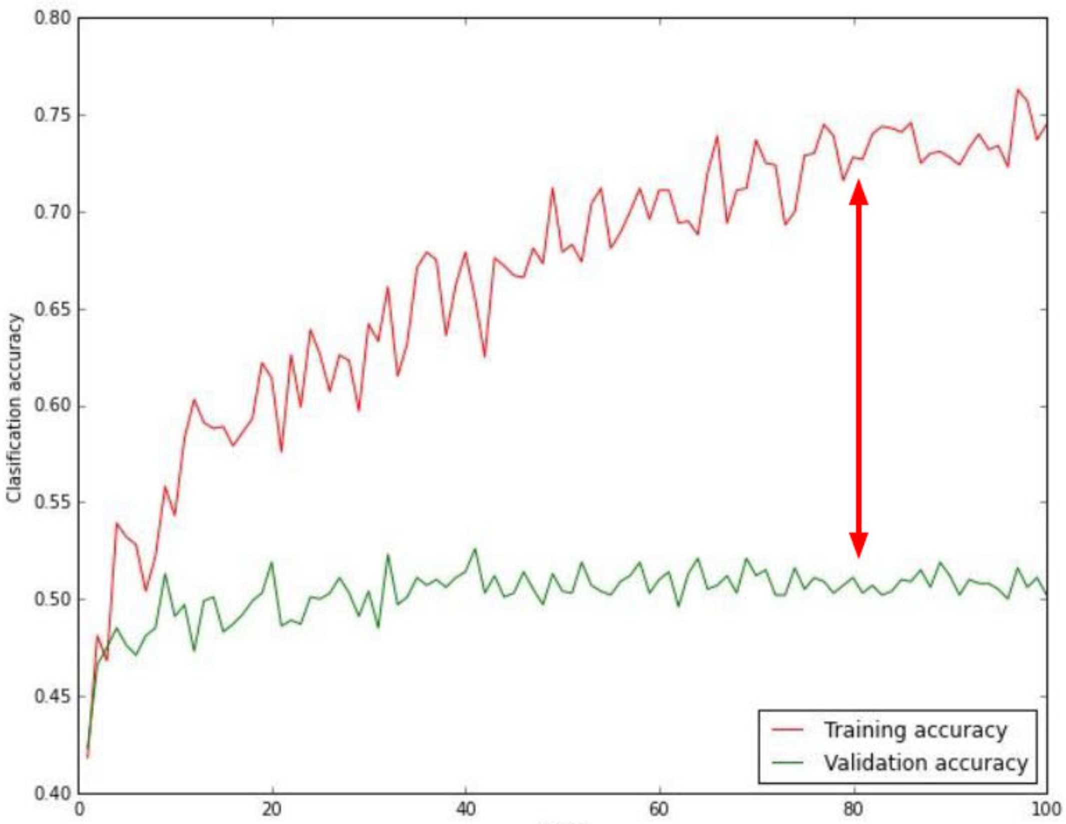
## نظارت بر منحنی اتلاف



# بهینه‌سازی هایپر-پارامترها

## نظارت بر منحنی اتلاف

Monitor and visualize the accuracy:




big gap = overfitting
=> increase regularization strength?

no gap
=> increase model capacity?



Ref: http://cs231n.stanford.edu/

# بهینه‌سازی هایپر-پارامترها

## دنبال کردن نسبت به‌هنگام‌سازی وزن‌ها به بزرگی وزن‌ها

Track the ratio of weight updates / weight magnitudes:

```
# assume parameter vector W and its gradient vector dW
param_scale = np.linalg.norm(W.ravel())
update = -learning_rate*dW # simple SGD update
update_scale = np.linalg.norm(update.ravel())
W += update # the actual update
print update_scale / param_scale # want ~1e-3
```

ratio between the updates and values: ~ 0.0002 / 0.02 = 0.01 (about okay)
**want this to be somewhere around 0.001 or so**

# خلاصه

چند توصیه‌ی کاربردی

- Activation Functions (use ReLU)
- Data Preprocessing (images: subtract mean)
- Weight Initialization (use Xavier/He init)
- Batch Normalization (use)
- Babysitting the Learning process
- Hyperparameter Optimization
  (random sample hyperparams, in log space when appropriate)

ملاحظاتی در
آموزش شبکه‌های عصبی عمیق

# ۷

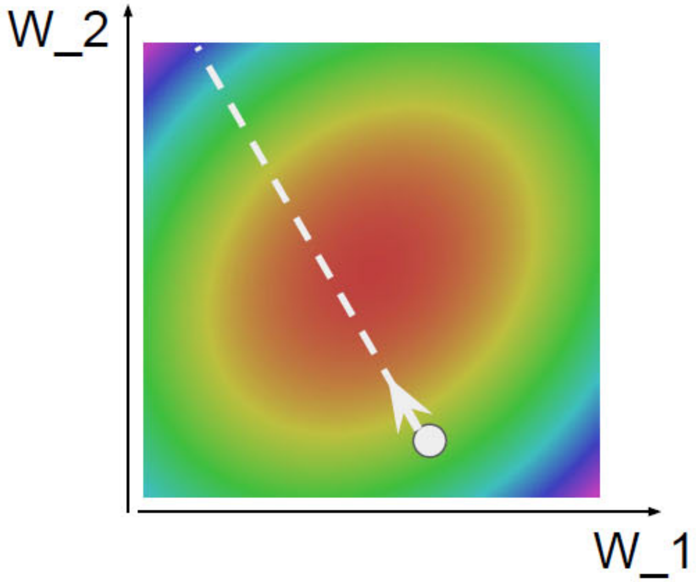
# بهینه‌سازی

بهینه‌سازی

# Optimization

```
# Vanilla Gradient Descent

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



# بهینه‌سازی

## کاهش گرادیانی دسته‌ای

## Batch Gradient Descent

**Algorithm 1** Batch Gradient Descent at Iteration $k$

**Require:** Learning rate $\epsilon_k$
**Require:** Initial Parameter $\theta$

**while** stopping criteria not met **do**
    Compute gradient estimate over $N$ examples:
    $\hat{\mathbf{g}} \leftarrow +\frac{1}{N}\nabla_\theta \sum_i L(f(\mathbf{x}^{(i)};\theta), \mathbf{y}^{(i)})$
    Apply Update: $\theta \leftarrow \theta - \epsilon\hat{\mathbf{g}}$
**end while**

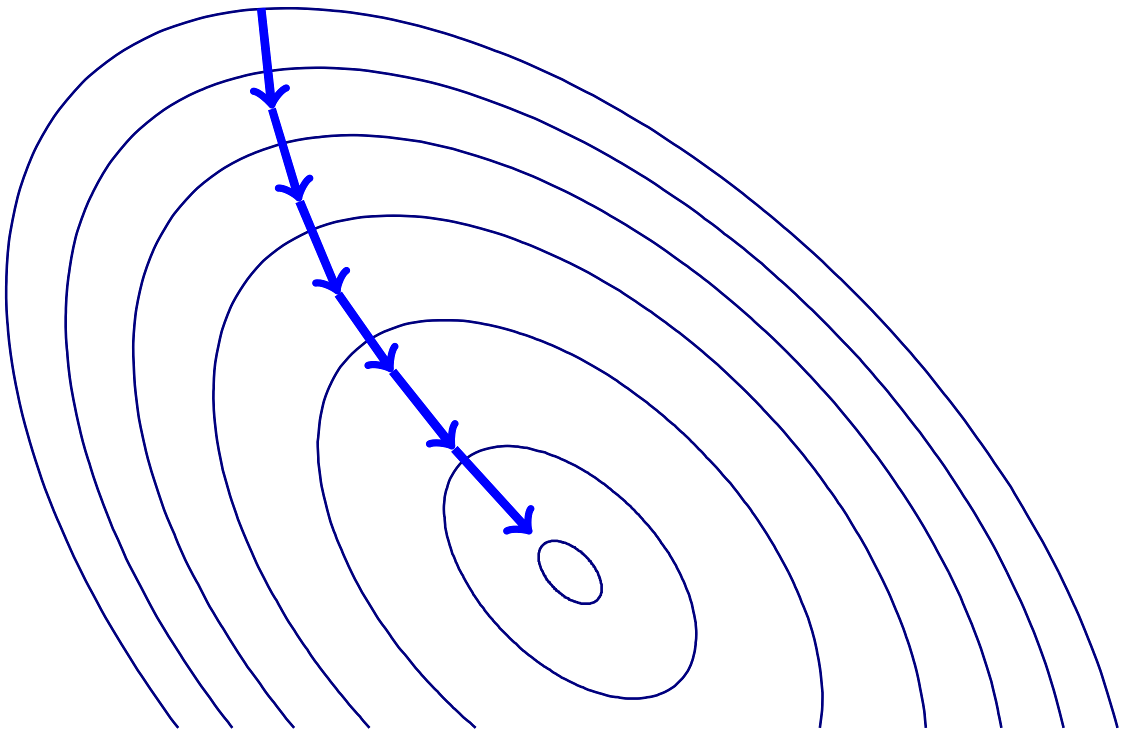


Ref: https://ttic.uchicago.edu/~shubhendu/Pages/CMSC35246.html

# بهینه‌سازی

## کاهش گرادیانی

## Gradient Descent



Ref: https://ttic.uchicago.edu/~shubhendu/Pages/CMSC35246.html

# بهینه‌سازی

## کاهش گرادیانی تصادفی

## Stochastic Gradient Descent

**Algorithm 2** Stochastic Gradient Descent at Iteration $k$

**Require:** Learning rate $\epsilon_k$
**Require:** Initial Parameter $\theta$
**while** stopping criteria not met **do**
    Sample example $(\mathbf{x}^{(i)}, \mathbf{y}^{(i)})$ from training set
    Compute gradient estimate:
    $\hat{\mathbf{g}} \leftarrow +\nabla_\theta L(f(\mathbf{x}^{(i)}; \theta), \mathbf{y}^{(i)})$
    Apply Update: $\theta \leftarrow \theta - \epsilon \hat{\mathbf{g}}$
**end while**

- $\epsilon_k$ is learning rate at step $k$
- Sufficient condition to guarantee convergence:

$$\sum_{k=1}^{\infty} \epsilon_k = \infty \text{ and } \sum_{k=1}^{\infty} \epsilon_k^2 < \infty$$



Ref: https://ttic.uchicago.edu/~shubhendu/Pages/CMSC35246.html

## Learning Rate Schedule

- In practice the learning rate is decayed linearly till iteration $\tau$

$$\epsilon_k = (1-\alpha)\epsilon_0 + \alpha\epsilon_\tau \text{ with } \alpha = \frac{k}{\tau}$$

- $\tau$ is usually set to the number of iterations needed for a large number of passes through the data
- $\epsilon_\tau$ should roughly be set to $1\%$ of $\epsilon_0$
- How to set $\epsilon_0$?



Ref: https://ttic.uchicago.edu/~shubhendu/Pages/CMSC35246.html

# بهینه‌سازی

مینی‌بچ سازی

## MINIBATCHING

- Potential Problem: Gradient estimates can be very noisy
- Obvious Solution: Use larger mini-batches
- Advantage: Computation time per update does not depend on number of training examples $N$
- This allows convergence on extremely large datasets
- See: Large Scale Learning with Stochastic Gradient Descent by Leon Bottou

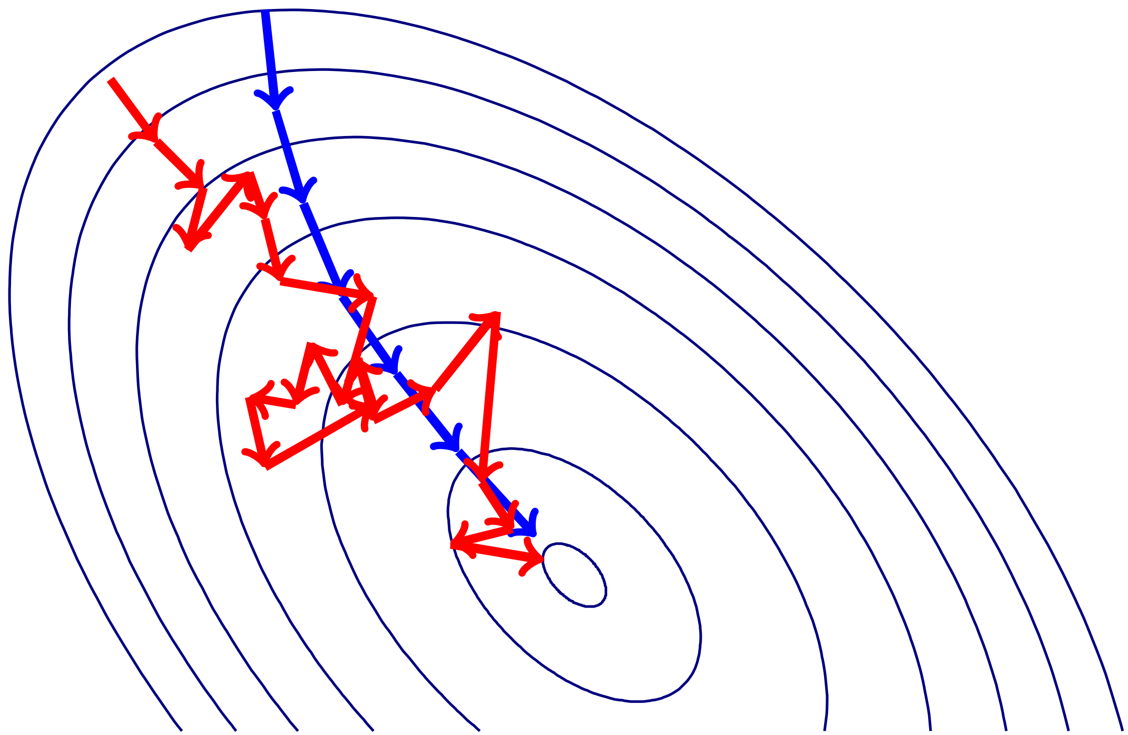


Ref: https://ttic.uchicago.edu/~shubhendu/Pages/CMSC35246.html

# بهینه‌سازی

## کاهش گرادیانی اتفاقی

### STOCHASTIC GRADIENT DESCENT





Ref: https://ttic.uchicago.edu/~shubhendu/Pages/CMSC35246.html

# بهینه‌سازی

## کاهش گرادیانی دسته‌ای در برابر کاهش گرادیانی اتفاقی

- Batch Gradient Descent:

$$\hat{\mathbf{g}} \leftarrow +\frac{1}{N}\nabla_\theta \sum_i L(f(\mathbf{x}^{(i)};\theta), \mathbf{y}^{(i)})$$

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

- SGD:

$$\hat{\mathbf{g}} \leftarrow +\nabla_\theta L(f(\mathbf{x}^{(i)};\theta), \mathbf{y}^{(i)})$$

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

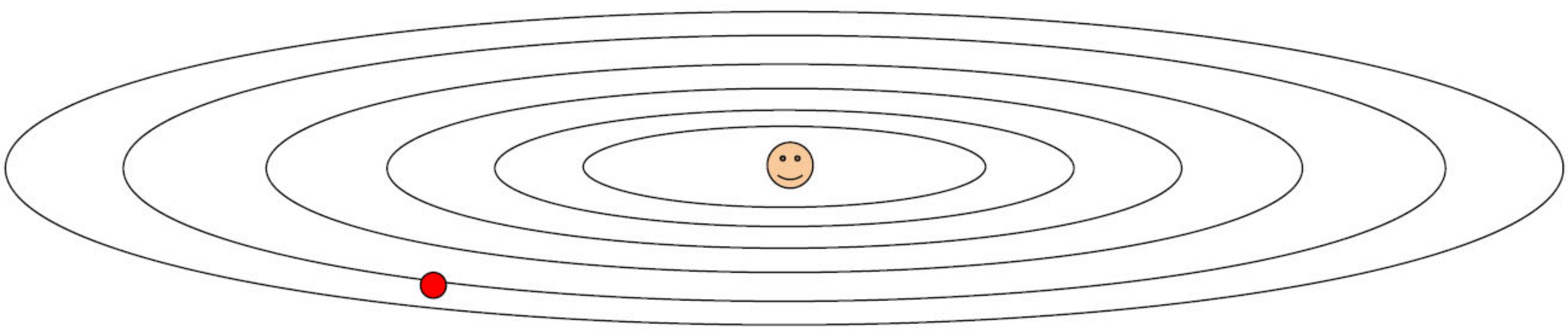


Ref: https://ttic.uchicago.edu/~shubhendu/Pages/CMSC35246.html

مشکلات کاهش گرادیانی اتفاقی

# Optimization: Problems with SGD

What if loss changes quickly in one direction and slowly in another?
What does gradient descent do?



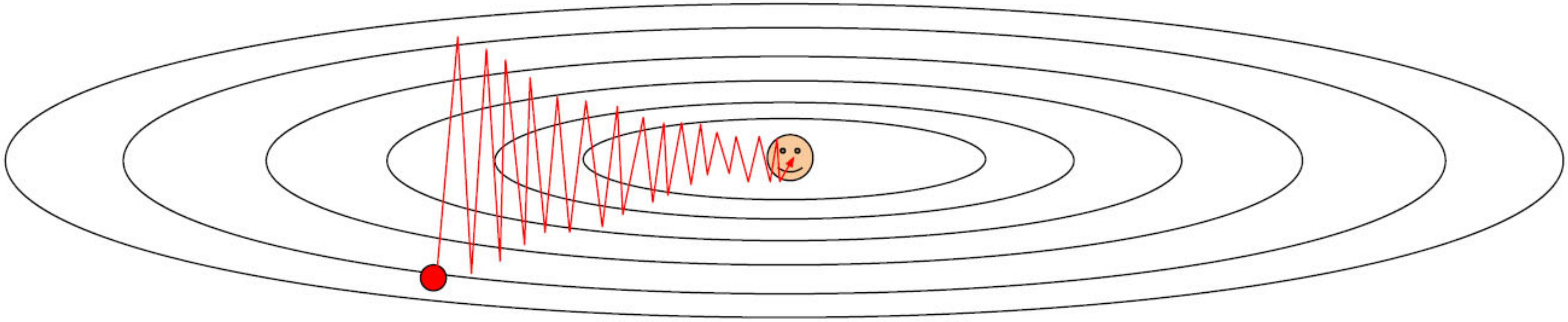
Loss function has high **condition number**: ratio of largest to smallest singular value of the Hessian matrix is large



Ref: http://cs231n.stanford.edu/

# Optimization: Problems with SGD

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

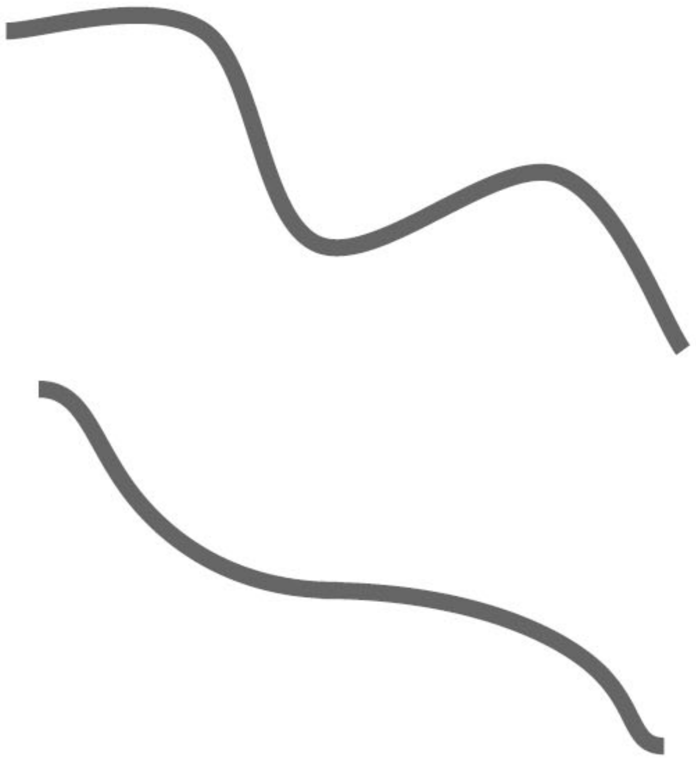


Loss function has high **condition number**: ratio of largest to smallest singular value of the Hessian matrix is large

مشکلات کاهش گرادیانی اتفاقی

# Optimization: Problems with SGD

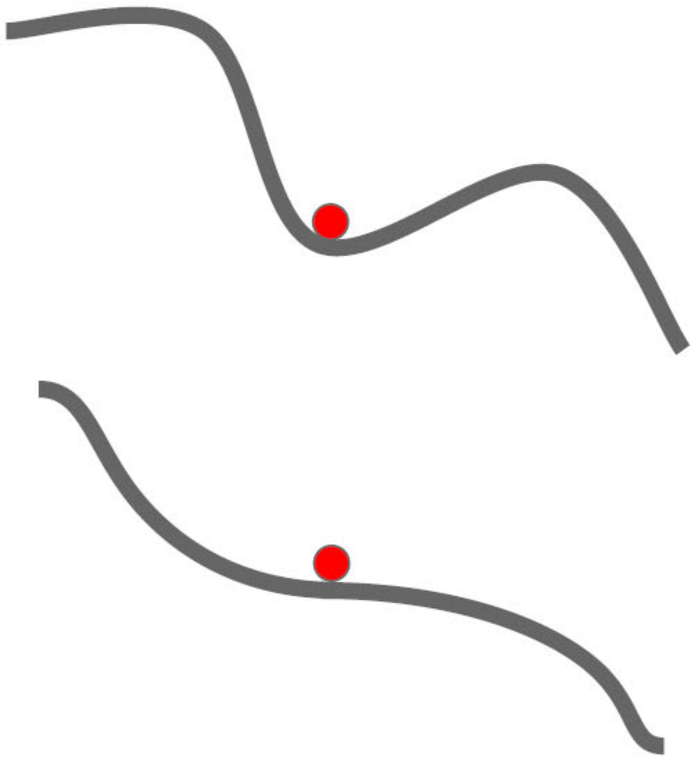
What if the loss function has a **local minima** or **saddle point**?



# Optimization: Problems with SGD

What if the loss function has a **local minima** or **saddle point**?

Zero gradient, gradient descent gets stuck





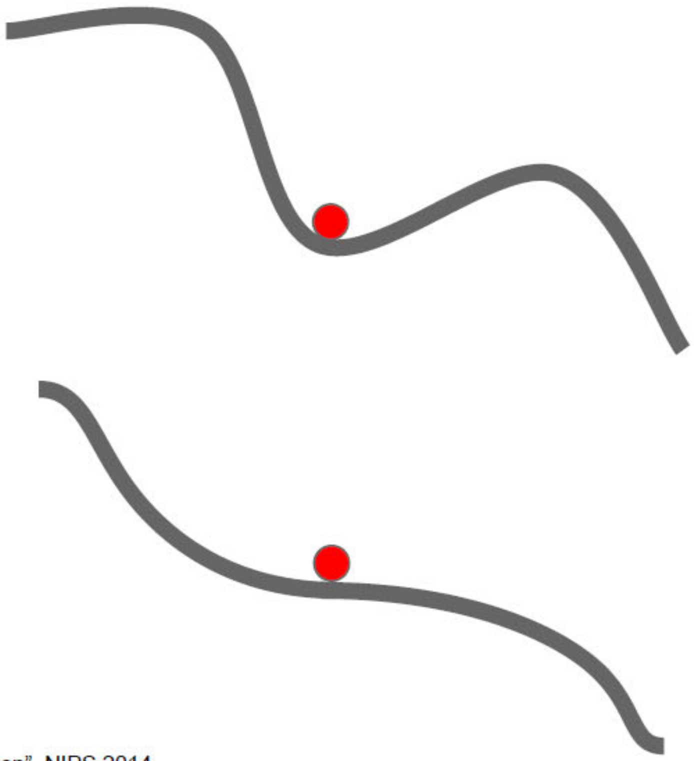
Ref: http://cs231n.stanford.edu/

مشکلات کاهش گرادیانی اتفاقی

# Optimization: Problems with SGD

What if the loss function has a **local minima** or **saddle point**?

Saddle points much more common in high dimension



Dauphin et al, "Identifying and attacking the saddle point problem in high-dimensional non-convex optimization", NIPS 2014

# Optimization: Problems with SGD

Our gradients come from minibatches so they can be noisy!

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

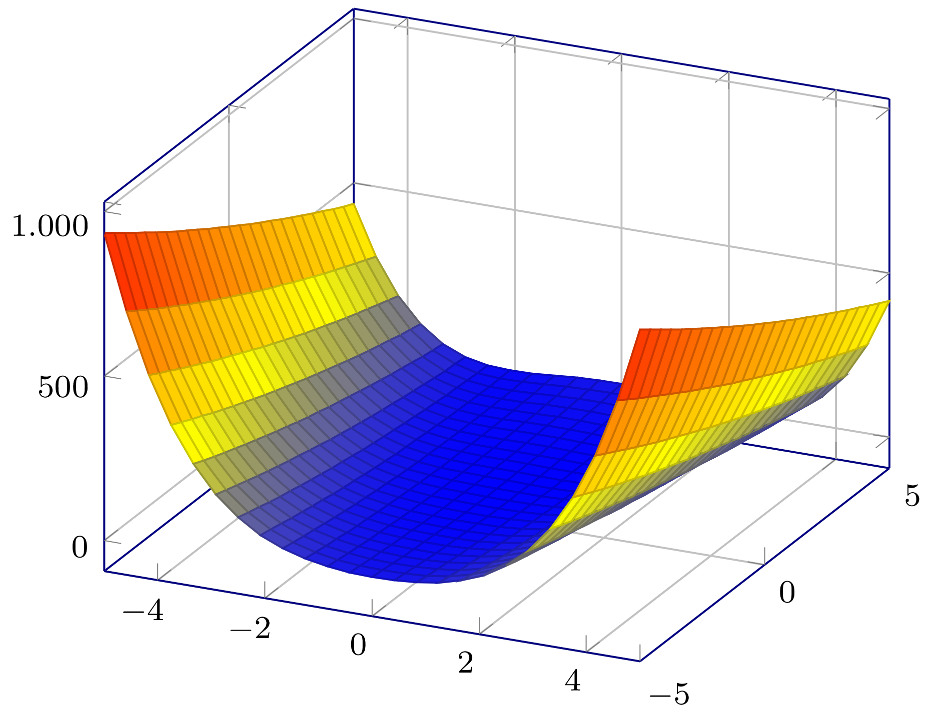
$$\nabla_W L(W) = \frac{1}{N}\sum_{i=1}^{N} \nabla_W L_i(x_i, y_i, W)$$

## MOMENTUM

- The Momentum method is a method to accelerate learning using SGD
- In particular SGD suffers in the following scenarios:
  - Error surface has high curvature
  - We get small but consistent gradients
  - The gradients are very noisy

## مومنتوم (تکانه)

## MOMENTUM






Ref: https://ttic.uchicago.edu/~shubhendu/Pages/CMSC35246.html

## MOMENTUM

- How do we try and solve this problem?
- Introduce a new variable $\mathbf{v}$, the velocity
- We think of $\mathbf{v}$ as the direction and speed by which the parameters move as the learning dynamics progresses
- The velocity is an exponentially decaying moving average of the negative gradients

$$\mathbf{v} \leftarrow \alpha \mathbf{v} - \epsilon \nabla_\theta \left( L(f(\mathbf{x}^{(i)}; \theta), \mathbf{y}^{(i)}) \right)$$

- $\alpha \in [0, 1)$Update rule: $\theta \leftarrow \theta + \mathbf{v}$


Ref: https://ttic.uchicago.edu/~shubhendu/Pages/CMSC35246.html

## MOMENTUM

- Let's look at the velocity term:

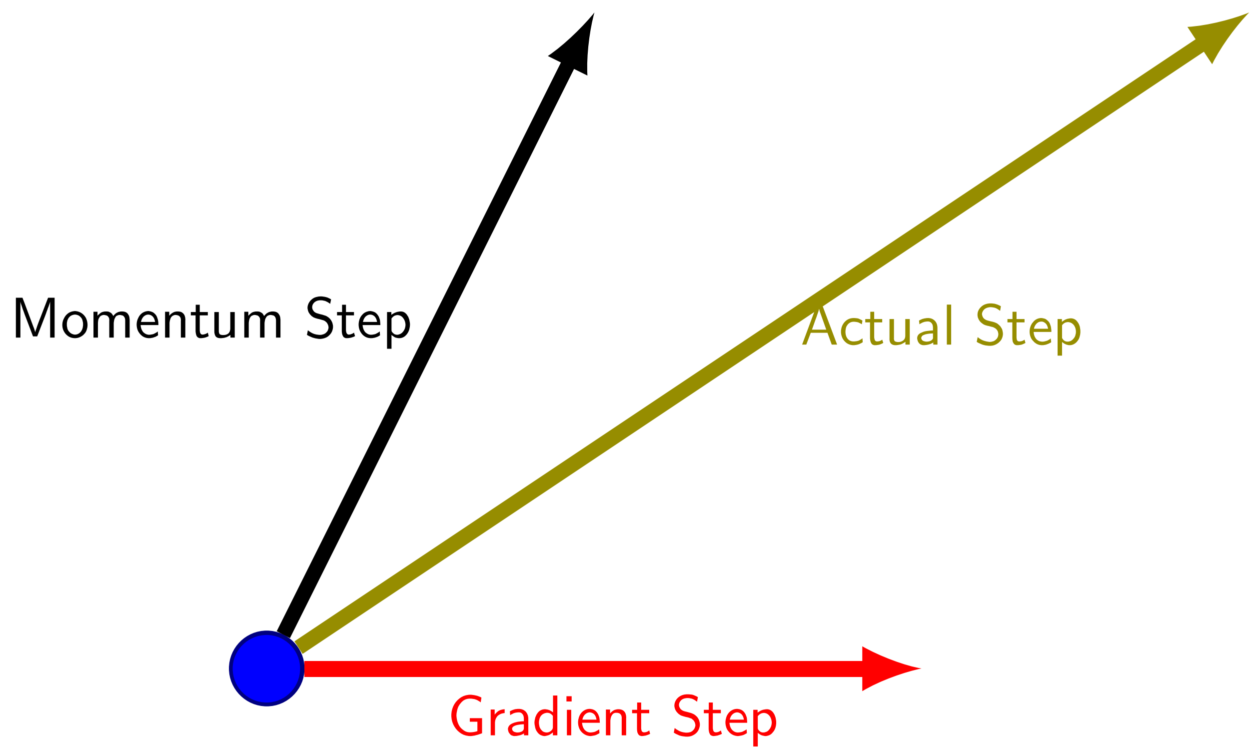
$$\mathbf{v} \leftarrow \alpha \mathbf{v} - \epsilon \nabla_\theta \left( L(f(\mathbf{x}^{(i)};\theta), \mathbf{y}^{(i)}) \right)$$

- The velocity accumulates the previous gradients
- What is the role of $\alpha$?
  - If $\alpha$ is larger than $\epsilon$ the current update is more affected by the previous gradients
  - Usually values for $\alpha$ are set high $\approx 0.8, 0.9$

# بهینه‌سازی

مومنتوم (تکانه)

## MOMENTUM






Ref: https://ttic.uchicago.edu/~shubhendu/Pages/CMSC35246.html

## Momentum: Step Sizes

- In SGD, the step size was the norm of the gradient scaled by the learning rate $\epsilon\|\mathbf{g}\|$. Why?
- While using momentum, the step size will also depend on the norm and alignment of a sequence of gradients
- For example, if at each step we observed $\mathbf{g}$, the step size would be (exercise!):

$$\epsilon\frac{\|\mathbf{g}\|}{1-\alpha}$$

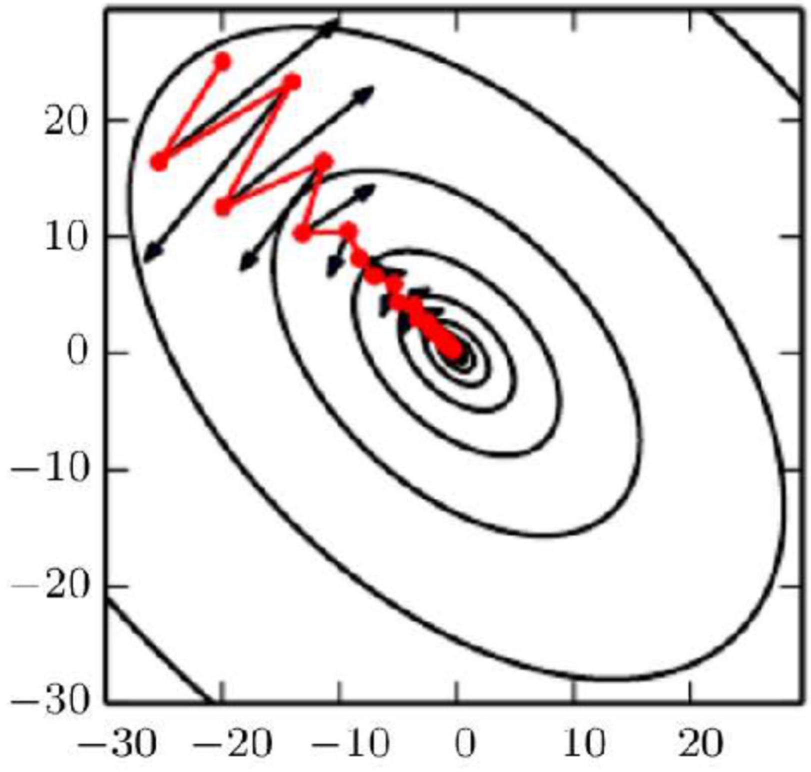
- If $\alpha = 0.9 \implies$ multiply the maximum speed by 10 relative to the current gradient direction


Ref: https://ttic.uchicago.edu/~shubhendu/Pages/CMSC35246.html

# بهینه‌سازی

موممنتوم (تکانه)

## MOMENTUM






Ref: https://ttic.uchicago.edu/~shubhendu/Pages/CMSC35246.html

# بهینه‌سازی

## کاهش گرادیانی اتفاقی با مومنتوم

### SGD WITH MOMENTUM

**Algorithm 2** Stochastic Gradient Descent with Momentum

**Require:** Learning rate $\epsilon_k$
**Require:** Momentum Parameter $\alpha$
**Require:** Initial Parameter $\theta$
**Require:** Initial Velocity $\mathbf{v}$

**while** stopping criteria not met **do**
  Sample example $(\mathbf{x}^{(i)}, \mathbf{y}^{(i)})$ from training set
  Compute gradient estimate:
  $\hat{\mathbf{g}} \leftarrow +\nabla_\theta L(f(\mathbf{x}^{(i)}; \theta), \mathbf{y}^{(i)})$
  Compute the velocity update:
  $\mathbf{v} \leftarrow \alpha \mathbf{v} - \epsilon \hat{\mathbf{g}}$
  Apply Update: $\theta \leftarrow \theta + \mathbf{v}$
**end while**




Ref: https://ttic.uchicago.edu/~shubhendu/Pages/CMSC35246.html

# SGD + Momentum

## SGD

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

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

## SGD+Momentum

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

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

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

Sutskever et al, “On the importance of initialization and momentum in deep learning”, ICML 2013

# بهینه‌سازی

کاهش گرادیانی اتفاقی + مومنتوم

# SGD + Momentum

## SGD+Momentum

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

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

## SGD+Momentum

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

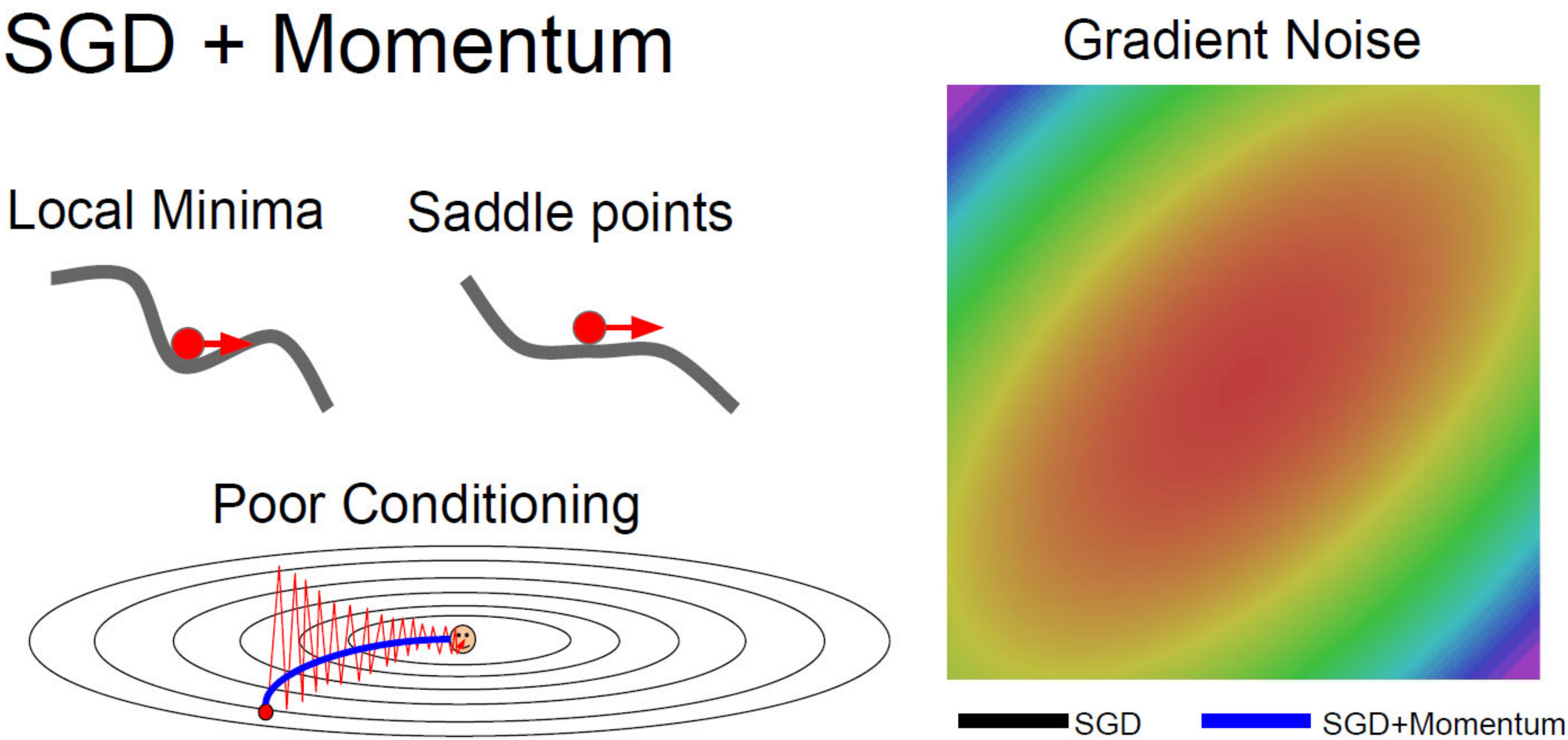
```
vx = 0
while True:
  dx = compute_gradient(x)
  vx = rho * vx + dx
  x -= learning_rate * vx
```

You may see SGD+Momentum formulated different ways, but they are equivalent - give same sequence of x

Sutskever et al, "On the importance of initialization and momentum in deep learning", ICML 2013

# بهینه‌سازی

## کاهش گرادیانی اتفاقی + مومنتوم





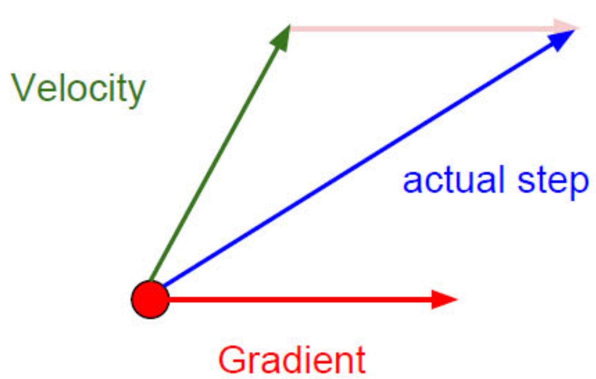

Ref: http://cs231n.stanford.edu/

# بهینه‌سازی

کاهش گرادیانی اتفاقی + مومنتوم

## SGD+Momentum

### Momentum update:




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

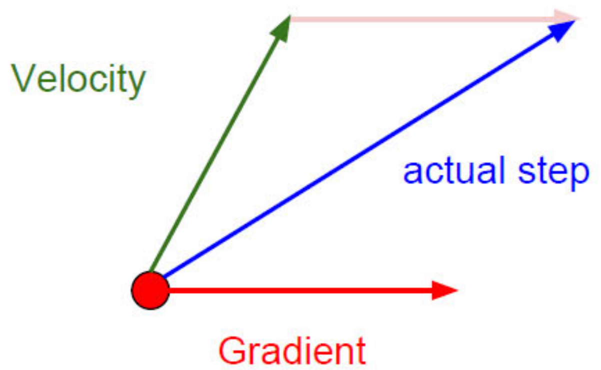
Nesterov, "A method of solving a convex programming problem with convergence rate O(1/k^2)", 1983
Nesterov, "Introductory lectures on convex optimization: a basic course", 2004
Sutskever et al, "On the importance of initialization and momentum in deep learning", ICML 2013

# بهینه‌سازی

مومنتوم نستروف

# Nesterov Momentum

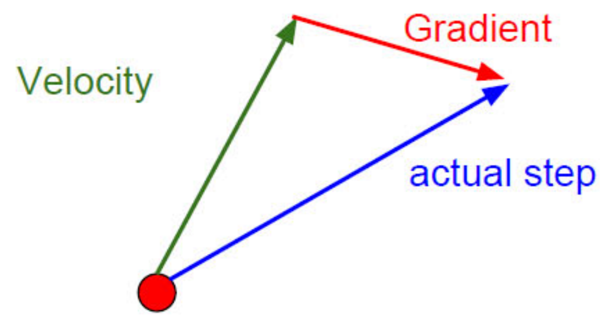
## Momentum update:




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



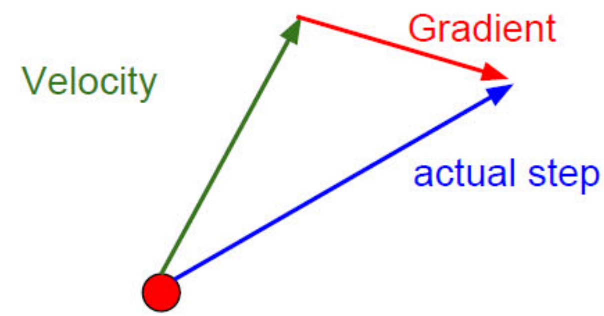

"Look ahead" to the point where updating using velocity would take us; compute gradient there and mix it with velocity to get actual update direction

مومنتوم نستروف

# Nesterov Momentum

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

$$x_{t+1} = x_t + v_{t+1}$$



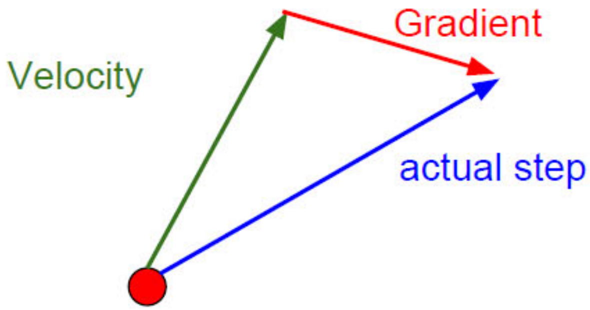

"Look ahead" to the point where updating using velocity would take us; compute gradient there and mix it with velocity to get actual update direction

مومنتوم نستروف

# Nesterov Momentum

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

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




"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

مومنتوم نستروف

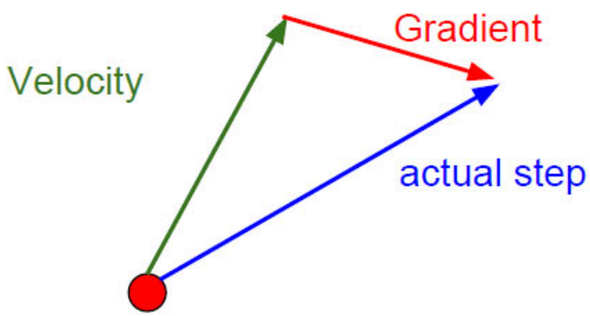
# Nesterov Momentum

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

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

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

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




"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

مومنتوم نستروف

# Nesterov Momentum

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

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

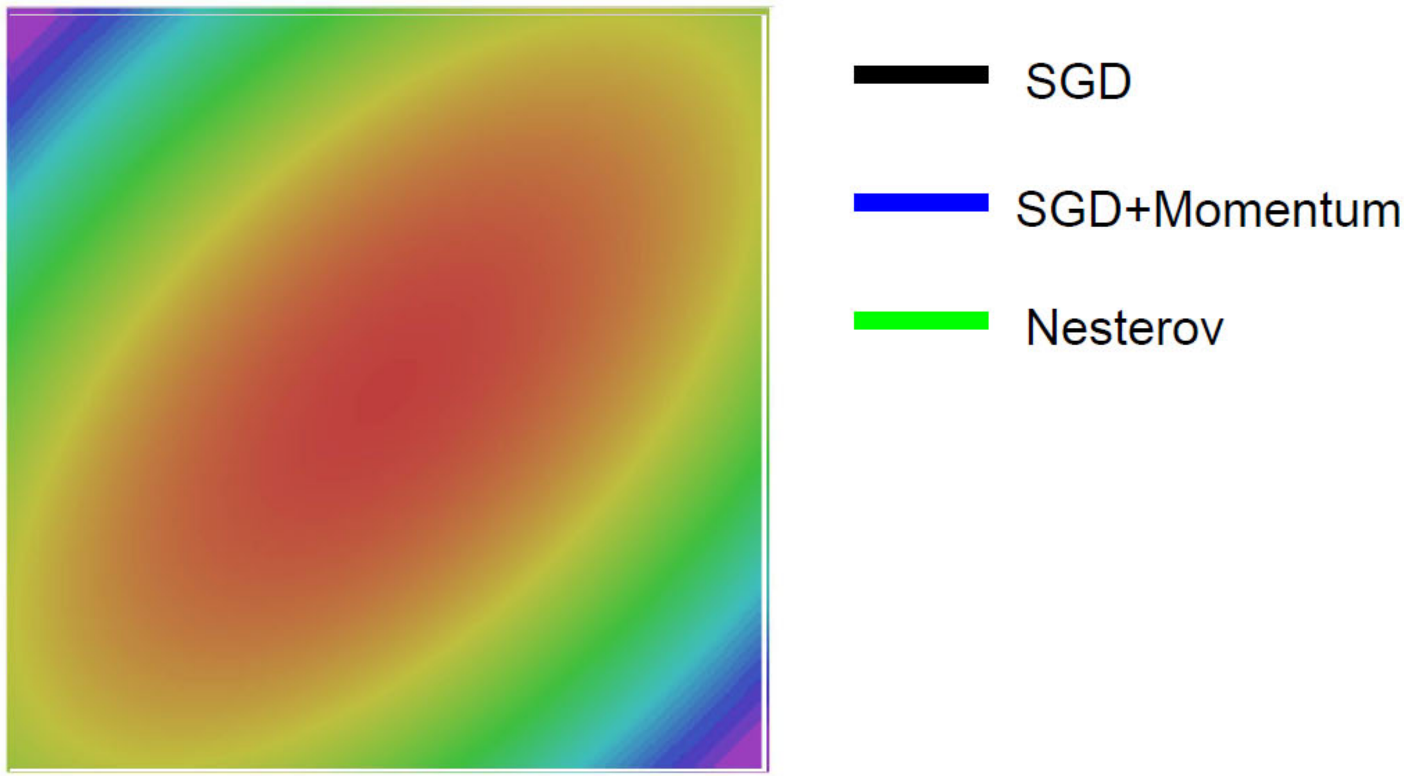
Change of variables $\tilde{x}_t = x_t + \rho v_t$ and rearrange:

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

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

مومنتوم نستروف

# Nesterov Momentum

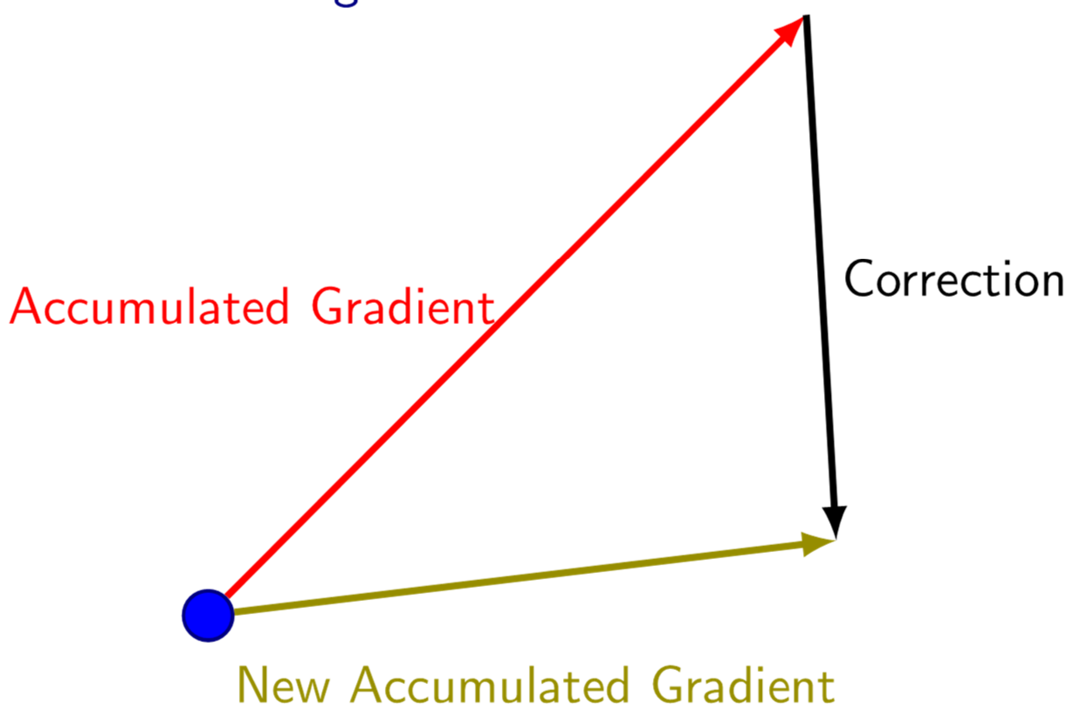





Ref: http://cs231n.stanford.edu/

## NESTEROV MOMENTUM

- Another approach: First take a step in the direction of the accumulated gradient
- Then calculate the gradient and make a correction





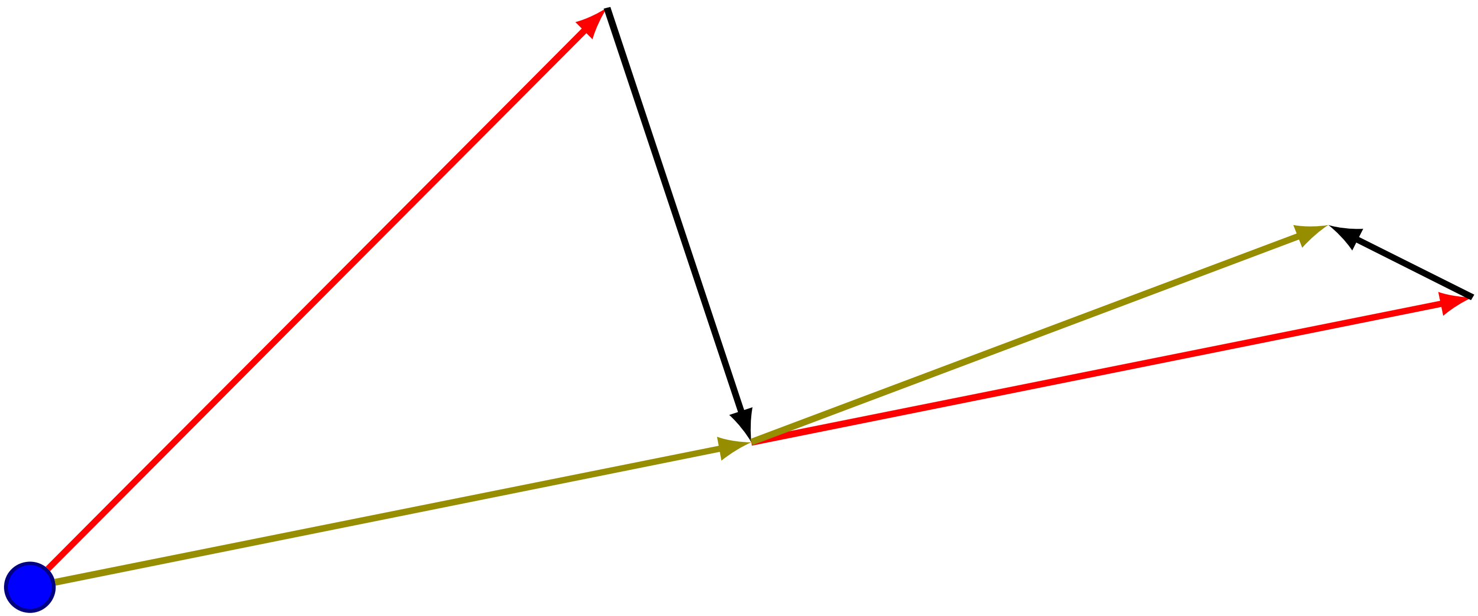

Ref: https://ttic.uchicago.edu/~shubhendu/Pages/CMSC35246.html

# بهینه‌سازی

مومنتوم نستروف

## NESTEROV MOMENTUM

Next Step





Ref: https://ttic.uchicago.edu/~shubhendu/Pages/CMSC35246.html

## Nesterov Momentum

- Recall the velocity term in the Momentum method:

$$\mathbf{v} \leftarrow \alpha \mathbf{v} - \epsilon \nabla_\theta \left( L(f(\mathbf{x}^{(i)}; \theta), \mathbf{y}^{(i)}) \right)$$

- Nesterov Momentum:

$$\mathbf{v} \leftarrow \alpha \mathbf{v} - \epsilon \nabla_\theta \left( L(f(\mathbf{x}^{(i)}; \theta + \alpha \mathbf{v}), \mathbf{y}^{(i)}) \right)$$

- Update: $\theta \leftarrow \theta + \mathbf{v}$



Ref: https://ttic.uchicago.edu/~shubhendu/Pages/CMSC35246.html

## SGD WITH NESTEROV MOMENTUM

**Algorithm 3** SGD with Nesterov Momentum

**Require:** Learning rate $\epsilon$
**Require:** Momentum Parameter $\alpha$
**Require:** Initial Parameter $\theta$
**Require:** Initial Velocity $\mathbf{v}$

1: **while** stopping criteria not met **do**
2: Sample example $(\mathbf{x}^{(i)}, \mathbf{y}^{(i)})$ from training set
3: Update parameters: $\tilde{\theta} \leftarrow \theta + \alpha \mathbf{v}$
4: Compute gradient estimate:
5: $\hat{\mathbf{g}} \leftarrow +\nabla_{\tilde{\theta}} L(f(\mathbf{x}^{(i)}; \tilde{\theta}), \mathbf{y}^{(i)})$
6: Compute the velocity update: $\mathbf{v} \leftarrow \alpha \mathbf{v} - \epsilon \hat{\mathbf{g}}$
7: Apply Update: $\theta \leftarrow \theta + \mathbf{v}$
8: **end while**

روش‌های نرخ یادگیری وفقی: انگیزه

ADAPTIVE LEARNING RATE METHODS

**Motivation**

- Till now we assign the same learning rate to all features
- If the features vary in importance and frequency, why is this a good idea?
- It's probably not!

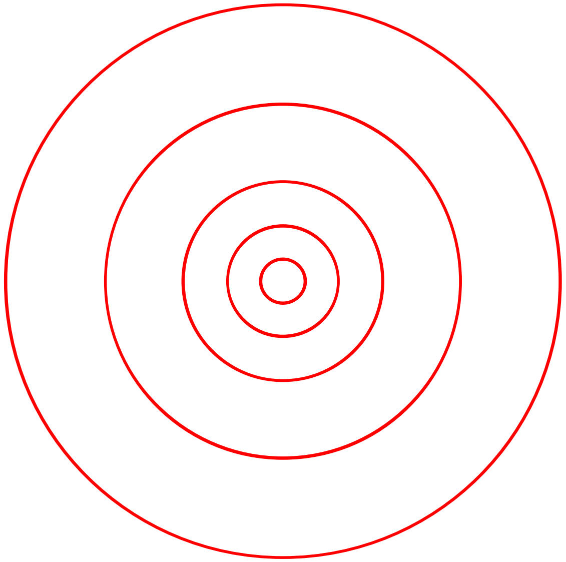


Ref: https://ttic.uchicago.edu/~shubhendu/Pages/CMSC35246.html

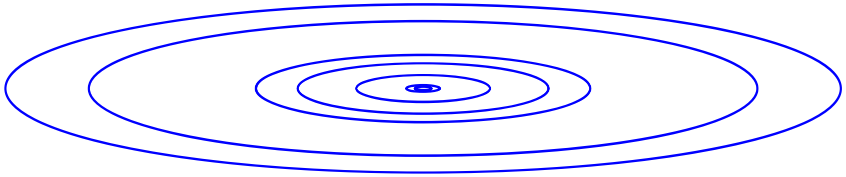
# بهینه‌سازی

## روش‌های نرخ یادگیری وفقی : انگیزه

## Adaptive Learning Rate Methods: Motivation



Nice (all features are equally important)



Harder!

آداگراد

# AdaGrad

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

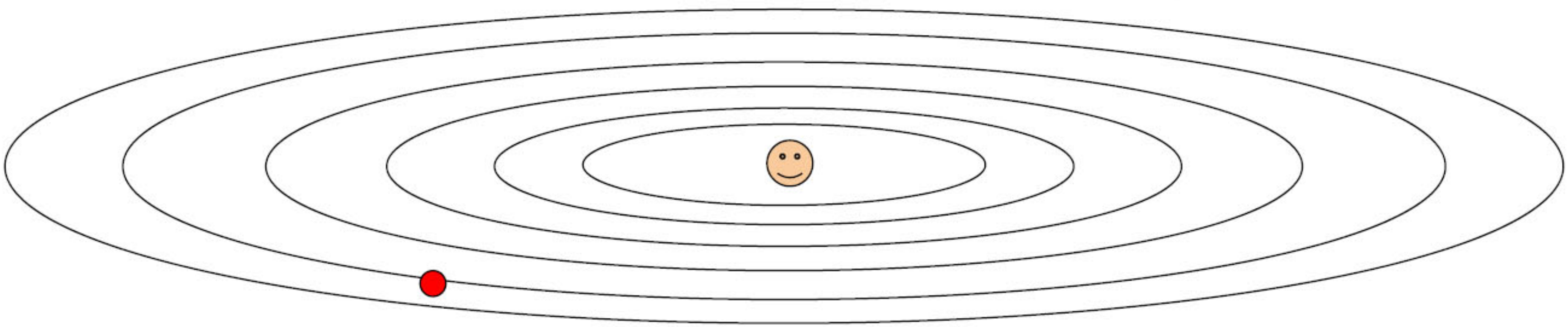
Added element-wise scaling of the gradient based on the historical sum of squares in each dimension

"Per-parameter learning rates" or "adaptive learning rates"

Duchi et al, "Adaptive subgradient methods for online learning and stochastic optimization", JMLR 2011

# AdaGrad

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



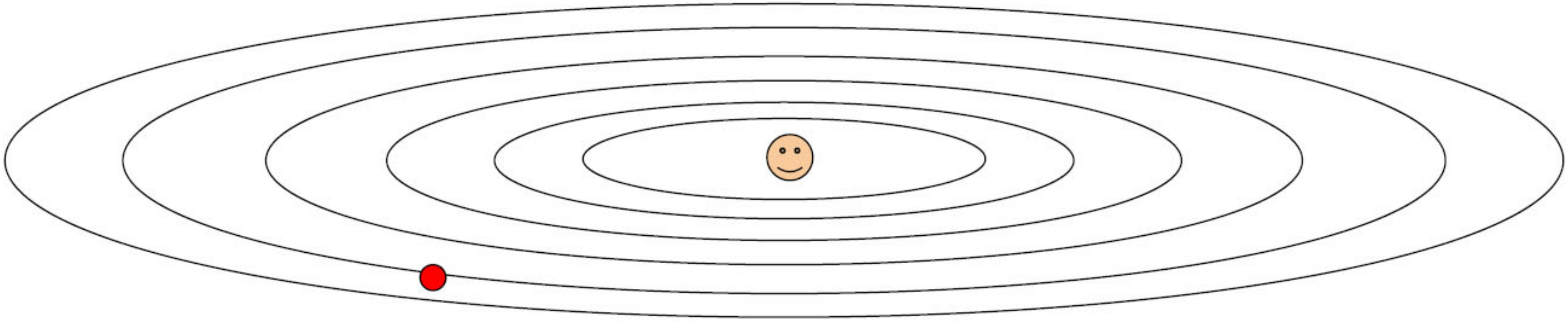
Q: What happens with AdaGrad?

## بهینه‌سازی

آداگراد

# AdaGrad

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



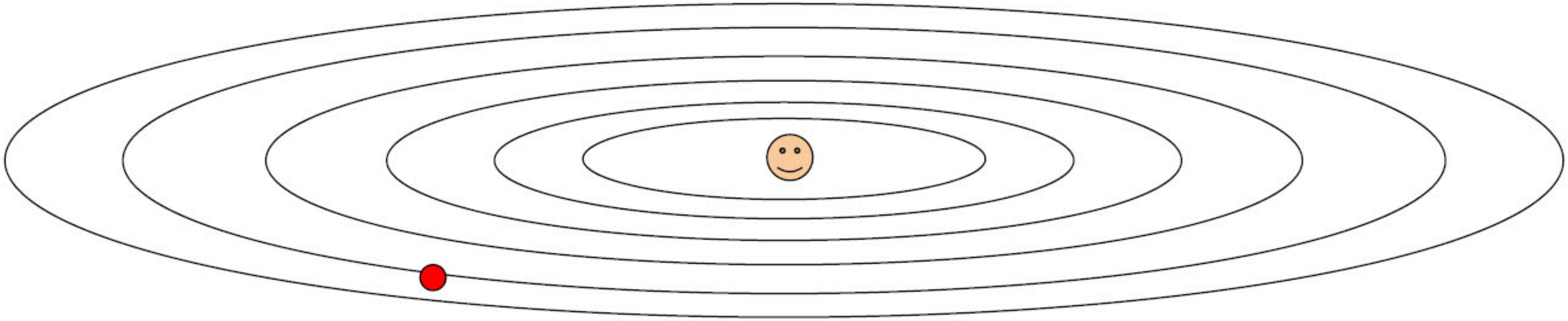
Q: What happens with AdaGrad?

Progress along “steep” directions is damped; progress along “flat” directions is accelerated

آداگراد

# AdaGrad

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

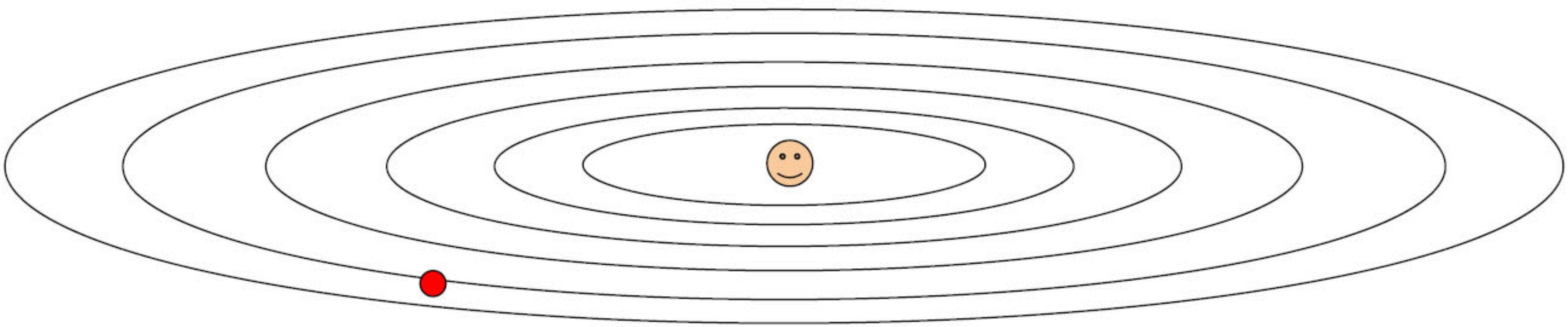


Q2: What happens to the step size over long time?

آداگراد

# AdaGrad

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



Q2: What happens to the step size over long time? Decays to zero

## AdaGrad

- Idea: Downscale a model parameter by square-root of sum of squares of all its historical values
- Parameters that have large partial derivative of the loss – learning rates for them are rapidly declined
- Some interesting theoretical properties

# بهینه‌سازی

## الگوریتم آداگراد

**Algorithm 4** AdaGrad

**Require:** Global Learning rate $\epsilon$, Initial Parameter $\theta$, $\delta$

Initialize $\mathbf{r} = 0$

1: **while** stopping criteria not met **do**

2: Sample example $(\mathbf{x}^{(i)}, \mathbf{y}^{(i)})$ from training set

3: Compute gradient estimate: $\hat{\mathbf{g}} \leftarrow +\nabla_\theta L(f(\mathbf{x}^{(i)}; \theta), \mathbf{y}^{(i)})$

4: Accumulate: $\mathbf{r} \leftarrow \mathbf{r} + \hat{\mathbf{g}} \odot \hat{\mathbf{g}}$

5: Compute update: $\Delta\theta \leftarrow -\frac{\epsilon}{\delta + \sqrt{\mathbf{r}}} \odot \hat{\mathbf{g}}$

6: Apply Update: $\theta \leftarrow \theta + \Delta\theta$

7: **end while**



Ref: https://ttic.uchicago.edu/~shubhendu/Pages/CMSC35246.html

# بهینه‌سازی

آر.ام.اس.پراپ

## RMSProp

AdaGrad

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

RMSProp

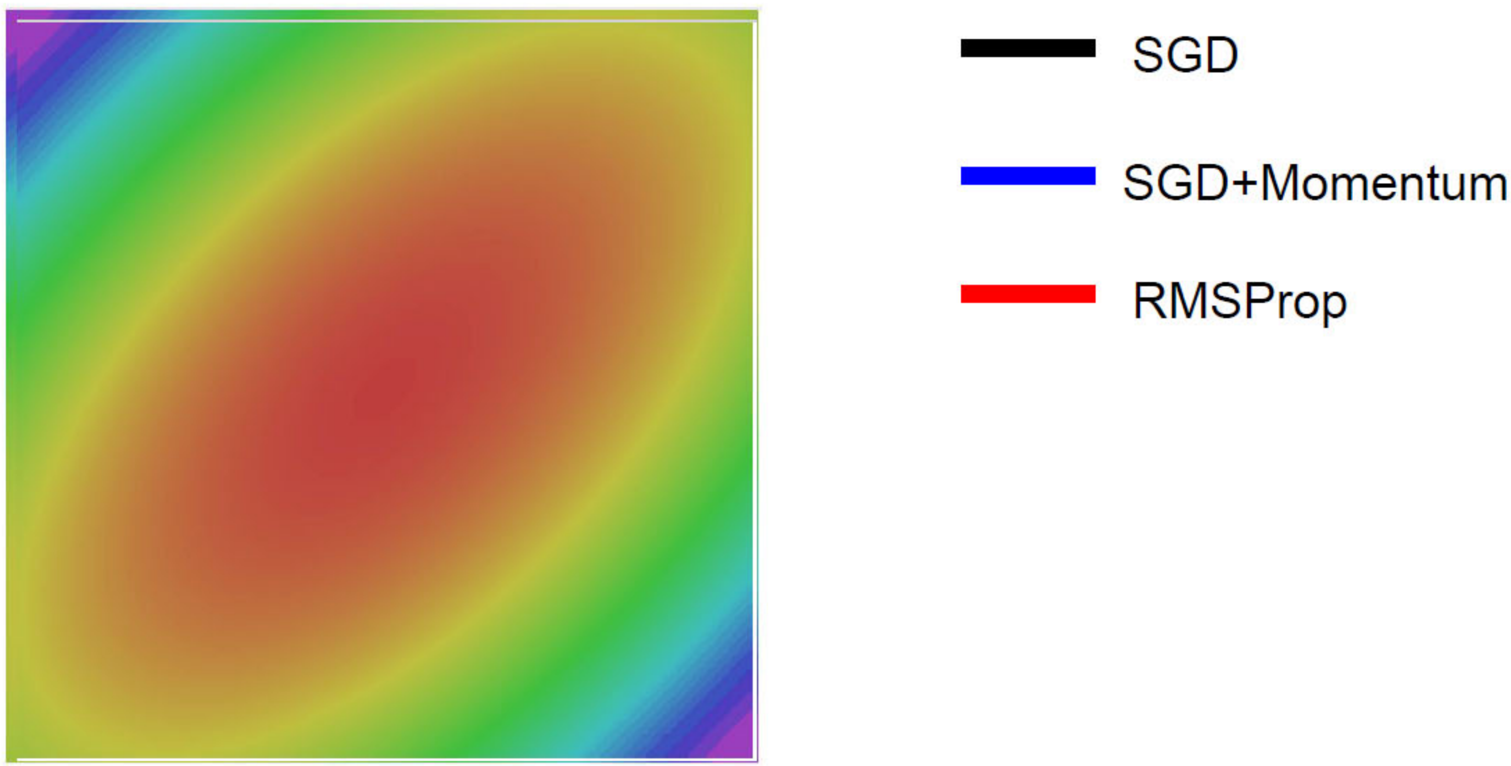
```
grad_squared = 0
while True:
  dx = compute_gradient(x)
  grad_squared = decay_rate * grad_squared + (1 - decay_rate) * dx * dx
  x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```

Tieleman and Hinton, 2012

# بهینه‌سازی

آر.ام.اس.پراپ

## RMSProp






Ref: http://cs231n.stanford.edu/

# بهینه‌سازی

آر.ام.اس.پراپ

## RMSProp

- AdaGrad is good when the objective is convex.
- AdaGrad might shrink the learning rate too aggressively, we want to keep the history in mind
- We can adapt it to perform better in non-convex settings by accumulating an exponentially decaying average of the gradient
- This is an idea that we use again and again in Neural Networks
- Currently has about 500 citations on scholar, but was proposed in a slide in Geoffrey Hinton's coursera course



Ref: https://ttic.uchicago.edu/~shubhendu/Pages/CMSC35246.html

الگوریتم آر.ام.اس.پراپ

## RMSProp

---

**Algorithm 5** RMSProp

**Require:** Global Learning rate $\epsilon$, decay parameter $\rho$, $\delta$

Initialize $\mathbf{r} = 0$

**while** stopping criteria not met **do**

    Sample example $(\mathbf{x}^{(i)}, \mathbf{y}^{(i)})$ from training set

    Compute gradient estimate: $\hat{\mathbf{g}} \leftarrow +\nabla_\theta L(f(\mathbf{x}^{(i)}; \theta), \mathbf{y}^{(i)})$

    Accumulate: $\mathbf{r} \leftarrow \rho \mathbf{r} + (1 - \rho)\hat{\mathbf{g}} \odot \hat{\mathbf{g}}$

    Compute update: $\Delta\theta \leftarrow -\frac{\epsilon}{\delta + \sqrt{\mathbf{r}}} \odot \hat{\mathbf{g}}$

    Apply Update: $\theta \leftarrow \theta + \Delta\theta$

**end while**

---

# بهینه‌سازی

الگوریتم آر.ام.اس.پراپ با نستروف

## RMSProp with Nesterov

**Algorithm 6** RMSProp with Nesterov

**Require:** Global Learning rate $\epsilon$, decay parameter $\rho$, $\delta$, $\alpha$, $\mathbf{v}$

Initialize $\mathbf{r} = 0$

```
while stopping criteria not met do
    Sample example (x^(i), y^(i)) from training set
    Compute Update: θ̃ ← θ + αv
    Compute gradient estimate: ĝ ← +∇_θ̃ L(f(x^(i); θ̃), y^(i))
    Accumulate: r ← ρr + (1 − ρ)ĝ ⊙ ĝ
    Compute Velocity: v ← αv − (ε/√r) ⊙ ĝ
    Apply Update: θ ← θ + v
end while
```

- **while** stopping criteria not met **do**
  - Sample example $(\mathbf{x}^{(i)}, \mathbf{y}^{(i)})$ from training set
  - Compute Update: $\tilde{\theta} \leftarrow \theta + \alpha \mathbf{v}$
  - Compute gradient estimate: $\hat{\mathbf{g}} \leftarrow +\nabla_{\tilde{\theta}} L(f(\mathbf{x}^{(i)}; \tilde{\theta}), \mathbf{y}^{(i)})$
  - Accumulate: $\mathbf{r} \leftarrow \rho \mathbf{r} + (1-\rho)\hat{\mathbf{g}} \odot \hat{\mathbf{g}}$
  - Compute Velocity: $\mathbf{v} \leftarrow \alpha \mathbf{v} - \frac{\epsilon}{\sqrt{\mathbf{r}}} \odot \hat{\mathbf{g}}$
  - Apply Update: $\theta \leftarrow \theta + \mathbf{v}$
- **end while**



Ref: https://ttic.uchicago.edu/~shubhendu/Pages/CMSC35246.html

آدام (تقریبی)

# Adam (almost)

```
first_moment = 0
second_moment = 0
while True:
  dx = compute_gradient(x)
  first_moment = beta1 * first_moment  + (1 - beta1) * dx
  second_moment = beta2 * second_moment + (1 - beta2) * dx * dx
  x -= learning_rate * first_moment / (np.sqrt(second_moment) + 1e-7))
```

Kingma and Ba, "Adam: A method for stochastic optimization", ICLR 2015

آدام (تقریبی)

# Adam (almost)

```
first_moment = 0
second_moment = 0
while True:
  dx = compute_gradient(x)
  first_moment = beta1 * first_moment  + (1 - beta1) * dx
  second_moment = beta2 * second_moment + (1 - beta2) * dx * dx
  x -= learning_rate * first_moment / (np.sqrt(second_moment) + 1e-7))
```

Momentum

AdaGrad / RMSProp

Sort of like RMSProp with momentum

Q: What happens at first timestep?

Kingma and Ba, "Adam: A method for stochastic optimization", ICLR 2015

# بهینه‌سازی

آدام (فرم کامل)

## Adam (full form)

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

Momentum

Bias correction

AdaGrad / RMSProp

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

Kingma and Ba, "Adam: A method for stochastic optimization", ICLR 2015

آدام (فرم کامل)

# Adam (full form)

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

Momentum

Bias correction

AdaGrad / RMSProp

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

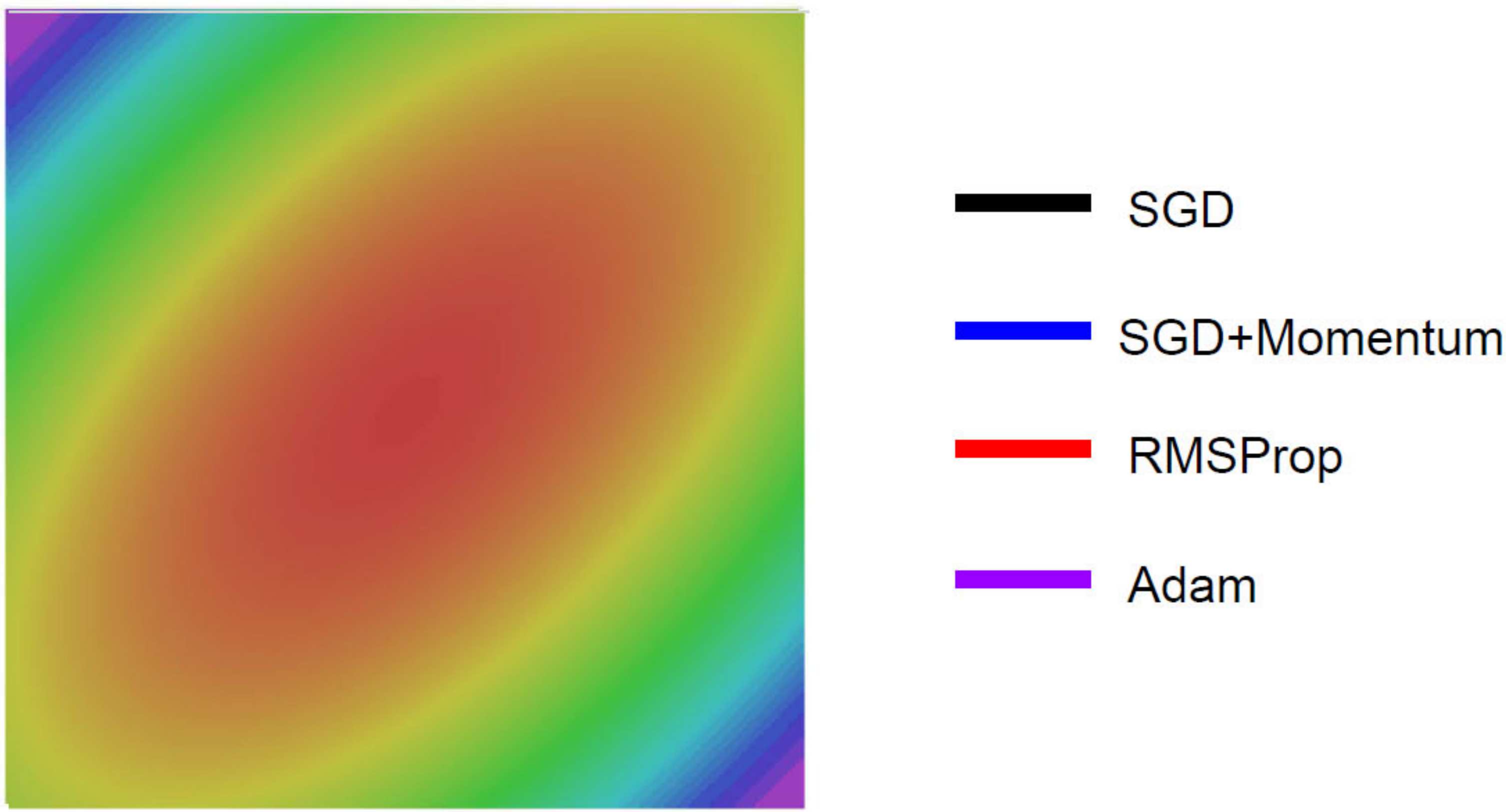
Adam with beta1 = 0.9, beta2 = 0.999, and learning_rate = 1e-3 or 5e-4 is a great starting point for many models!

Kingma and Ba, “Adam: A method for stochastic optimization”, ICLR 2015

# بهینه‌سازی

آدام (فرم کامل)

## Adam






Ref: http://cs231n.stanford.edu/

# بهینه‌سازی

آدام

## ADAM

- We could have used RMSProp with momentum
- Use of Momentum with rescaling is not well motivated
- Adam is like RMSProp with Momentum but with bias correction terms for the first and second moments

# بهینه‌سازی

## الگوریتم آدام

ADAM: ADAPTIVE MOMENTS

**Algorithm 7** RMSProp with Nesterov

**Require:** $\epsilon$ (set to 0.0001), decay rates $\rho_1$ (set to 0.9), $\rho_2$ (set to 0.9), $\theta$, $\delta$

Initialize moments variables $\mathbf{s} = 0$ and $\mathbf{r} = 0$, time step t = 0

**while** stopping criteria not met **do**
  Sample example $(\mathbf{x}^{(i)}, \mathbf{y}^{(i)})$ from training set
  Compute gradient estimate: $\hat{\mathbf{g}} \leftarrow +\nabla_\theta L(f(\mathbf{x}^{(i)}; \theta), \mathbf{y}^{(i)})$
  $t \leftarrow t + 1$
  Update: $\mathbf{s} \leftarrow \rho_1 \mathbf{s} + (1 - \rho_1)\hat{\mathbf{g}}$
  Update: $\mathbf{r} \leftarrow \rho_2 \mathbf{r} + (1 - \rho_2)\hat{\mathbf{g}} \odot \hat{\mathbf{g}}$
  Correct Biases: $\hat{\mathbf{s}} \leftarrow \frac{\mathbf{s}}{1-\rho_1^t}, \hat{\mathbf{r}} \leftarrow \frac{\mathbf{r}}{1-\rho_2^t}$
  Compute Update: $\Delta\theta = -\epsilon \frac{\hat{\mathbf{s}}}{\sqrt{\hat{\mathbf{r}}} + \delta}$
  Apply Update: $\theta \leftarrow \theta + \Delta\theta$
**end while**

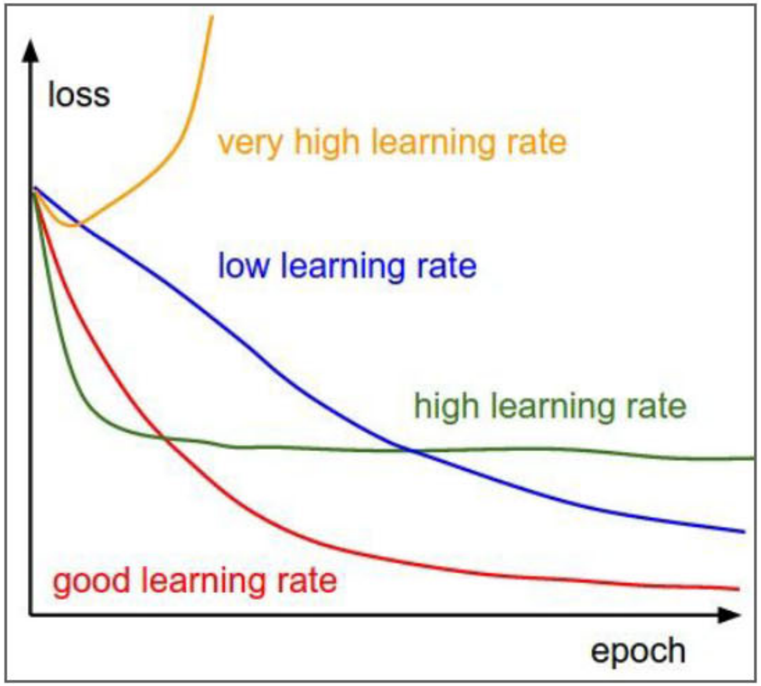


Ref: https://ttic.uchicago.edu/~shubhendu/Pages/CMSC35246.html

# بهینه‌سازی

## نرخ یادگیری: هاپیر–پارامتر روش‌های بهینه‌سازی

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

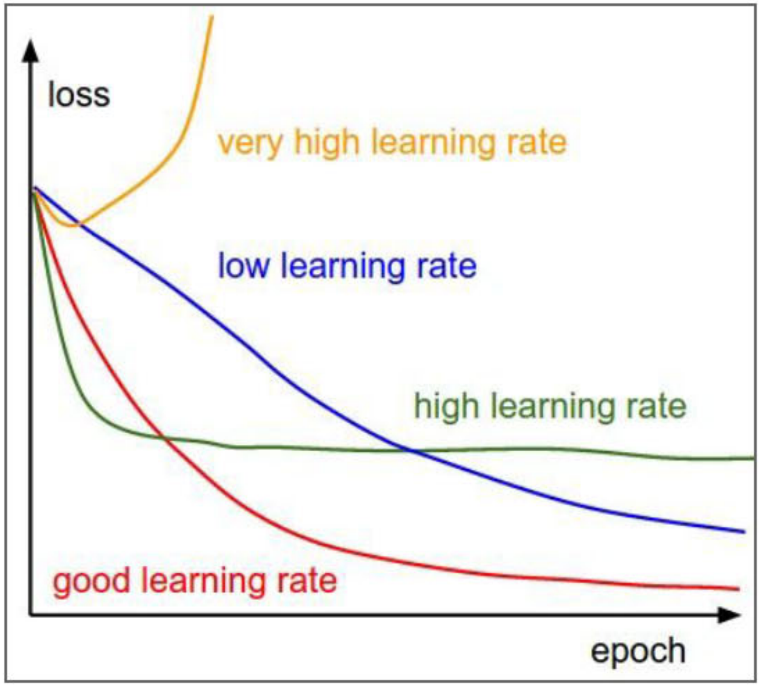



Q: Which one of these learning rates is best to use?

# بهینه‌سازی

نرخ یادگیری : هاپیر–پارامتر روش‌های بهینه‌سازی

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




**=> Learning rate decay over time!**

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

**exponential decay:**

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

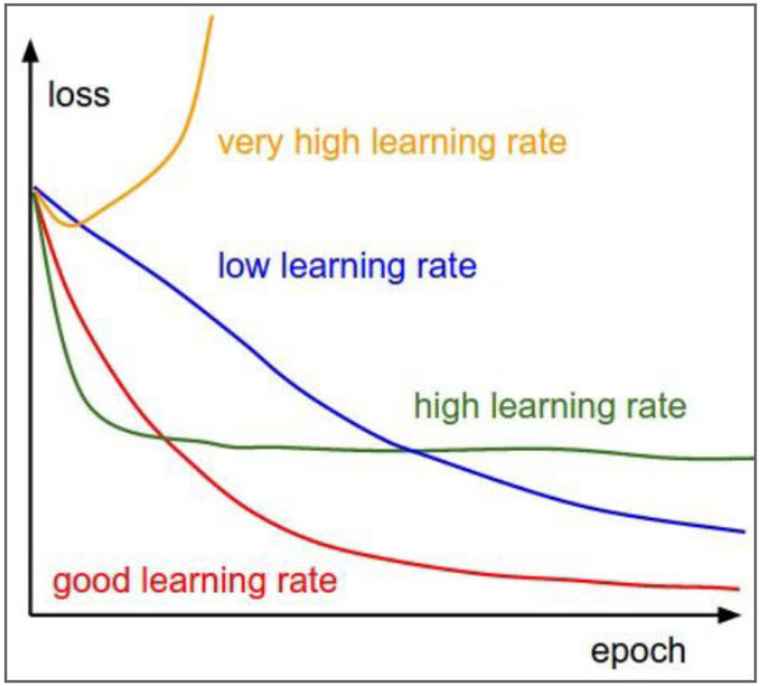
**1/t decay:**

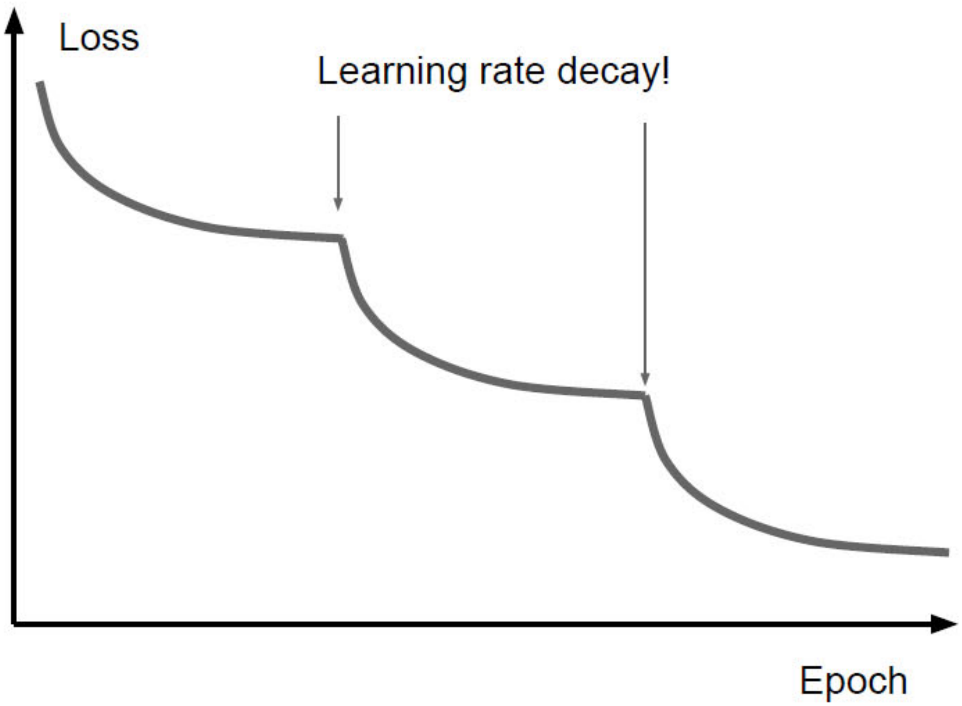
$$\alpha = \alpha_0 / (1 + kt)$$

# بهینه‌سازی

نرخ یادگیری: هاپیر-پارامتر روش‌های بهینه‌سازی

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



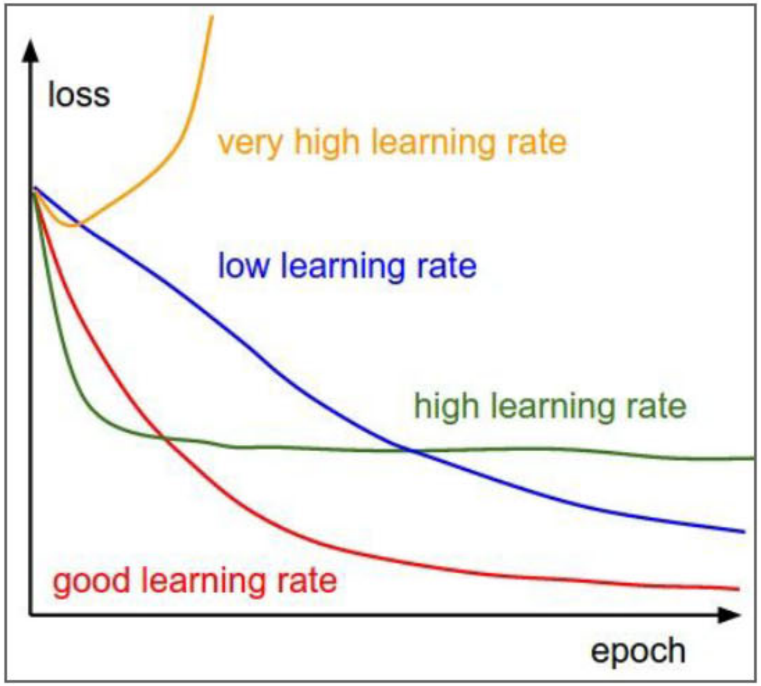


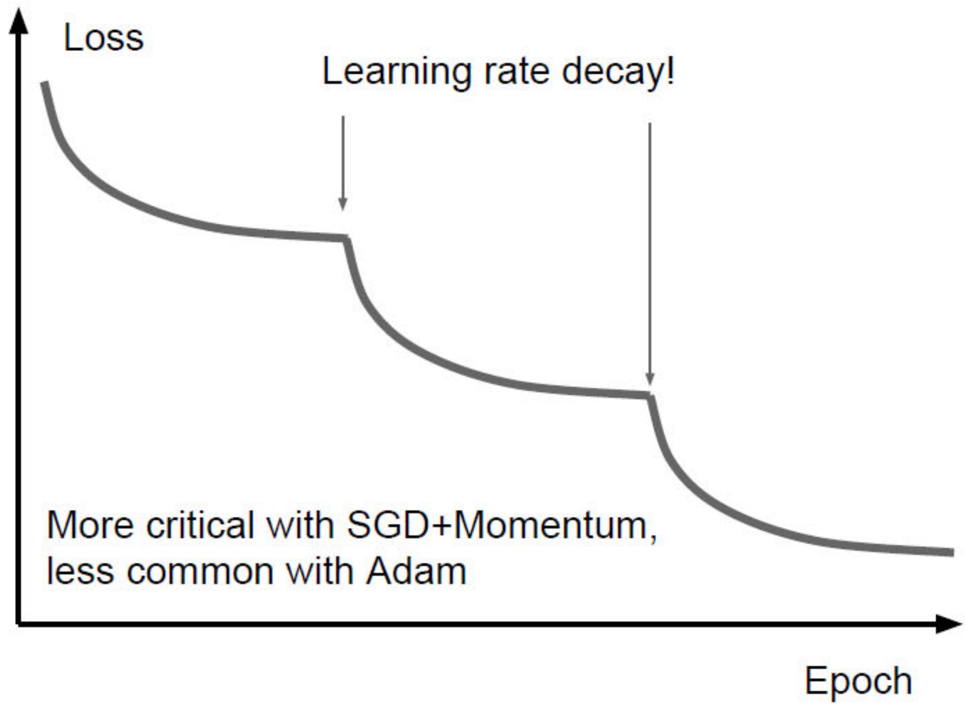

Ref: http://cs231n.stanford.edu/

# بهینه‌سازی

## نرخ یادگیری: هاپیر-پارامتر روش‌های بهینه‌سازی

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

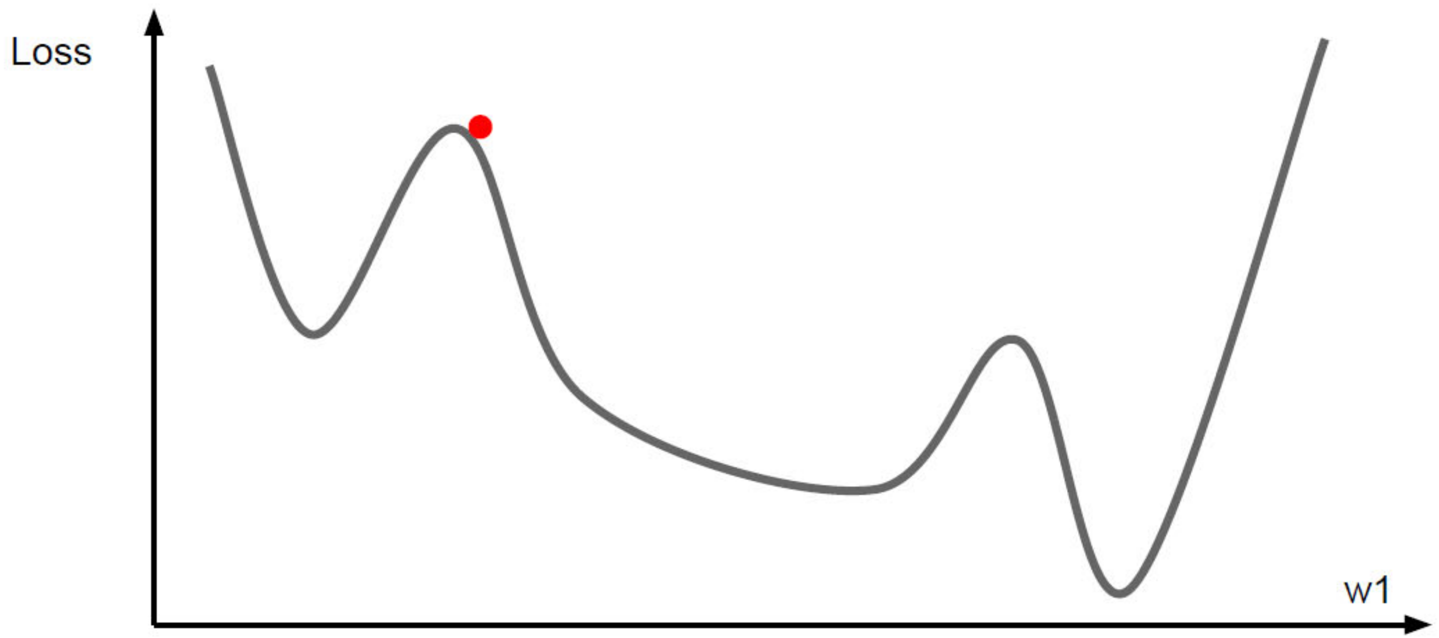





Ref: http://cs231n.stanford.edu/

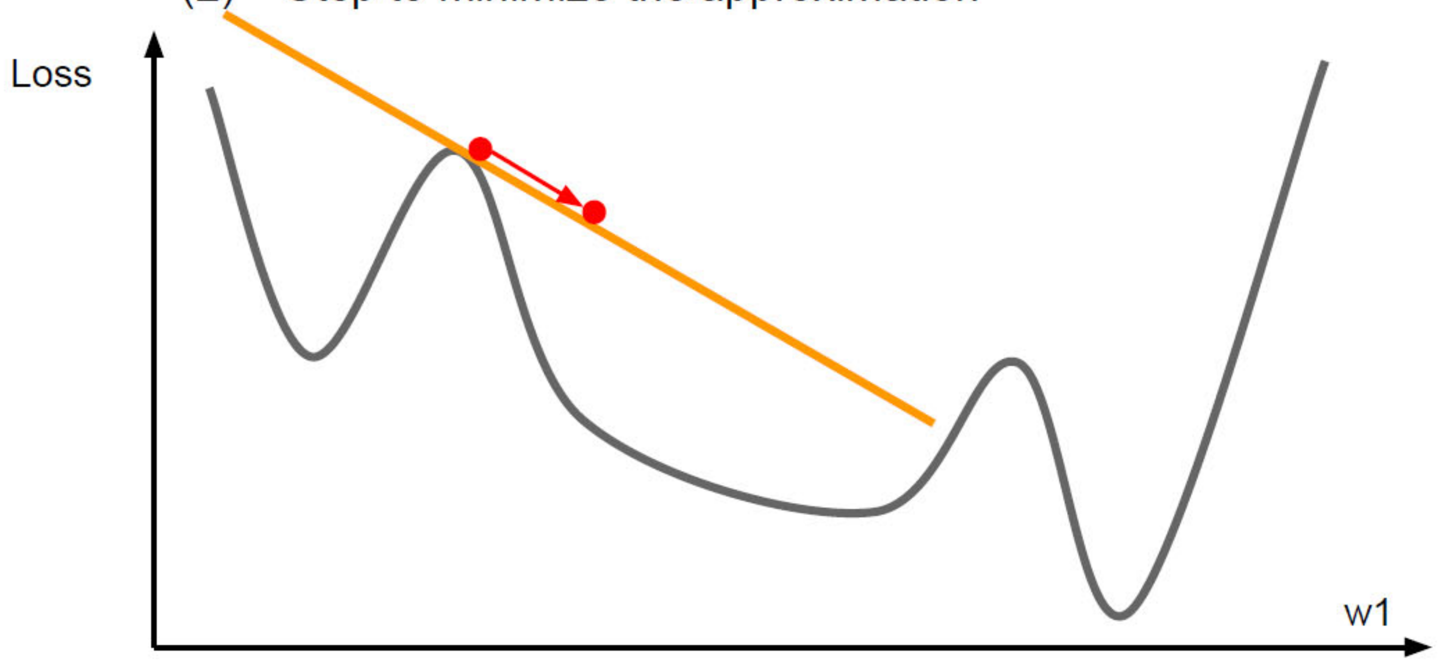
# First-Order Optimization



بهینه‌سازی مرتبه–اول

# First-Order Optimization

(1) Use gradient form linear approximation
(2) Step to minimize the approximation



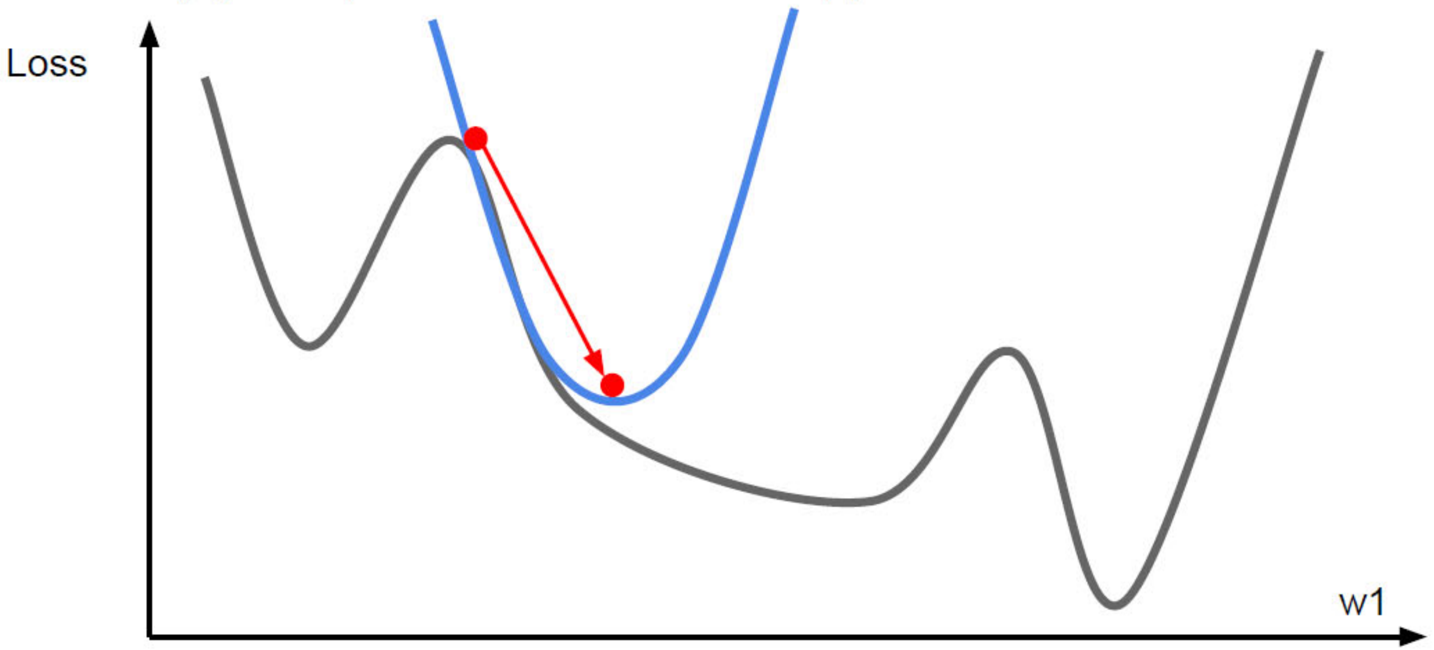
## بهینه‌سازی

بهینه‌سازی مرتبه–دوم

# Second-Order Optimization

(1) Use gradient **and Hessian** to form **quadratic** approximation

(2) Step to the **minima** of the approximation






Ref: http://cs231n.stanford.edu/

# Second-Order Optimization

second-order Taylor expansion:

$$J(\boldsymbol{\theta}) \approx J(\boldsymbol{\theta}_0) + (\boldsymbol{\theta} - \boldsymbol{\theta}_0)^\top \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0) + \frac{1}{2} (\boldsymbol{\theta} - \boldsymbol{\theta}_0)^\top \boldsymbol{H} (\boldsymbol{\theta} - \boldsymbol{\theta}_0)$$

Solving for the critical point we obtain the Newton parameter update:

$$\boldsymbol{\theta}^* = \boldsymbol{\theta}_0 - \boldsymbol{H}^{-1} \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0)$$

Q: What is nice about this update?

# Second-Order Optimization

second-order Taylor expansion:

$$J(\boldsymbol{\theta}) \approx J(\boldsymbol{\theta}_0) + (\boldsymbol{\theta} - \boldsymbol{\theta}_0)^\top \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0) + \frac{1}{2}(\boldsymbol{\theta} - \boldsymbol{\theta}_0)^\top \boldsymbol{H}(\boldsymbol{\theta} - \boldsymbol{\theta}_0)$$

Solving for the critical point we obtain the Newton parameter update:

$$\boldsymbol{\theta}^* = \boldsymbol{\theta}_0 - \boldsymbol{H}^{-1} \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0)$$

No hyperparameters!
No learning rate!
(Though you might use one in practice)

Q: What is nice about this update?

# Second-Order Optimization

second-order Taylor expansion:

$$J(\boldsymbol{\theta}) \approx J(\boldsymbol{\theta}_0) + (\boldsymbol{\theta} - \boldsymbol{\theta}_0)^\top \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0) + \frac{1}{2}(\boldsymbol{\theta} - \boldsymbol{\theta}_0)^\top \boldsymbol{H}(\boldsymbol{\theta} - \boldsymbol{\theta}_0)$$

Solving for the critical point we obtain the Newton parameter update:

$$\boldsymbol{\theta}^* = \boldsymbol{\theta}_0 - \boldsymbol{H}^{-1} \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0)$$

Q2: Why is this bad for deep learning?

# Second-Order Optimization

second-order Taylor expansion:

$$J(\boldsymbol{\theta}) \approx J(\boldsymbol{\theta}_0) + (\boldsymbol{\theta} - \boldsymbol{\theta}_0)^\top \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0) + \frac{1}{2}(\boldsymbol{\theta} - \boldsymbol{\theta}_0)^\top \boldsymbol{H}(\boldsymbol{\theta} - \boldsymbol{\theta}_0)$$

Solving for the critical point we obtain the Newton parameter update:

$$\boldsymbol{\theta}^* = \boldsymbol{\theta}_0 - \boldsymbol{H}^{-1} \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0)$$

Hessian has O(N^2) elements
Inverting takes O(N^3)
N = (Tens or Hundreds of) Millions

Q2: Why is this bad for deep learning?

# Second-Order Optimization

$$\boldsymbol{\theta}^* = \boldsymbol{\theta}_0 - \boldsymbol{H}^{-1}\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0)$$

- Quasi-Newton methods (**BGFS** most popular):
  *instead of inverting the Hessian (O(n^3)), approximate inverse Hessian with rank 1 updates over time (O(n^2) each).*

- **L-BFGS** (Limited memory BFGS):
  *Does not form/store the full inverse Hessian.*

# L-BFGS

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

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

Le et al, "On optimization methods for deep learning, ICML 2011"
Ba et al, "Distributed second-order optimization using Kronecker-factored approximations", ICLR 2017

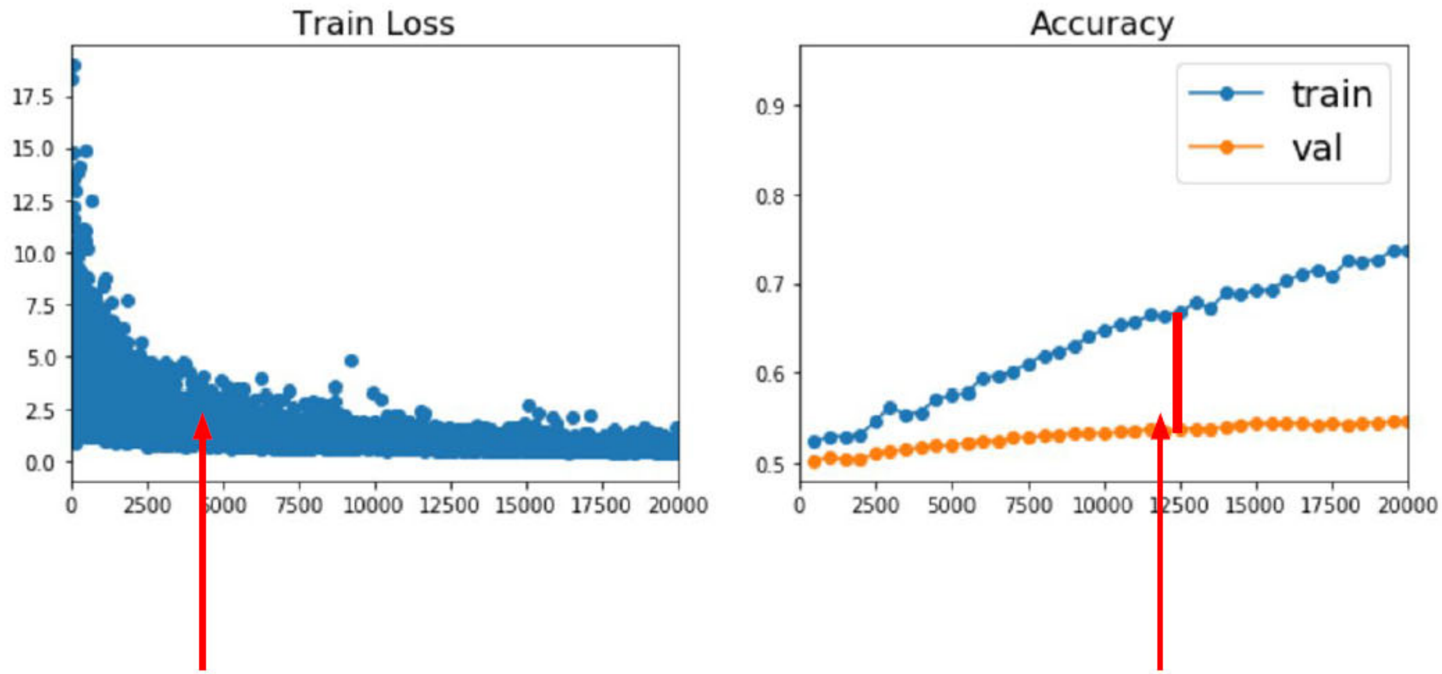
# In practice:

- **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

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

فراتر از خطای آموزش

# Beyond Training Error




Better optimization algorithms help reduce training loss

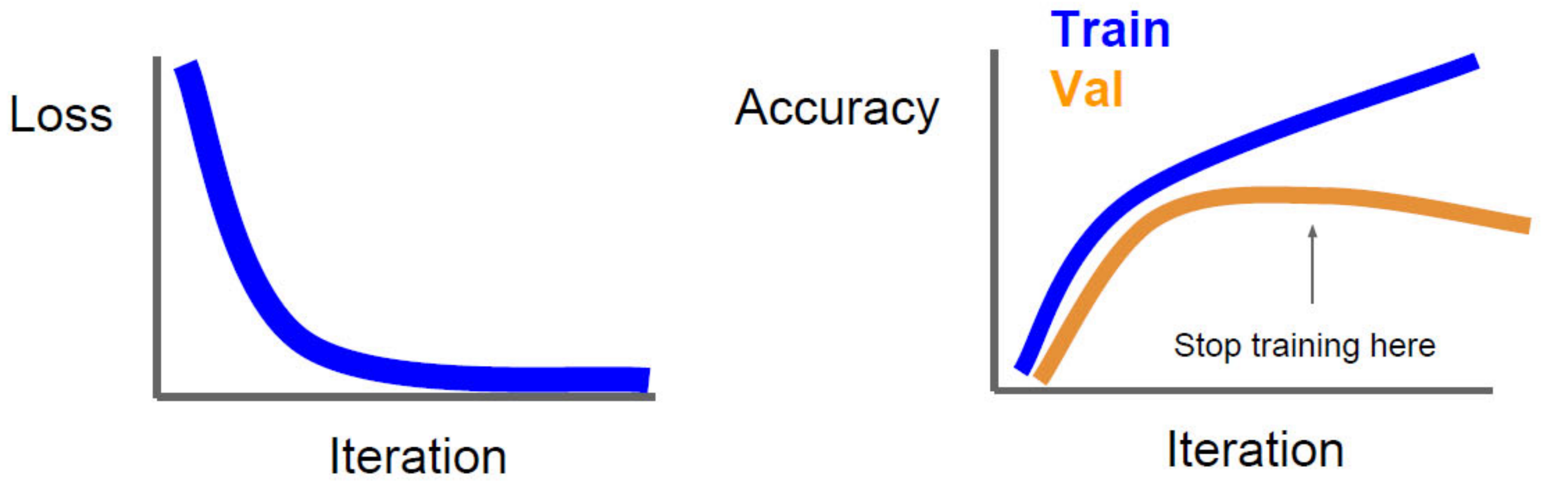
But we really care about error on new data - how to reduce the gap?



Ref: http://cs231n.stanford.edu/

توقف زودهنگام

# Early Stopping




Stop training the model when accuracy on the validation set decreases
Or train for a long time, but always keep track of the model snapshot that worked best on val



Ref: http://cs231n.stanford.edu/

# بهینه‌سازی

## خلاصه‌ی الگوریتم‌ها

SGD: $\theta \leftarrow \theta - \epsilon \hat{\mathbf{g}}$

Momentum: $\mathbf{v} \leftarrow \alpha \mathbf{v} - \epsilon \hat{\mathbf{g}}$ then $\theta \leftarrow \theta + \mathbf{v}$

Nesterov: $\mathbf{v} \leftarrow \alpha \mathbf{v} - \epsilon \nabla_\theta \left( L(f(\mathbf{x}^{(i)}; \theta + \alpha \mathbf{v}), \mathbf{y}^{(i)}) \right)$ then $\theta \leftarrow \theta + \mathbf{v}$

AdaGrad: $\mathbf{r} \leftarrow \mathbf{r} + \mathbf{g} \odot \mathbf{g}$ then $\Delta\theta - \leftarrow \dfrac{\epsilon}{\delta + \sqrt{\mathbf{r}}} \odot \mathbf{g}$ then $\theta \leftarrow \theta + \Delta\theta$

RMSProp: $\mathbf{r} \leftarrow \rho \mathbf{r} + (1-\rho) \hat{\mathbf{g}} \odot \hat{\mathbf{g}}$ then $\Delta\theta \leftarrow -\dfrac{\epsilon}{\delta + \sqrt{\mathbf{r}}} \odot \hat{\mathbf{g}}$ then $\theta \leftarrow \theta + \Delta\theta$

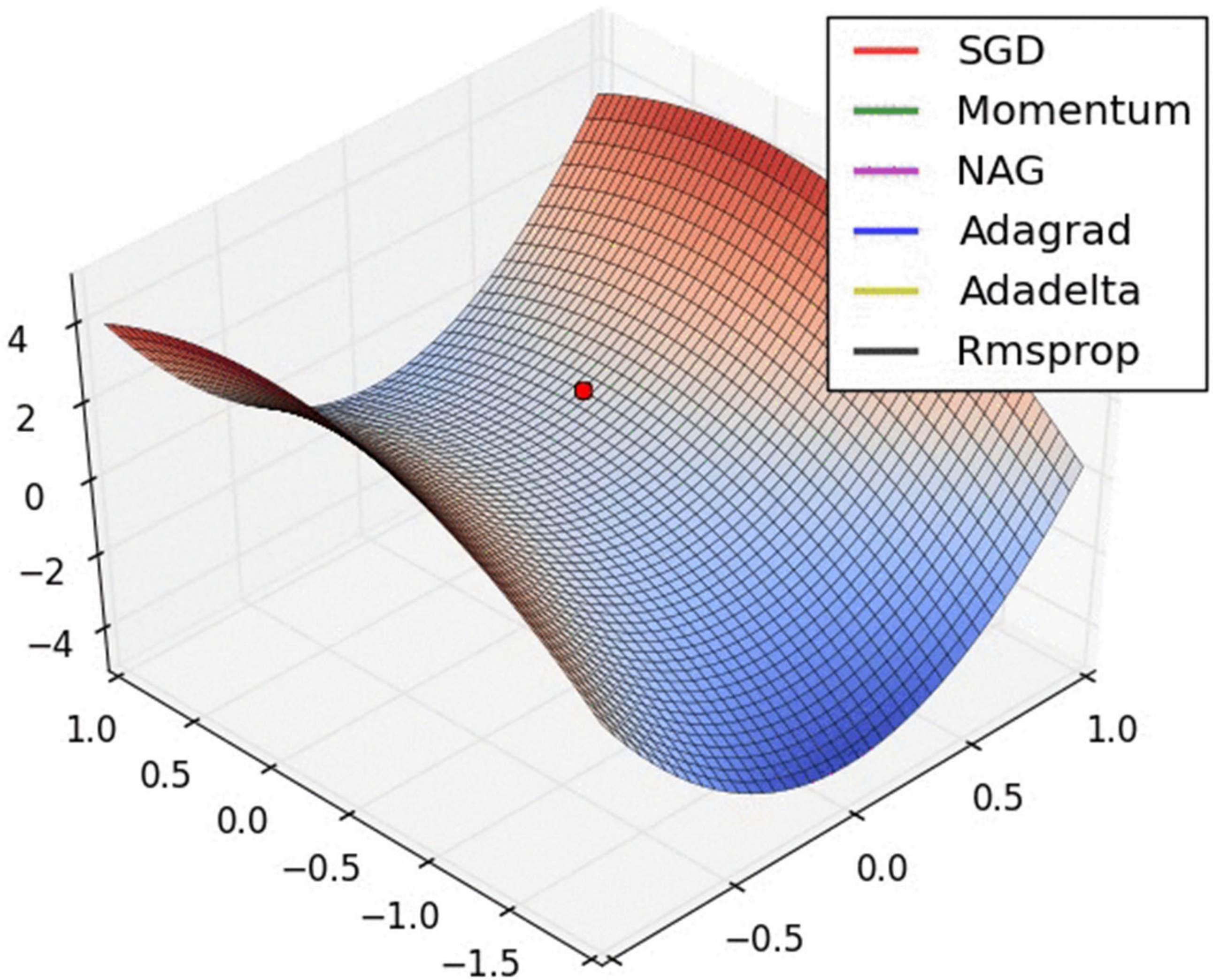
Adam: $\hat{\mathbf{s}} \leftarrow \dfrac{\mathbf{s}}{1-\rho_1^t}, \hat{\mathbf{r}} \leftarrow \dfrac{\mathbf{r}}{1-\rho_2^t}$ then $\Delta\theta = -\epsilon \dfrac{\hat{\mathbf{s}}}{\sqrt{\hat{\mathbf{r}}} + \delta}$ then $\theta \leftarrow \theta + \Delta\theta$



Ref: https://ttic.uchicago.edu/~shubhendu/Pages/CMSC35246.html

# بهینه‌سازی

## تراجکتوری نمونه برای روش‌ها (سه‌بعدی)

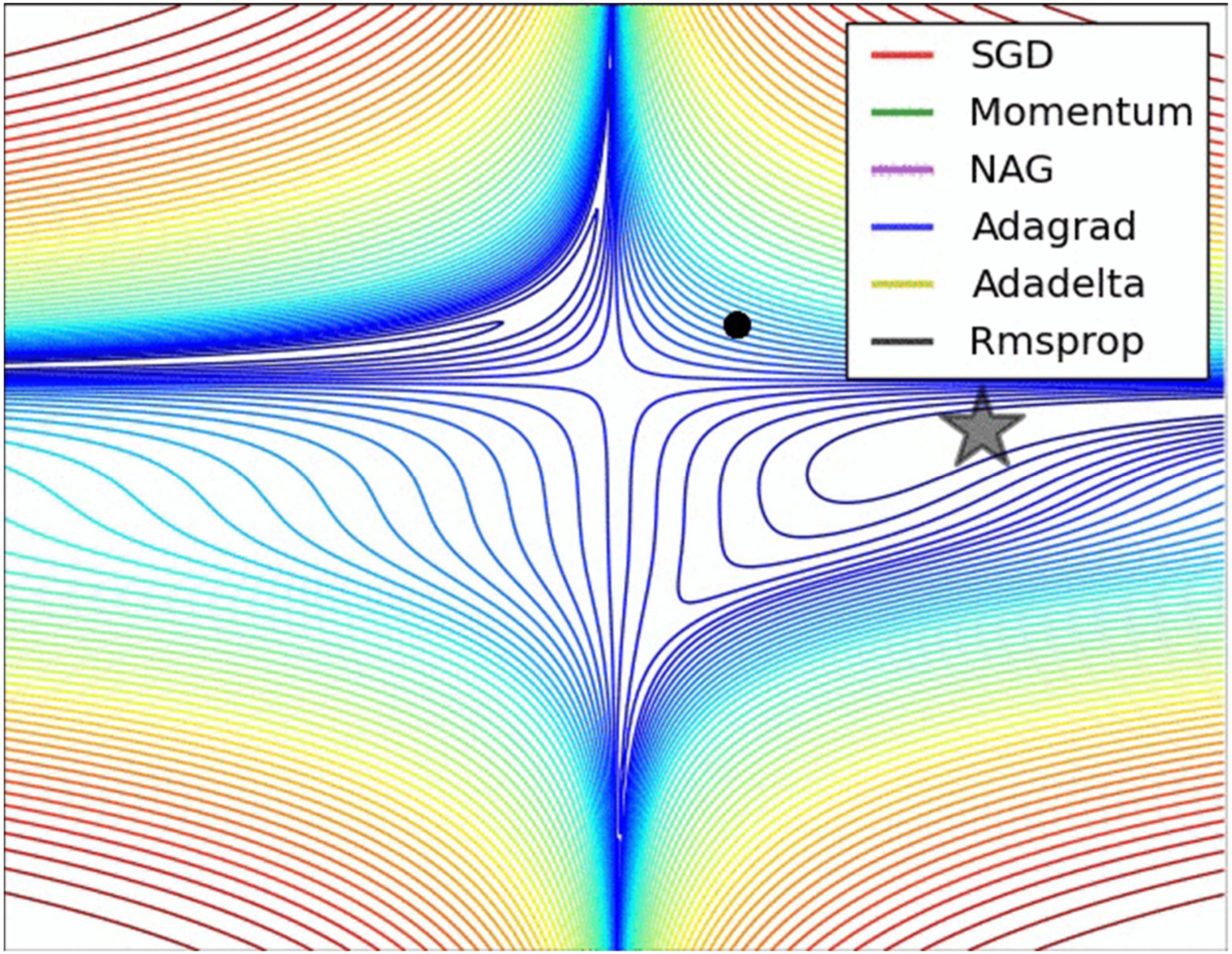





Ref: http://cs231n.stanford.edu/

# بهینه‌سازی

تراجکتوری نمونه برای روش‌ها (بر روی نمودارهای کانتوری دوبعدی)






Ref: http://cs231n.stanford.edu/

**یادگیری عمیق**

ملاحظاتی در
آموزش شبکه‌های عصبی عمیق

# ۸

# مدل‌های دسته‌جمعی

ترکیب مدل‌ها

# Model Ensembles

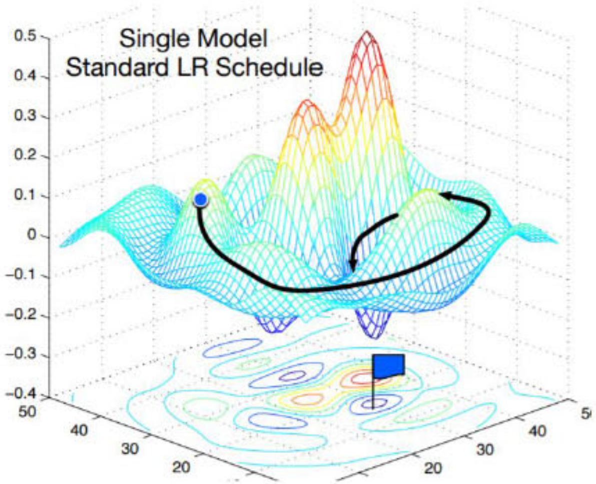
1. Train multiple independent models
2. At test time average their results

(Take average of predicted probability distributions, then choose argmax)

Enjoy 2% extra performance

# Model Ensembles: Tips and Tricks

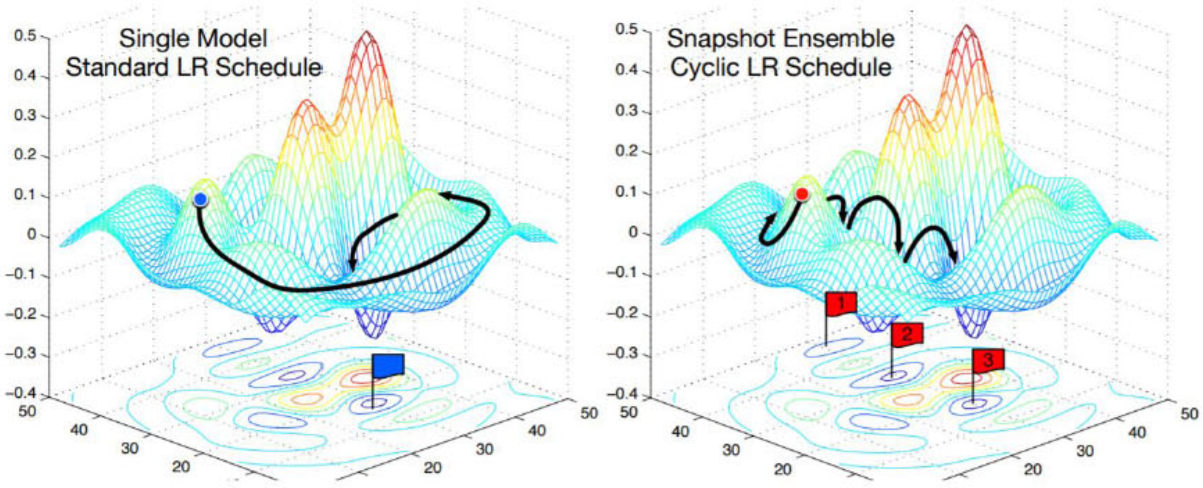
Instead of training independent models, use multiple snapshots of a single model during training!

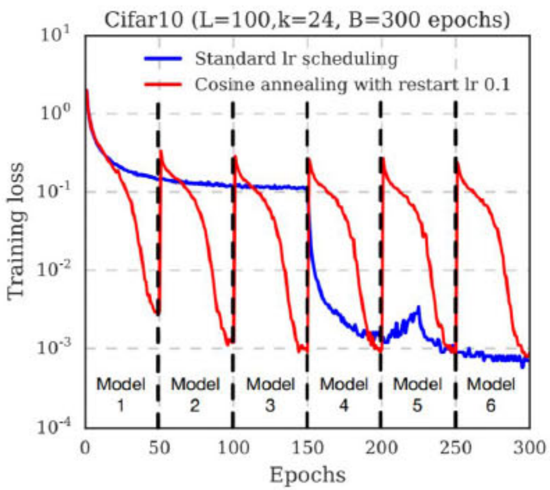



Loshchilov and Hutter, "SGDR: Stochastic gradient descent with restarts", arXiv 2016
Huang et al, "Snapshot ensembles: train 1, get M for free", ICLR 2017

# Model Ensembles: Tips and Tricks

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




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

Cyclic learning rate schedules can make this work even better!

نکات و ترفندها

# Model Ensembles: Tips and Tricks

Instead of using actual parameter vector, keep a moving average of the parameter vector and use that at test time (Polyak averaging)

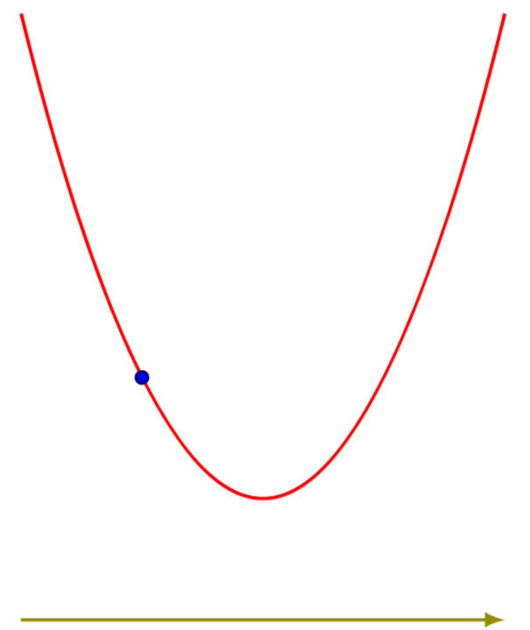
```
while True:
  data_batch = dataset.sample_data_batch()
  loss = network.forward(data_batch)
  dx = network.backward()
  x += - learning_rate * dx
  x_test = 0.995*x_test + 0.005*x # use for test set
```

Polyak and Juditsky, "Acceleration of stochastic approximation by averaging", SIAM Journal on Control and Optimization, 1992.

# متوسط‌گیری پولیاک

انگیزه

## POLYAK AVERAGING: MOTIVATION




- Consider gradient descent above with high step size $\epsilon$

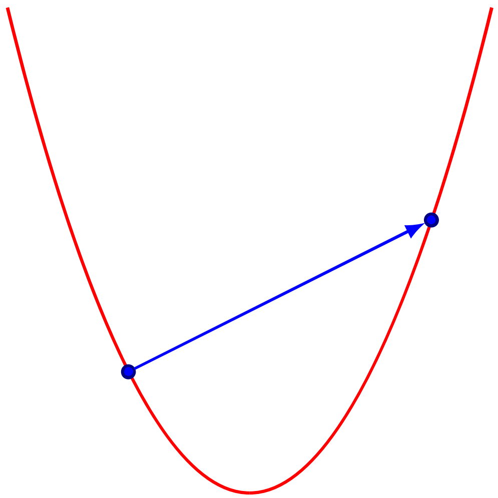


Ref: https://ttic.uchicago.edu/~shubhendu/Pages/CMSC35246.html

# متوسطگیری پولیاک

انگیزه

## Polyak Averaging: Motivation



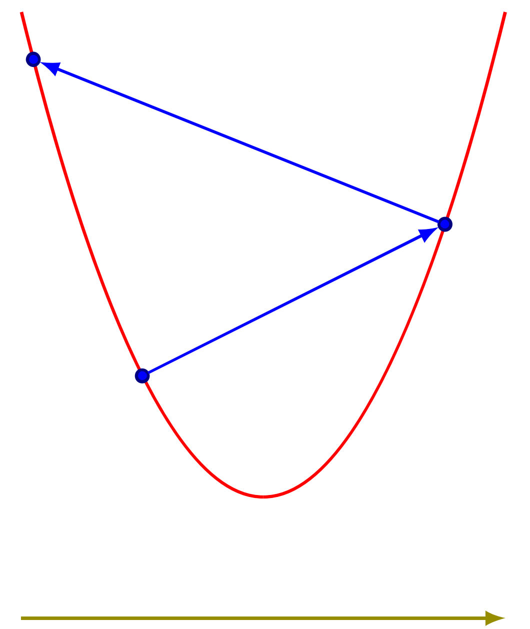



Ref: https://ttic.uchicago.edu/~shubhendu/Pages/CMSC35246.html

# متوسط‌گیری پولیاک

انگیزه

## Polyak Averaging: Motivation



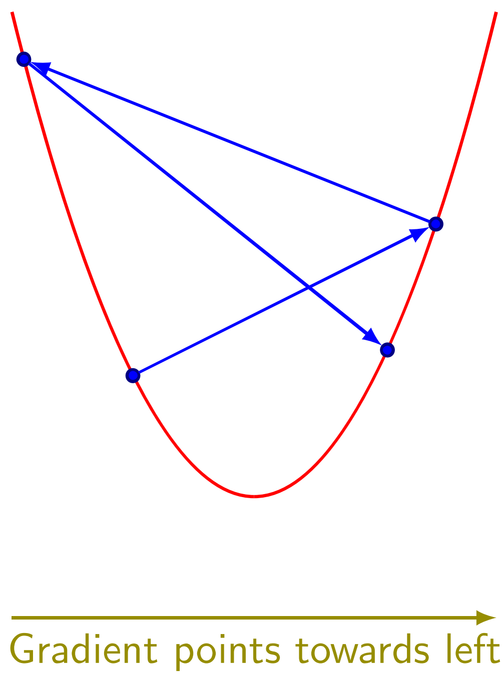



Ref: https://ttic.uchicago.edu/~shubhendu/Pages/CMSC35246.html

# متوسطگیری پولیاک

انگیزه

## POLYAK AVERAGING: MOTIVATION

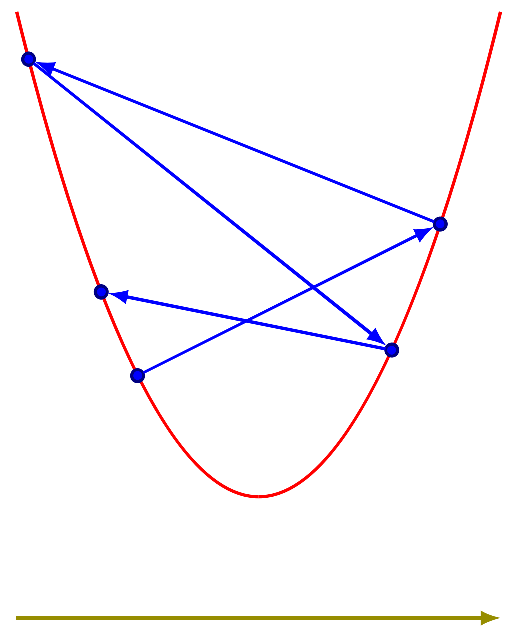





Ref: https://ttic.uchicago.edu/~shubhendu/Pages/CMSC35246.html

# متوسطگیری پولیاک

## Polyak Averaging: Motivation






Ref: https://ttic.uchicago.edu/~shubhendu/Pages/CMSC35246.html

# متوسط‌گیری پولیاک

راه‌حل

## A Solution

- Suppose in $t$ iterations you have parameters $\theta^{(1)}, \theta^{(2)}, \dots, \theta^{(t)}$
- Polyak Averaging suggests setting $\hat{\theta}^{(t)} = \frac{1}{t} \sum_i \theta^{(i)}$
- Has strong convergence guarantees in convex settings
- Is this a good idea in non-convex problems?



Ref: https://ttic.uchicago.edu/~shubhendu/Pages/CMSC35246.html

# متوسطگیری پولیاک

اصلاح ساده

## SIMPLE MODIFICATION

- In non-convex surfaces the parameter space can differ greatly in different regions
- Averaging is not useful
- Typical to consider the exponentially decaying average instead:

$$\hat{\theta}^{(t)} = \alpha\hat{\theta}^{(t-1)} + (1-\alpha)\hat{\theta}^{(t)} \text{ with } \alpha \in [0,1]$$



Ref: https://ttic.uchicago.edu/~shubhendu/Pages/CMSC35246.html

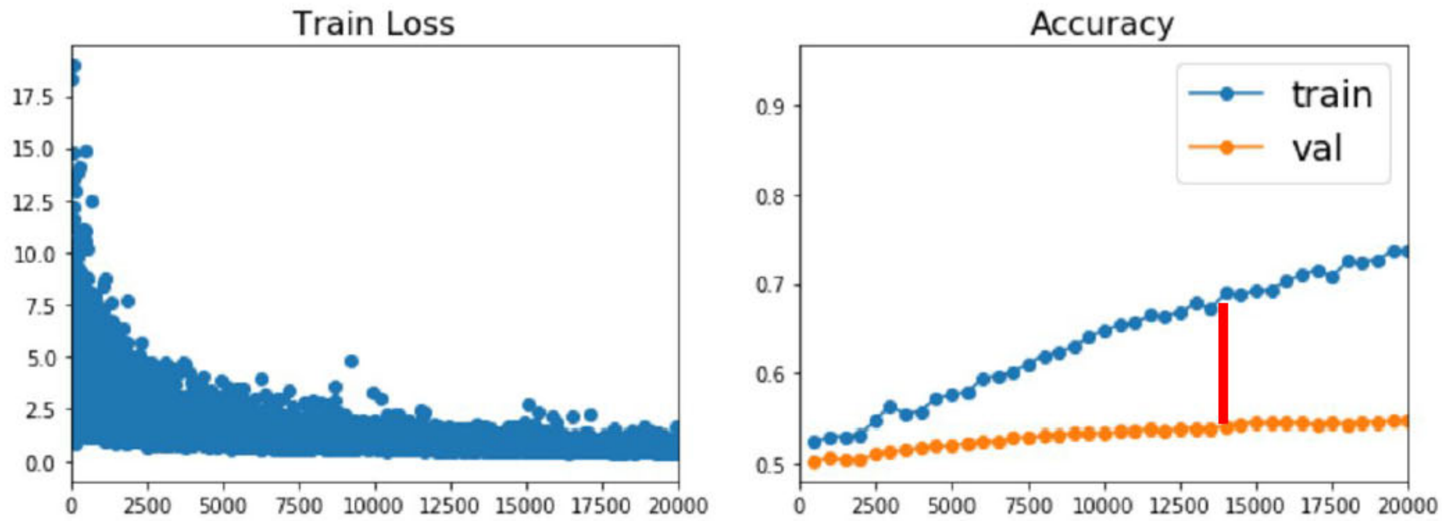
ملاحظاتی در
آموزش شبکه‌های عصبی عمیق

# ۹

# رگولاریزاسیون (منظم‌سازی)

# رگولاریزاسیون (منظم‌سازی)

## How to improve single-model performance?




Regularization



Ref: http://cs231n.stanford.edu/

# رگولاریزاسیون

اضافه کردن یک جمله به تابع اتلاف

## Regularization: Add term to loss

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

In common use:

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

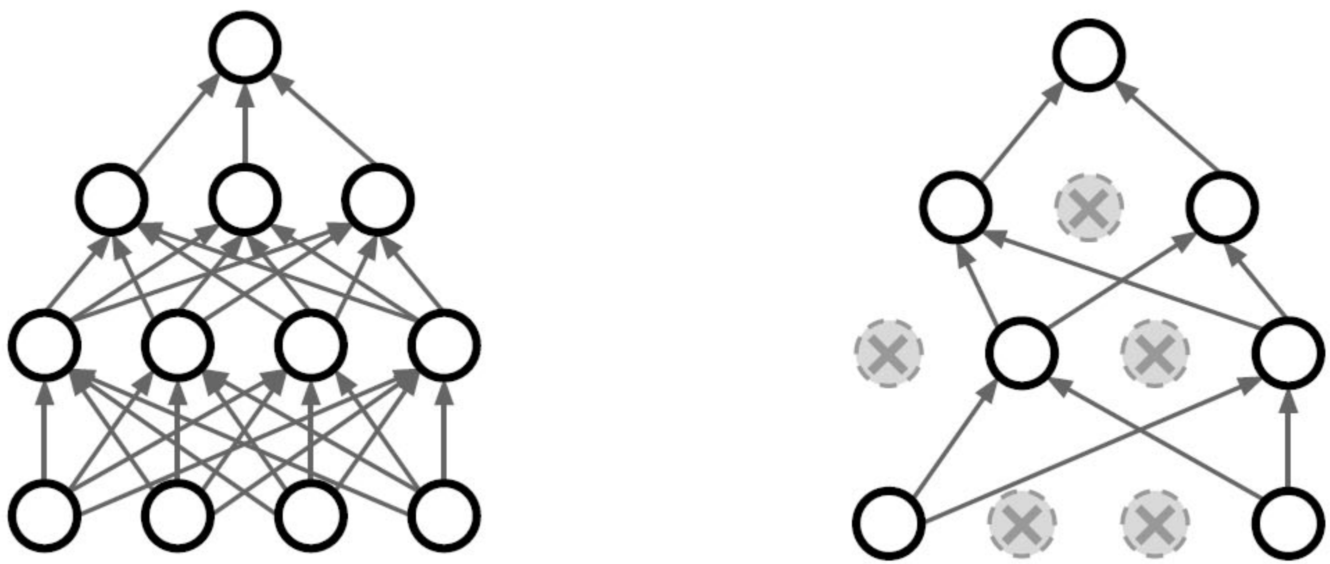
L1 regularization $R(W) = \sum_k \sum_l |W_{k,l}|$

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

برون‌اندازی

# Regularization: Dropout

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



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

## رگولاریزاسیون

برون‌اندازی

# Regularization: Dropout

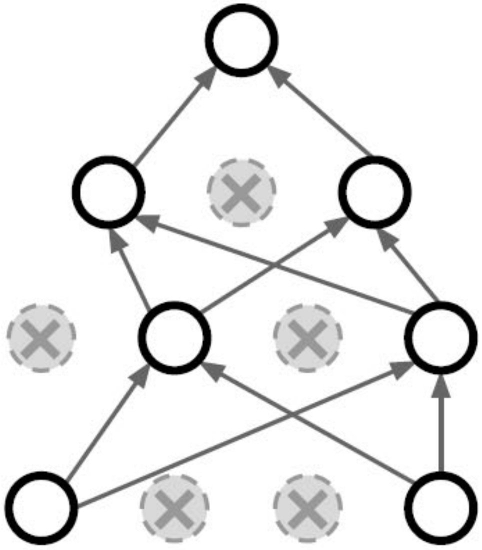
```
p = 0.5 # probability of keeping a unit active. higher = less dropout

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)
```

Example forward pass with a 3-layer network using dropout

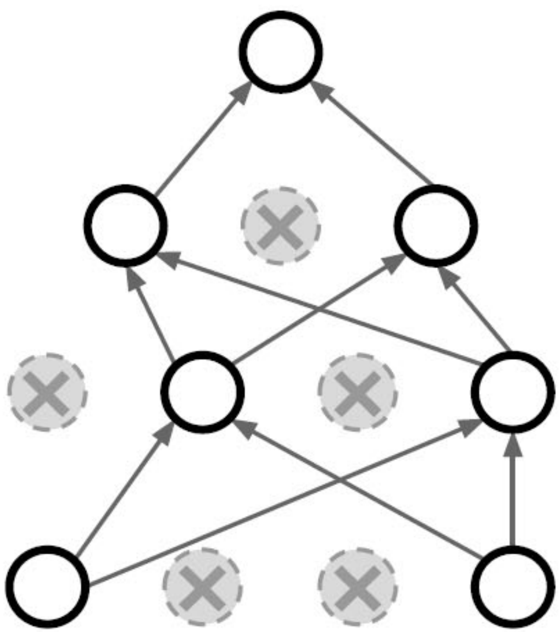


## رگولاریزاسیون

برون‌اندازی

# Regularization: Dropout

## How can this possibly be a good idea?



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

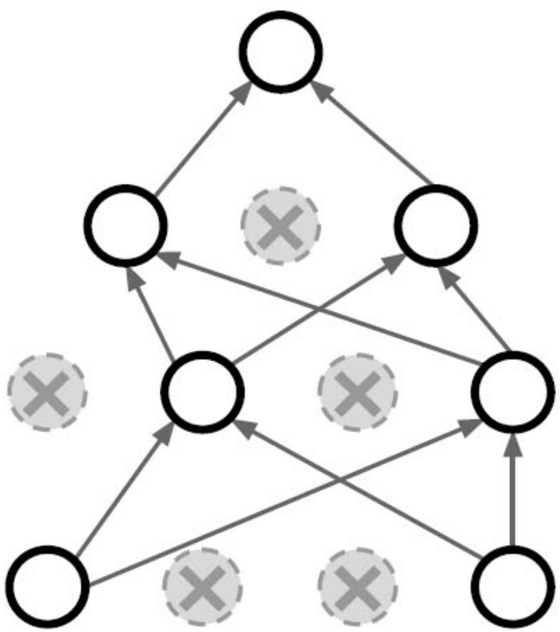



Ref: http://cs231n.stanford.edu/

برون‌اندازی

# Regularization: Dropout

How can this possibly be a good idea?



Another interpretation:

Dropout is training a large **ensemble** of models (that share parameters).

Each binary mask is one model

An FC layer with 4096 units has $2^{4096} \sim 10^{1233}$ possible masks!
Only $\sim 10^{82}$ atoms in the universe...



Ref: http://cs231n.stanford.edu/

# Dropout: Test time

Dropout makes our output random!

Output (label) Input (image) Random mask

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

Want to “average out” the randomness at test-time

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

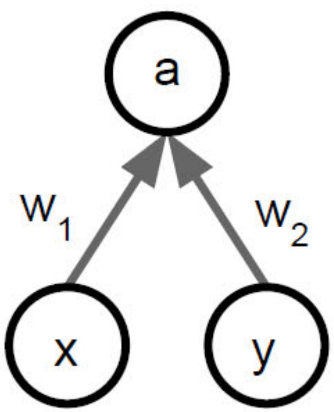
But this integral seems hard …

# Dropout: Test time

Want to approximate the integral

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

Consider a single neuron.

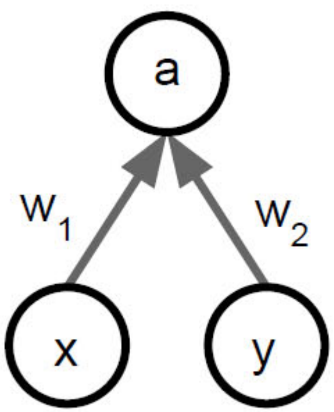


برون‌اندازی: در زمان آزمایش

# Dropout: Test time

Want to approximate the integral

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




Consider a single neuron.

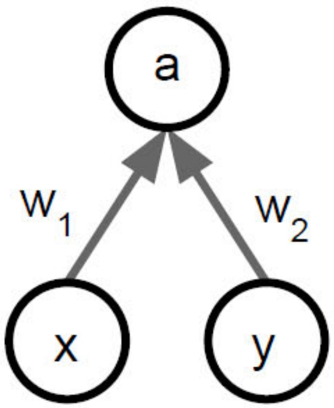
At test time we have: $E\big[a\big] = w_1 x + w_2 y$

بروناندازی: در زمان آزمایش

# Dropout: Test time

Want to approximate the integral

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




Consider a single neuron.

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

During training we have:

$$\begin{aligned} E\left[a\right] &= \frac{1}{4}(w_1 x + w_2 y) + \frac{1}{4}(w_1 x + 0y) \\ &+ \frac{1}{4}(0x + 0y) + \frac{1}{4}(0x + w_2 y) \\ &= \frac{1}{2}(w_1 x + w_2 y) \end{aligned}$$

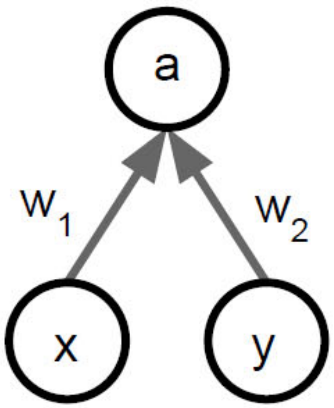


Ref: http://cs231n.stanford.edu/

# Dropout: Test time

Want to approximate the integral

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




Consider a single neuron.

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

During training we have:
$$E\left[a\right] = \frac{1}{4}(w_1 x + w_2 y) + \frac{1}{4}(w_1 x + 0y) + \frac{1}{4}(0x + 0y) + \frac{1}{4}(0x + w_2 y) = \frac{1}{2}(w_1 x + w_2 y)$$

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

# Dropout: Test time

```
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
```

At test time all neurons are active always
=> We must scale the activations so that for each neuron:
<u>output at test time</u> = <u>expected output at training time</u>



Ref: http://cs231n.stanford.edu/

# رگولاریزاسیون

## برون‌اندازی: خلاصه

## 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



Ref: http://cs231n.stanford.edu/

# رگولاریزاسیون

## برون‌اندازی وارون

## More common: “Inverted dropout”

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

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

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

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

test time is unchanged!

Ref: http://cs231n.stanford.edu/

## رگولاریزاسیون

یک الگوی متداول

# Regularization: A common pattern

**Training**: Add some kind of randomness

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

**Testing:** Average out randomness (sometimes approximate)

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

# Regularization: A common pattern

**Training**: Add some kind of randomness

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

**Testing:** Average out randomness (sometimes approximate)

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

**Example**: Batch Normalization

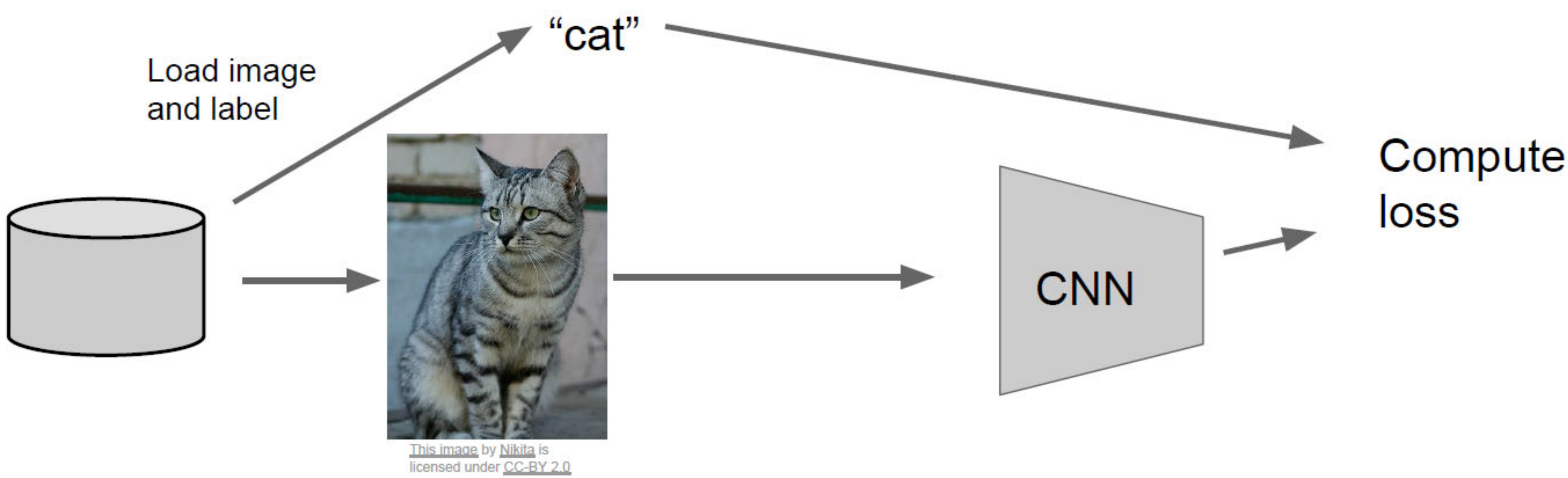
**Training**: Normalize using stats from random minibatches

**Testing**: Use fixed stats to normalize

## رگولاریزاسیون

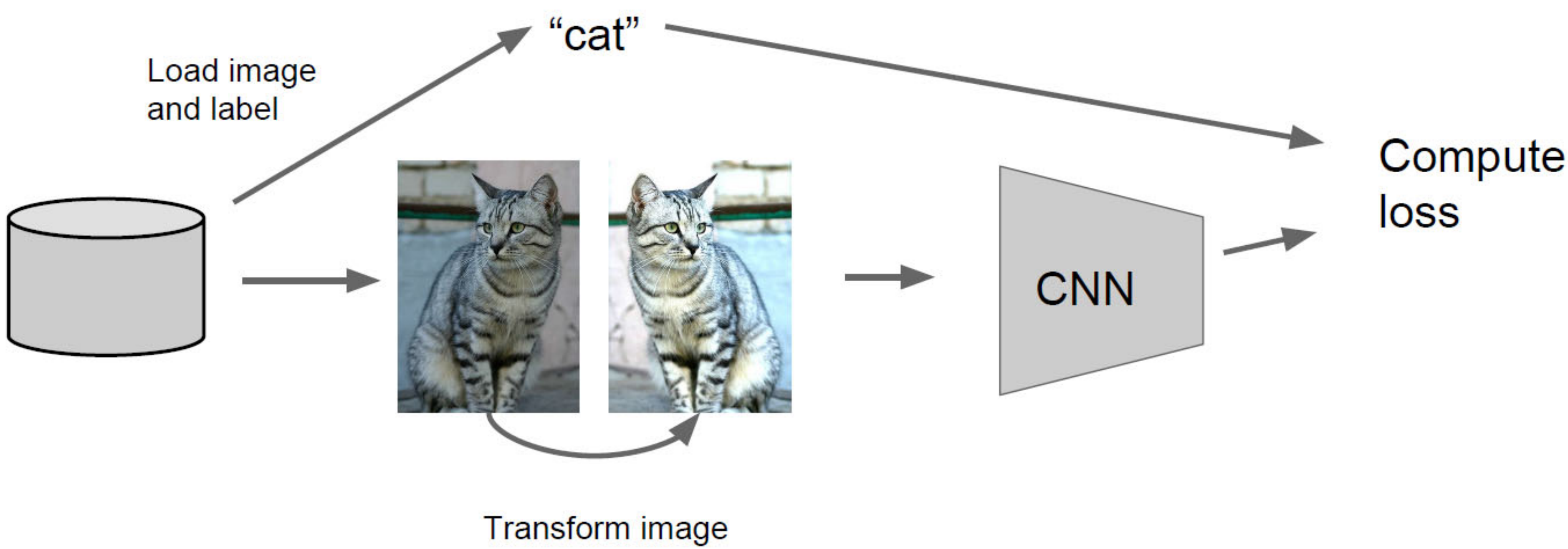
داده‌افزایی

# Regularization: Data Augmentation



دادهافزایی

# Regularization: Data Augmentation



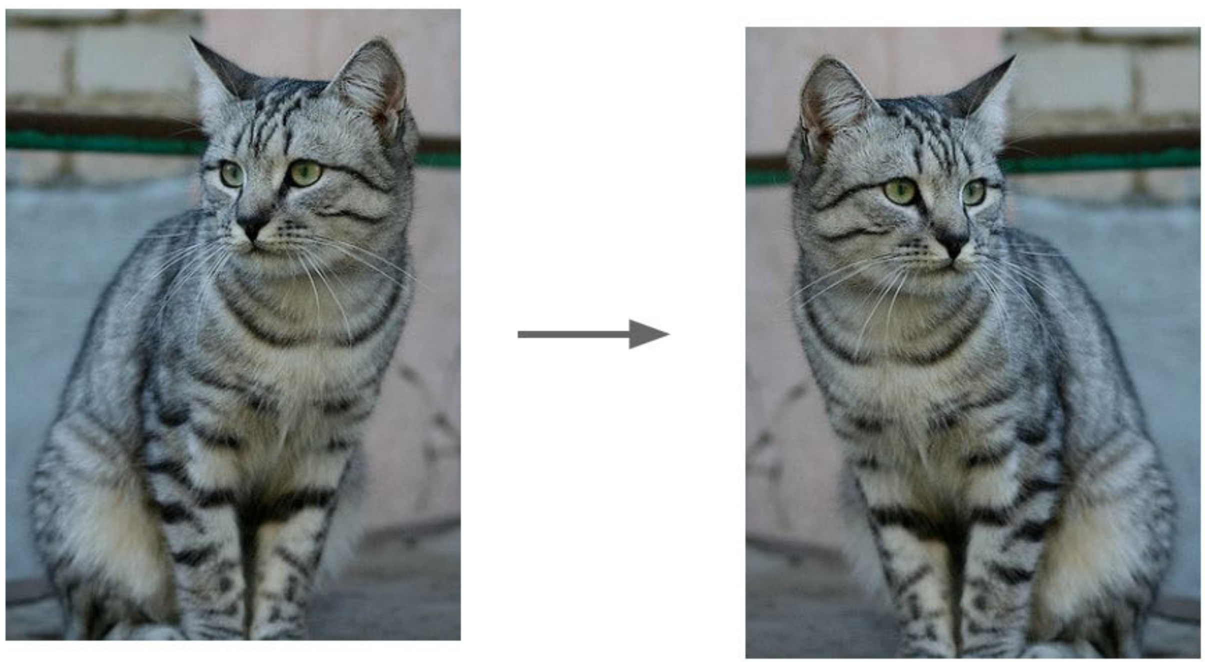



Ref: http://cs231n.stanford.edu/

# رگولاریزاسیون

## داده‌افزایی: برگردان افقی

# Data Augmentation
## Horizontal Flips





Ref: http://cs231n.stanford.edu/

دادهافزایی: کراپ‌ها و مقیاس‌های تصادفی

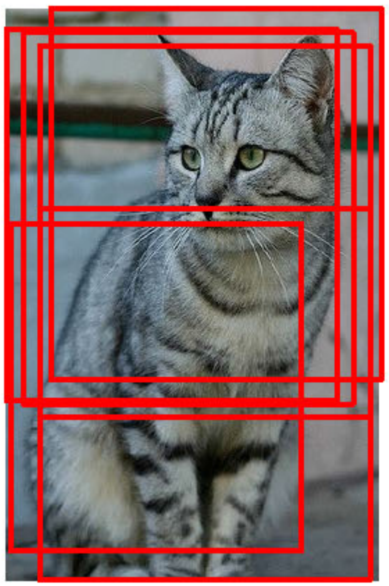
# Data Augmentation

## Random crops and scales

**Training**: sample random crops / scales

ResNet:

1. Pick random L in range [256, 480]
2. Resize training image, short side = L
3. Sample random 224 x 224 patch



داده‌افزایی : کراپ‌ها و مقیاس‌های تصادفی

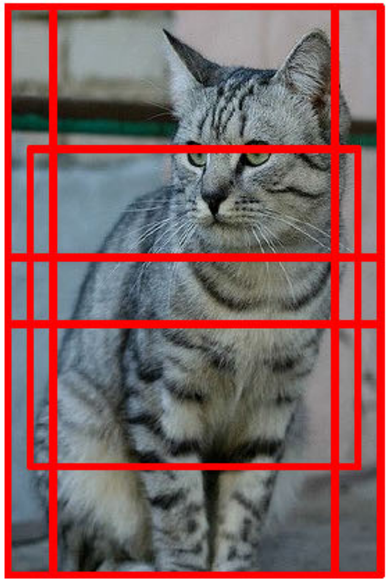
# Data Augmentation

## Random crops and scales

**Training**: sample random crops / scales

ResNet:

1. Pick random L in range [256, 480]
2. Resize training image, short side = L
3. Sample random 224 x 224 patch



**Testing**: average a fixed set of crops

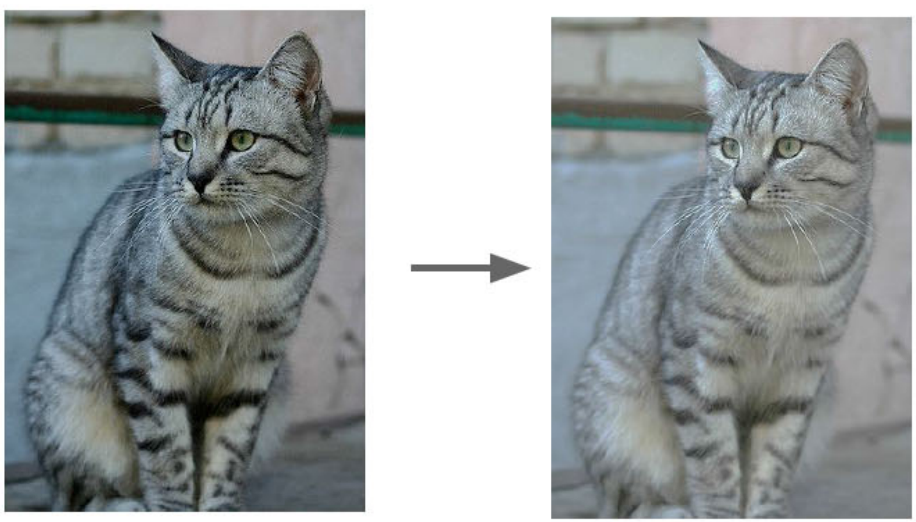
ResNet:

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

داده‌افزایی: رنگ‌پرانی

# Data Augmentation
## Color Jitter

Simple: Randomize
contrast and brightness





Ref: http://cs231n.stanford.edu/

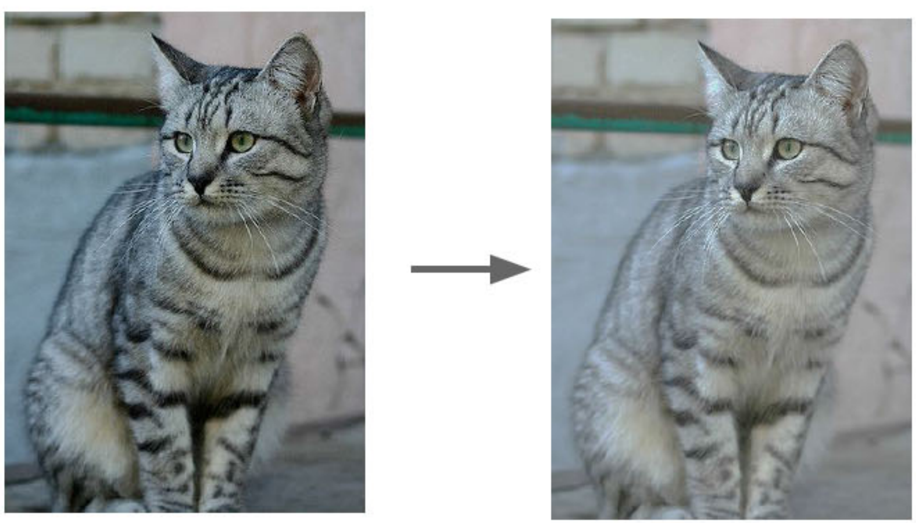
## رگولاریزاسیون

دادهافزایی: رنگ‌پرانی

# Data Augmentation

## Color Jitter

Simple: Randomize contrast and brightness



**More Complex**:

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

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



Ref: http://cs231n.stanford.edu/

# Data Augmentation

Get creative for your problem!

Random mix/combinations of :
- translation
- rotation
- stretching
- shearing,
- lens distortions, … (go crazy)



Ref: http://cs231n.stanford.edu/

# Regularization: A common pattern

**Training**: Add random noise
**Testing**: Marginalize over the noise

**Examples**:
Dropout
Batch Normalization
Data Augmentation

# Regularization: A common pattern

**Training**: Add random noise

**Testing**: Marginalize over the noise

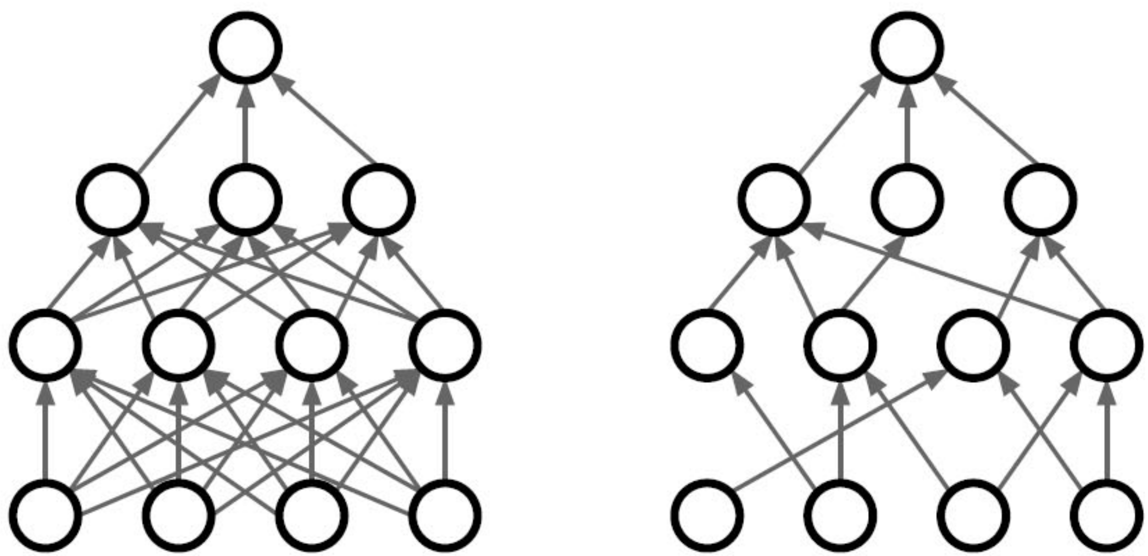
**Examples**:

Dropout

Batch Normalization

Data Augmentation

DropConnect



Wan et al, "Regularization of Neural Networks using DropConnect", ICML 2013

تلفیق ماکزیمم کسری

# Regularization: A common pattern

**Training**: Add random noise

**Testing**: Marginalize over the noise

**Examples**:

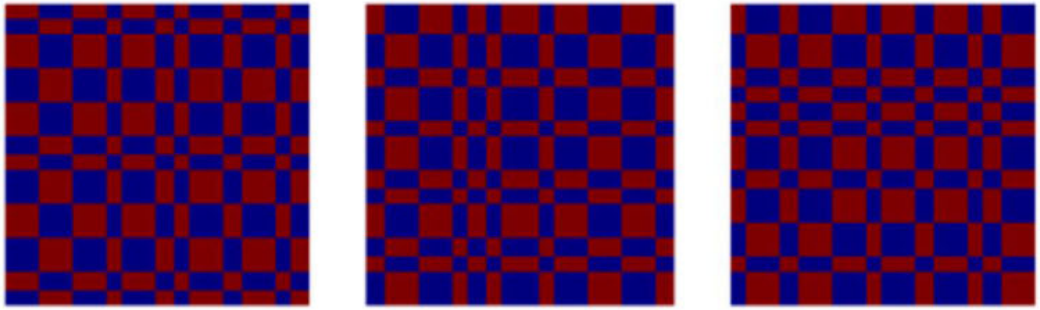
Dropout

Batch Normalization

Data Augmentation

DropConnect

Fractional Max Pooling



Graham, "Fractional Max Pooling", arXiv 2014

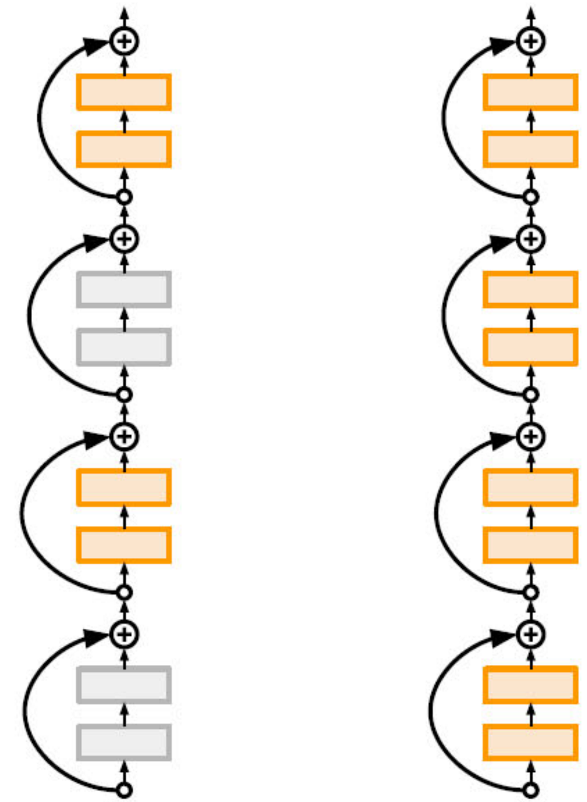
# رگولاریزاسیون

عمق اتفاقی

# Regularization: A common pattern

**Training**: Add random noise
**Testing**: Marginalize over the noise

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



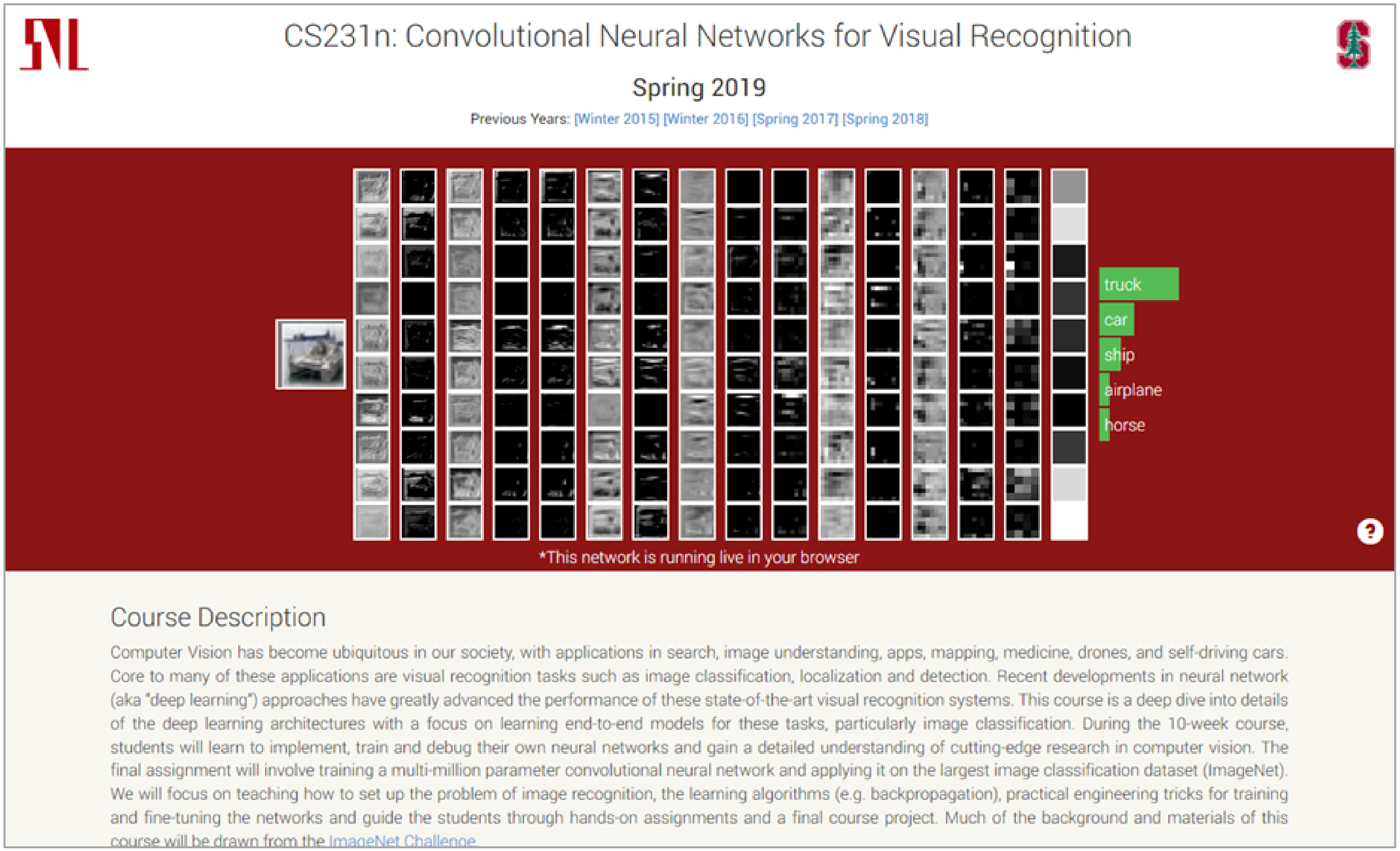
Huang et al, "Deep Networks with Stochastic Depth", ECCV 2016

مبانی یادگیری ماشینی

# ۱۰

# منابع

# منبع اصلی




## http://cs231n.stanford.edu

http://cs231n.github.io/neural-networks-1/
http://cs231n.github.io/neural-networks-2/
http://cs231n.github.io/neural-networks-3/
http://cs231n.github.io/neural-networks-case-study/

# منبع کمکی

## CMSC 35246 Deep Learning

### Spring 2017, University of Chicago

In many real world Machine Learning tasks, in particular those with perceptual input, such as vision and speech, the mapping from raw data to the output is often a complicated function with many factors of variation. Prior to 2010, to achieve decent performance on such tasks, significant effort had to be put to engineer hand crafted features. Deep Learning algorithms aim to learn feature hierarchies with features at higher levels in the hierarchy formed by the composition of lower level features. This automatic feature learning has been demonstrated to uncover underlying structure in the data leading to state-of-the-art results in tasks in vision, speech and rapidly in other domains as well.

This course aims to cover the basics of Deep Learning and some of the underlying theory with a particular focus on supervised Deep Learning, with a good coverage of unsupervised methods.

**Instructors:** Shubhendu Trivedi and Risi Kondor

- shubhendu@cs.uchicago.edu
- risi@cs.uchicago.edu

**Time:** Mondays and Wednesdays, 3.00pm-4.20pm, Ryerson 277

**Office hours:**

- Trivedi: M/W 4.30pm-5.30pm; F 4.30pm-6.30pm; Sa 3.00pm-5.00pm; by appointment
- Kondor: By appointment

### Prerequisites

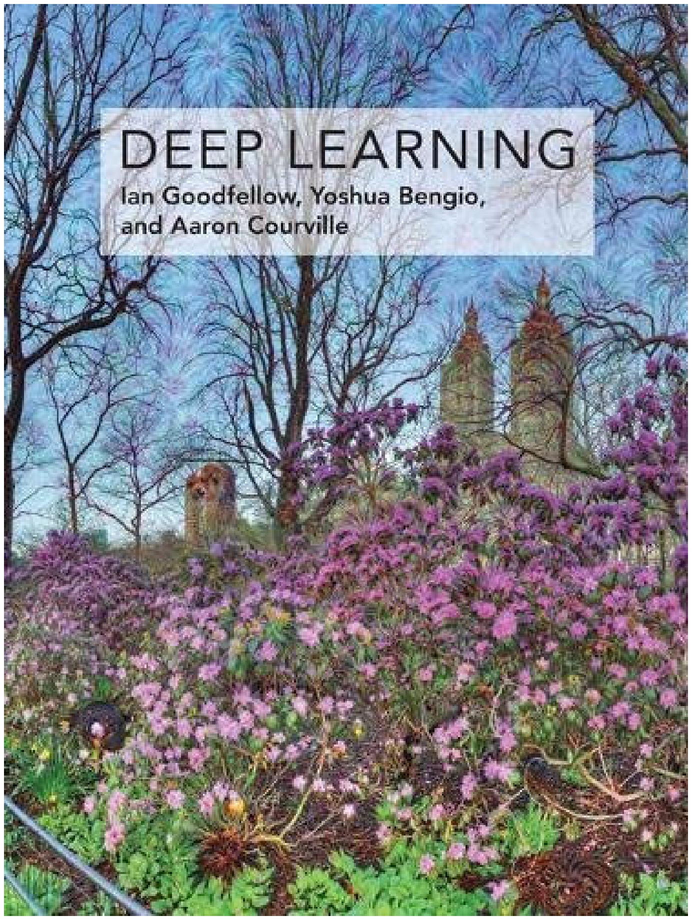
1. Graduate Machine Learning courses at the level of STAT 37710/CMSC 35400 or TTIC 31020 (STAT 27725/CMSC 25400 should be OK).
2. Familiarity with basic Probability Theory, Linear Algebra, Calculus
3. Programming proficiency in Python (although you should be fine if you have extensive experience in some other high level language)

### Syllabus

See schedule

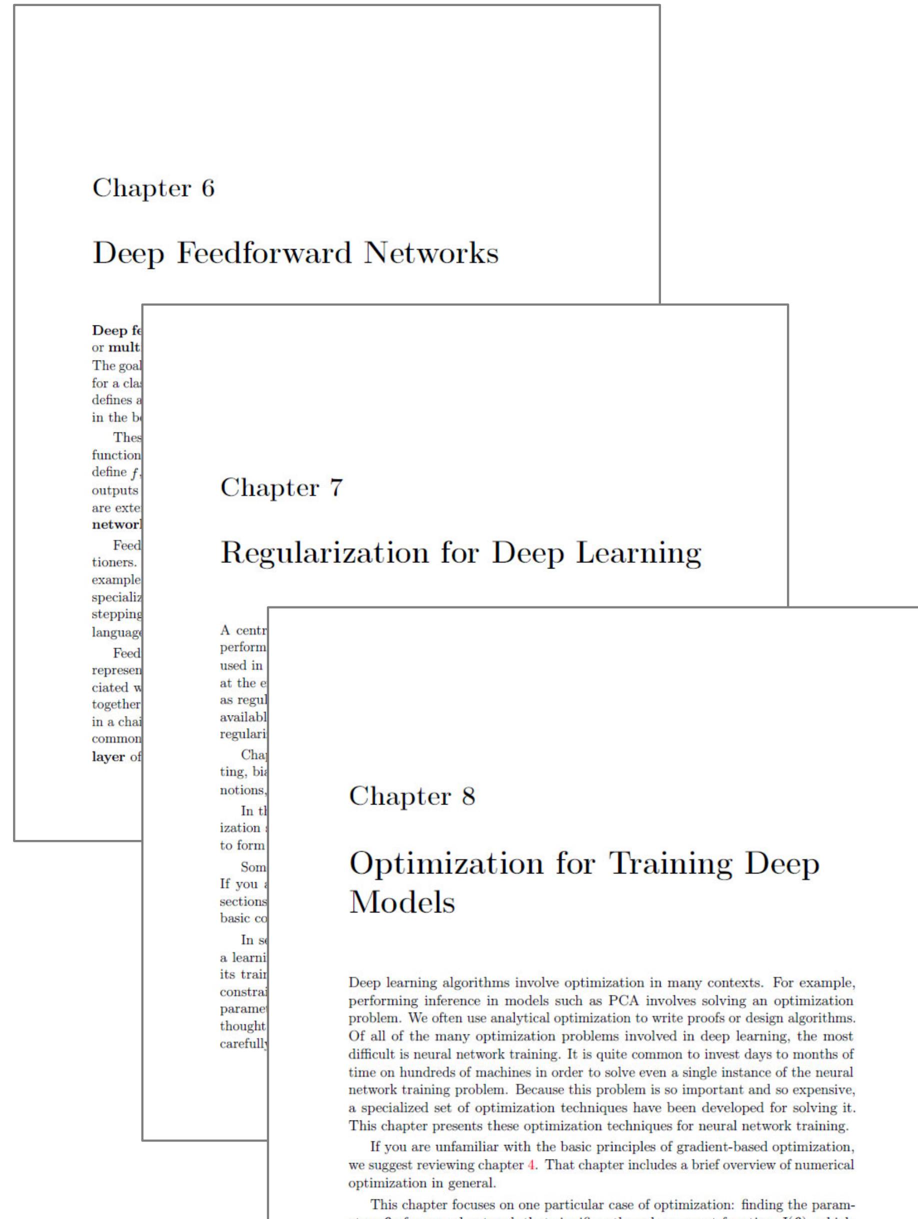
https://ttic.uchicago.edu/~shubhendu/Pages/CMSC35246.html

# منبع کمکی




I. Goodfellow, Y. Bengio, A. Courville,
**Deep Learning**,
MIT Press, 2016.

**Chapters 6, 7, 8**


Chapter 6

Deep Feedforward Networks

Chapter 7

Regularization for Deep Learning

Chapter 8

Optimization for Training Deep Models

Deep learning algorithms involve optimization in many contexts. For example, performing inference in models such as PCA involves solving an optimization problem. We often use analytical optimization to write proofs or design algorithms. Of all of the many optimization problems involved in deep learning, the most difficult is neural network training. It is quite common to invest days to months of time on hundreds of machines in order to solve even a single instance of the neural network training problem. Because this problem is so important and so expensive, a specialized set of optimization techniques have been developed for solving it. This chapter presents these optimization techniques for neural network training.

If you are unfamiliar with the basic principles of gradient-based optimization, we suggest reviewing chapter 4. That chapter includes a brief overview of numerical optimization in general.

This chapter focuses on one particular case of optimization: finding the parameters $\boldsymbol{\theta}$ of a neural network that significantly reduce a cost function $J(\boldsymbol{\theta})$, which

Prepared by Kazim Fouladi | Spring 2025 | 4th Edition