

# Deep Learning (for Computer Vision)

Arjun Jain

# Agenda this Week

- Building blocks
  - Linear
  - Convolution
  - Max Pooling
  - Dropout
  - Cross Entropy
  - Batch Normalization
- Weight initializations, data preprocessing
- Choosing hyperparameters, baby sitting the learning process

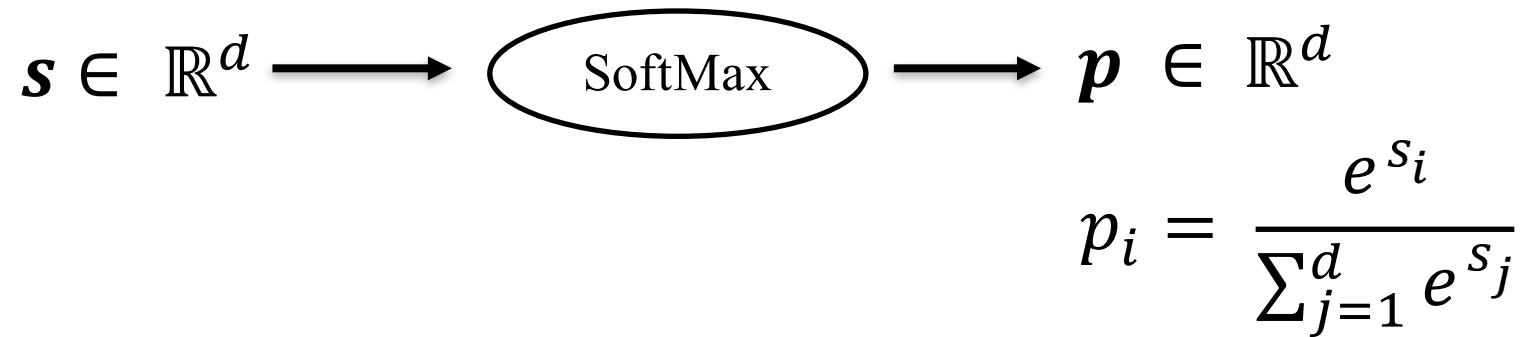
# Sources

A lot of the material has been shamelessly and gratefully collected from:

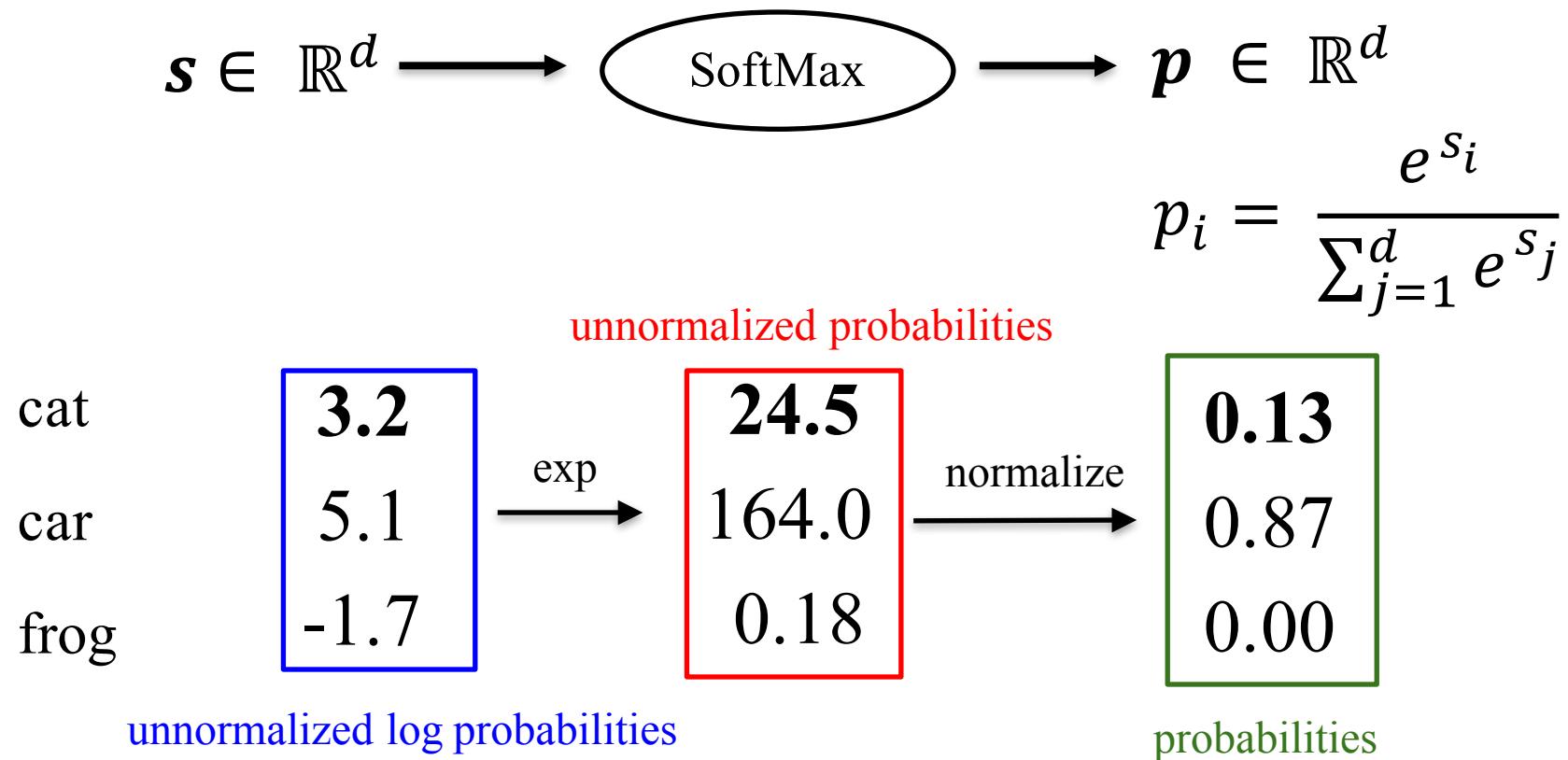
- <http://cs231n.stanford.edu/>

# Building Blocks: **SoftMax**

# Building Blocks – SoftMax



# Building Blocks – SoftMax



# Building Blocks – SoftMax



$$p_i = \frac{e^{s_i}}{\sum_{j=1}^d e^{s_j}}$$

```
> n = torch.DoubleTensor({3.2,5.1,-1.7})  
> smax = nn.SoftMax()  
> m = smax:forward(n)  
> m_ = nn.Exp():forward(n)  
> m_ = m_ / m_:sum()
```

```
> =n  
3.2000  
5.1000  
-1.7000  
[torch.DoubleTensor of size 3]
```

```
> =m  
0.1300  
0.8690  
0.0010  
[torch.DoubleTensor of size 3]
```

```
> =m_  
0.1300  
0.8690  
0.0010  
[torch.DoubleTensor of size 3]
```

$$\frac{\partial p_j}{\partial s_k} = \begin{cases} p_j(1 - p_j) & \text{if } j = k \\ -p_j p_k & \text{if } j \neq k \end{cases}$$

# Building Blocks

## Cross Entropy

# Building Blocks – Cross Entropy

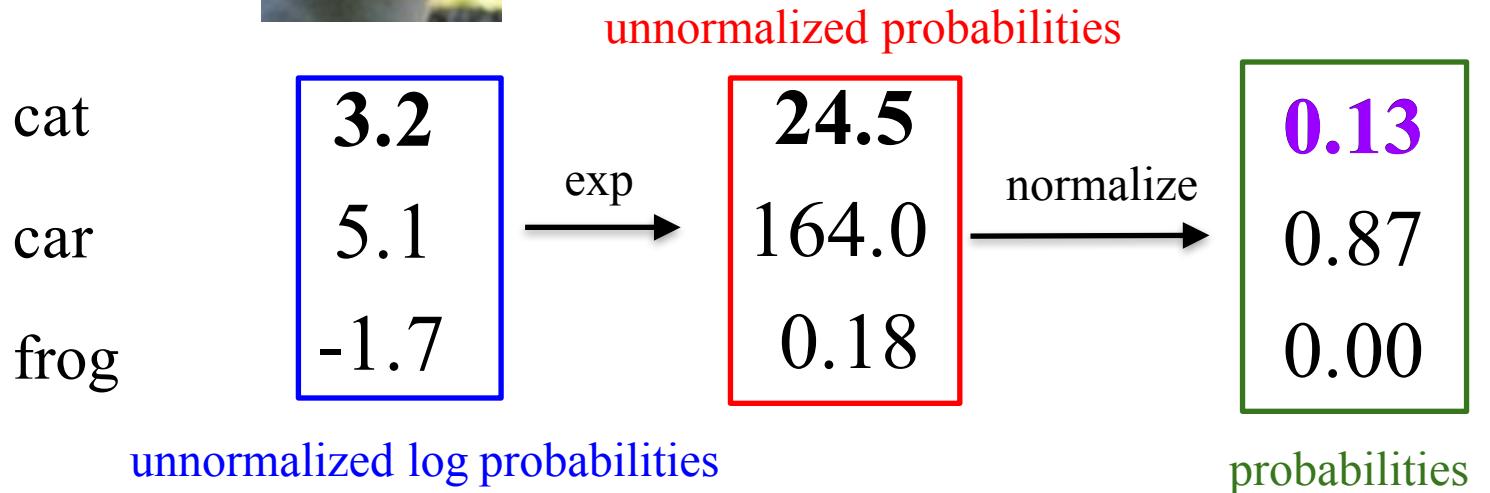


$$s = f(x_i; W) \quad L_i = -y_i \cdot \log\left(\frac{e^{s_i}}{\sum_j e^{s_j}}\right) \quad L = \sum_i L_i$$

$y_i$  is 1 (and 0 otherwise) if and only if sample belongs to class  $i$

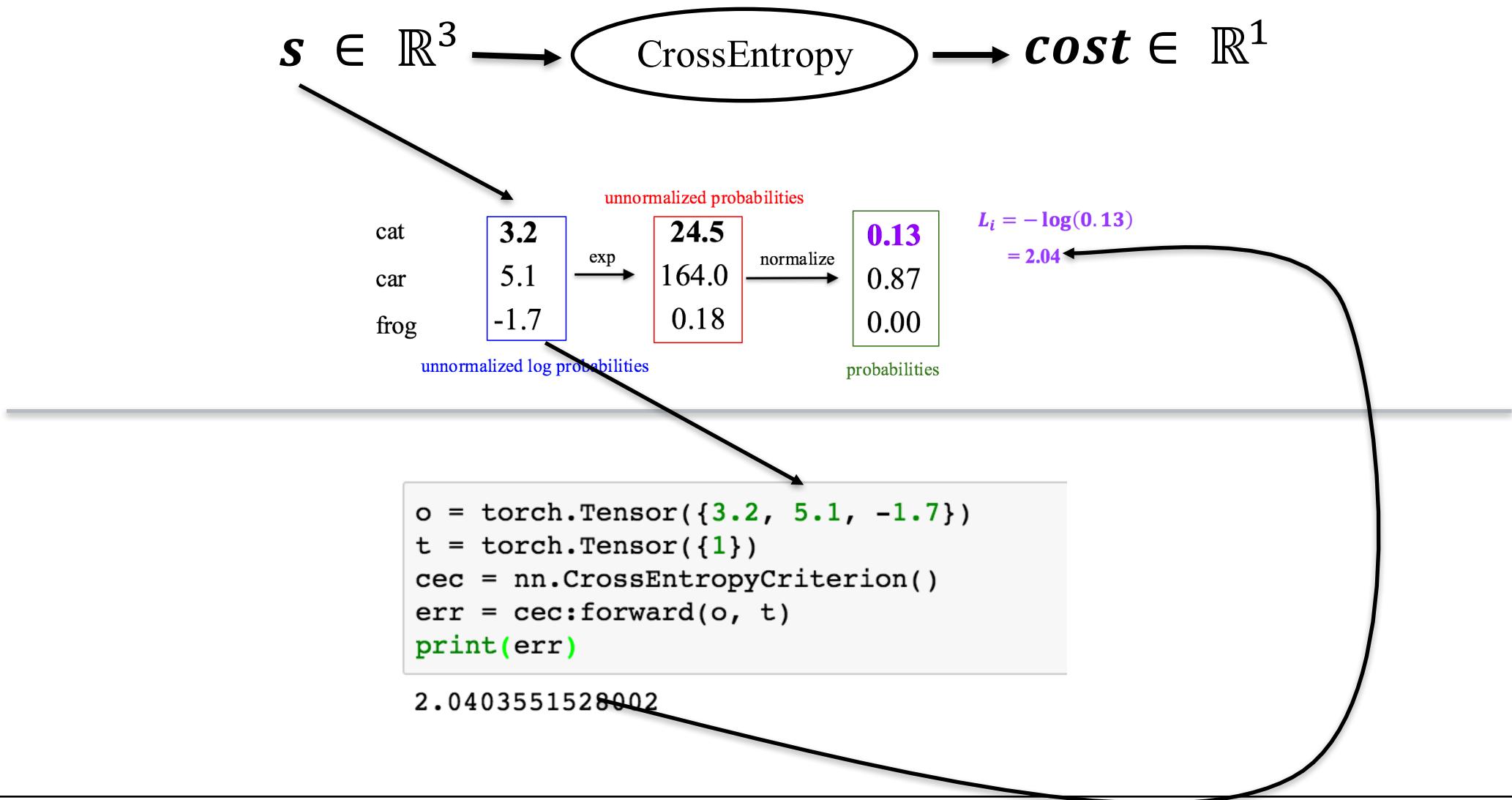
# Building Blocks – Cross Entropy

$s \in \mathbb{R}^d \rightarrow \text{CrossEntropy} \rightarrow \text{cost} \in \mathbb{R}^1$

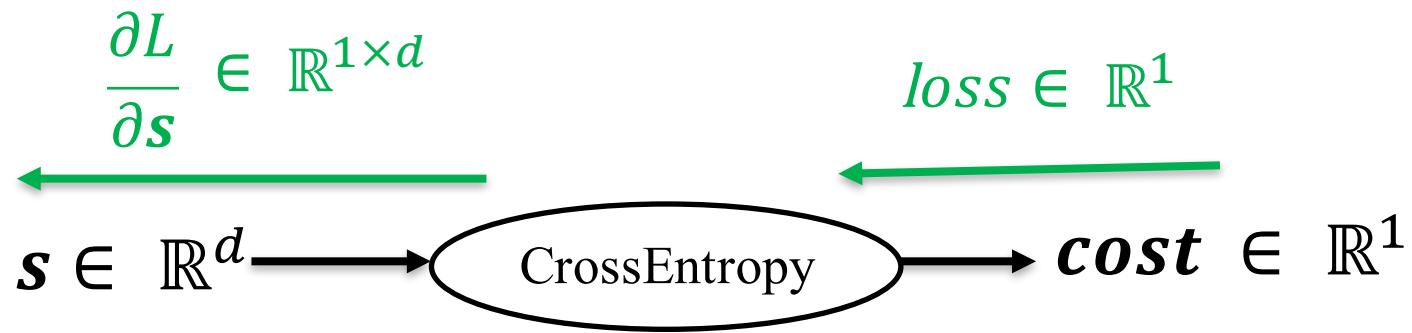


$$\begin{aligned}L_i &= -\log(0.13) \\&= 2.04\end{aligned}$$

# Building Blocks – Cross Entropy (in torch7)



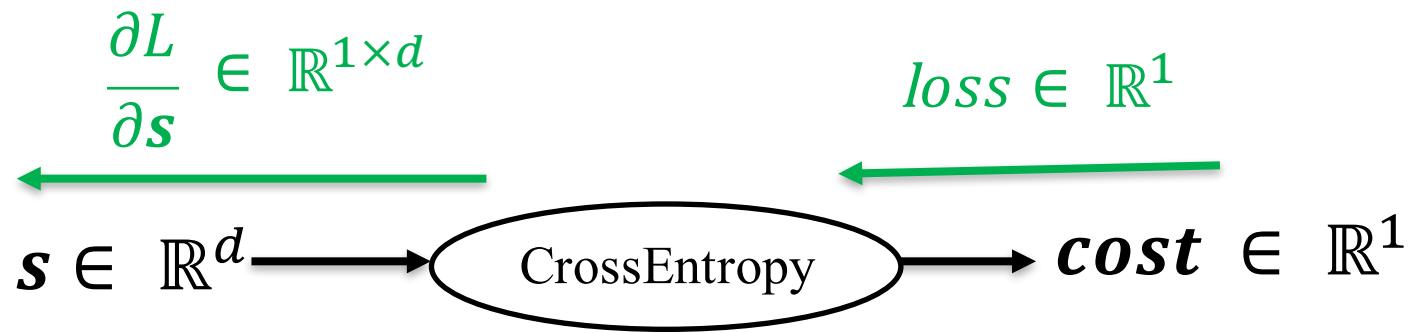
# Building Blocks – Cross Entropy



$$s = f(x_i; W) \quad L_i = -y_i \cdot \log\left(\frac{e^{s_i}}{\sum_j e^{s_j}}\right) \in \mathbb{R}^{1 \times d}$$

$y_i$  is 1 (and 0 otherwise) if and only if sample belongs to class  $i$

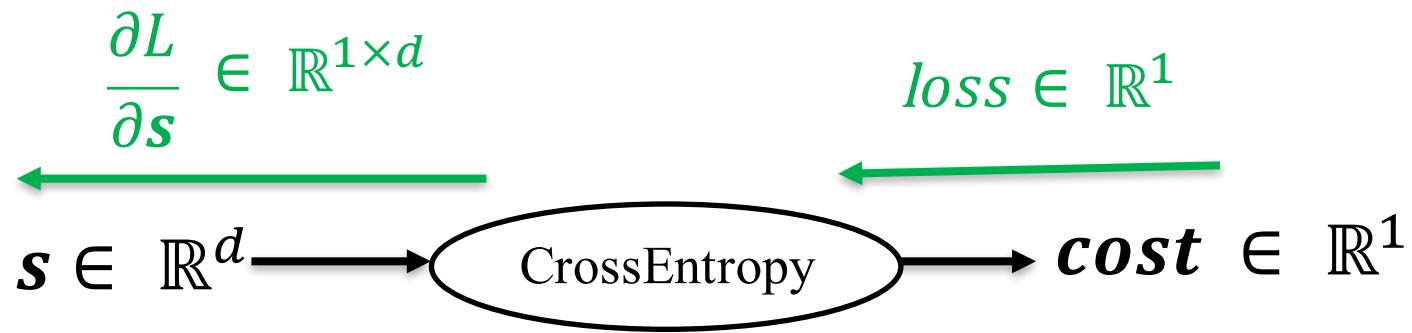
# Building Blocks – Cross Entropy



$$s = f(x_i; W) \quad p_i = \frac{e^{s_i}}{\sum_j e^{s_j}} \quad L_i = -y_i \cdot \log\left(\frac{e^{s_i}}{\sum_j e^{s_j}}\right) \quad L = \sum_i L_i$$

$y_i$  is 1 (and 0 otherwise) if and only if sample belongs to class  $i$

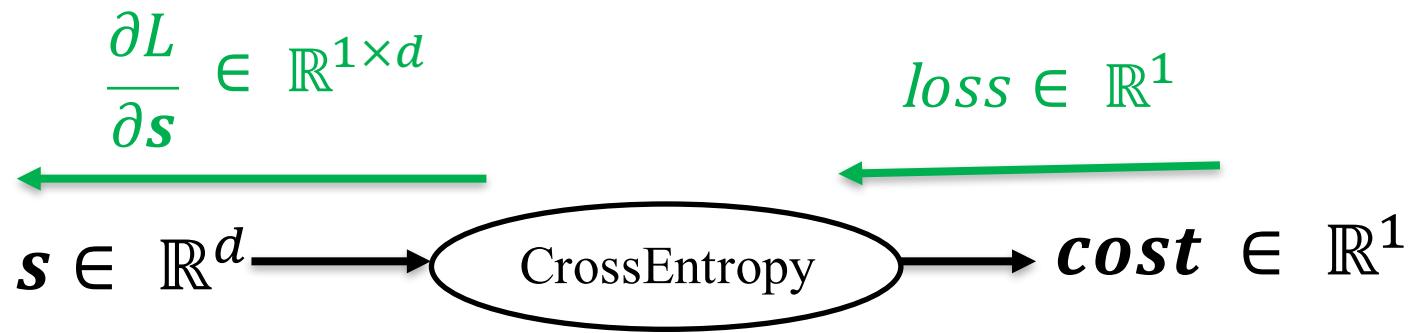
# Building Blocks – Cross Entropy



$$s = f(x_i; W) \quad p_i = \frac{e^{s_i}}{\sum_j e^{s_j}} \quad L_i = -y_i \cdot \log(p_i) \quad L = \sum_i L_i$$

$y_i$  is 1 (and 0 otherwise) if and only if sample belongs to class  $i$

# Building Blocks – Cross Entropy

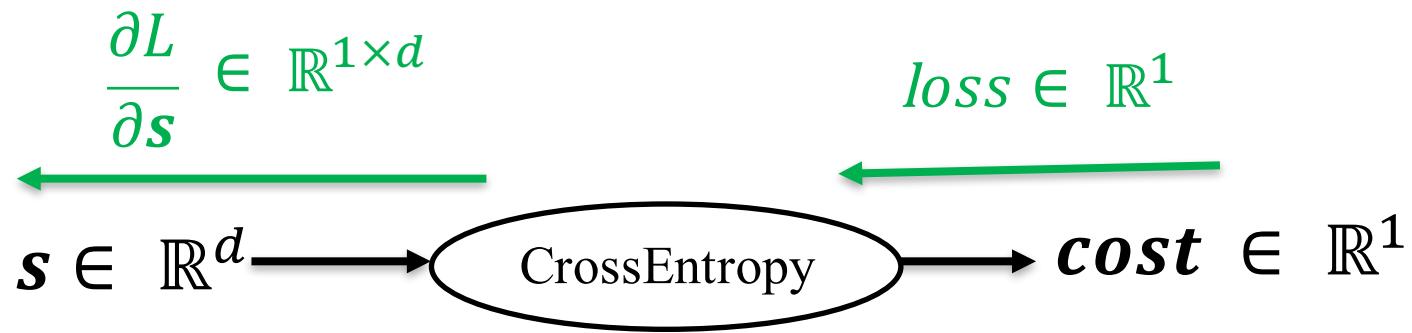


$$s = f(x_i; W) \quad p_i = \frac{e^{s_i}}{\sum_j e^{s_j}} \quad L_i = -y_i \cdot \log(p_i) \quad L = \sum_i L_i$$

$y_i$  is 1 (and 0 otherwise) if and only if sample belongs to class  $i$

What is  $\frac{\partial L}{\partial s_k}$  ????

# Building Blocks – Cross Entropy



$$s = f(x_i; W) \quad p_i = \frac{e^{s_i}}{\sum_j e^{s_j}} \quad L_i = -y_i \cdot \log(p_i) \quad L = \sum_i L_i$$

$y_i$  is 1 (and 0 otherwise) if and only if sample belongs to class  $i$

$$\frac{\partial L}{\partial s_k} = p_k - y_k$$

# Building Blocks – Cross Entropy

$$\frac{\partial L}{\partial s_k} = p_k - y_k$$

$$\frac{\partial L}{\partial s_k} = - \sum_j y_j \frac{\partial \log p_j}{\partial s_k} = - \sum_j y_j \frac{1}{p_j} \frac{\partial p_j}{\partial s_k}$$

$$\frac{\partial p_j}{\partial s_k} = \begin{cases} p_j(1-p_j) & \text{if } j = k \\ -p_j p_k & \text{if } j \neq k \end{cases}$$

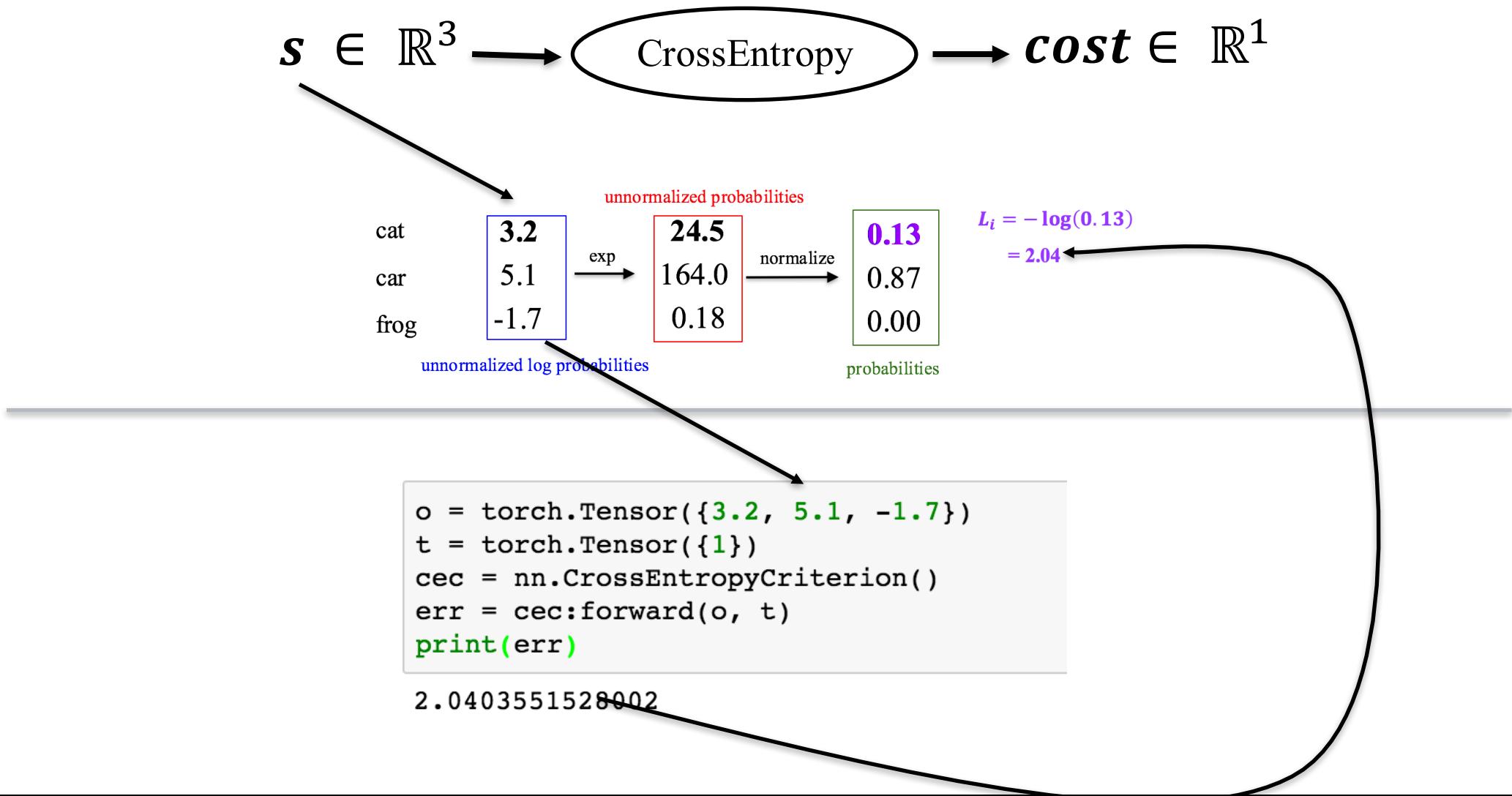
$$= -y_k(1-p_k) - \sum_{j \neq k} y_j \frac{1}{p_j} (-p_j p_k)$$

$$= -y_k(1-p_k) + \sum_{j \neq k} y_j(p_k)$$

$$= -y_k + y_k p_k + \sum_{j \neq k} y_j(p_k)$$

$$= p_k \left( \sum_j y_j \right) - y_k = p_k - y_k$$

# Building Blocks – Cross Entropy (in torch7)



# Building Blocks – Cross Entropy (in torch7)

$$\frac{\partial L}{\partial s} \in \mathbb{R}^{1 \times d}$$

$$s \in \mathbb{R}^3 \quad \text{CrossEntropy} \quad loss \in \mathbb{R}^1$$

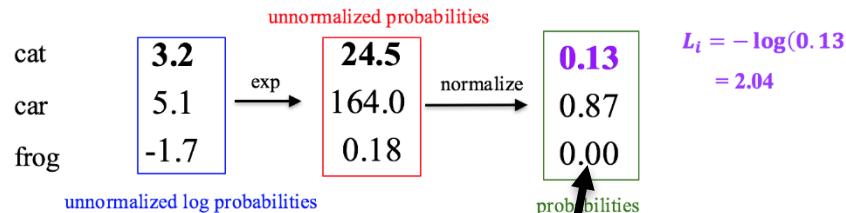
$$cost \in \mathbb{R}^1$$

```
dl_do = cec:backward(o, t)  
print(dl_do)
```

```
-0.8700  
0.8690  
0.0010
```

```
target = torch.Tensor([1, 0, 0])  
dl_do_manual = ohat - target  
print(dl_do_manual)
```

```
-0.8700  
0.8690  
0.0010
```

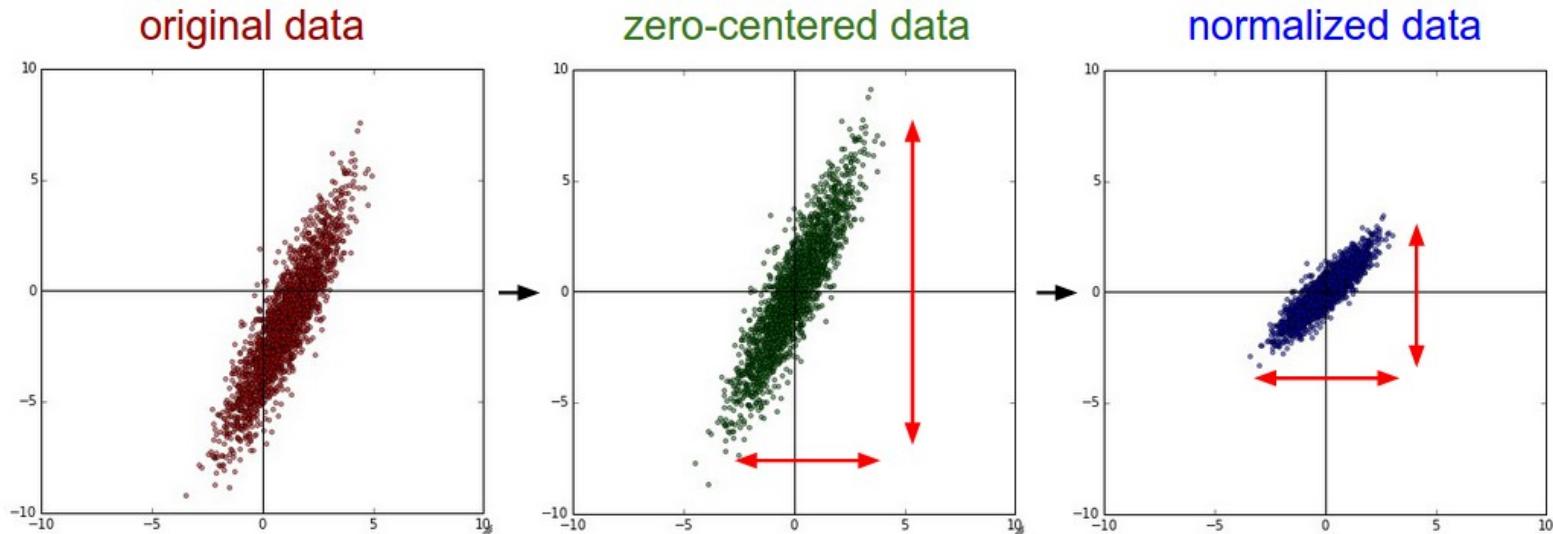


```
ohat = nn.SoftMax():forward(o)  
print(ohat)
```

```
0.1300  
0.8690  
0.0010
```

# Data Preprocessing

# Data Preprocessing



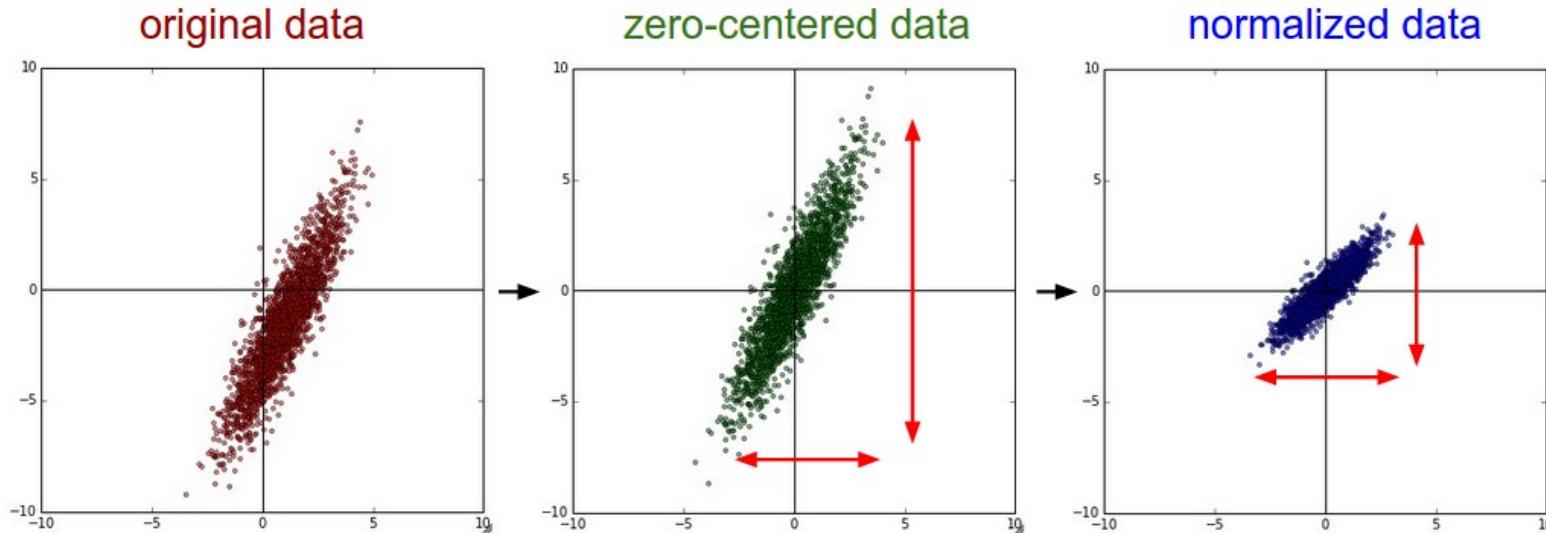
- Assume  $\mathbf{X}$  [ $N \times D$ ] is data matrix, each example in a row
- So, in our case we have 10 examples, each 4D

```
[> X = torch.rand(10,4)
 [> mean = X:mean(1)
 [> std = X:std(1)
 [> meanrepeated = torch.repeatTensor(mean, 10,1)
 [> stdrepeated = torch.repeatTensor(std, 10,1)
 [> =X
  0.4822  0.2837  0.6833  0.9955
  0.1640  0.0252  0.2801  0.5854
  0.7129  0.8695  0.2564  0.8241
  0.7037  0.8237  0.2027  0.1058
  0.6759  0.2490  0.0476  0.3699
  0.5486  0.4739  0.2690  0.7482
  0.4623  0.5303  0.5062  0.1874
  0.3652  0.8691  0.6733  0.3607
  0.4448  0.8708  0.8118  0.7951
  0.2013  0.6639  0.5152  0.7776
 [torch.DoubleTensor of size 10x4]

 [> =mean
  0.4761  0.5659  0.4246  0.5750
 [torch.DoubleTensor of size 1x4]

 [> =std
  0.1942  0.3047  0.2491  0.3013
 [torch.DoubleTensor of size 1x4]]
```

# Data Preprocessing

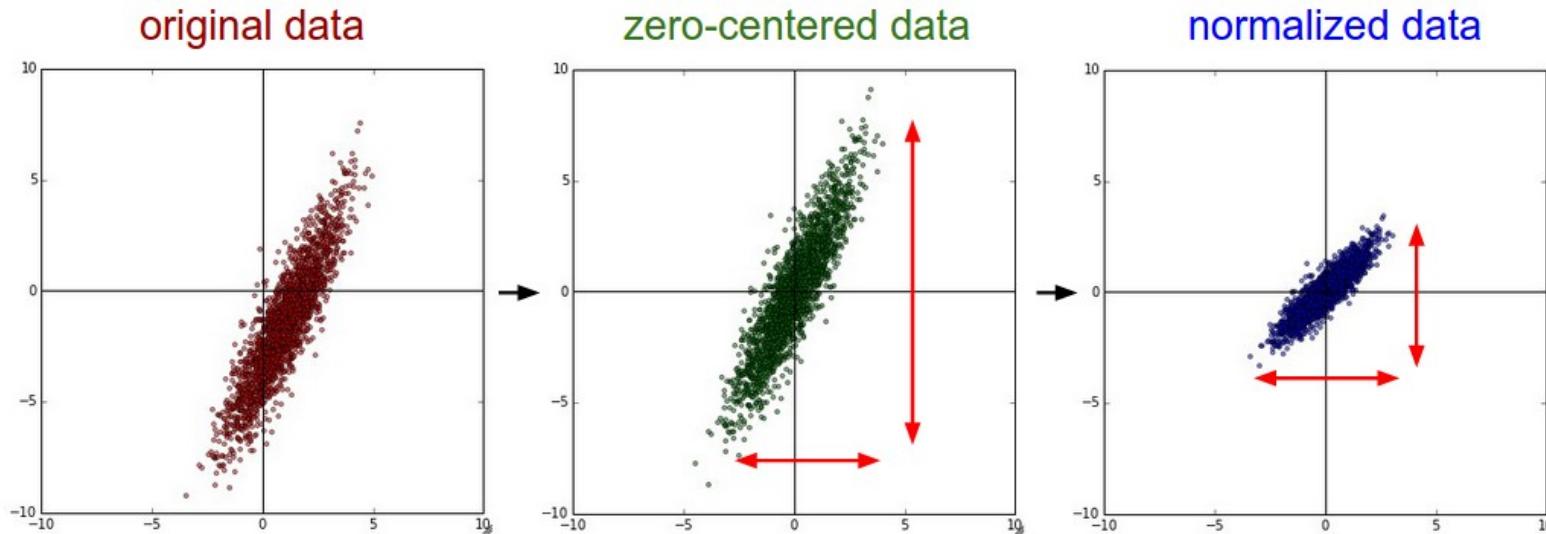


- Assume  $\mathbf{X}$  [ $N \times D$ ] is data matrix, each example in a row
- So, in our case we have 10 examples, each 4D

```
[> X = torch.rand(10,4)
[> mean = X:mean(1)
[> std = X:std(1)
[> meanrepeated = torch.repeatTensor(mean, 10,1)
[> stdrepeated = torch.repeatTensor(std, 10,1)
[> =meanrepeated
 0.4761  0.5659  0.4246  0.5750
 0.4761  0.5659  0.4246  0.5750
 0.4761  0.5659  0.4246  0.5750
 0.4761  0.5659  0.4246  0.5750
 0.4761  0.5659  0.4246  0.5750
 0.4761  0.5659  0.4246  0.5750
 0.4761  0.5659  0.4246  0.5750
 0.4761  0.5659  0.4246  0.5750
 0.4761  0.5659  0.4246  0.5750
 0.4761  0.5659  0.4246  0.5750
[torch.DoubleTensor of size 10x4]
```

```
[> =stdrepeated
 0.1942  0.3047  0.2491  0.3013
 0.1942  0.3047  0.2491  0.3013
 0.1942  0.3047  0.2491  0.3013
 0.1942  0.3047  0.2491  0.3013
 0.1942  0.3047  0.2491  0.3013
 0.1942  0.3047  0.2491  0.3013
 0.1942  0.3047  0.2491  0.3013
 0.1942  0.3047  0.2491  0.3013
 0.1942  0.3047  0.2491  0.3013
 0.1942  0.3047  0.2491  0.3013
[torch.DoubleTensor of size 10x4]
```

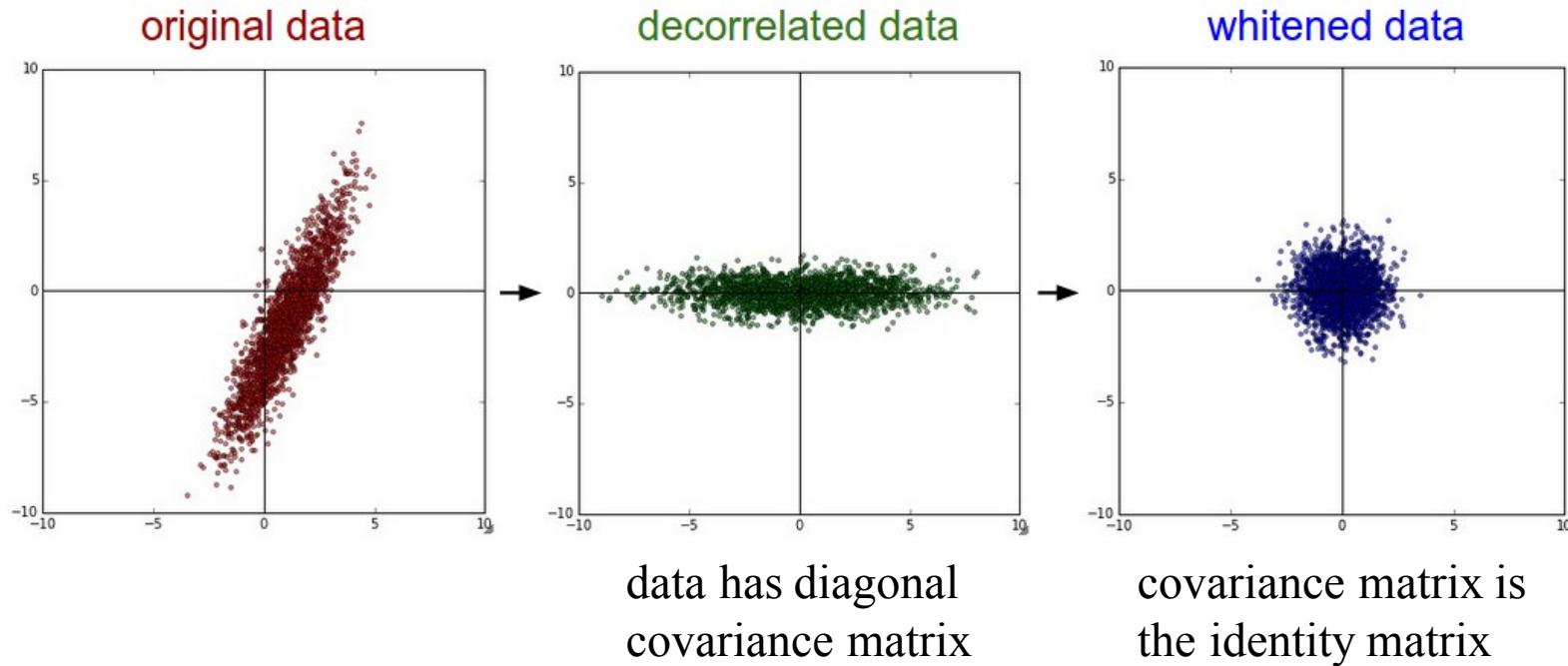
# Data Preprocessing



- Assume  $\mathbf{X}$  [ $N \times D$ ] is data matrix, each example in a row
- So, in our case we have 10 examples, each 4D

```
[> X = torch.rand(10,4)
[> mean = X:mean(1)
[> std = X:std(1)
[> meanrepeated = torch.repeatTensor(mean, 10,1)
[> stdrepeated = torch.repeatTensor(std, 10,1)
[> X = X - meanrepeated
[> X=X:cdiv(stdrepeated)
[> =X
 0.0315 -0.9264  1.0390  1.3959
-1.6070 -1.7749 -0.5799  0.0346
 1.2192  0.9964 -0.6751  0.8270
 1.1719  0.8461 -0.8909 -1.5573
 1.0287 -1.0402 -1.5134 -0.6806
 0.3735 -0.3019 -0.6247  0.5749
-0.0709 -0.1170  0.3280 -1.2866
-0.5710  0.9953  0.9986 -0.7111
-0.1611  1.0008  1.5545  0.7308
-1.4149  0.3216  0.3639  0.6725
[torch.DoubleTensor of size 10x4]
```

# Data Preprocessing



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

# Data Preprocessing

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

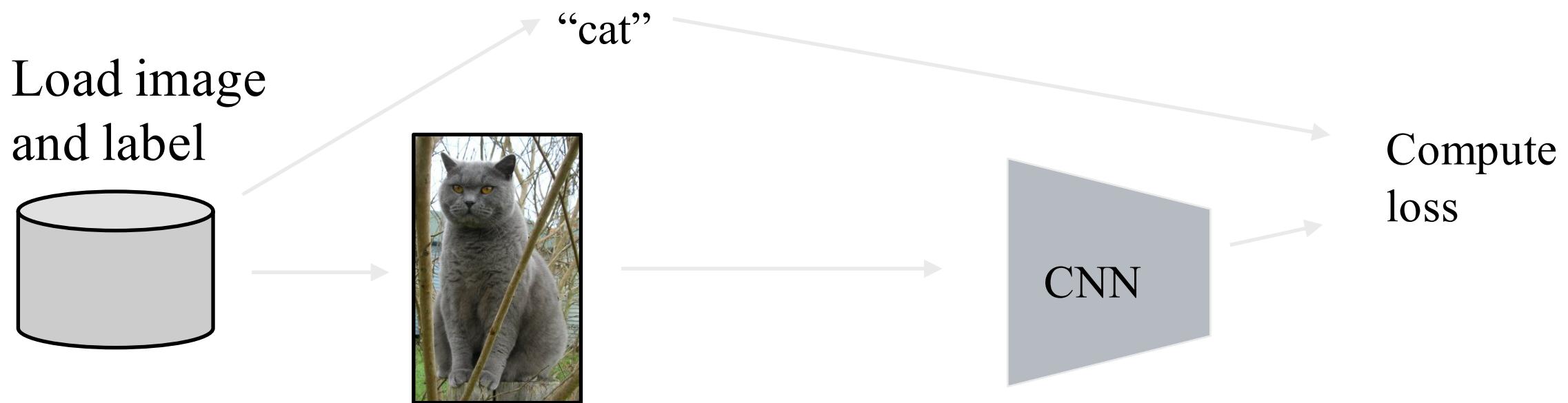
e.g. consider CIFAR-10 example with [32,32,3] images

- 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)

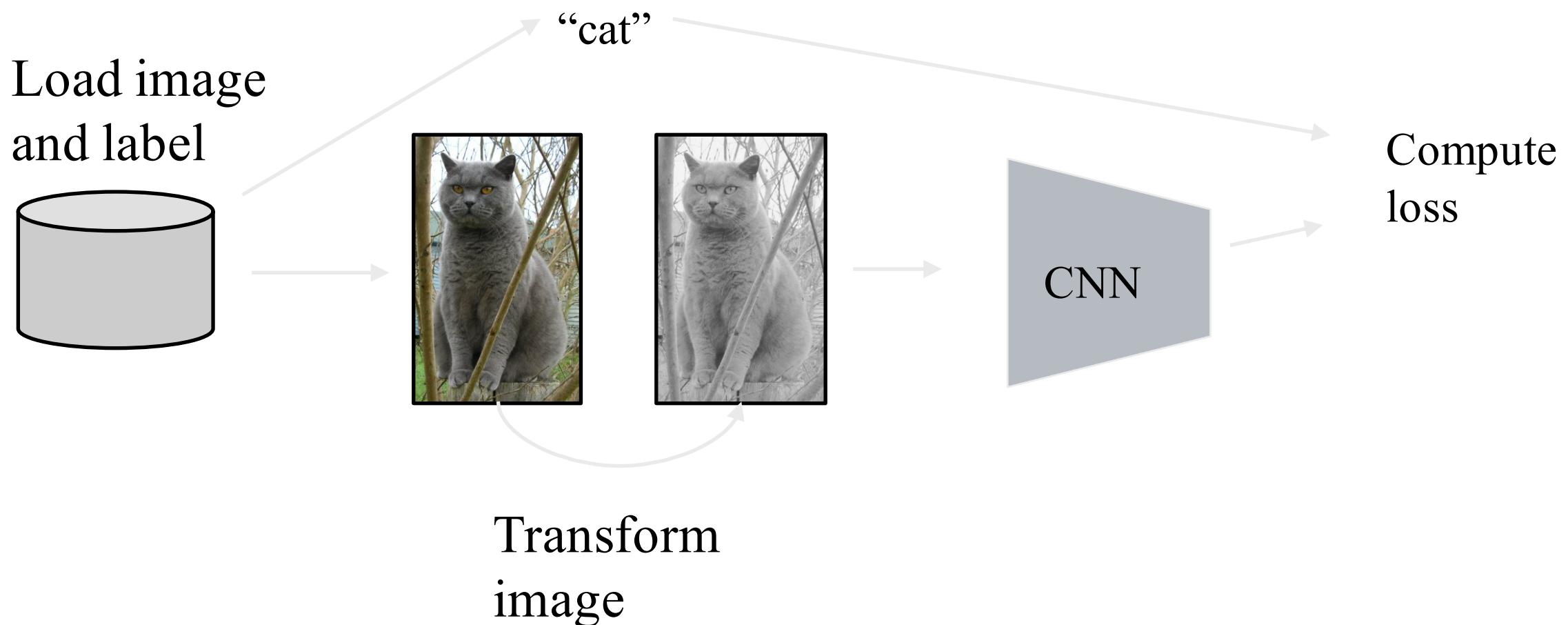
Not common to normalize variance, to do PCA or whitening

# Data Augmentation

# Data Augmentation

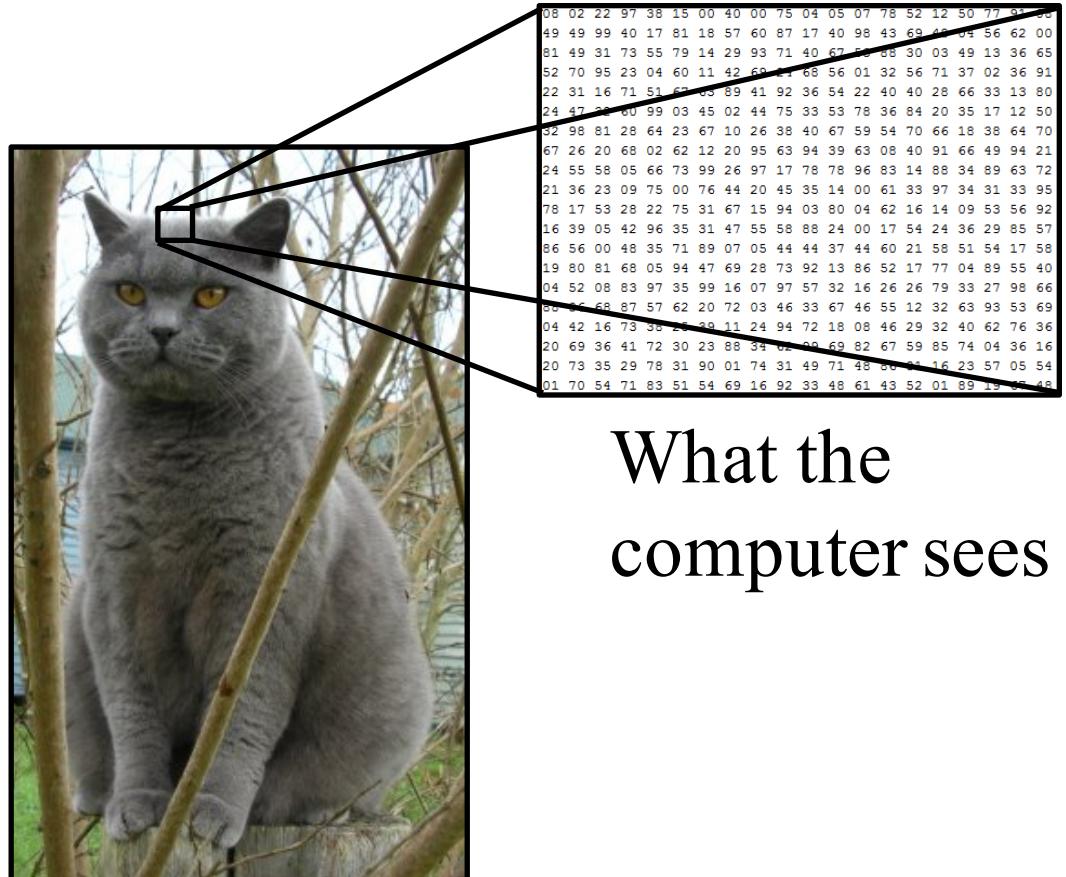


# Data Augmentation



# Data Augmentation

- Change the pixels without changing the label
- Train on transformed data
- VERY widely used



# Data Augmentation

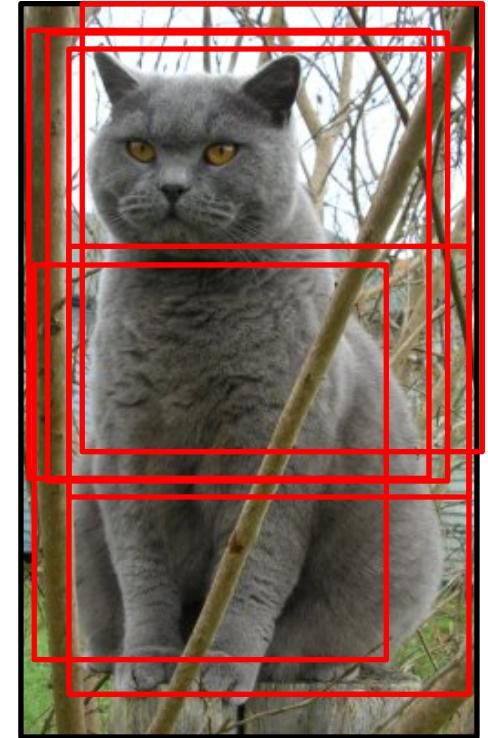
## 1. Horizontal flips



# Data Augmentation

## 2. Random crops/scales

**Training:** sample random crops / scales



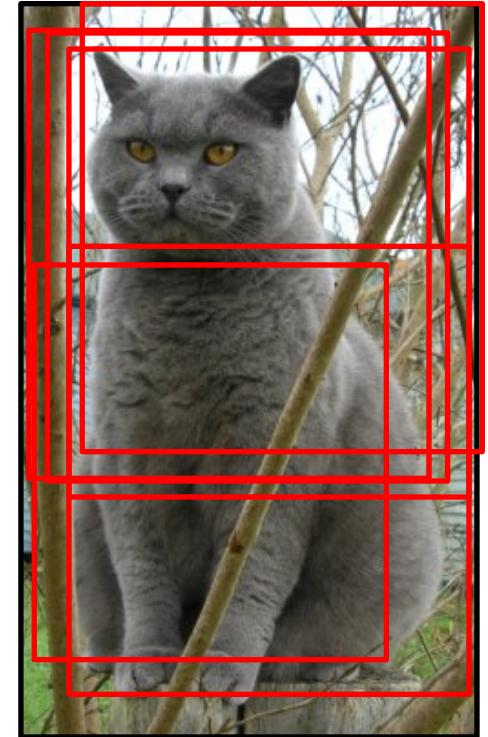
# Data Augmentation

## 2. Random crops/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  $224 \times 224$  patch



# Data Augmentation

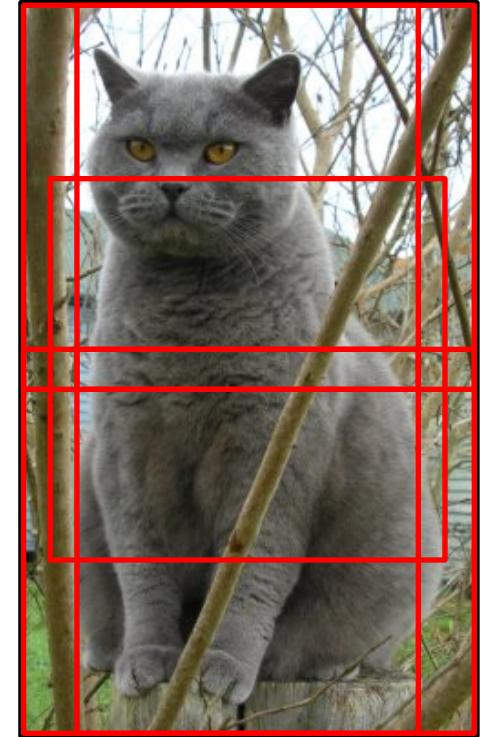
## 2. Random crops/scales

**Training:** sample random crops / scales

ResNet:

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

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



# Data Augmentation

## 3. Color jitter

Simple:

Randomly jitter contrast



# Data Augmentation

## 3. Color jitter

**Simple:**

Randomly jitter contrast



**Complex:**

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

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

# Data Augmentation

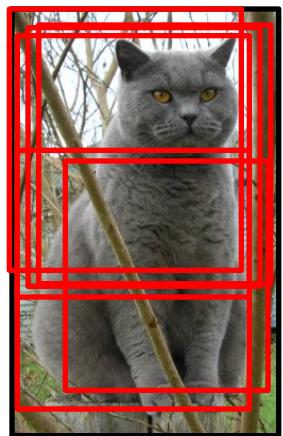
## 4. Get creative!

Random mix/combinations of :

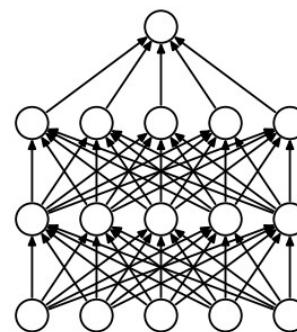
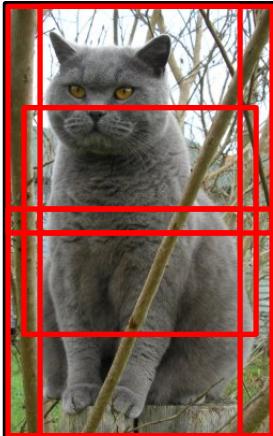
- translation (what about a pure ConvNet?)
- rotation
- stretching
- shearing,
- lens distortions, ... (go crazy)

# A general theme:

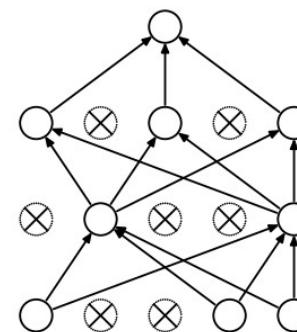
1. **Training:** Add random noise
2. **Testing:** Marginalize over the noise



Data Augmentation

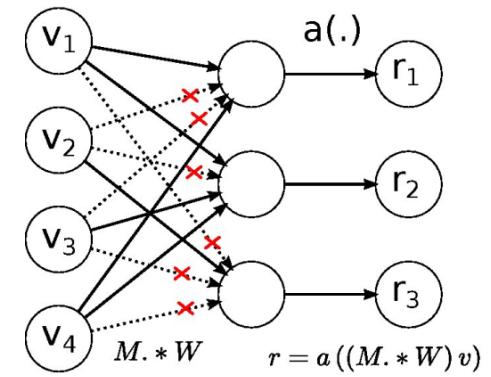


(a) Standard Neural Net



(b) After applying dropout.

Dropout



DropConnect

Batch normalization, Model ensembles

# Data Augmentation: Takeaway

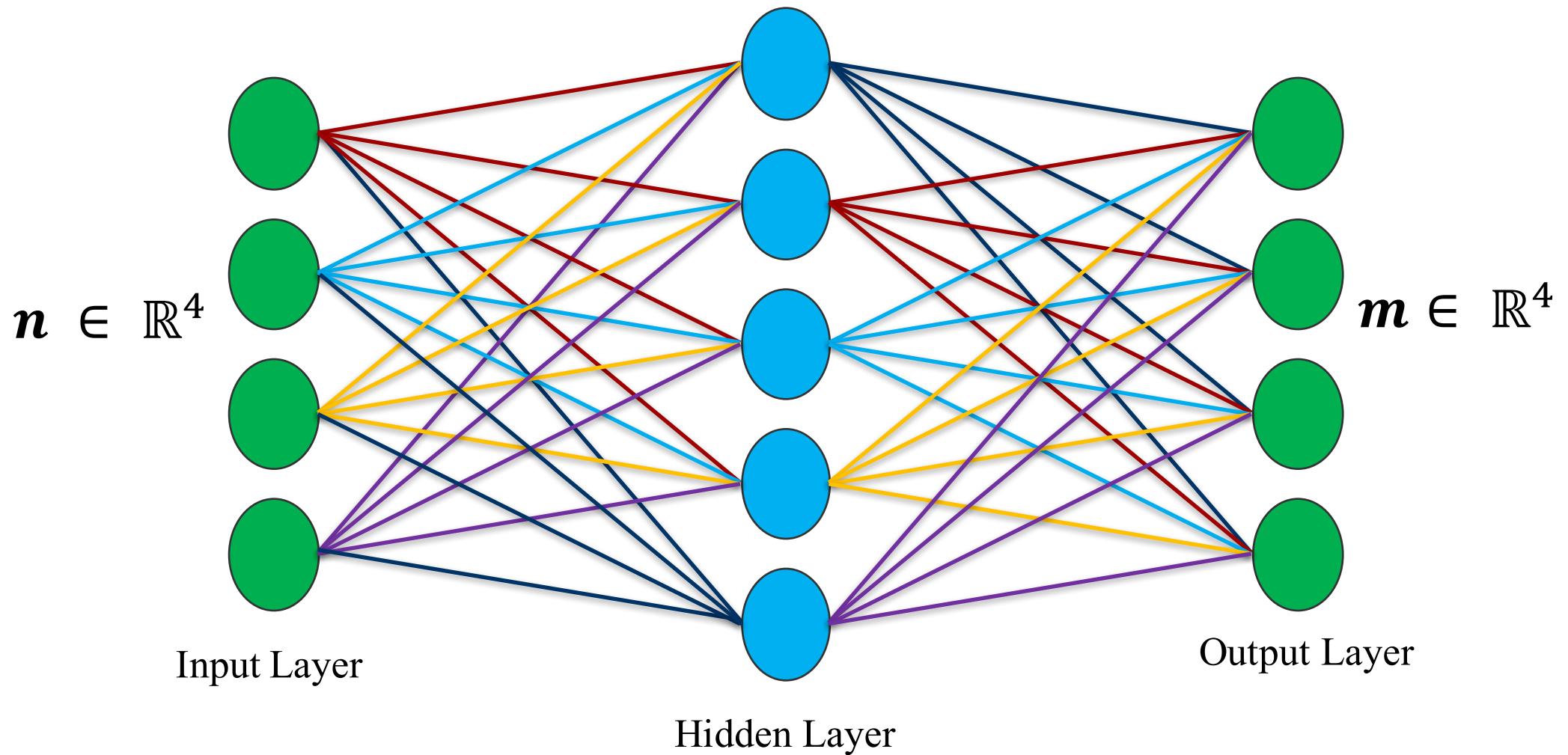
Simple to implement, use it

Especially useful for small datasets

Fits into framework of noise / marginalization

# Weight Initialization

# What happens when $W=0$ init is used?



- First idea: **Small random numbers**  
(normal distribution mean 0, sigma 1 scaled by -0.01 to 0.01)

```
self.W = torch.randn(fan_out, fan_in) * 0.01  
self.b = torch.randn(fan_out)* 0.01
```

- First idea: **Small random numbers**  
(normal distribution mean 0, sigma 1 scaled by -0.01 to 0.01)

```
self.W = torch.randn(fan_out, fan_in) * 0.01  
self.b = torch.randn(fan_out)* 0.01
```

Works ~okay for small networks (like our one layer cifar-10 classifier), but can lead to non-homogeneous distributions of activations across the layers of a network.

# Lets look at some activation statistics

E.g. 10-layer net with 1000 neurons on each layer, using tanh nonlinearities, and initializing as described in last slide.

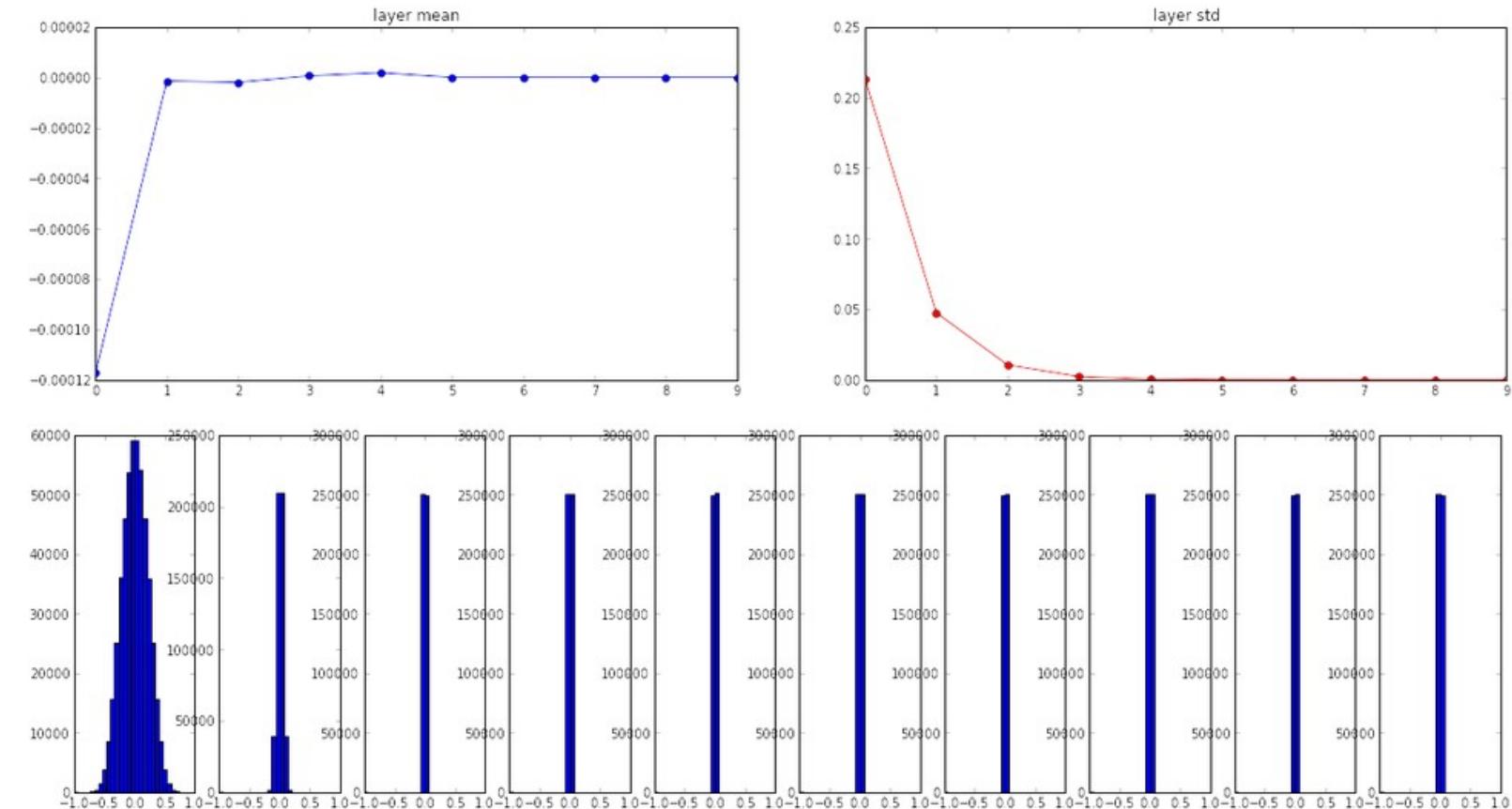
```
local Linear = torch.class("Linear")
function Linear:_init(fan_in, fan_out)
    self.output = nil
    self.W = torch.randn(fan_out, fan_in) * 0.01
    self.b = torch.randn(fan_out)* 0.01
end

-- Wx + b
function Linear:forward(xi)
    self.output = self.W * xi:reshape(x:size(1),1) + self.b
    return self.output
end

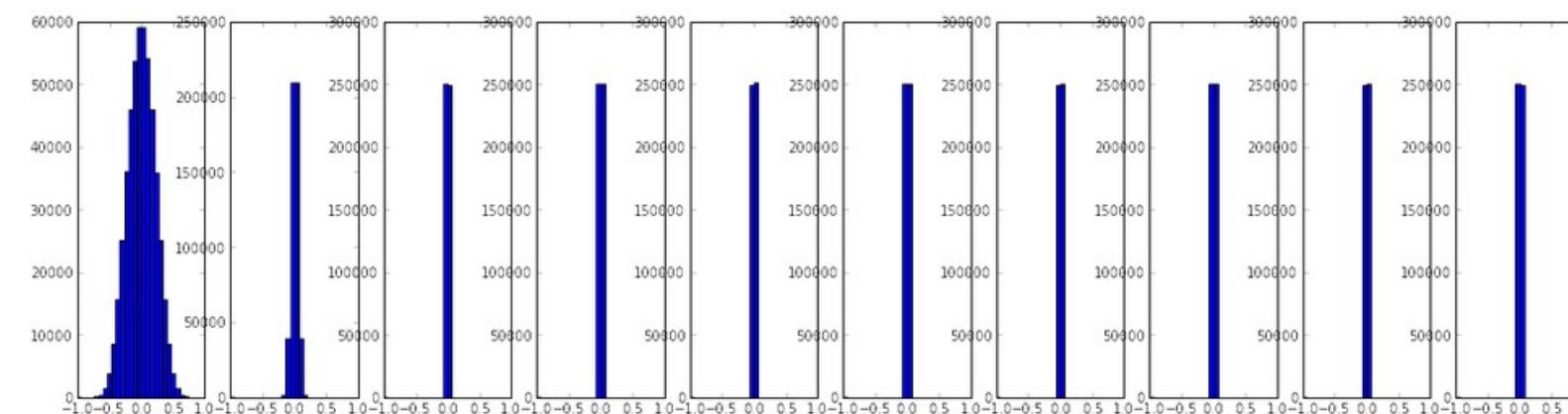
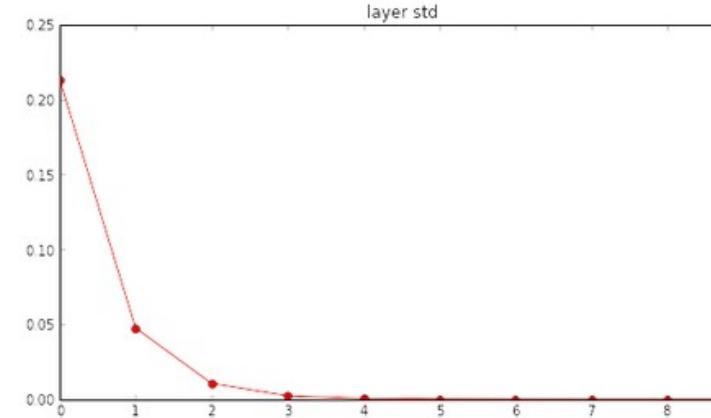
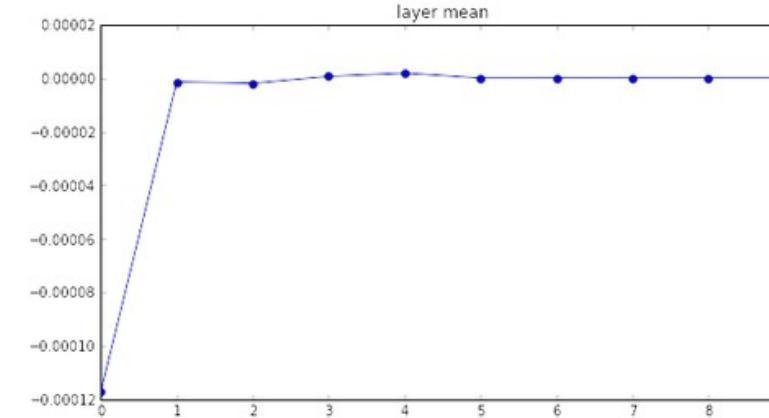
x = torch.randn(1000)
l = Linear.new(1000, 1000)
input = x
for i = 1, 10 do
    h = l:forward(input)
    h = nn.Tanh():forward(h)
    input = h
end
```

# Lets look at some activation statistics

```
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
```



```
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!

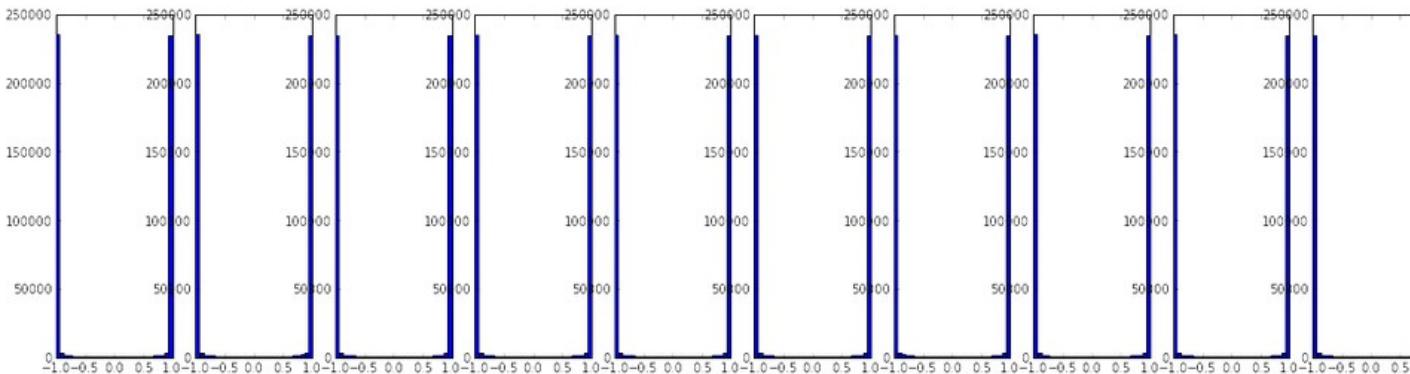
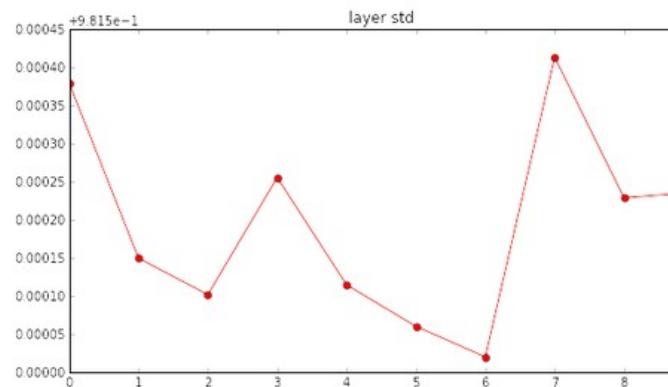
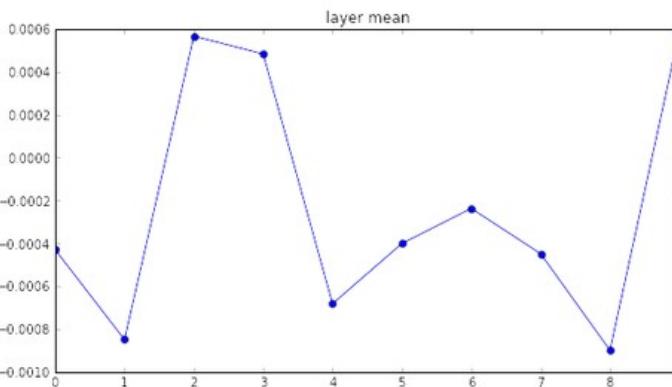
Q: think about the  
backward pass. What  
do the gradients look  
like?

Hint: think about backward  
update for  $\frac{\partial L}{\partial W}$

```
self.W = torch.randn(fan_out, fan_in) * 1.0  
self.b = torch.randn(fan_out)* 1.0
```

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



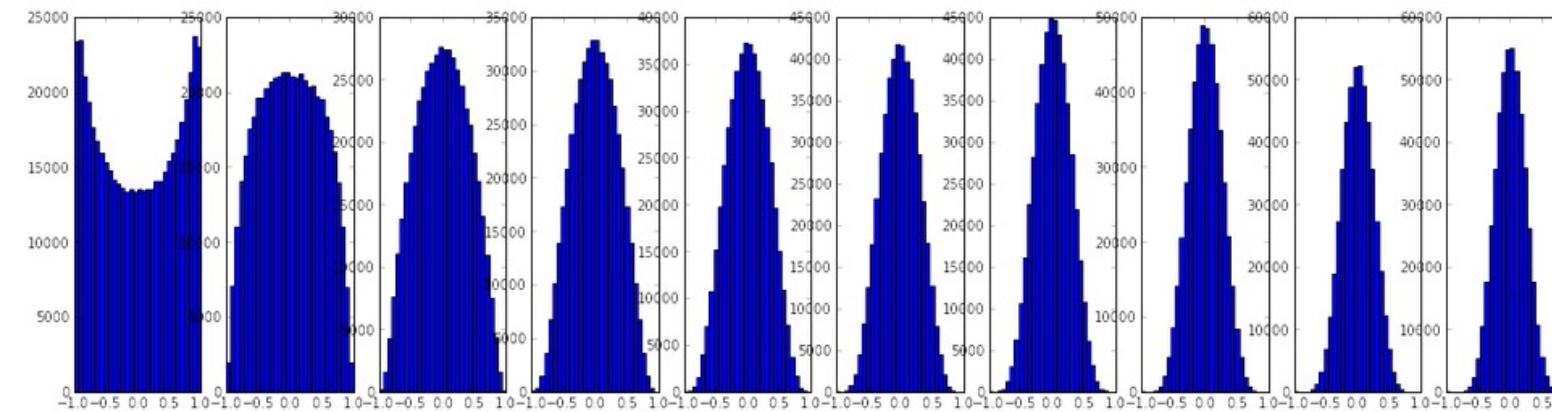
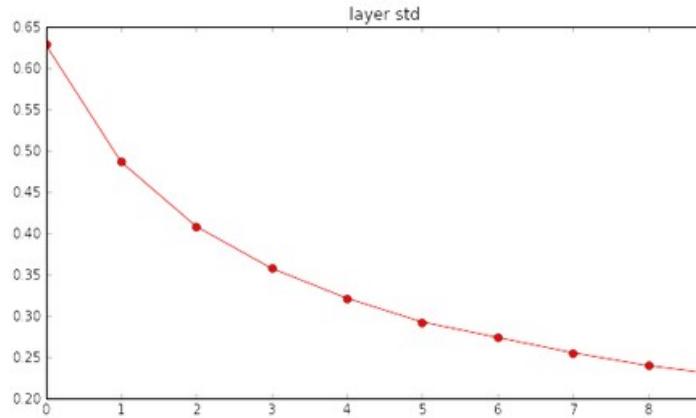
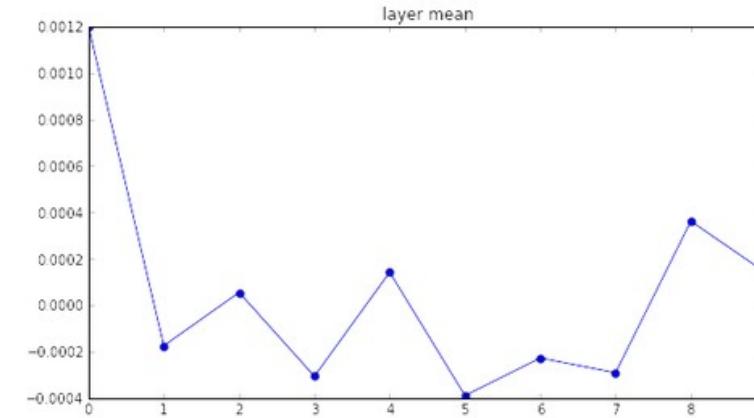
Almost all neurons completely saturated, either -1 and 1. Gradients will be all zero.

```
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
```

```
self.W = torch.randn(fan_out, fan_in) / math.sqrt(fan_in)  
self.b = torch.randn(fan_out) / math.sqrt(fan_in)
```

“Xavier initialization”  
[Glorot et al., 2010]

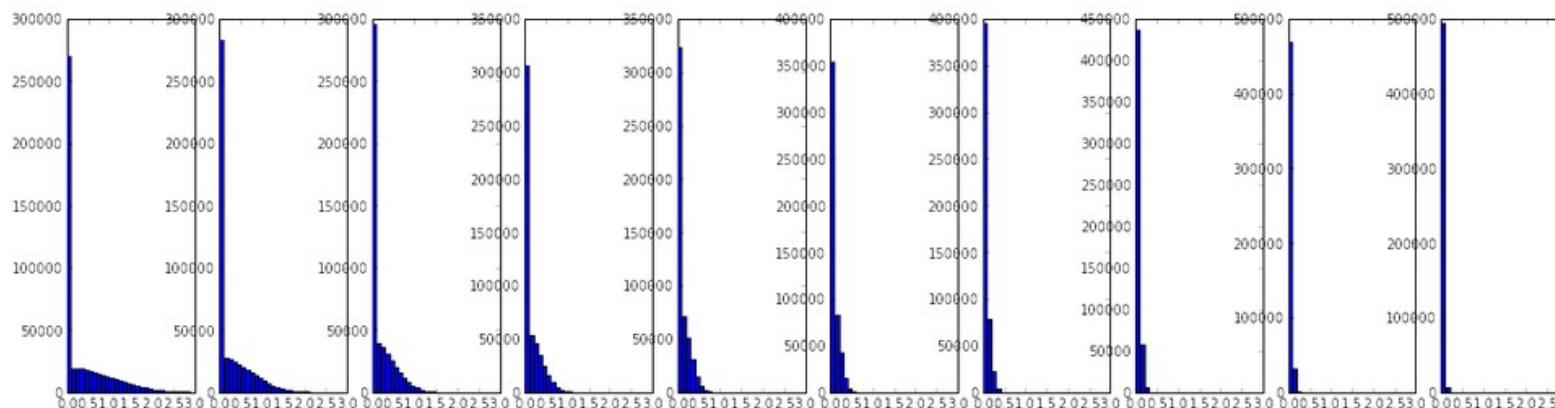
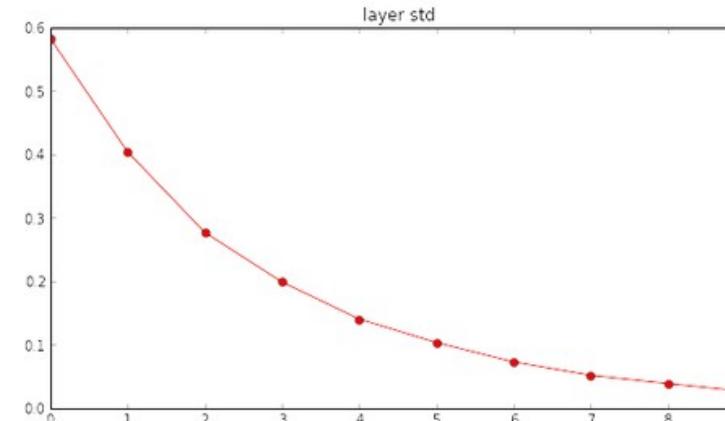
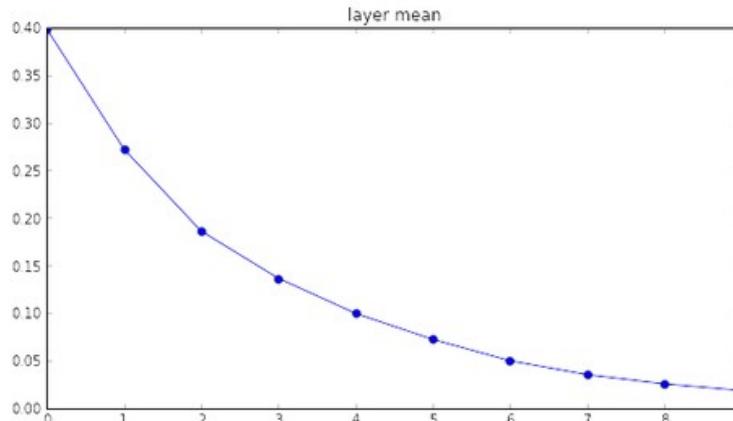
**Reasonable initialization.**  
(Mathematical derivation  
assumes linear activations)



```
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
```

```
self.W = torch.randn(fan_out, fan_in) / math.sqrt(fan_in)  
self.b = torch.randn(fan_out) / math.sqrt(fan_in)
```

but when using the ReLU nonlinearity it breaks.



```

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

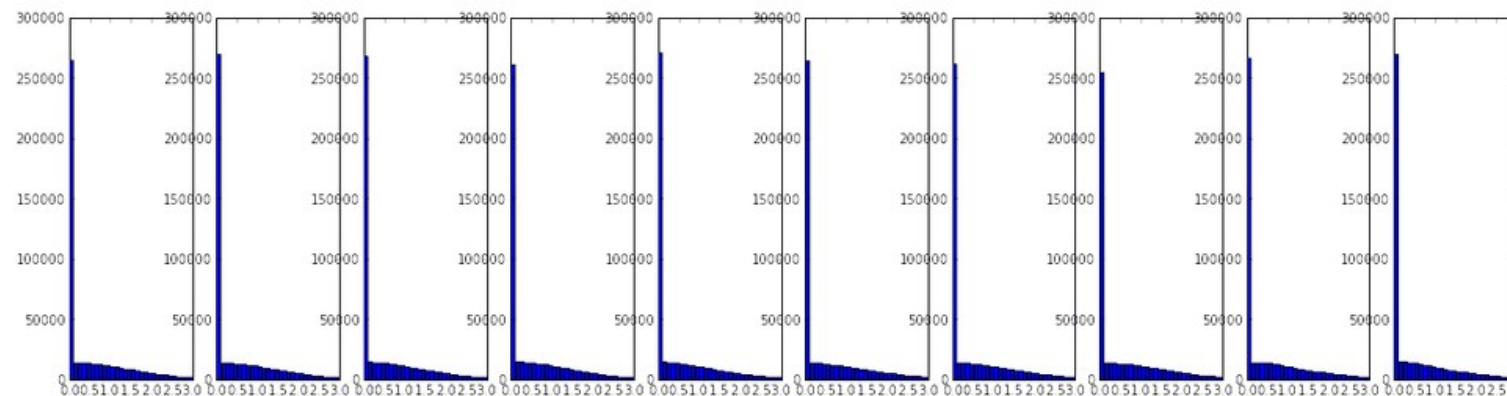
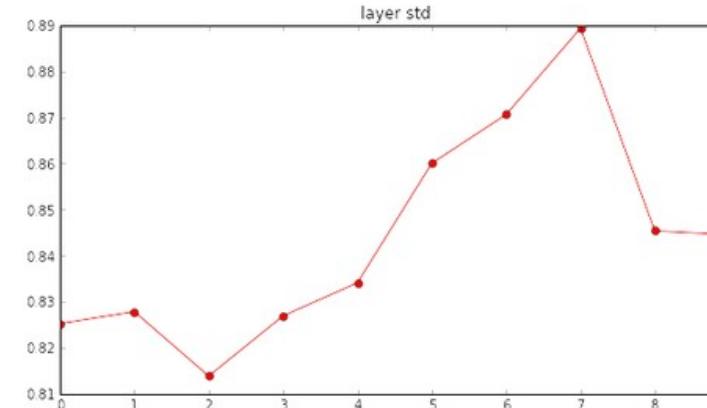
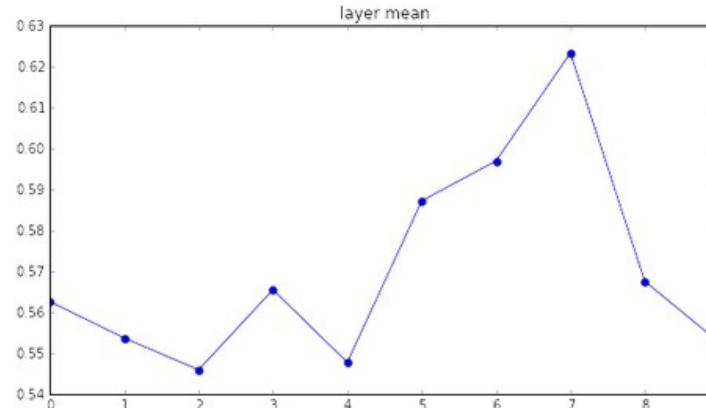
```

```

self.W = torch.randn(fan_out, fan_in) / math.sqrt(fan_in/2)
self.b = torch.randn(fan_out) / math.sqrt(fan_in/2)

```

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

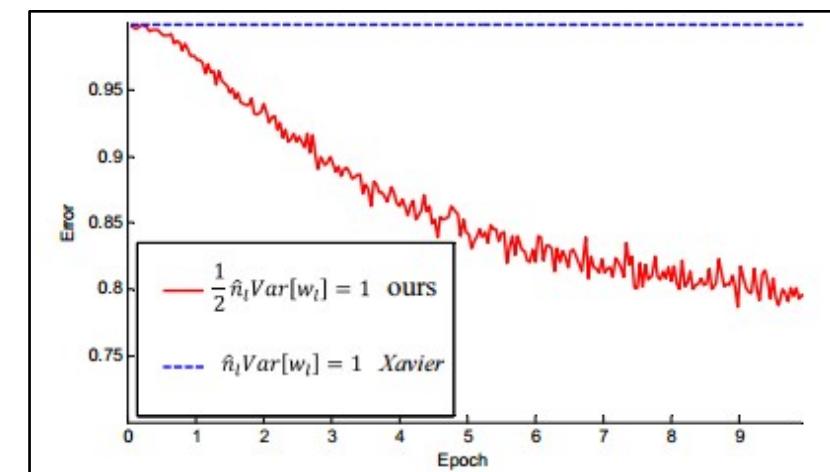
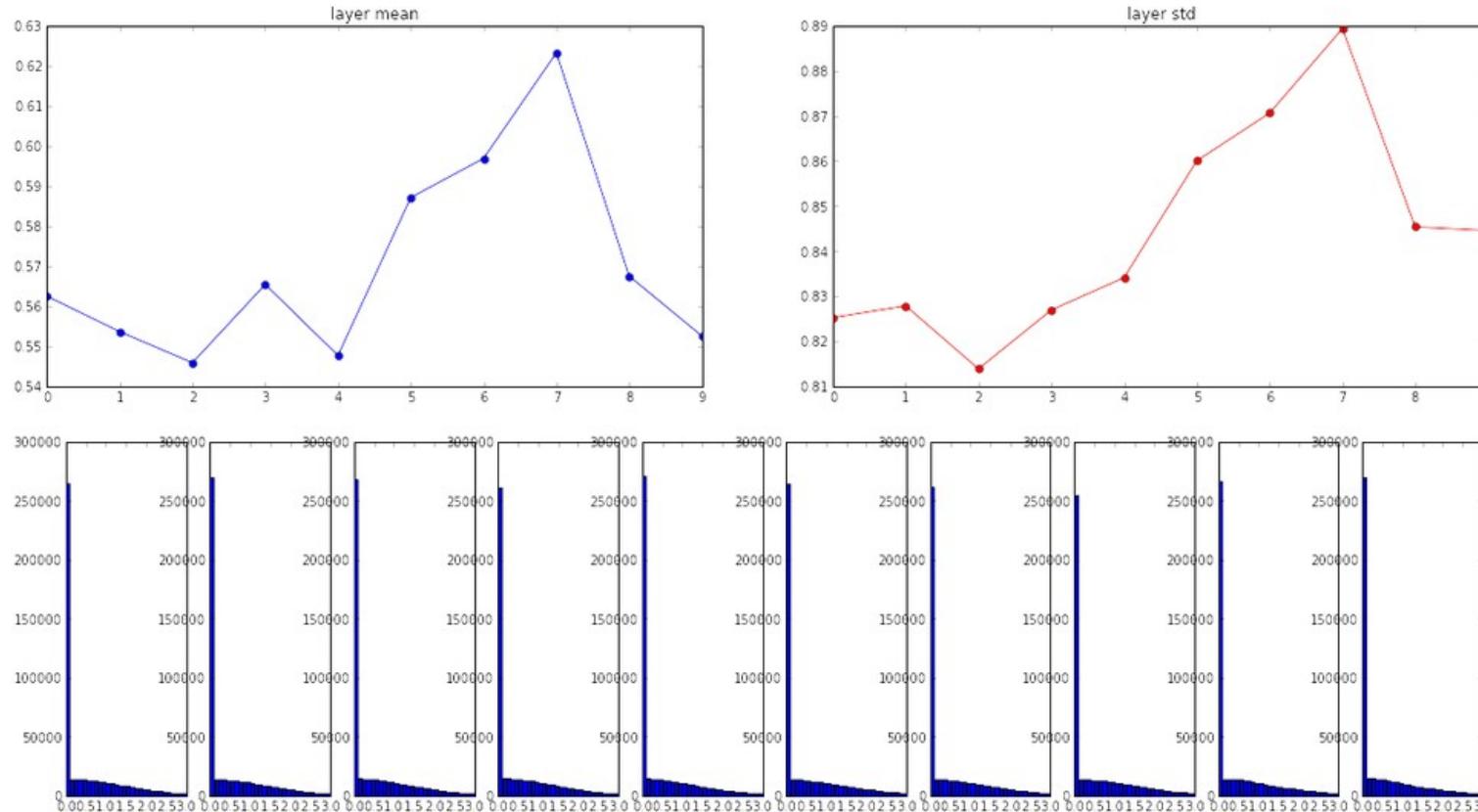


input layer had mean 0.000501 and std 0.999444  
 hidden layer 1 had mean 0.562488 and std 0.825232  
 hidden layer 2 had mean 0.553614 and std 0.827835  
 hidden layer 3 had mean 0.545867 and std 0.813855  
 hidden layer 4 had mean 0.565396 and std 0.826902  
 hidden layer 5 had mean 0.547678 and std 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

```

self.W = torch.randn(fan_out, fan_in) / math.sqrt(fan_in/2)
self.b = torch.randn(fan_out) / math.sqrt(fan_in/2)
  
```

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



```
9  -- "Efficient backprop"
10 -- Yann Lecun, 1998
11 local function w_init_heuristic(fan_in, fan_out)
12     return math.sqrt(1/(3*fan_in))
13 end
14
15
16 -- "Understanding the difficulty of training deep feedforward neural networks"
17 -- Xavier Glorot, 2010
18 local function w_init_xavier(fan_in, fan_out)
19     return math.sqrt(2/(fan_in + fan_out))
20 end
21
22
23 -- "Understanding the difficulty of training deep feedforward neural networks"
24 -- Xavier Glorot, 2010
25 local function w_init_xavier_caffe(fan_in, fan_out)
26     return math.sqrt(1/fan_in)
27 end
28
29
30 -- "Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification"
31 -- Kaiming He, 2015
32 local function w_init_kaiming(fan_in, fan_out)
33     return math.sqrt(4/(fan_in + fan_out))
34 end
```

<https://github.com/e-lab/torch-toolbox/blob/master/Weight-init/weight-init.lua>

# 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

*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*

by Mishkin and Matas, 2015

*Centered Weight Normalization in Accelerating Training of Deep Neural Networks*

by Huang et al., 2017

...

# Batch Normalization

“you want unit gaussian activations? just make them so.”

[Ioffe and Szegedy, 2015]

# Batch Normalization

[Ioffe and Szegedy, 2015]

“you want unit gaussian activations? just make them so.”

consider a batch of activations at some layer. To make each feature dimension unit gaussian, apply:

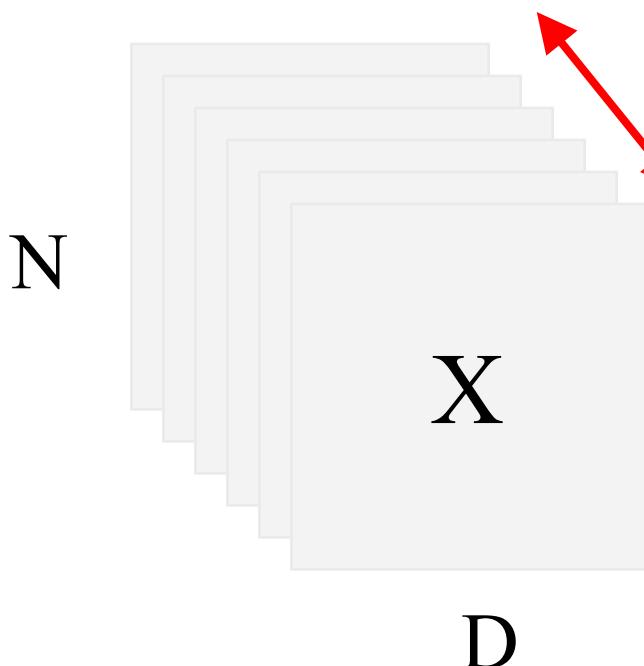
$$\hat{x}^{(k)} = \frac{x^{(k)} - \text{E}[x^{(k)}]}{\sqrt{\text{Var}[x^{(k)}]}}$$

this is a vanilla  
differentiable function...

# Batch Normalization

[Ioffe and Szegedy, 2015]

“you want unit gaussian activations?  
just make them so.”



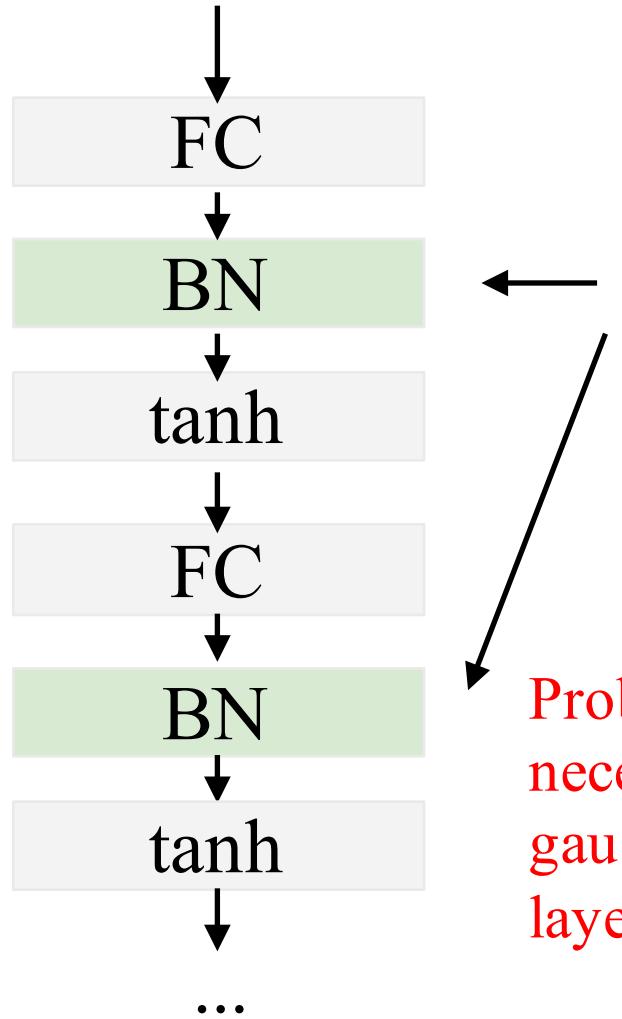
1. compute the empirical mean and variance independently for each feature dimension.

2. Normalize

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

# Batch Normalization

[Ioffe and Szegedy, 2015]



Usually inserted after Fully Connected / (or Convolutional, as we'll see soon) layers, and before nonlinearity.

Problem: do we necessarily want a unit gaussian input to a tanh layer?

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

# Batch Normalization

[Ioffe and Szegedy, 2015]

Normalize:

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

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

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

Note, the network can learn:

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

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

to recover the identity mapping.

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

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

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

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

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

- 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

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

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

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

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

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

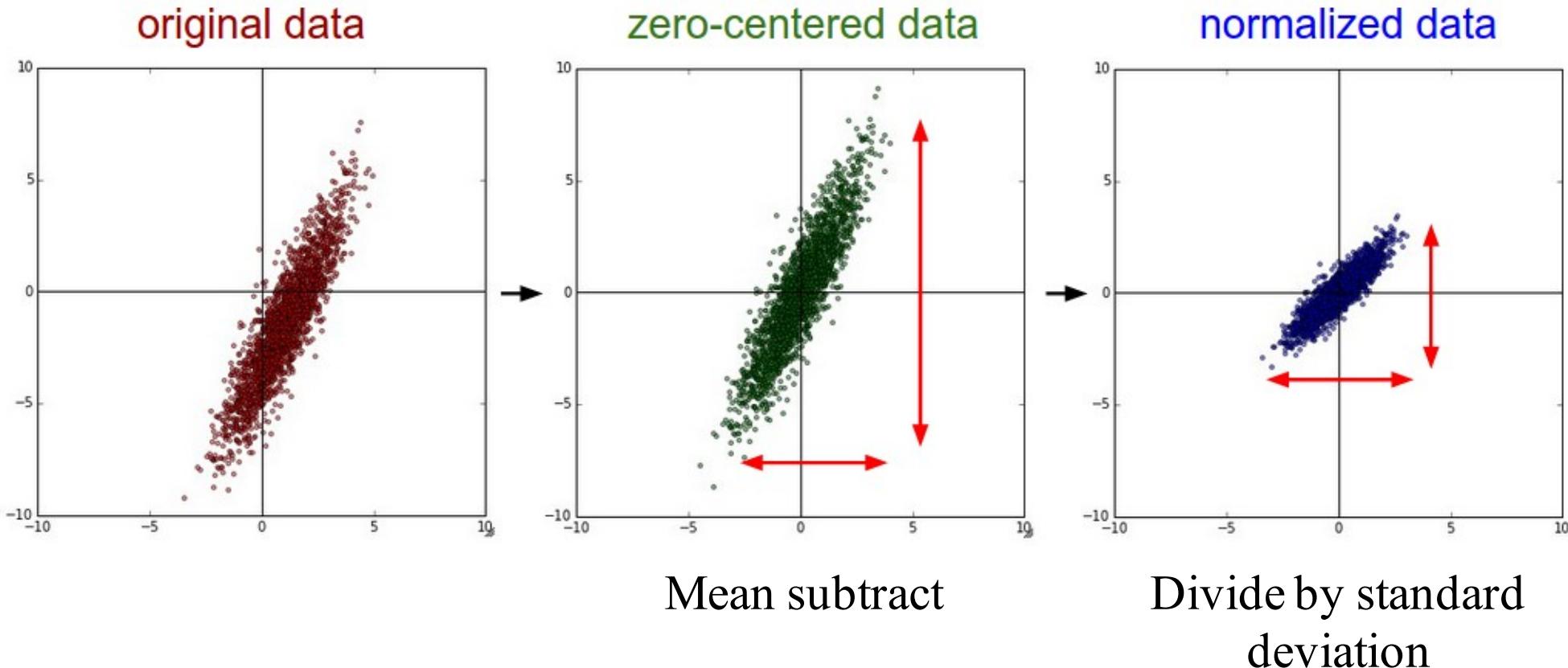
**Note: at test time BatchNorm layer functions differently:**

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)

# Babysitting the Learning Process

# Step 1: Preprocess the data



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

# Notebook

[https://github.com/stencilman/CS763\\_Spring2017/blob/master/Lec3%2C4/CrossEntropy-Linear.ipynb](https://github.com/stencilman/CS763_Spring2017/blob/master/Lec3%2C4/CrossEntropy-Linear.ipynb)

**2. Data Preprocessing: We compute the mean and standard deviation 'images' and then subtract and divide by the same respectively (like AlexNet). We also visualize them.**

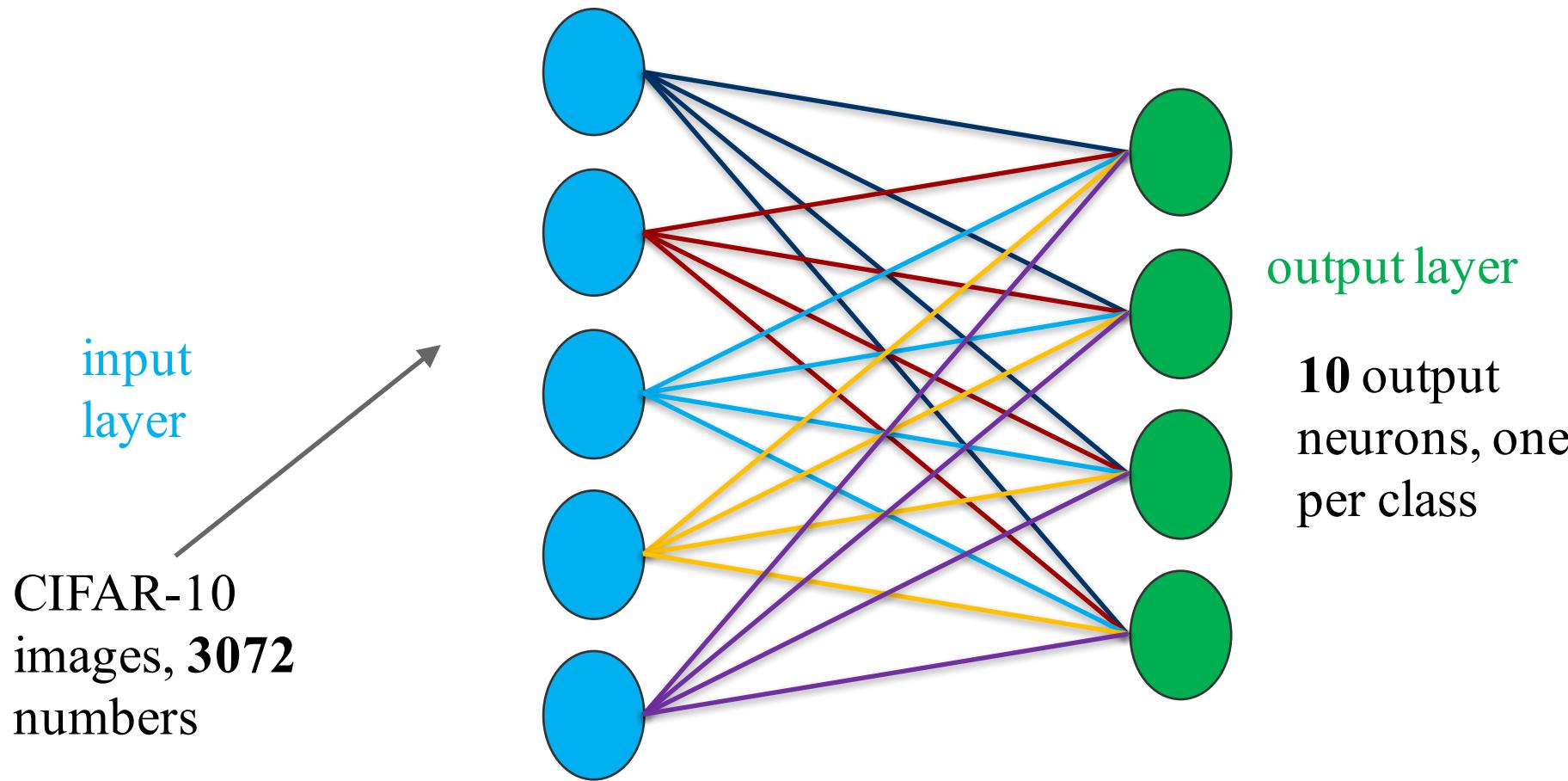
```
In [3]: x_mean = torch.mean(tr_x:float(), 1)
x_std = torch.std(tr_x:float(), 1)
itorch.image(x_mean)
itorch.image(x_std)
```



```
In [7]: function get_xi(data_x, idx)
    xi = (data_x[idx]:float() - x_mean)
    xi = xi:cdiv(x_std)
    xi = xi:reshape(3*32*32)
    return xi
end
```

# Step 2: Choose the architecture:

Say we start with **single** layer network:

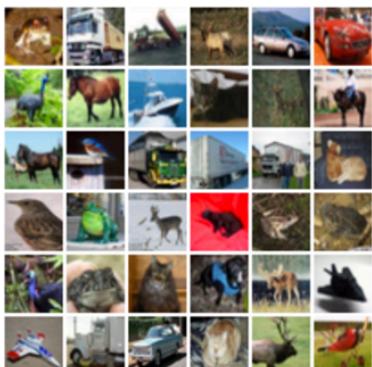


**1. Data Loading:** Let us load the training and the test data and check the size of the tensors. Let us also display the first few images from the training set.

```
In [1]: -- load trainin images  
tr_x = torch.load('cifar10/tr_data.bin')  
-- load trainin labels  
tr_y = torch.load('cifar10/tr_labels.bin'):double() + 1  
-- load test images  
te_x = torch.load('cifar10/te_data.bin')  
-- load test labels  
te_y = torch.load('cifar10/te_labels.bin'):double() + 1  
print(tr_x:size())  
print(tr_y:size())
```

```
Out[1]: 50000  
3  
32  
32  
[torch.LongStorage of size 4]  
  
50000  
[torch.LongStorage of size 1]
```

```
In [2]: -- display the first 36 training set images  
require 'image';  
itorch.image(tr_x[{{1,36}, {}, {}, {}}])
```



```
function train_and_test_loop(no_iterations, lr, lambda)
    for i = 0, no_iterations do

        -- trainin input and target
        xi = get_xi(tr_x, i)
        ti = tr_y[i]

        -- Train
        op = model:forward(xi)
        loss_tr = criterion:forward(op, ti)
        dl_do = criterion:backward(op, ti)
        model:backward(xi, dl_do)

        -- Test
        idx = shuffle_te[mod(i, te_x:size(1)) + 1]
        xi = get_xi(te_x, idx)
        ti = te_y[idx]
        -- Compute loss
        op = model:forward(xi)
        loss_te = criterion:forward(op, ti, model, lambda)

        -- udapte model weights
        gradient_descent(model, lr)

        err = evaluate(model, tr_x, tr_y)
        if (err < besterr) then
            besterr = err
            bestmodel:copy(model)
        end

    end
    return (1 - besterr)*100 -- Accuracy
end
```

# Double check that the loss is reasonable:

```
op = model:forward(xi)
loss_tr = criterion:forward(op, ti)
print(loss_tr)
```

```
-- run it
lr = 0.00001    disable regularization
lambda = 0.0
train_and_test_loop(1, lr, lambda)
```

Out[11]: 2.2656910718829

loss ~2.3.  
“correct” for  
10 classes

Print Loss

# Double check that the loss is reasonable:

```
op = model:forward(xi)
loss_tr = criterion:forward(op, ti,
print(loss_tr)
```

```
-- run it
lr = 0.00001
lambda = 1e3
```

Crank it way up regularization

```
train_and_test_loop(1, lr, lambda)
```

Out[12]: 12.582525307612



Print Loss

loss went up, good. (sanity check)

Lets try to train now...

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

```
tr_x = tr_x[{{1,20}, {}, {}, {}}]
te_x = tr_x[{{1,20}, {}, {}, {}}]
tr_y = tr_y[{{1,20}}]
print(tr_x:size())
print(tr_y:size())
```

```
Out[14]: 20
          3
          32
          32
[torch.LongStorage of size 4]

          20
[torch.LongStorage of size 1]
```

```
-- run it
lr = 0.0001
lambda = 0
train_and_test_loop(100000, lr, lambda)
```

The above code:

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

Lets try to train now...

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

Very small loss,  
train accuracy 100,  
nice!

```
-- run it
lr = 0.0001
lambda = 0
train_and_test_loop(100000, lr, lambda)
```

```
Out[54]: iter: 0, accuracy: 100% Loss: 0.023342480719671
-- best accuracy achieved: 20%
Out[54]: iter: 500, accuracy: 20% Loss: 6.4891701533306
-- best accuracy achieved: 100%
Out[54]: iter: 1000, accuracy: 100% Loss: 3.363490690347
Out[54]: iter: 1500, accuracy: 100% Loss: 2.3995975677242
Out[54]: iter: 2000, accuracy: 100% Loss: 1.8909617506362
Out[54]: iter: 2500, accuracy: 100% Loss: 1.5617572159784
Out[54]: iter: 3000, accuracy: 100% Loss: 1.3375534142717
Out[54]: iter: 3500, accuracy: 100% Loss: 1.1668484200641
Out[54]: iter: 4000, accuracy: 100% Loss: 1.0398030826978
Out[54]: iter: 98000, accuracy: 100% Loss: 0.075056174474324
Out[54]: iter: 98500, accuracy: 100% Loss: 0.074695131101785
Out[54]: iter: 99000, accuracy: 100% Loss: 0.074675841566382
Out[54]: iter: 99500, accuracy: 100% Loss: 0.074908872365756
Out[54]: iter: 100000, accuracy: 100% Loss: 0.074439254969025
```

Lets try to train now...

```
-- run it
lr = 1e-7
lambda = 1e-7
train_and_test_loop(10000, lr, lambda)
```

I like to start with small regularization and find learning rate that makes the loss go down.

Lets try to train now...

I like to start with small regularization and find learning rate that makes the loss go down.

```
-- run it
lr = 1e-7
lambda = 1e-7
train_and_test_loop(10000, lr, lambda)
```

```
Out[18]: iter: 0, accuracy: 10% Loss: 0.023248429529449
          -- best accuracy achieved: 10%
Out[18]: iter: 500, accuracy: 10% Loss: 11.522416713458
Out[18]: iter: 1000, accuracy: 10% Loss: 11.517536122735
Out[18]: iter: 1500, accuracy: 11% Loss: 11.508510566527
          -- best accuracy achieved 11%
Out[18]: iter: 2000, accuracy: 13% Loss: 11.510842908524
          -- best accuracy achieved 13%
Out[18]: iter: 2500, accuracy: 13% Loss: 11.501224886344
Out[18]: iter: 3000, accuracy: 14% Loss: 11.49398984774
          -- best accuracy achieved 14%
Out[18]: iter: 3500, accuracy: 16% Loss: 11.487628759524
          -- best accuracy achieved 16%
Out[18]: iter: 4000, accuracy: 17% Loss: 11.492140238992
          -- best accuracy achieved: 17%
```

Loss barely changing: Learning rate is probably too low

Lets try to train now...

I like to start with small regularization and find learning rate that makes the loss go down.

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

```
-- run it
lr = 1e-7
lambda = 1e-7
train_and_test_loop(10000, lr, lambda)
```

```
Out[18]: iter: 0, accuracy: 10% Loss: 0.023248429529449
-- best accuracy achieved: 10%
Out[18]: iter: 500, accuracy: 10% Loss: 11.522416713458
Out[18]: iter: 1000, accuracy: 10% Loss: 11.517536122735
Out[18]: iter: 1500, accuracy: 11% Loss: 11.508510566527
-- best accuracy achieved 11%
Out[18]: iter: 2000, accuracy: 13% Loss: 11.510842908524
-- best accuracy achieved 13%
Out[18]: iter: 2500, accuracy: 13% Loss: 11.501224886344
Out[18]: iter: 3000, accuracy: 14% Loss: 11.49398984774
-- best accuracy achieved 14%
Out[18]: iter: 3500, accuracy: 16% Loss: 11.487628759524
-- best accuracy achieved 16%
Out[18]: iter: 4000, accuracy: 17% Loss: 11.492140238992
-- best accuracy achieved: 17%
```

Loss barely changing: Learning rate is probably too low

Lets try to train now...

I like to start with small regularization and find learning rate that makes the loss go down.

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

Notice train/val accuracy goes to 17%  
though, what's up with that? (remember this  
is softmax)

```
-- run it
lr = 1e-7
lambda = 1e-7
train_and_test_loop(10000, lr, lambda)

Out[18]: iter: 0, accuracy: 10% Loss: 0.023248429529449
          -- best accuracy achieved: 10%
Out[18]: iter: 500, accuracy: 10% Loss: 11.522416713458
Out[18]: iter: 1000, accuracy: 10% Loss: 11.517536122735
Out[18]: iter: 1500, accuracy: 11% Loss: 11.508510566527
          -- best accuracy achieved 11%
Out[18]: iter: 2000, accuracy: 13% Loss: 11.510842908524
          -- best accuracy achieved 13%
Out[18]: iter: 2500, accuracy: 13% Loss: 11.501224886344
Out[18]: iter: 3000, accuracy: 14% Loss: 11.49398984774
          -- best accuracy achieved 14%
Out[18]: iter: 3500, accuracy: 16% Loss: 11.487628759524
          -- best accuracy achieved 16%
Out[18]: iter: 4000, accuracy: 17% Loss: 11.492140238992
          -- best accuracy achieved: 17%
```

Loss barely changing: Learning rate is  
probably too low

Lets try to train now...

I like to start with small regularization and find learning rate that makes the loss go down.

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

```
-- run it
lr = 1e6
lambda = 1e-7
train_and_test_loop(10000, lr, lambda)
```

Okay now lets try learning rate 1e6. What could possibly go wrong?

Lets try to train now...

I like to 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

```
-- run it
lr = 1e6
lambda = 1e-7
train_and_test_loop(10000, lr, lambda)

Out[19]: iter: 0, accuracy: 11% Loss: 0.023115084740835
          -- best accuracy achieved: 11%

Out[19]: iter: 500, accuracy: 13% Loss: nan
          -- best accuracy achieved: 13%

Out[19]: iter: 1000, accuracy: 13% Loss: nan

Out[19]: iter: 1500, accuracy: 13% Loss: nan

Out[19]: iter: 2000, accuracy: 13% Loss: nan
```

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

Lets try to train now...

I like to 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

```
-- run it
lr = 1e-3
lambda = 1e-7
train_and_test_loop(3000, lr, lambda)

Out[29]: iter: 0, accuracy: 20% Loss: 0.02357119788693
-- best accuracy achieved: 20%

Out[29]: iter: 500, accuracy: 13% Loss: nan

Out[29]: iter: 1000, accuracy: 13% Loss: nan

Out[29]: iter: 1500, accuracy: 13% Loss: nan

Out[29]: iter: 2000, accuracy: 13% Loss: nan

Out[29]: iter: 2500, accuracy: 13% Loss: nan

Out[29]: iter: 3000, accuracy: 13% Loss: nan
```

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

=> Rough range for learning rate we should be cross-validating is somewhere [1e-3 ... 1e-7]

# Hyperparameter Optimization

# Cross-validation strategy

I like to do **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  $> 3 * \text{original cost}$ , break out early

# For example: run coarse search for 2000 iterations

```
for i = 1, 100 do
    init_model()
    lr = math.pow(10, torch.uniform(-7.0, -3.0))
    lambda = math.pow(10, torch.uniform(-5, 5))
    best_acc = train_and_test_loop(2000, lr, lambda)
    print(string.format("Try %d/%d Best val accuracy: %d, lr: %f, lambda: %f", i, 100, best_acc, lr, lambda))
end
```

note it's best to optimize in log space!

Out[10]: Try 1/100 Best val accuracy: 16, lr: 0.000045, lambda: 4996.489302

Out[10]: Try 2/100 Best val accuracy: 31, lr: 0.000003, lambda: 0.001315

Out[10]: Try 3/100 Best val accuracy: 25, lr: 0.000001, lambda: 0.000012

Out[10]: Try 4/100 Best val accuracy: 24, lr: 0.000002, lambda: 216.397129

Out[10]: Try 5/100 Best val accuracy: 26, lr: 0.000007, lambda: 0.000012

Out[10]: Try 6/100 Best val accuracy: 29, lr: 0.000009, lambda: 275.964597

Out[10]: Try 7/100 Best val accuracy: 30, lr: 0.000021, lambda: 0.000253

Out[10]: Try 8/100 Best val accuracy: 13, lr: 0.000809, lambda: 4.339235

Out[10]: Try 9/100 Best val accuracy: 26, lr: 0.000003, lambda: 0.000062

Out[10]: Try 10/100 Best val accuracy: 27, lr: 0.000095, lambda: 18.288190

Out[10]: Try 11/100 Best val accuracy: 14, lr: 0.000000, lambda: 1333.400659

Out[10]: Try 12/100 Best val accuracy: 8, lr: 0.000311, lambda: 0.000020

Out[10]: Try 13/100 Best val accuracy: 8, lr: 0.000617, lambda: 0.000050

Out[10]: Try 14/100 Best val accuracy: 34, lr: 0.000013, lambda: 0.124955

Out[10]: Try 15/100 Best val accuracy: 17, lr: 0.000013, lambda: 5262.631955

nice

# Now run finer search...

```
for i = 1, 100 do
    init_model()
    lr = math.pow(10, torch.uniform(-7.0, -3.0))
    lambda = math.pow(10, torch.uniform(-5, 5))
    best_acc = train_and_test_loop(2000, lr, lambda)
    print(string.format("Try %d/%d Best val accuracy: %d, lr: %f, lambda: %f", i, 100, best_acc, lr, lambda))
end
```

adjust range →

```
for i = 1, 100 do
    init_model()
    lr = math.pow(10, torch.uniform(-6.0, -4.0))
    lambda = math.pow(10, torch.uniform(-3, 1))
    best_acc = train_and_test_loop(2000, lr, lambda)
    print(string.format("Try %d/%d Best val accuracy: %d, lr: %f, lambda: %f", i, 100, best_acc, lr, lambda))
end
```

```
Out[11]: Try 1/100 Best val accuracy: 35, lr: 0.000055, lambda: 0.002026
Out[11]: Try 2/100 Best val accuracy: 28, lr: 0.000001, lambda: 1.994656
Out[11]: Try 3/100 Best val accuracy: 32, lr: 0.000003, lambda: 0.483409
Out[11]: Try 4/100 Best val accuracy: 37, lr: 0.000032, lambda: 1.981563
Out[11]: Try 5/100 Best val accuracy: 27, lr: 0.000003, lambda: 0.004578
Out[11]: Try 6/100 Best val accuracy: 28, lr: 0.000004, lambda: 0.082862
Out[11]: Try 7/100 Best val accuracy: 34, lr: 0.000020, lambda: 0.003083
Out[11]: Try 8/100 Best val accuracy: 28, lr: 0.000054, lambda: 0.064499
Out[11]: Try 9/100 Best val accuracy: 31, lr: 0.000003, lambda: 0.004361
Out[11]: Try 10/100 Best val accuracy: 32, lr: 0.000004, lambda: 0.001610
Out[11]: Try 11/100 Best val accuracy: 31, lr: 0.000006, lambda: 0.300821
```

37% - relatively good  
for a 1-layer neural net  
and only 2000  
iterations

# Now run finer search...

```
for i = 1, 100 do
    init_model()
    lr = math.pow(10, torch.uniform(-7.0, -3.0))
    lambda = math.pow(10, torch.uniform(-5, 5))
    best_acc = train_and_test_loop(2000, lr, lambda)
    print(string.format("Try %d/%d Best val accuracy: %d, lr: %f", i, 100, best_acc, lr))
end
```

adjust range →

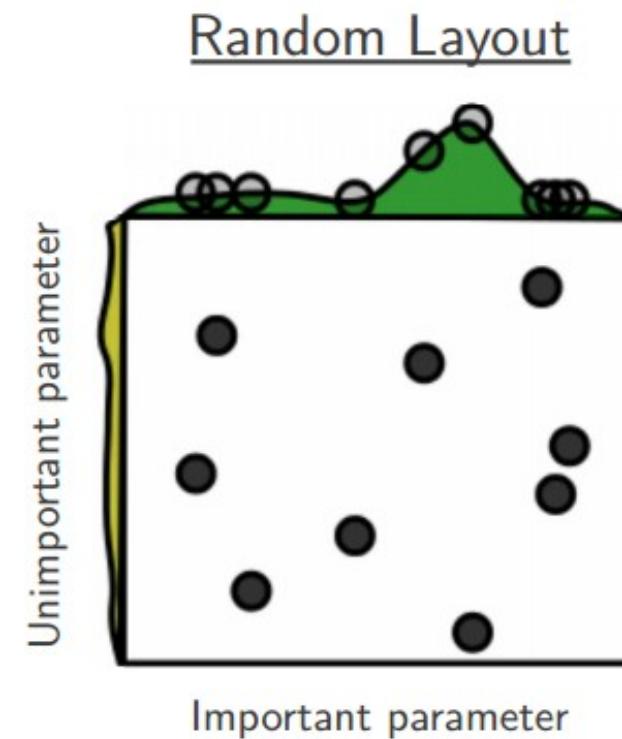
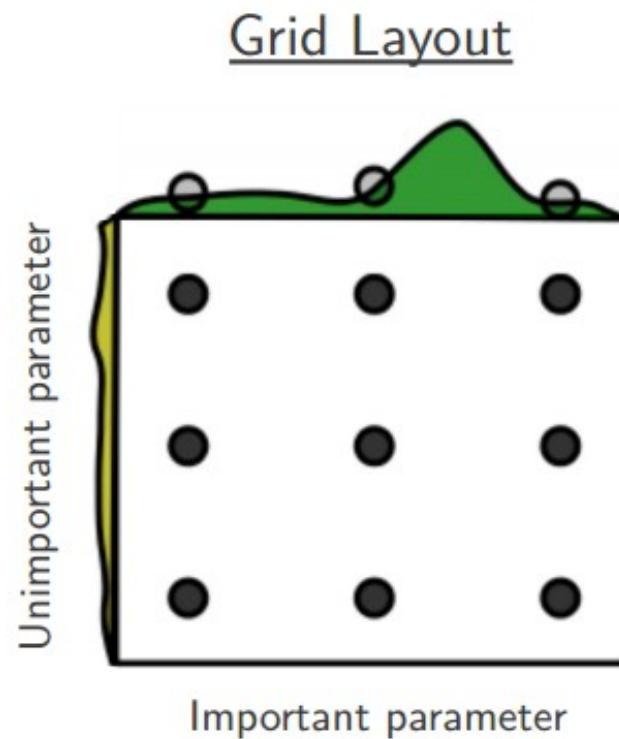
```
for i = 1, 100 do
    init_model()
    lr = math.pow(10, torch.uniform(-6.0, -4.0))
    lambda = math.pow(10, torch.uniform(-3, 1))
    best_acc = train_and_test_loop(2000, lr, lambda)
    print(string.format("Try %d/%d Best val accuracy: %d, lr: %f", i, 100, best_acc, lr))
end
```

```
Out[11]: Try 1/100 Best val accuracy: 35, lr: 0.000055, lambda: 0.002026
Out[11]: Try 2/100 Best val accuracy: 28, lr: 0.000001, lambda: 1.994656
Out[11]: Try 3/100 Best val accuracy: 32, lr: 0.000003, lambda: 0.483409
Out[11]: Try 4/100 Best val accuracy: 37, lr: 0.000032, lambda: 1.981563
Out[11]: Try 5/100 Best val accuracy: 27, lr: 0.000003, lambda: 0.004578
Out[11]: Try 6/100 Best val accuracy: 28, lr: 0.000004, lambda: 0.082862
Out[11]: Try 7/100 Best val accuracy: 34, lr: 0.000020, lambda: 0.003083
Out[11]: Try 8/100 Best val accuracy: 28, lr: 0.000054, lambda: 0.064499
Out[11]: Try 9/100 Best val accuracy: 31, lr: 0.000003, lambda: 0.004361
Out[11]: Try 10/100 Best val accuracy: 32, lr: 0.000004, lambda: 0.001610
Out[11]: Try 11/100 Best val accuracy: 31, lr: 0.000006, lambda: 0.300821
```

37% - relatively good  
for a 1-layer neural net  
and only 2000  
iterations

Make sure the best  
ones are not on the  
boundary

# Random Search vs. Grid Search



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

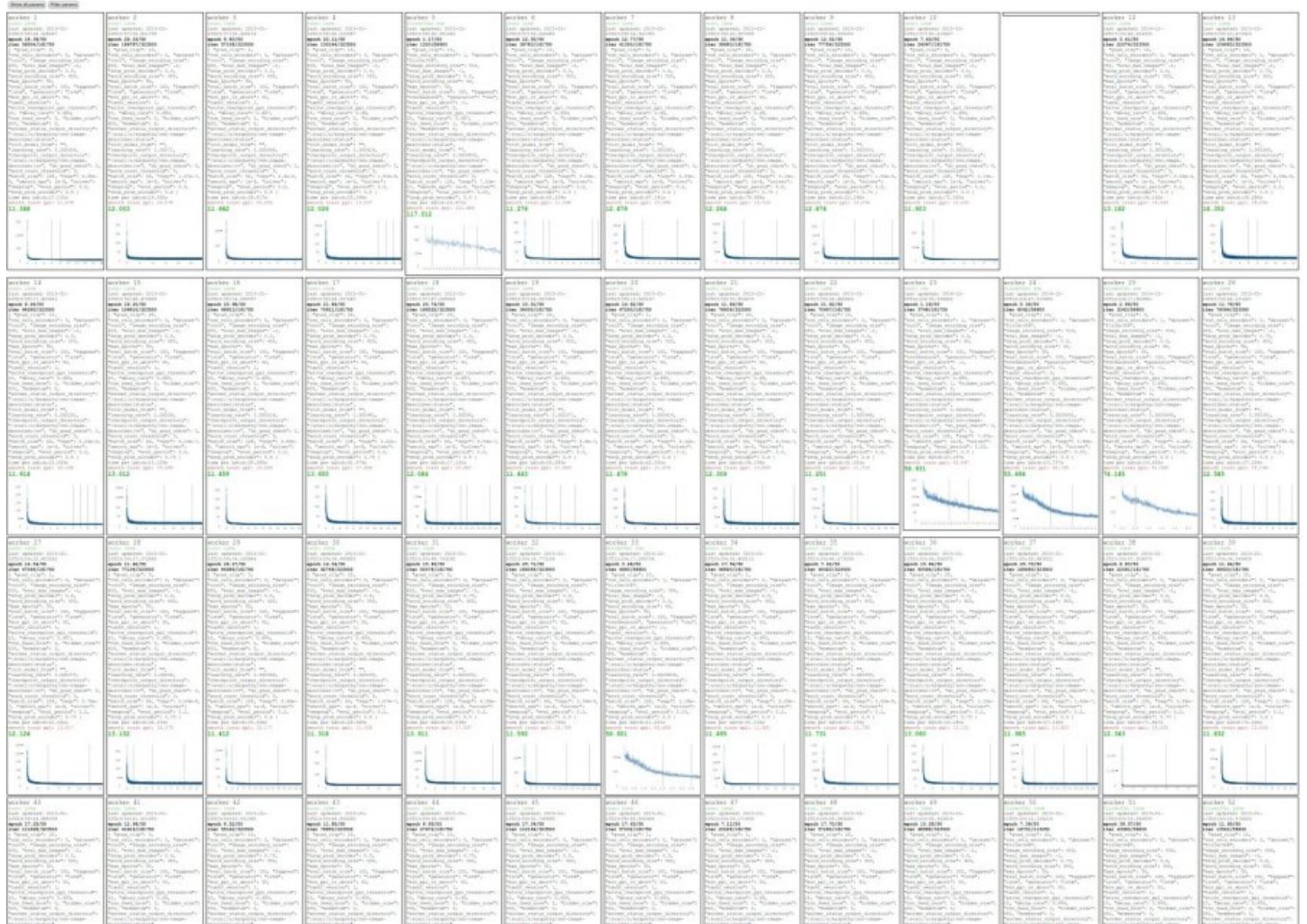
# Hyperparameters to play with:

- network architecture
- learning rate, its multiplier schedule
- regularization (L2/Dropout strength)

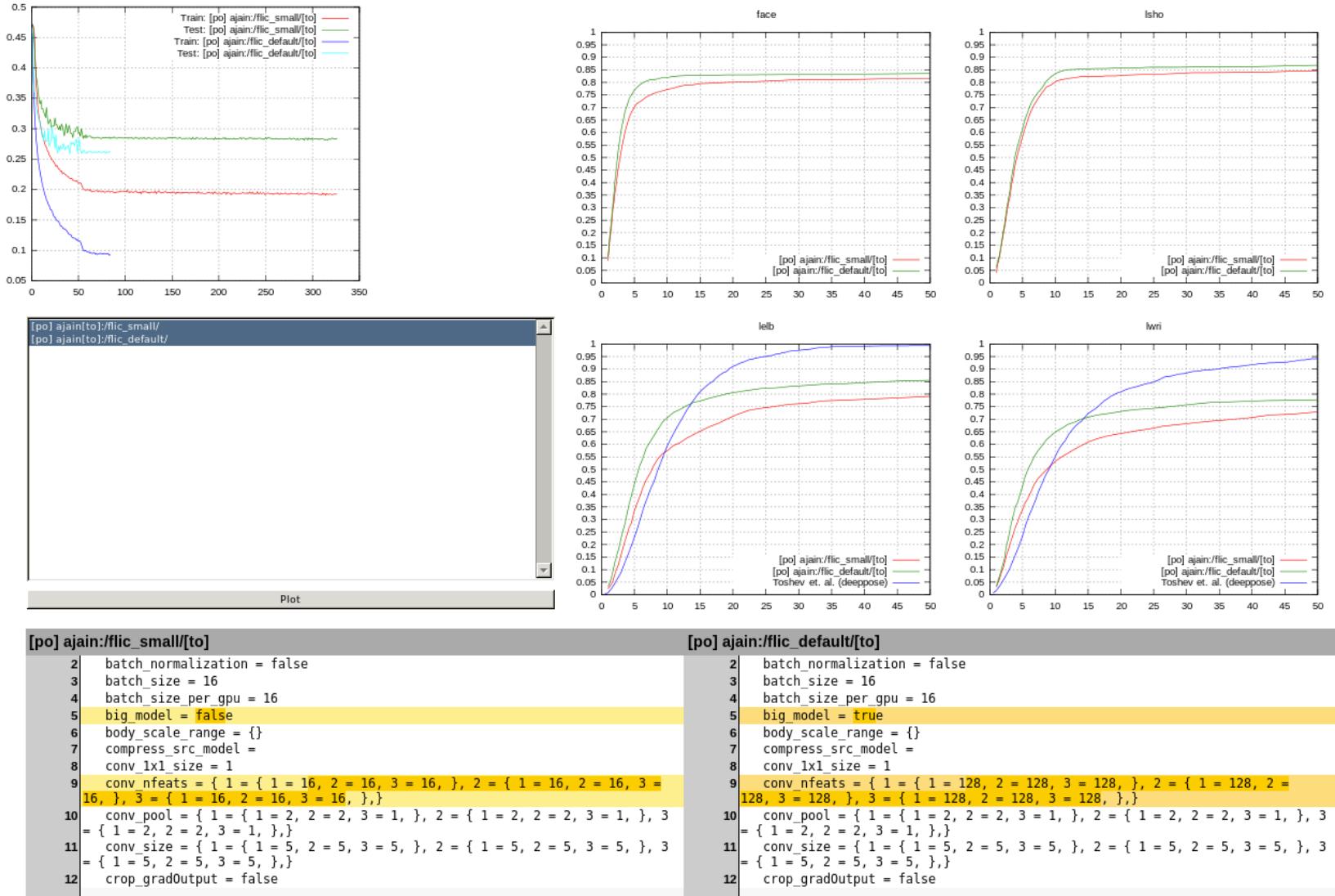
neural networks practitioner  
music = loss function



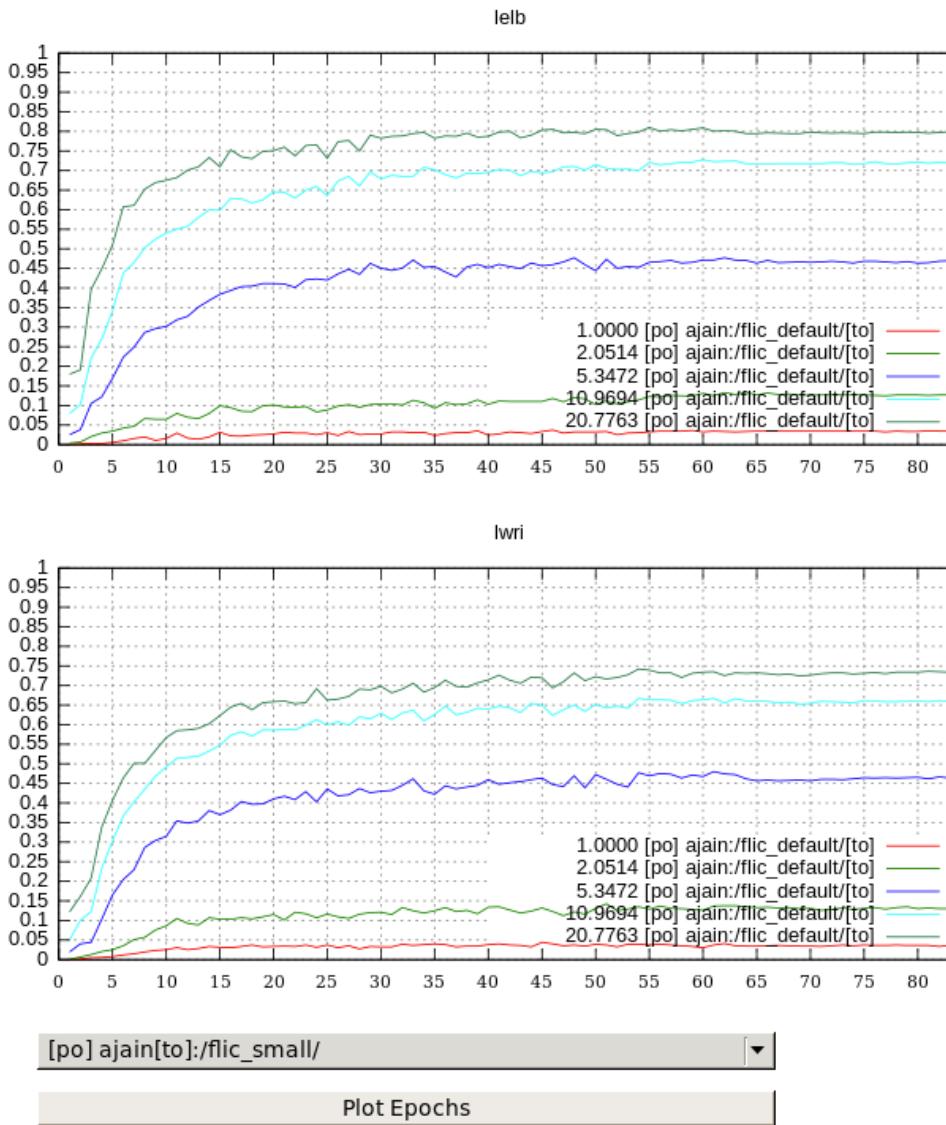
# Karpathy's cross-validation “command center”



# My cross-validation “command center”



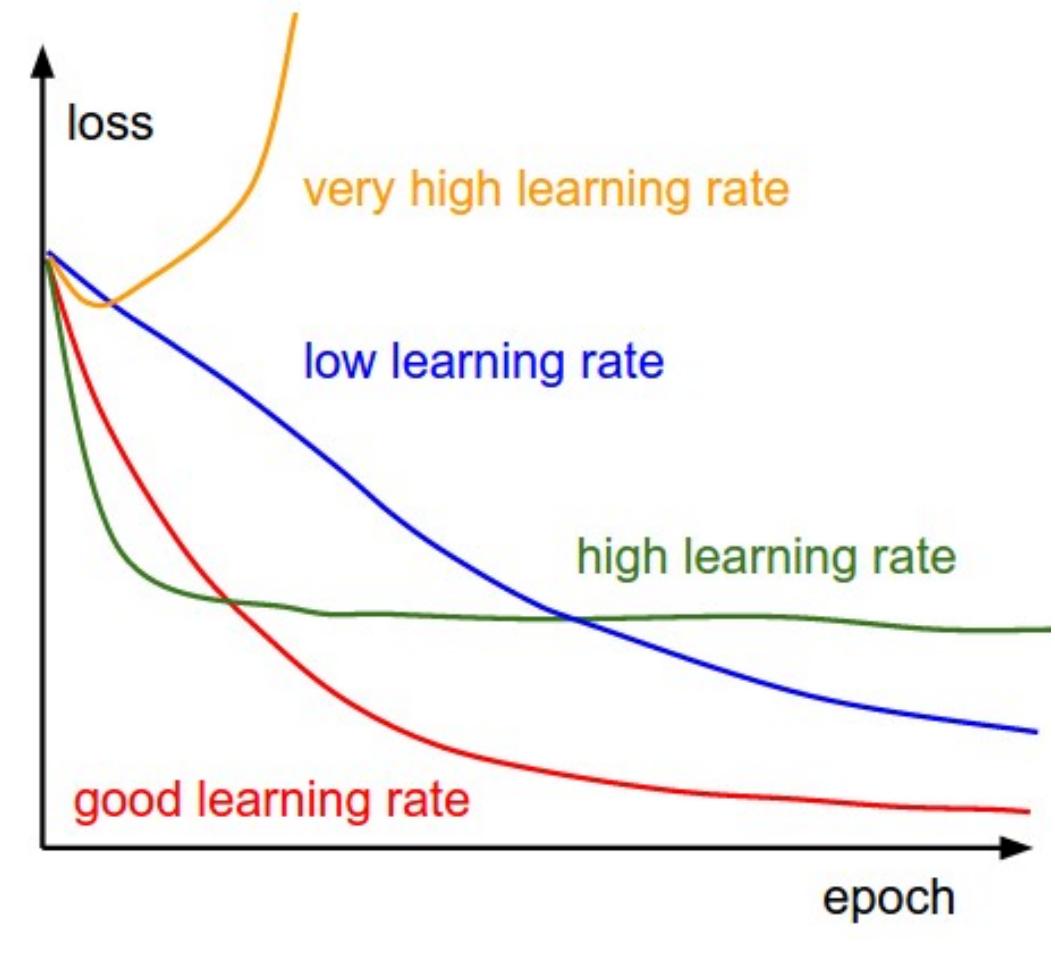
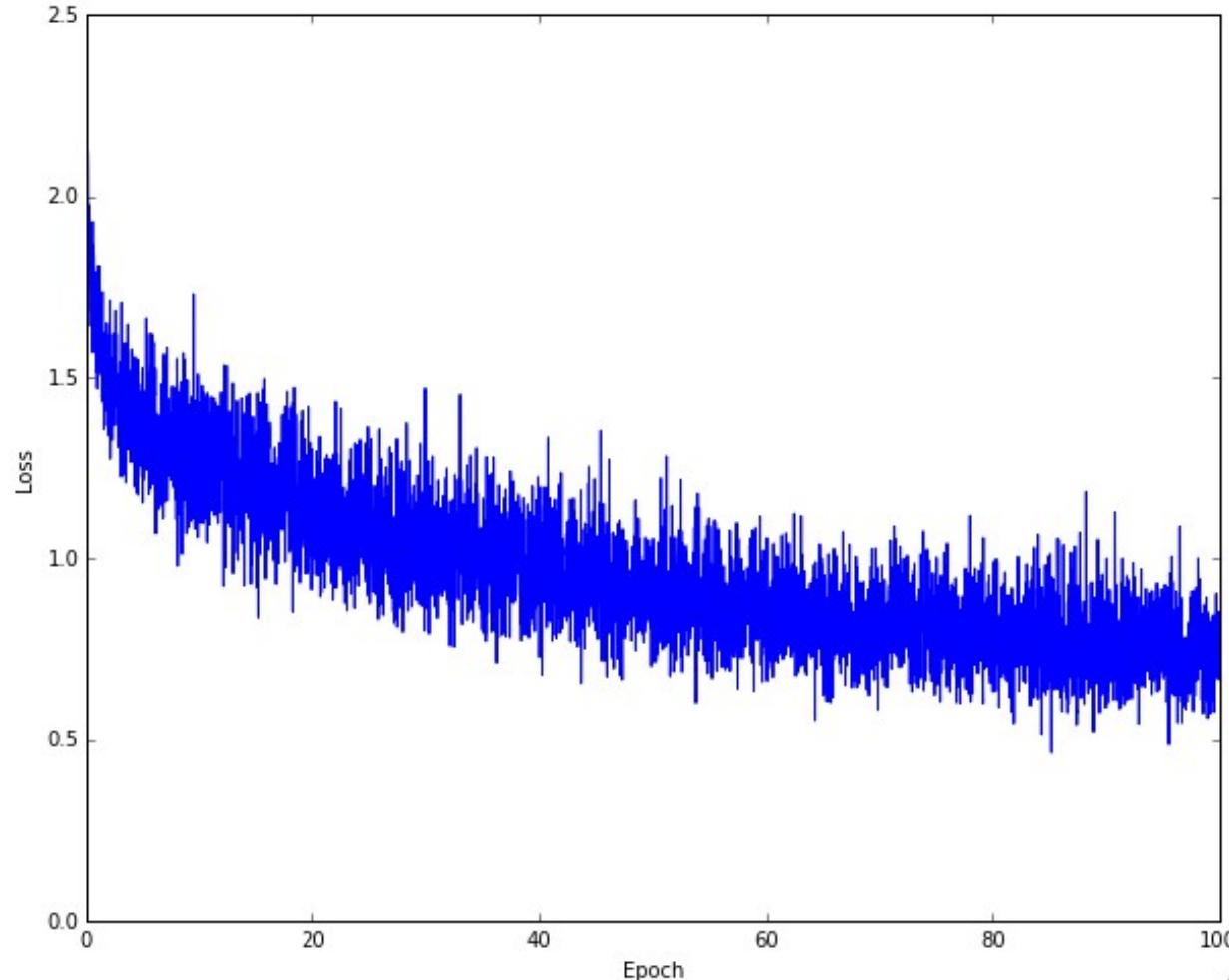
# My cross-validation “command center”

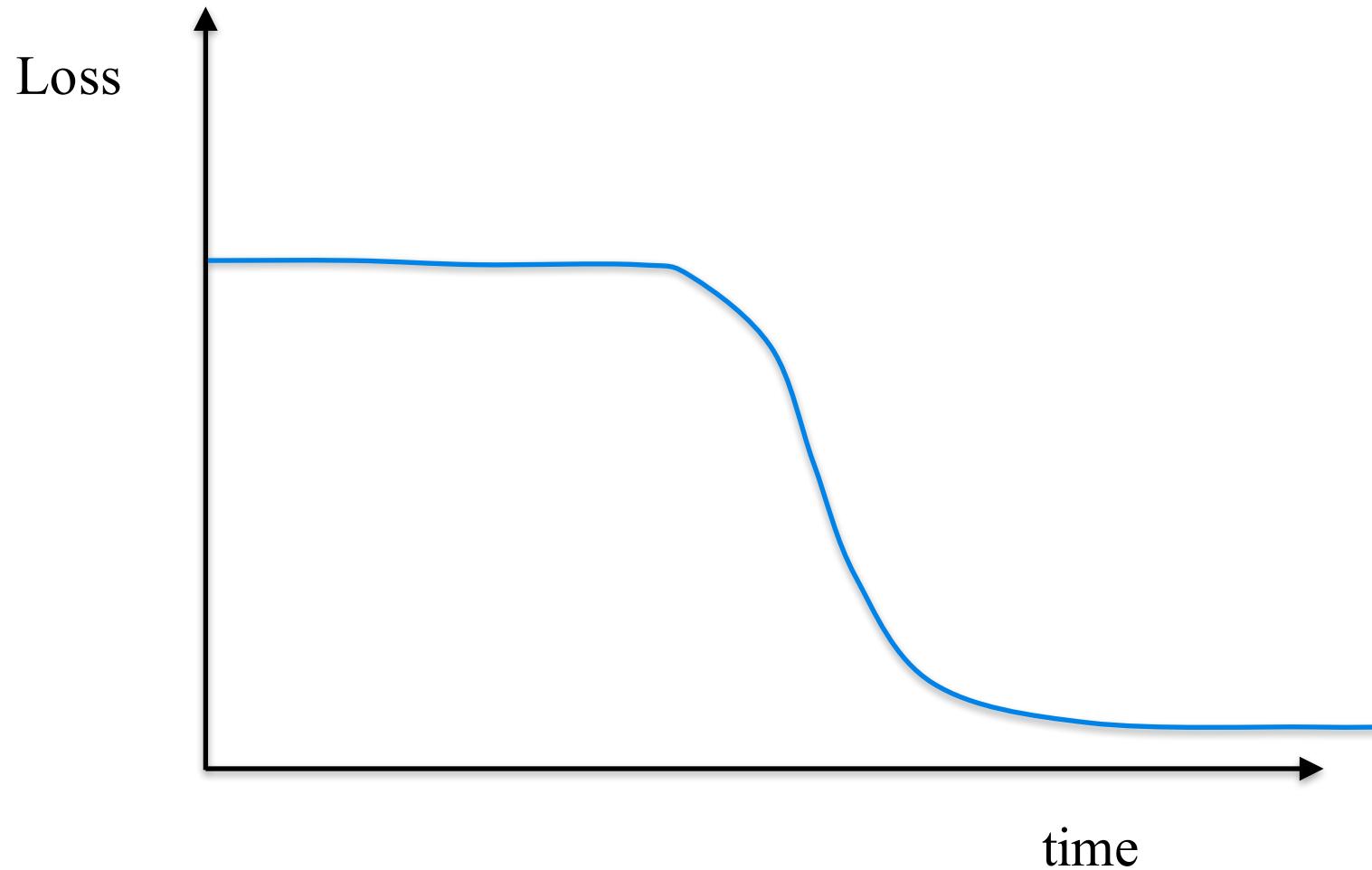


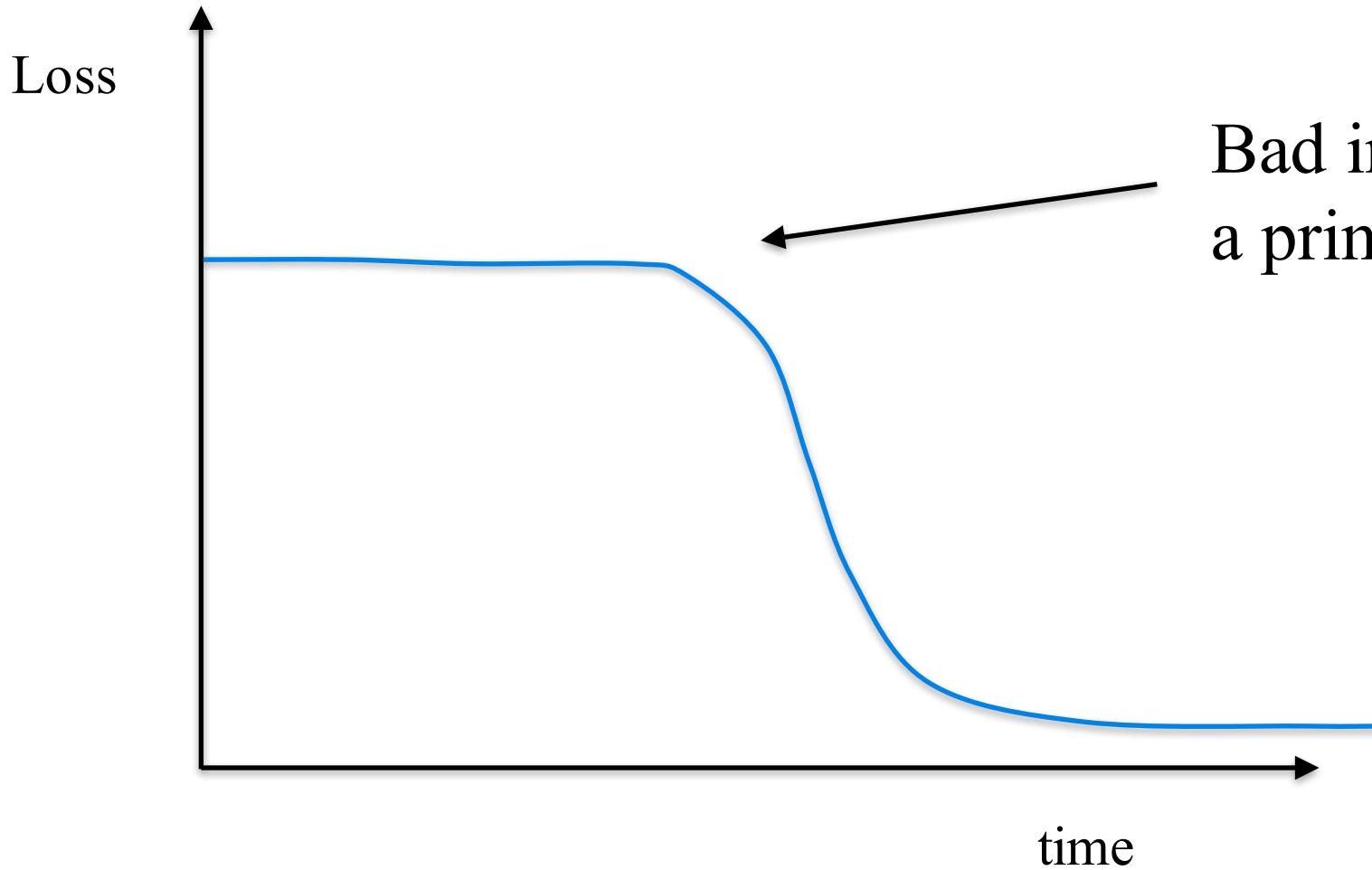
# My cross-validation “command center”



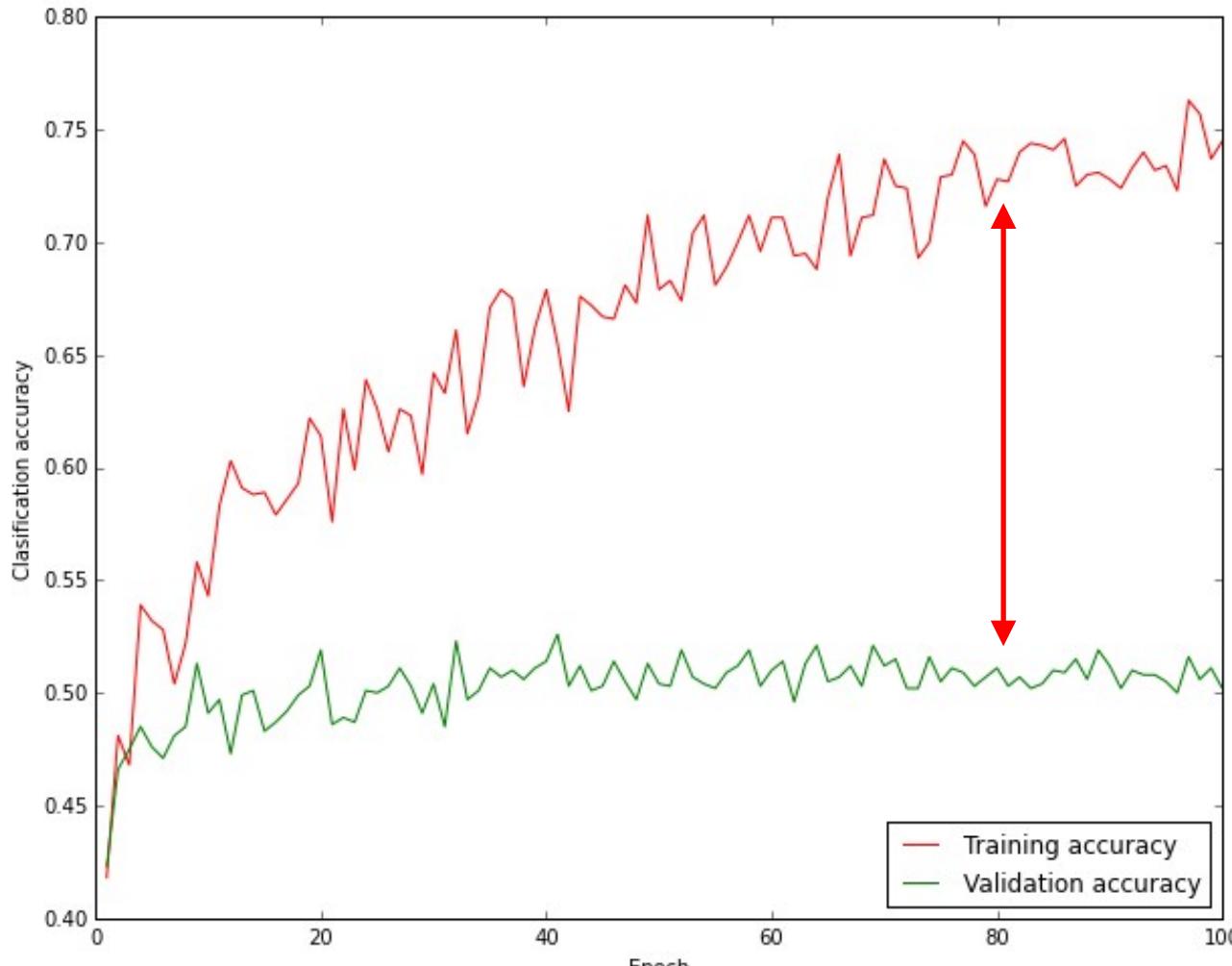
# Monitor and visualize the loss curve







# Monitor and visualize the accuracy:



big gap = overfitting  
=> increase regularization strength?

no gap  
=> increase model capacity?

# Track the ratio of weight updates / weight magnitudes:

```
function gradient_descent(model, lr)
    w_scale = torch.norm(model.W:view(model.W:nElement()), 2, 1)
    update_scale = torch.norm(lr * model.gradW:view(model.gradW:nElement()), 2, 1)
    model.W = model.W + lr * model.gradW
    model.b = model.b + lr * model.gradb
    print(update_scale/w_scale) -- Want ~1e-3
end
```

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

# Visualize Activations

- Visualize features (feature maps need to be uncorrelated) and have high variance.

**samples**



**hidden unit**

Good training: hidden units are sparse across samples and across features.

**samples**

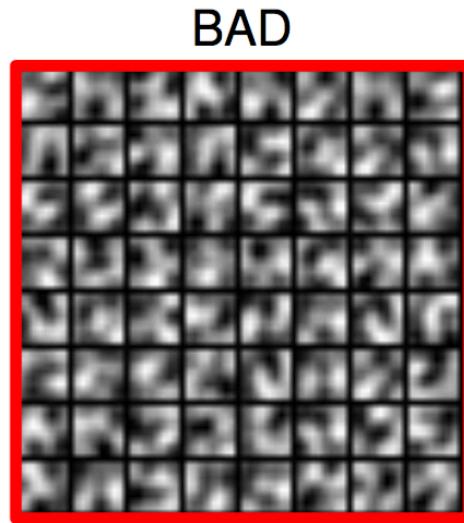
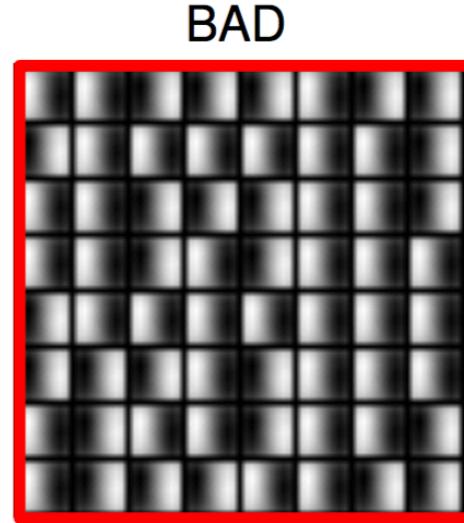
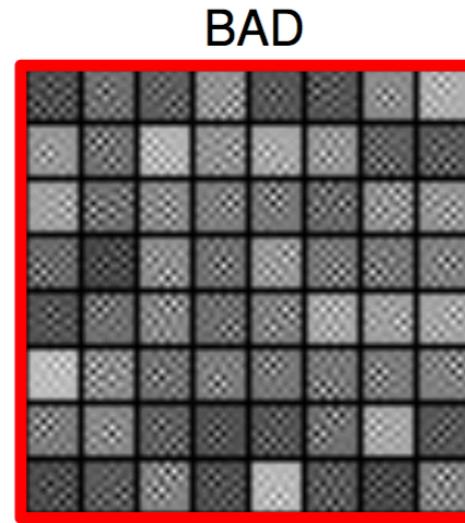
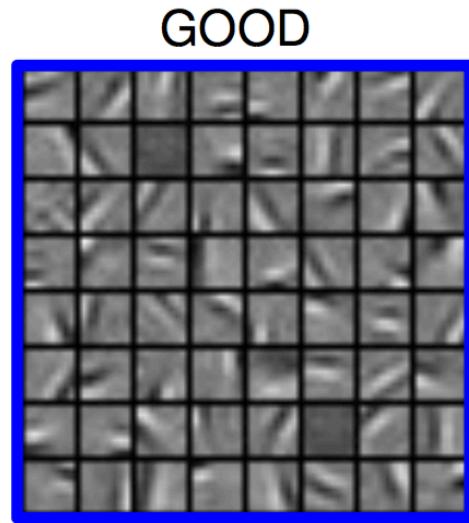


**hidden unit**

Bad training: many hidden units ignore the input and/or exhibit strong correlations.

# Visualize (initial) Convolution Layer Weights

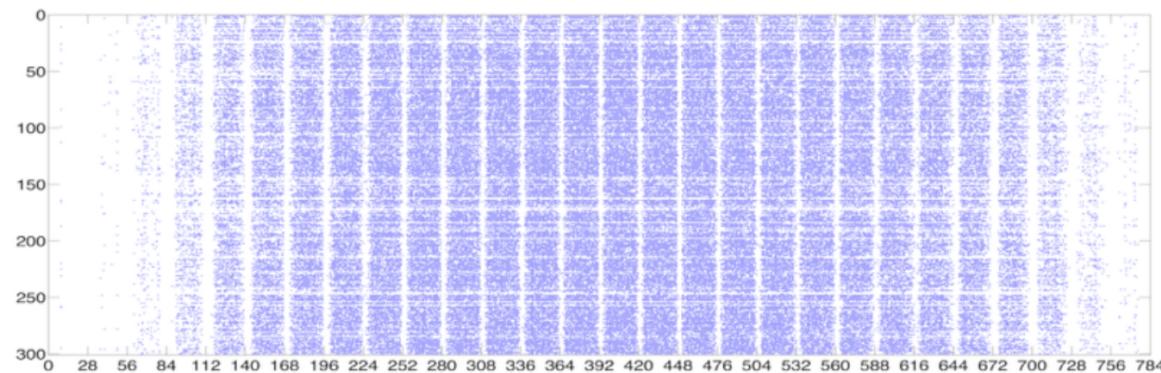
- Visualize features (feature maps need to be uncorrelated) and have high variance.



Good training: learned filters exhibit structure and are uncorrelated.

# Visualize Linear Layer (Fully-Connected) Weights

- Visualization of Linear layer weights for some networks
- It has a banded structure repeated 28 times (Why?!) Hint: Images are 28x28
- Thus, looking at the weights we get some intuition



Thank you!