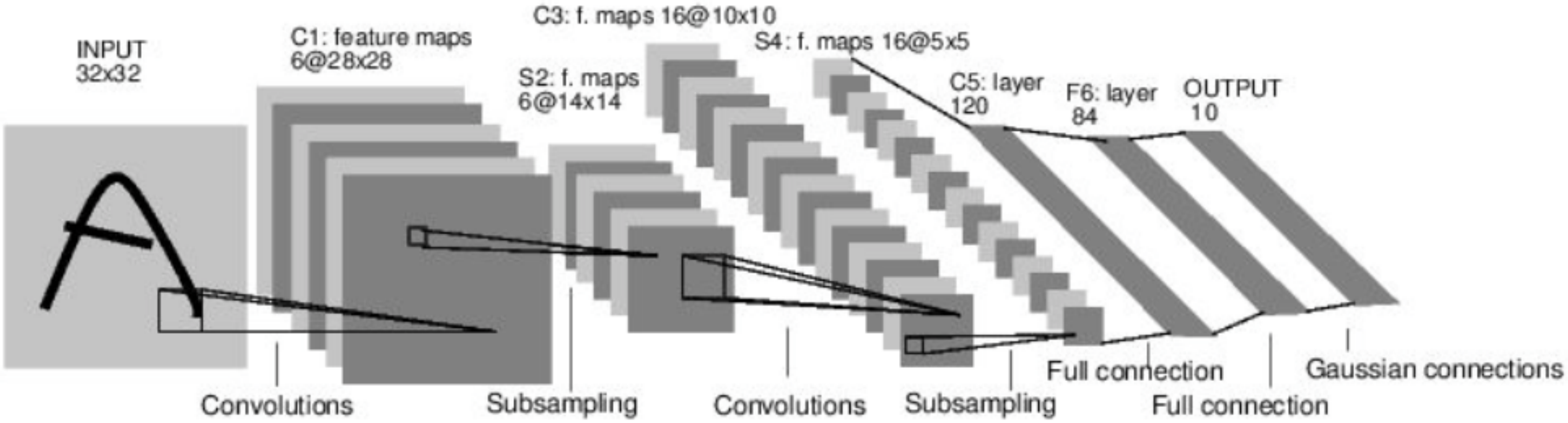
ELEC/COMP 576: Introduction to Convnets Lecture 4

Ankit B. Patel
Baylor College of Medicine (Neuroscience Dept.)
Rice University (ECE Dept.)

Convolutional Networks
(Convnets)

Convolutional Neural Network



[LeNet-5, LeCun 1980]

History of Convolutional Neural Network

- In 1962, Hubel and Wiesel describe simple and complex cells in visual area V1 (inspiration for later NNs: S-->template matching for pattern specificity and C-->pooling for robustness to nuisances)

- In 1979, Fukushima introduces the Neocognitron. It foreshadows current deep NNs: convolutional layers, weight replication, and WTA-subsampling. However its unsupervised

- In 1989, LeCun applies Backprop to Fukushima's Neocognitron to do supervised learning. This is the first incarnation of modern convolutional neural nets (CNNs) and subsequently used by US Post Office for address reading.

- In 1999, Riesenhuber and Poggio introduce HMAX, a computational model that summarizes the basic facts about the ventral visual stream

- In 2012, Krizhevsky introduces AlexNet which is implemented in GPUs and win the ImageNet Challenge

Image classification

Year	Codename	Error (percent)	99.9% Conf Int
2014	**GoogLeNet**	**6.66**	**6.40 - 6.92**
2014	VGG	7.32	7.05 - 7.60
2014	MSRA	8.06	7.78 - 8.34
2014	AHoward	8.11	7.83 - 8.39
2014	DeeperVision	9.51	9.21 - 9.82
2013	Clarifai†	11.20	10.87 - 11.53
2014	CASIAWS†	11.36	11.03 - 11.69
2014	Trimps†	11.46	11.13 - 11.80
2014	Adobe†	11.58	11.25 - 11.91
2013	**Clarifai**	**11.74**	**11.41 - 12.08**
2013	NUS	12.95	12.60 - 13.30
2013	ZF	13.51	13.14 - 13.87
2013	AHoward	13.55	13.20 - 13.91
2013	OverFeat	14.18	13.83 - 14.54
2014	Orange†	14.80	14.43 - 15.17
2012	SuperVision†	15.32	14.94 - 15.69
2012	**SuperVision**	**16.42**	**16.04 - 16.80**
2012	ISI	26.17	25.71 - 26.65
2012	VGG	26.98	26.53 - 27.43
2012	XRCE	27.06	26.60 - 27.52
2012	UvA	29.58	29.09 - 30.04

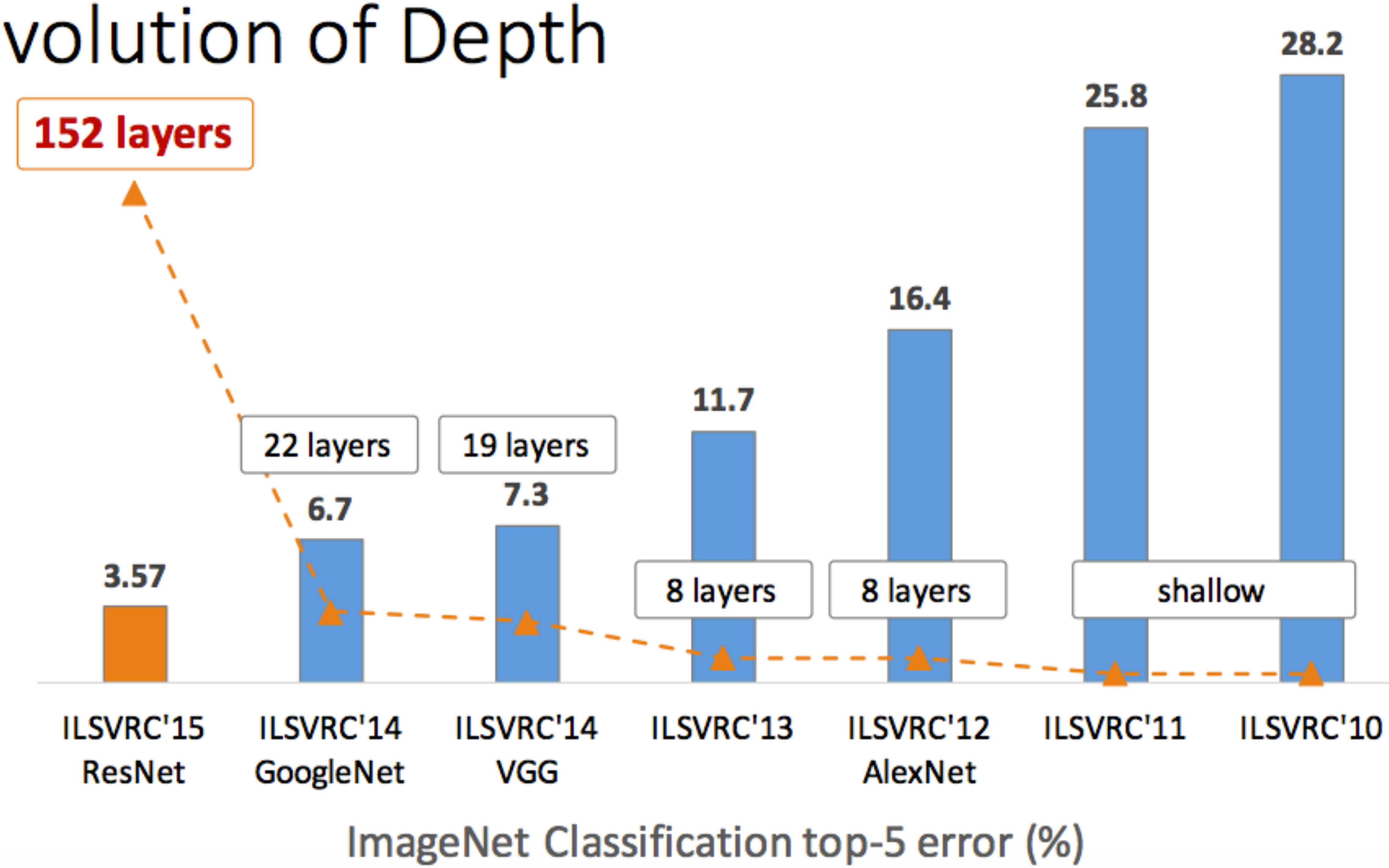
Single-object localization

Year	Codename	Error (percent)	99.9% Conf Int
2014	**VGG**	**25.32**	**24.87 - 25.78**
2014	GoogLeNet	26.44	25.98 - 26.92
2013	**OverFeat**	**29.88**	**29.38 - 30.35**
2014	Adobe†	30.10	29.61 - 30.58
2014	SYSU	31.90	31.40 - 32.40
2012	SuperVision†	33.55	33.05 - 34.04
2014	MIL	33.74	33.24 - 34.25
2012	**SuperVision**	**34.19**	**33.67 - 34.69**
2014	MSRA	35.48	34.97 - 35.99
2014	Trimps†	42.22	41.69 - 42.75
2014	Orange†	42.70	42.18 - 43.24
2013	VGG	46.42	45.90 - 46.95
2012	VGG	50.03	49.50 - 50.57
2012	ISI	53.65	53.10 - 54.17
2014	CASIAWS†	61.96	61.44 - 62.48

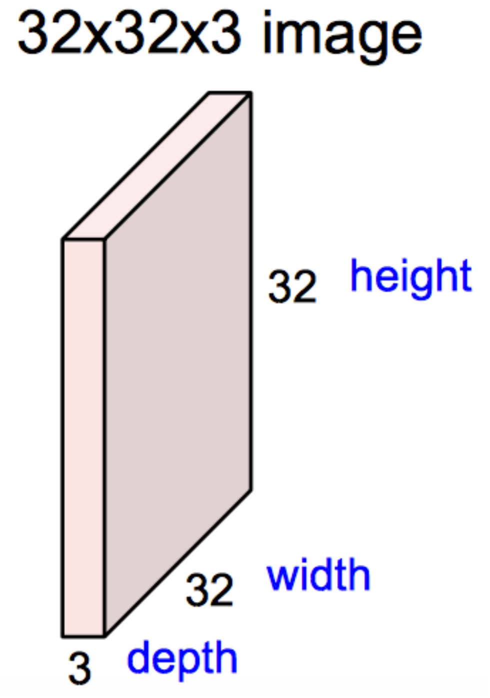
Object detection

Year	Codename	AP (percent)	99.9% Conf Int
2014	GoogLeNet†	43.93	42.92 - 45.65
2014	CUHK†	40.67	39.68 - 42.30
2014	DeepInsight†	40.45	39.49 - 42.06
2014	**NUS**	**37.21**	**36.29 - 38.80**
2014	UvA†	35.42	34.63 - 36.92
2014	MSRA	35.11	34.36 - 36.70
2014	Berkeley†	34.52	33.67 - 36.12
2014	UvA	32.03	31.28 - 33.49
2014	Southeast	30.48	29.70 - 31.93
2014	HKUST	28.87	28.03 - 30.20
2013	**UvA**	**22.58**	**22.00 - 23.82**
2013	NEC†	20.90	20.40 - 22.15
2013	NEC	19.62	19.14 - 20.85
2013	OverFeat†	19.40	18.82 - 20.61
2013	Toronto	11.46	10.98 - 12.34
2013	SYSU	10.45	10.04 - 11.32
2013	UCLA	9.83	9.48 - 10.77

Revolution of Depth

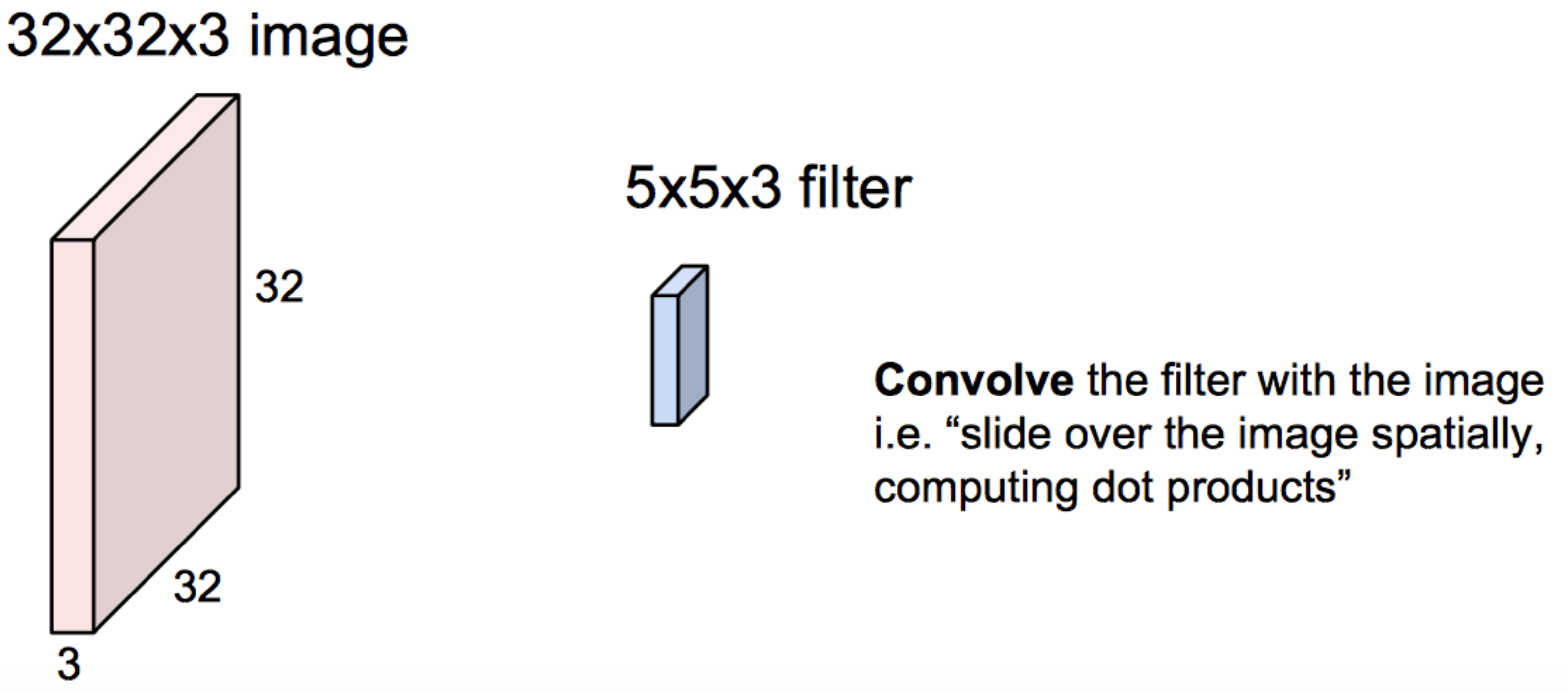


Convolutional Layer



[Fei-Fei Li, Andrej Karpathy, Justin Johns

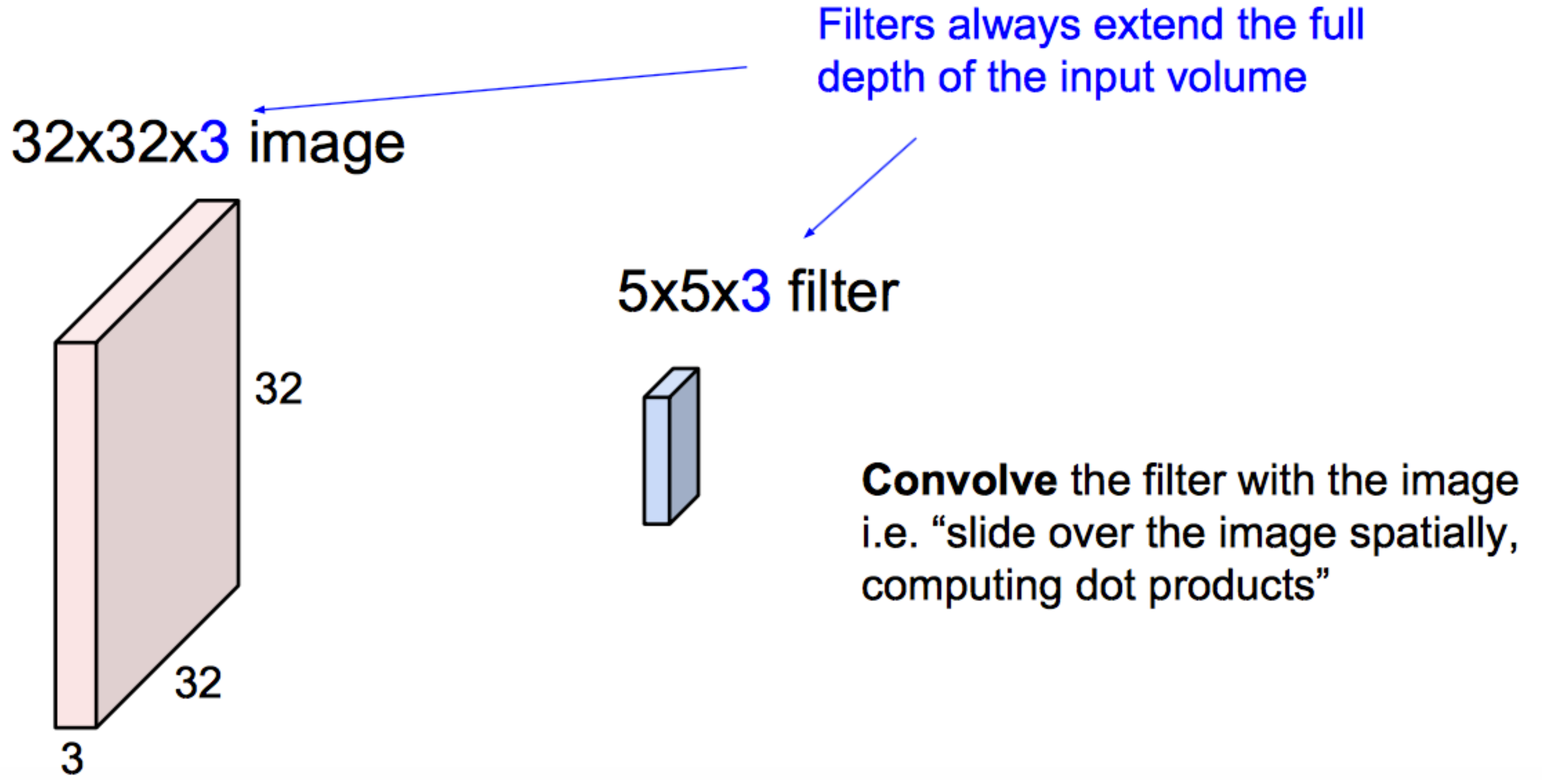
Convolutional Layer



Convolve the filter with the image i.e. “slide over the image spatially, computing dot products”

[Fei-Fei Li, Andrej Karpathy, Justin Johns

Convolutional Layer



Filters always extend the full depth of the input volume

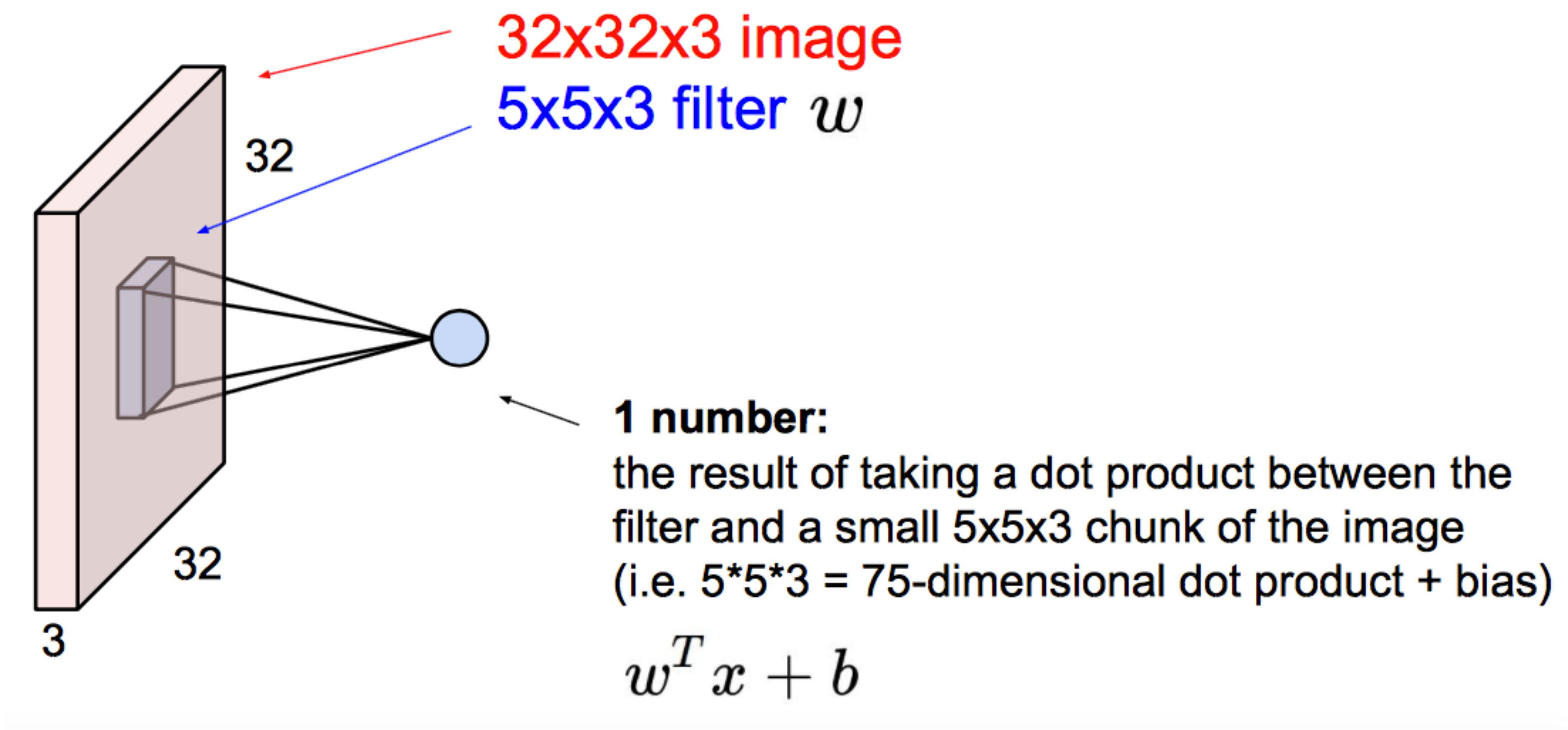
32x32x3 image

5x5x3 filter

Convolve the filter with the image i.e. “slide over the image spatially, computing dot products”

[Fei-Fei Li, Andrej Karpathy, Justin Johns

Convolutional Layer

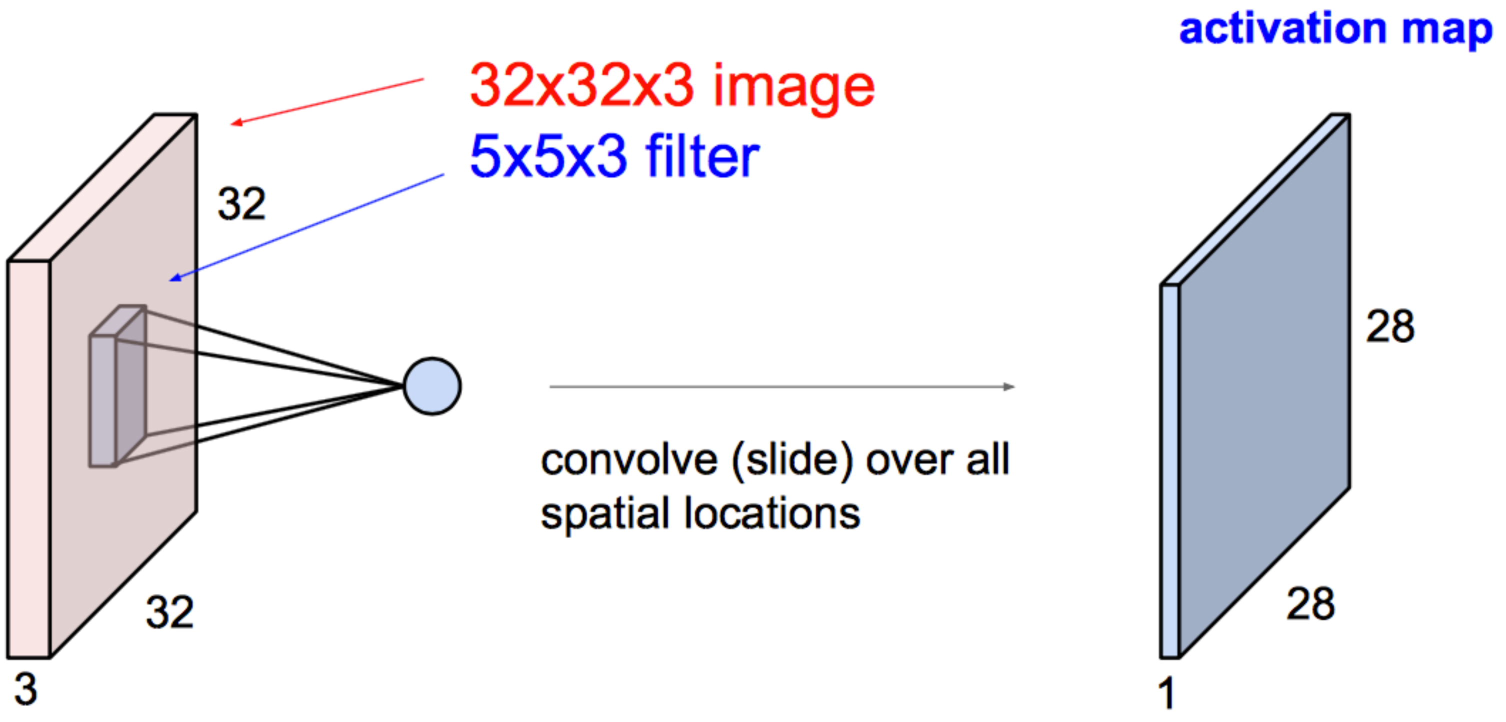


1 number:
the result of taking a dot product between the filter and a small 5x5x3 chunk of the image (i.e. 5*5*3 = 75-dimensional dot product + bias)

$$w^T x + b$$

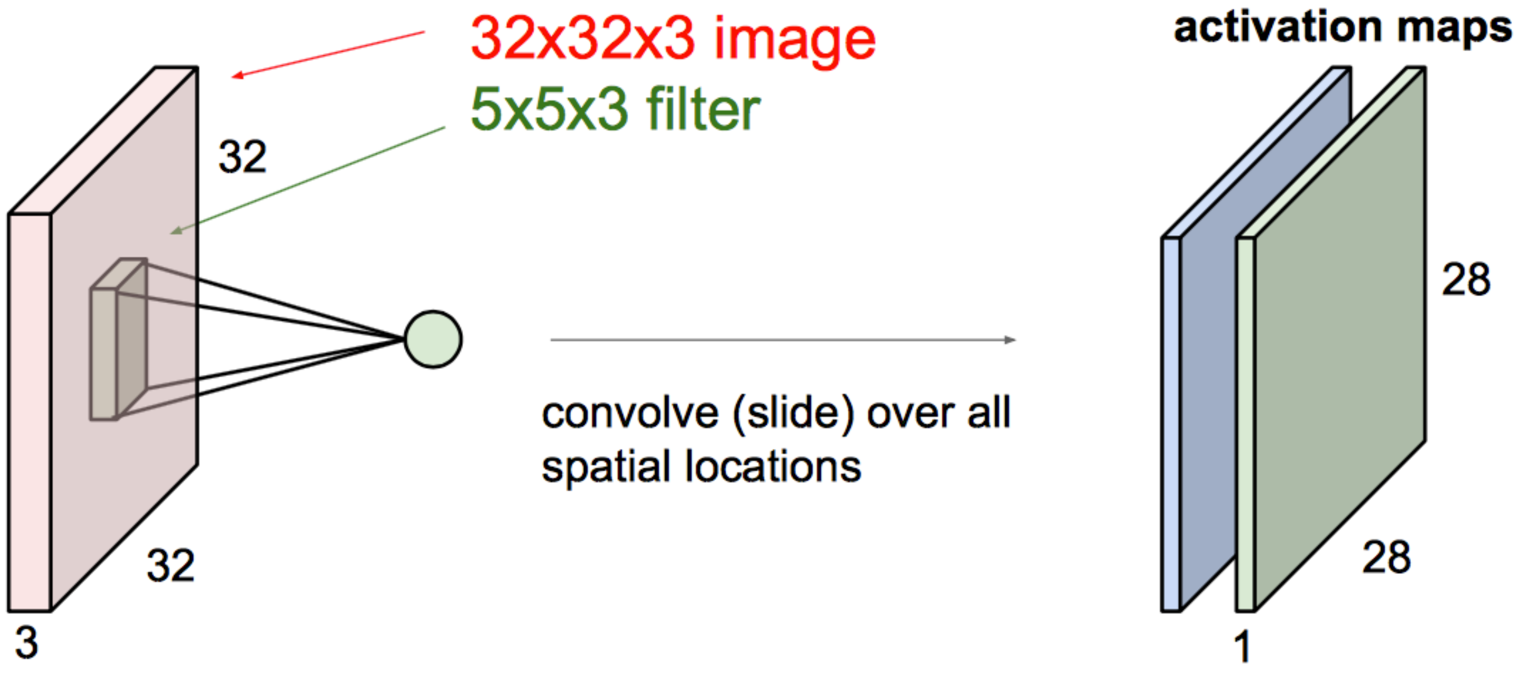
[Fei-Fei Li, Andrej Karpathy, Justin Johns

Convolutional Layer



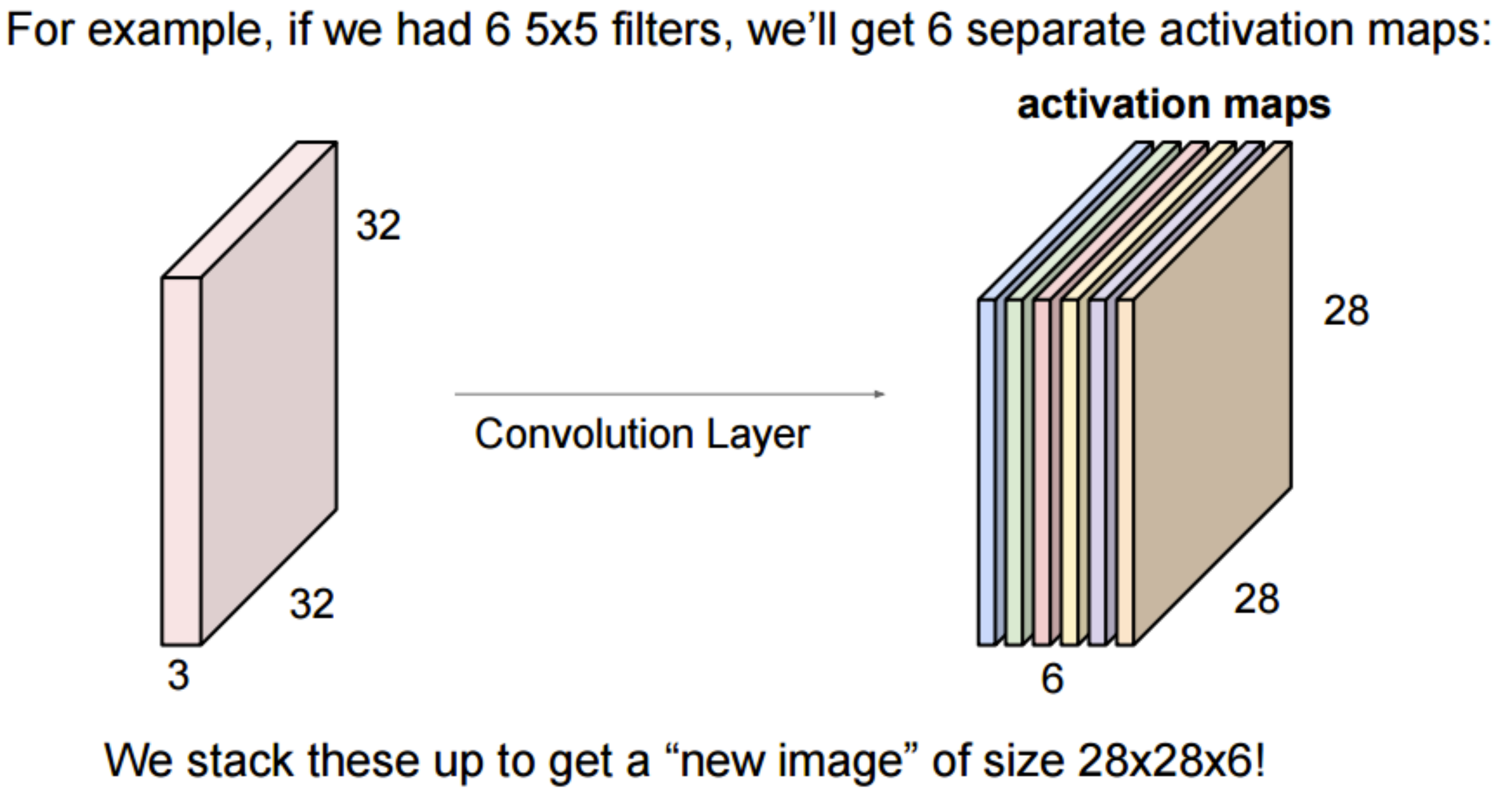
[Fei-Fei Li, Andrej Karpathy, Justin Johns

Convolutional Layer



[Fei-Fei Li, Andrej Karpathy, Justin Johns

Convolutional Layer



For example, if we had 6 5x5 filters, we'll get 6 separate activation maps:

We stack these up to get a "new image" of size 28x28x6!

Common to zero-pad the border

[Fei-Fei Li, Andrej Karpathy, Justin Johnson]

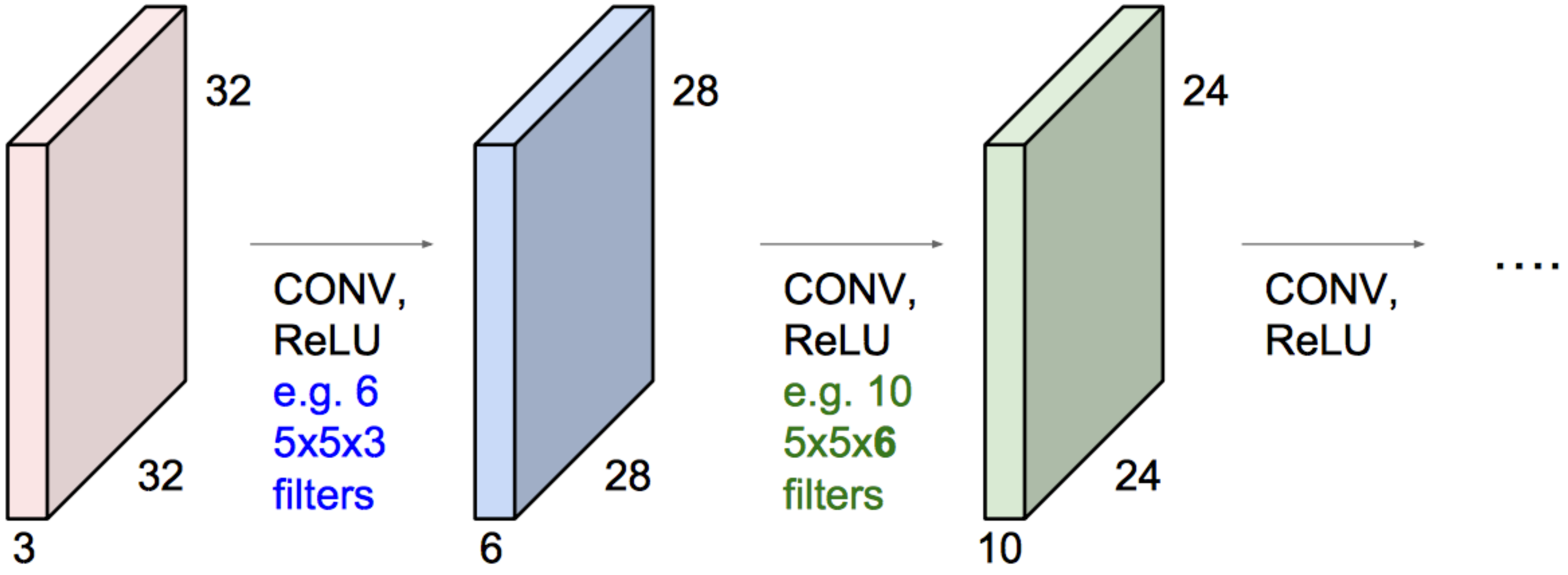
Question (3 min):

What is the special structure of the matrix that corresponds to a Convolution operation?

Can you exploit this structure to design a more efficient algorithm for computing the convolution?

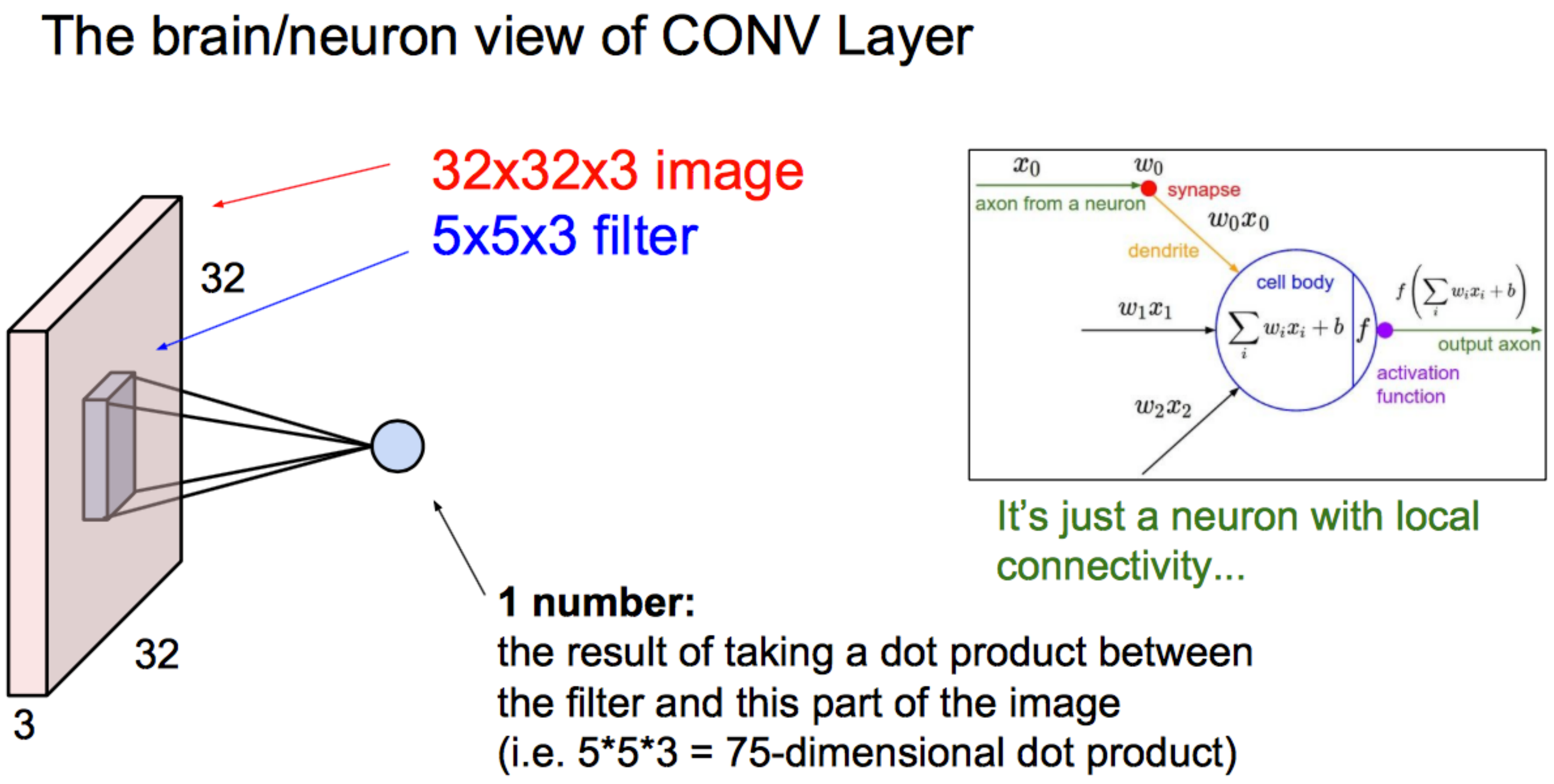
Convolutional Network

Preview: ConvNet is a sequence of Convolutional Layers, interspersed with activation functions



[Fei-Fei Li, Andrej Karpathy, Justin Johns

A Neural View of Convolutional Layer



The brain/neuron view of CONV Layer

32x32x3 image

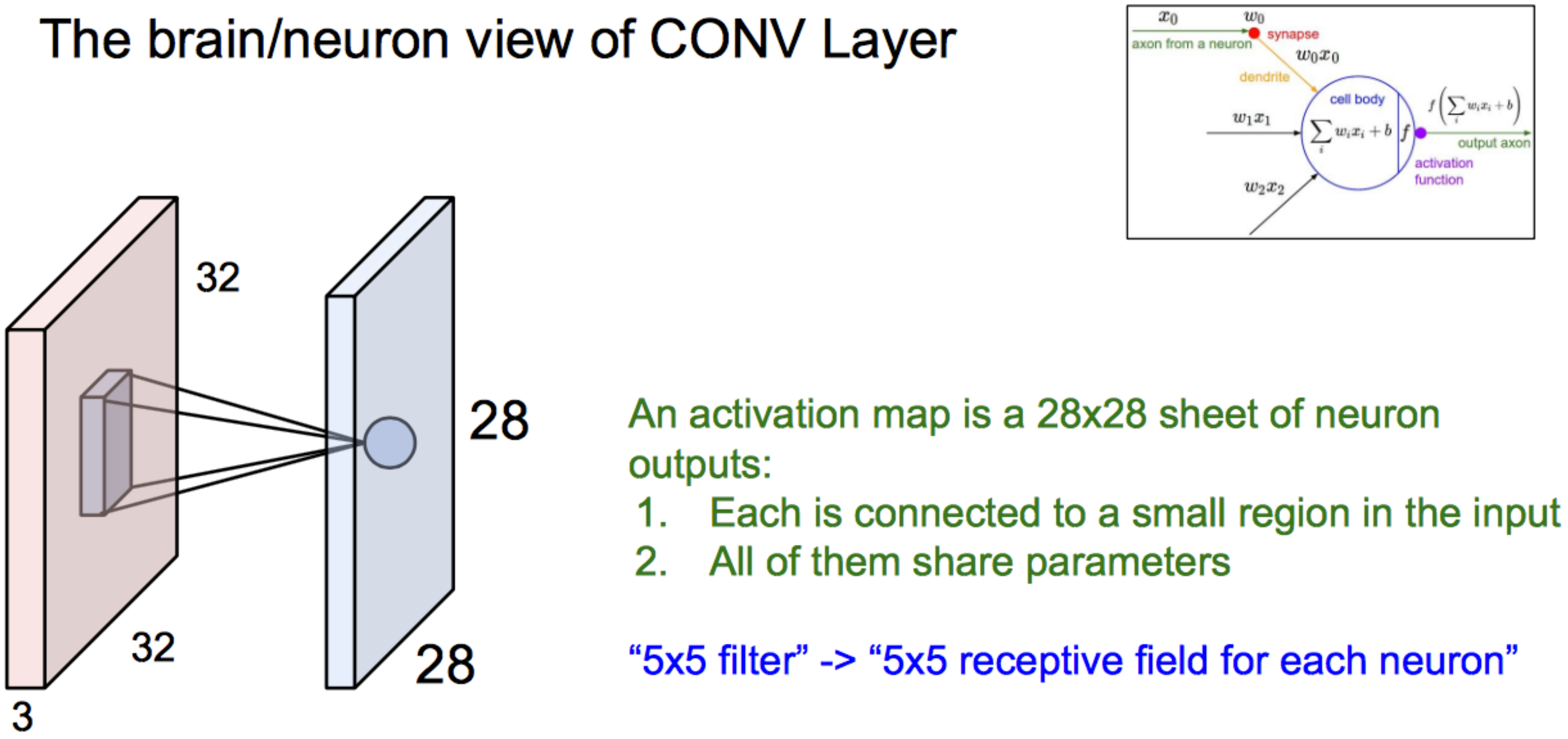
5x5x3 filter

1 number:
the result of taking a dot product between
the filter and this part of the image
(i.e. 5*5*3 = 75-dimensional dot product)

It's just a neuron with local connectivity...

[Fei-Fei Li, Andrej Karpathy, Justin Johns

A Neural View of Convolutional Layer



The brain/neuron view of CONV Layer

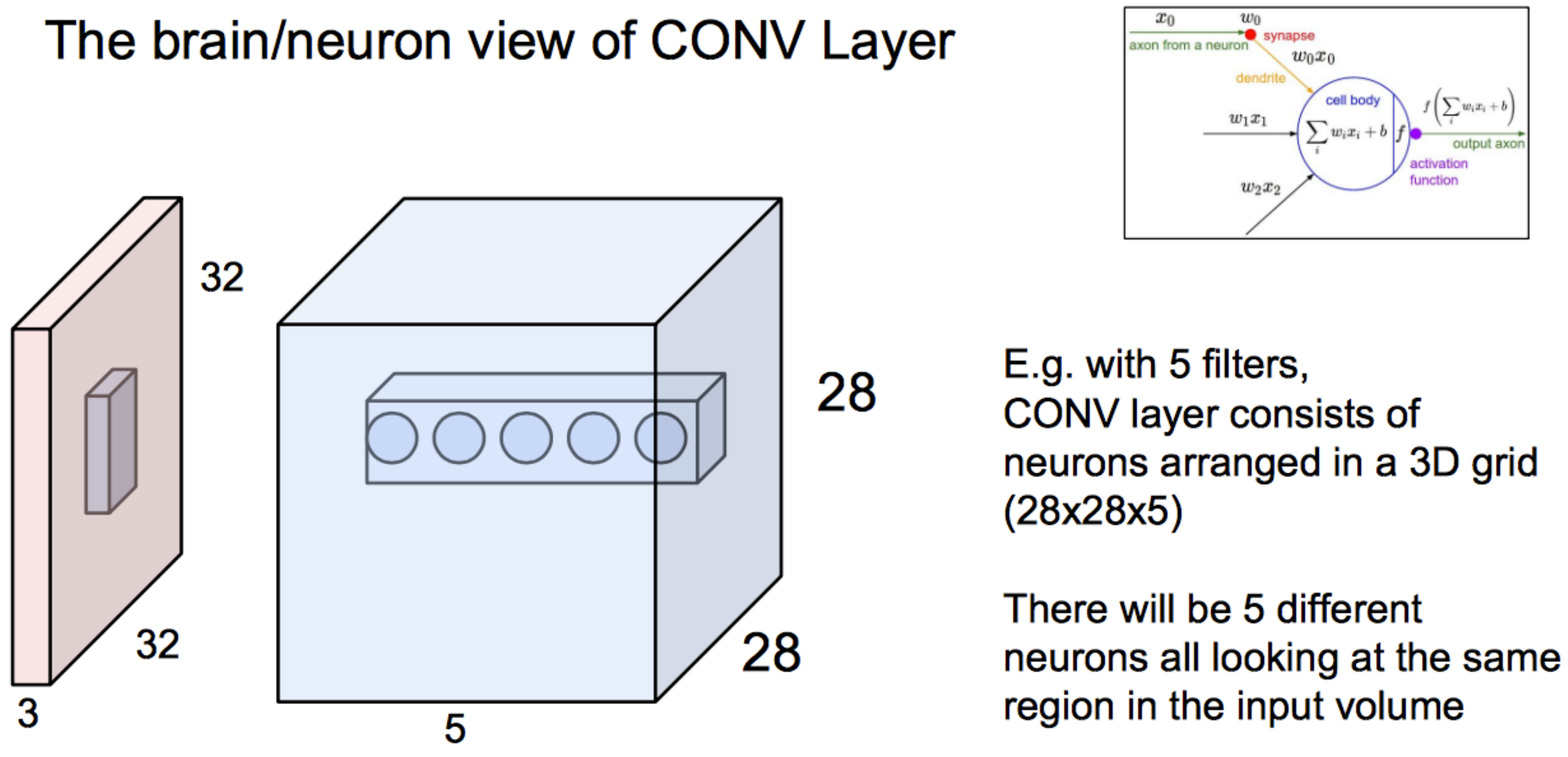
An activation map is a 28x28 sheet of neuron outputs:

1. Each is connected to a small region in the input
2. All of them share parameters

“5x5 filter” -> “5x5 receptive field for each neuron”

[Fei-Fei Li, Andrej Karpathy, Justin Johns

A Neural View of Convolutional Layer



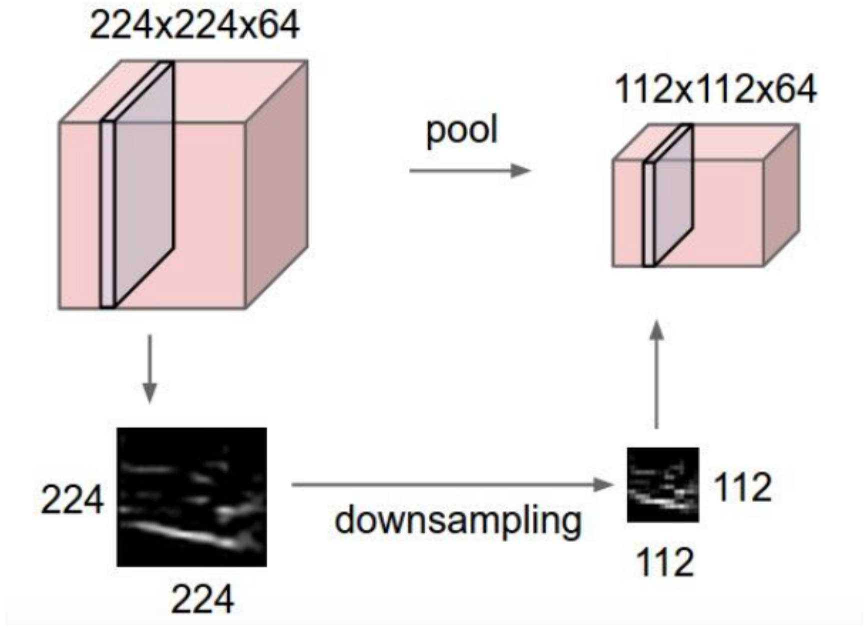
The brain/neuron view of CONV Layer

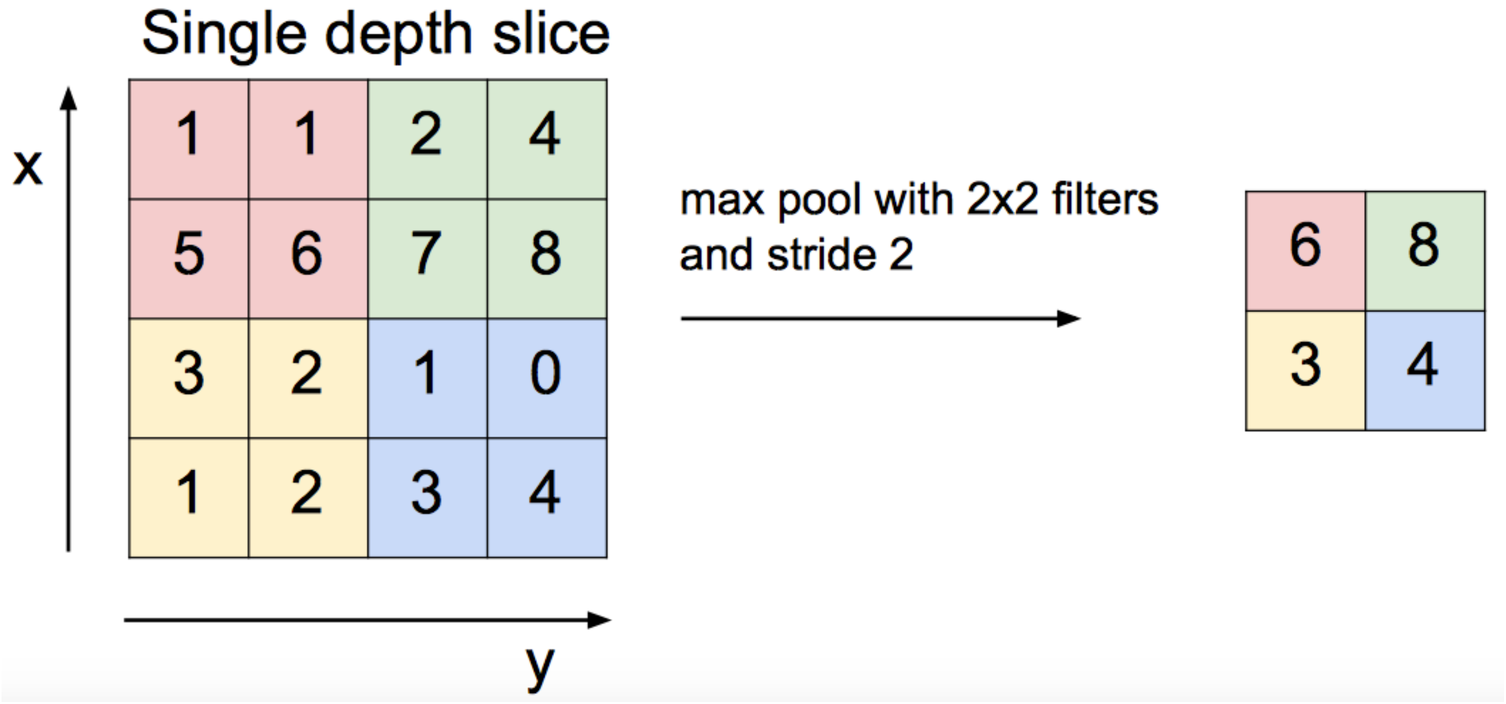
E.g. with 5 filters,
CONV layer consists of
neurons arranged in a 3D grid
(28x28x5)

There will be 5 different
neurons all looking at the same
region in the input volume

[Fei-Fei Li, Andrej Karpathy, Justin Johns

Pooling Layer



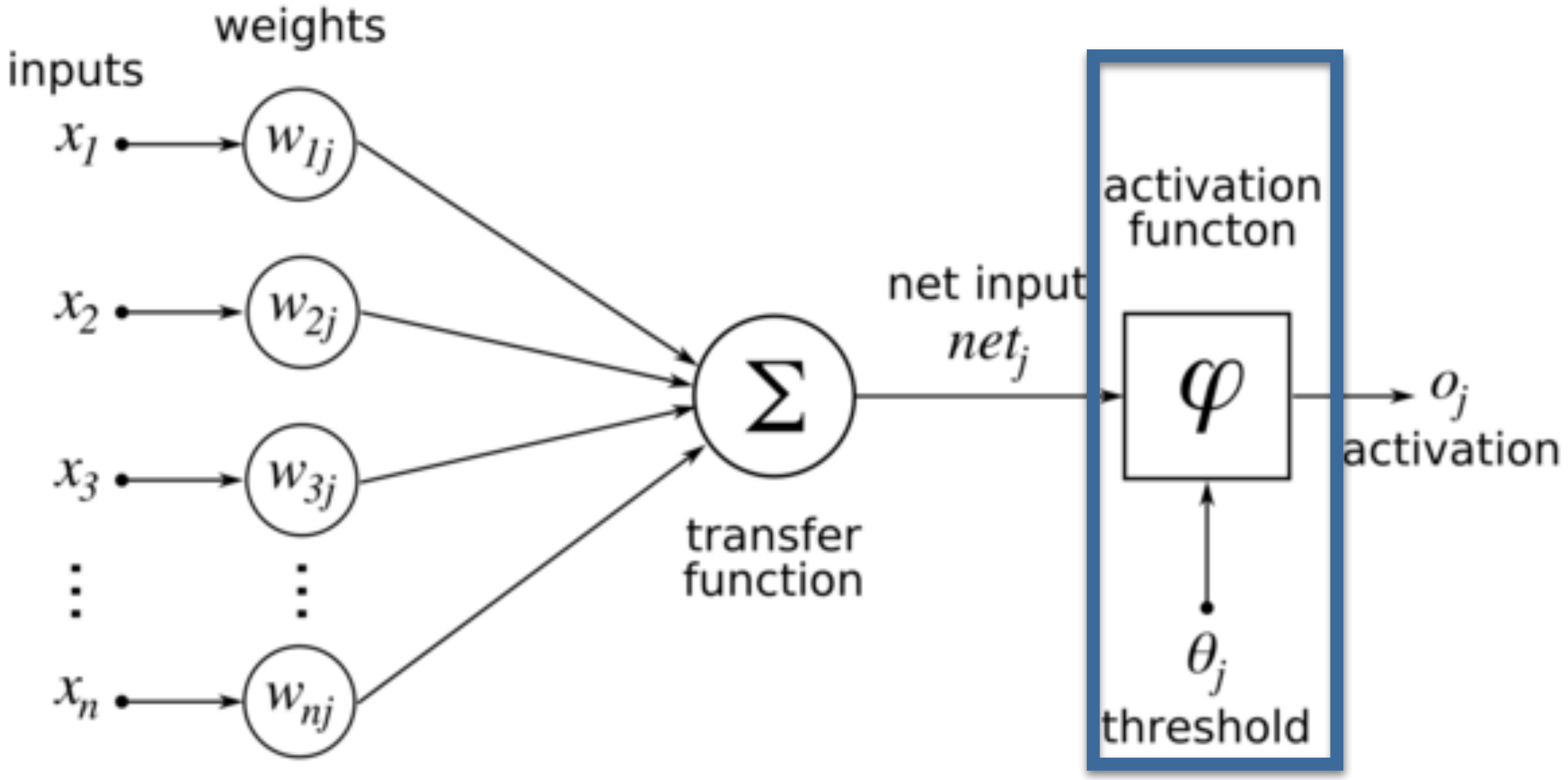


1	1	2	4
5	6	7	8
3	2	1	0
1	2	3	4

6	8
3	4

[Fei-Fei Li, Andrej Karpathy, Justin Johnson]

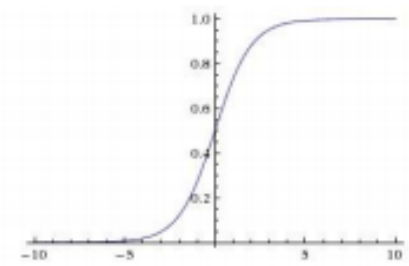
Activation Functions



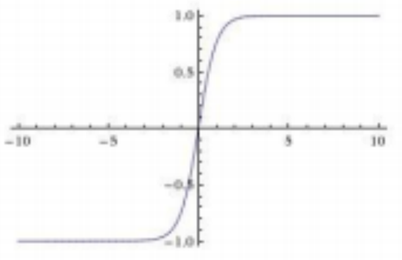
Activation Functions

Sigmoid

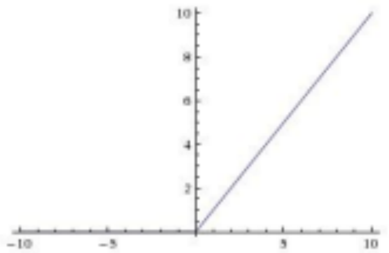
$$\sigma(x) = 1/(1+e^{-x})$$



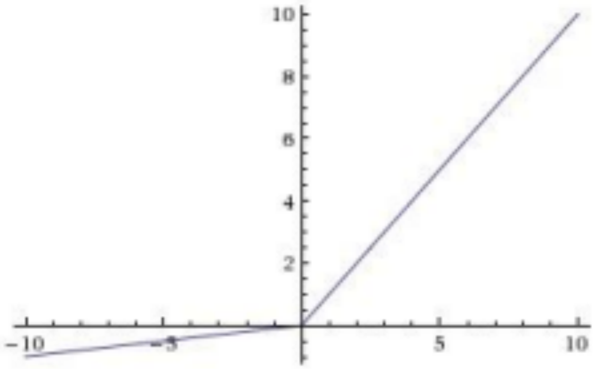
tanh tanh(x)



ReLU max(0,x)



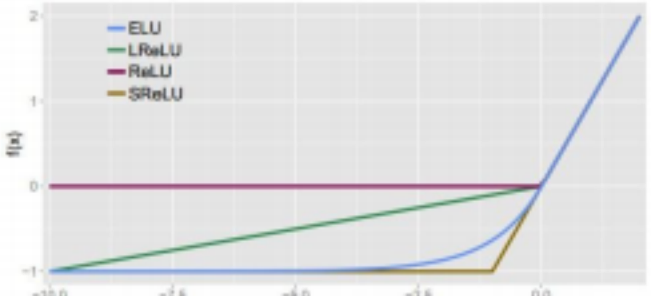
Leaky ReLU
max(0.1x, x)



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

ELU

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



[Fei-Fei Li, Andrej Karpathy, Justin Johnson]

Classification with 2-Layer Convnet: Visualizing the Mechanism Inside

- http://cs.stanford.edu/people/karpathy/convnetjs/demo/classify2d.html
- Try playing around with this app to build intuition:
    - change datapoints to see how decision boundaries change
    - change network layer types, widths, activation functions, etc.
    - try shallower vs deeper

Training on CIFAR10

- http://cs.stanford.edu/people/karpathy/convnetjs/demo/cifar10.html

[Fei-Fei Li, Andrej Karpathy, Justin Johns

Training Convnets: Problems and Solutions

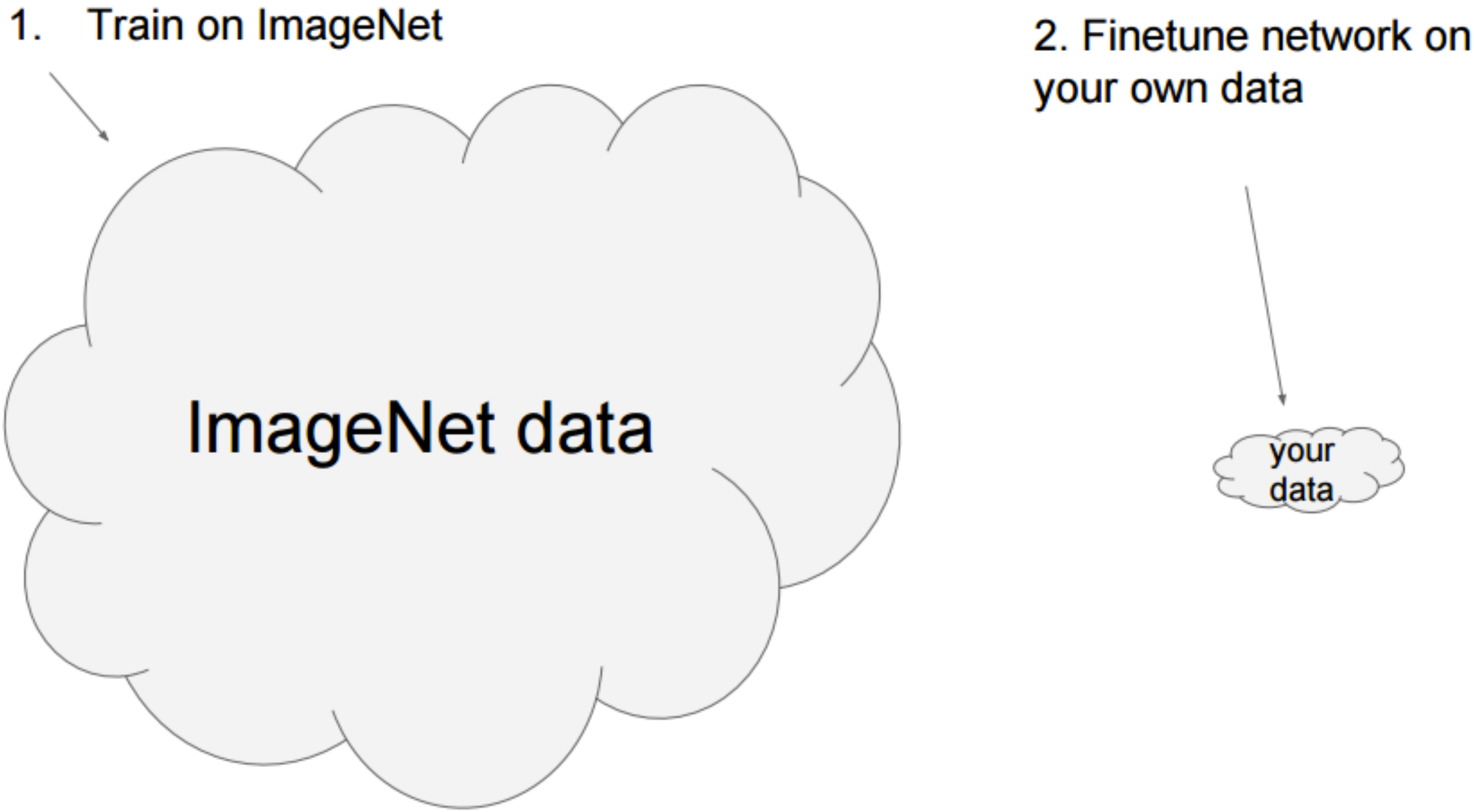
First Lesson: Transfer Learning

“ConvNets need a lot of data to train”

MYTH

finetuning! we rarely ever train ConvNets from scratch.

Transfer Learning



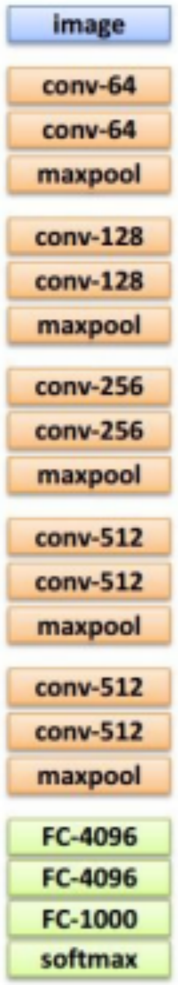
1. Train on ImageNet

ImageNet data

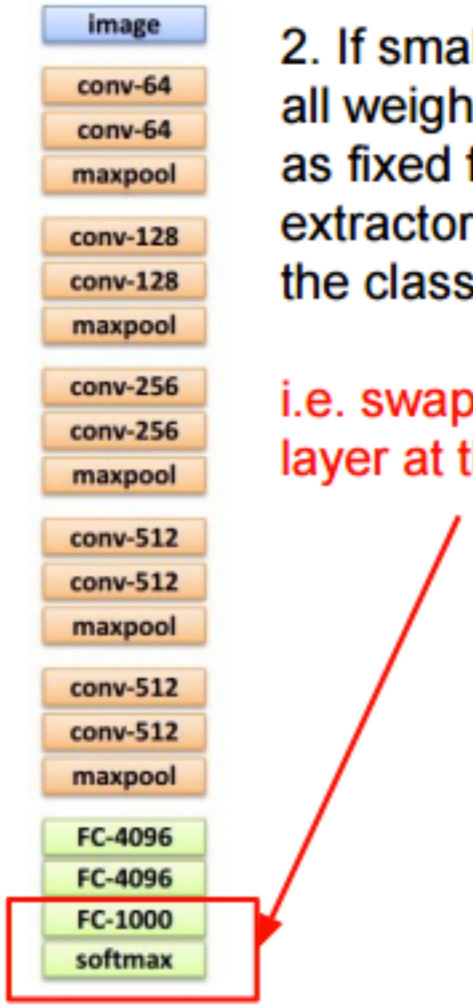
2. Finetune network on your own data

your data

Transfer Learning

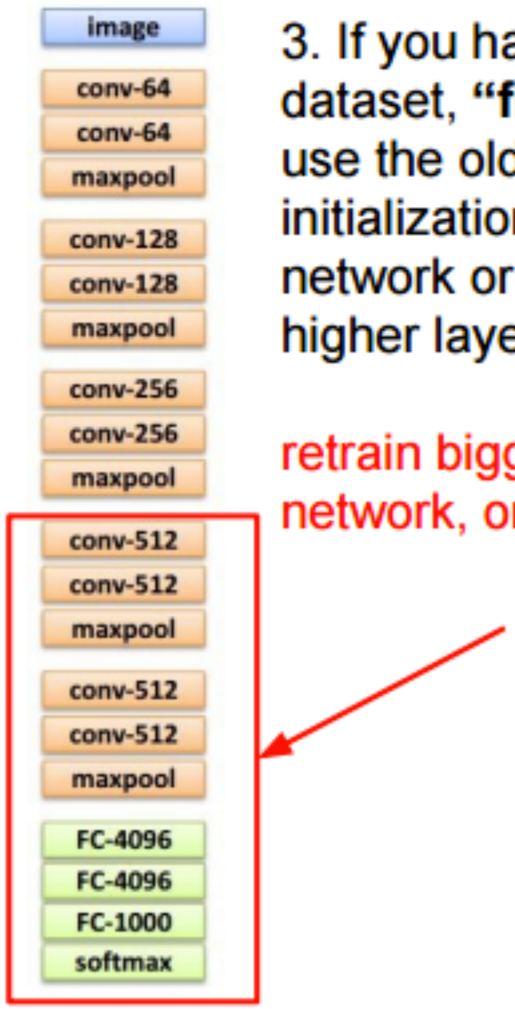


1. Train on ImageNet



2. If small dataset: fix all weights (treat CNN as fixed feature extractor), retrain only the classifier

i.e. swap the Softmax layer at the end

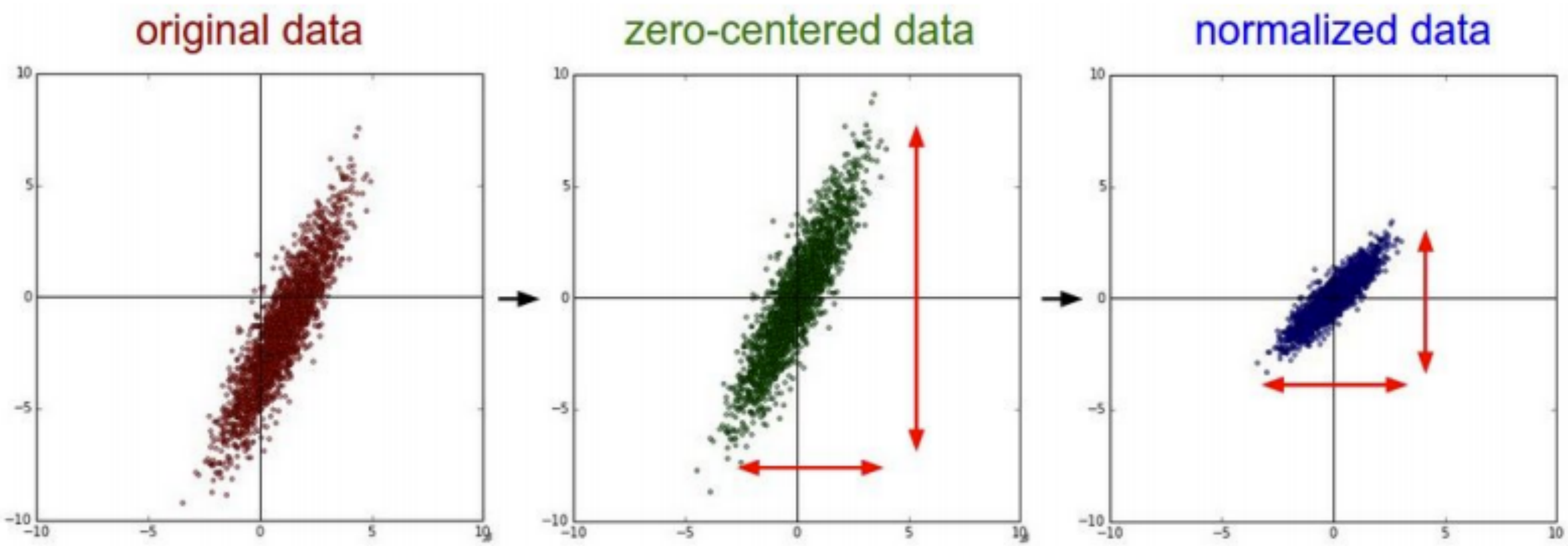


3. If you have medium sized dataset, **“finetune”** instead: use the old weights as initialization, train the full network or only some of the higher layers

retrain bigger portion of the network, or even all of it.

Data Preprocessing

Zero-Center & Normalize Data

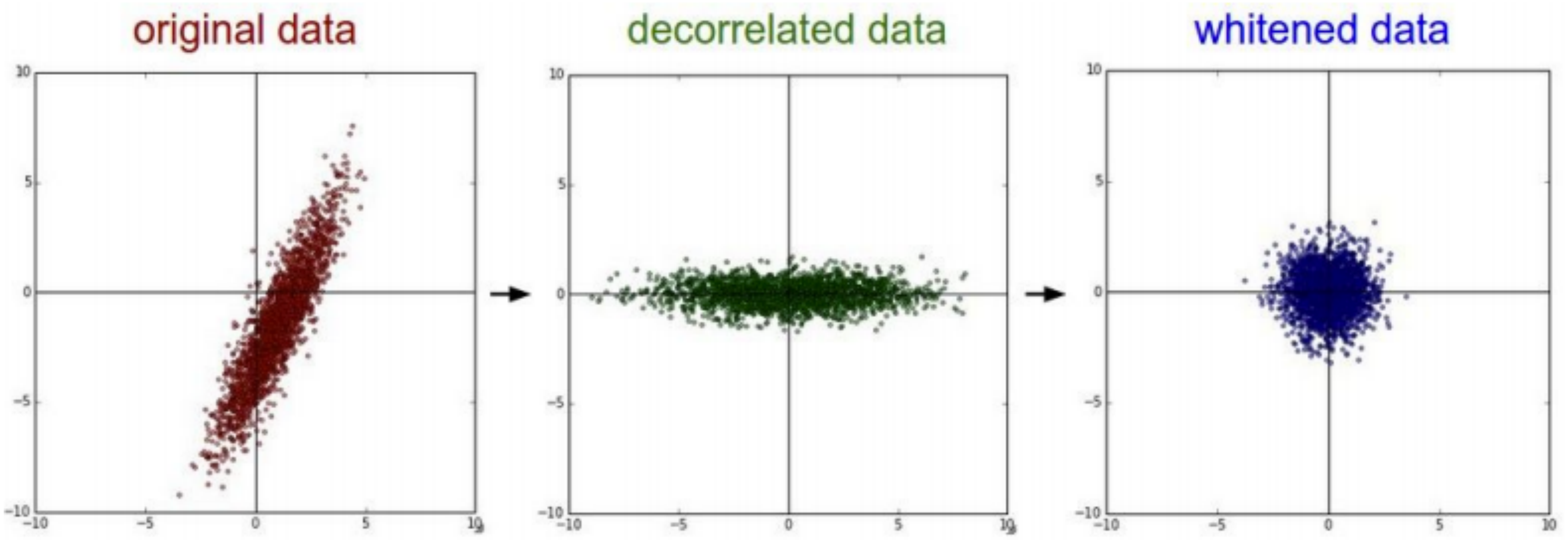


```
X -= np.mean(X, axis = 0)
```

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

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

PCA & Whitening



(data has diagonal covariance matrix)

(covariance matrix is the identity matrix)

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

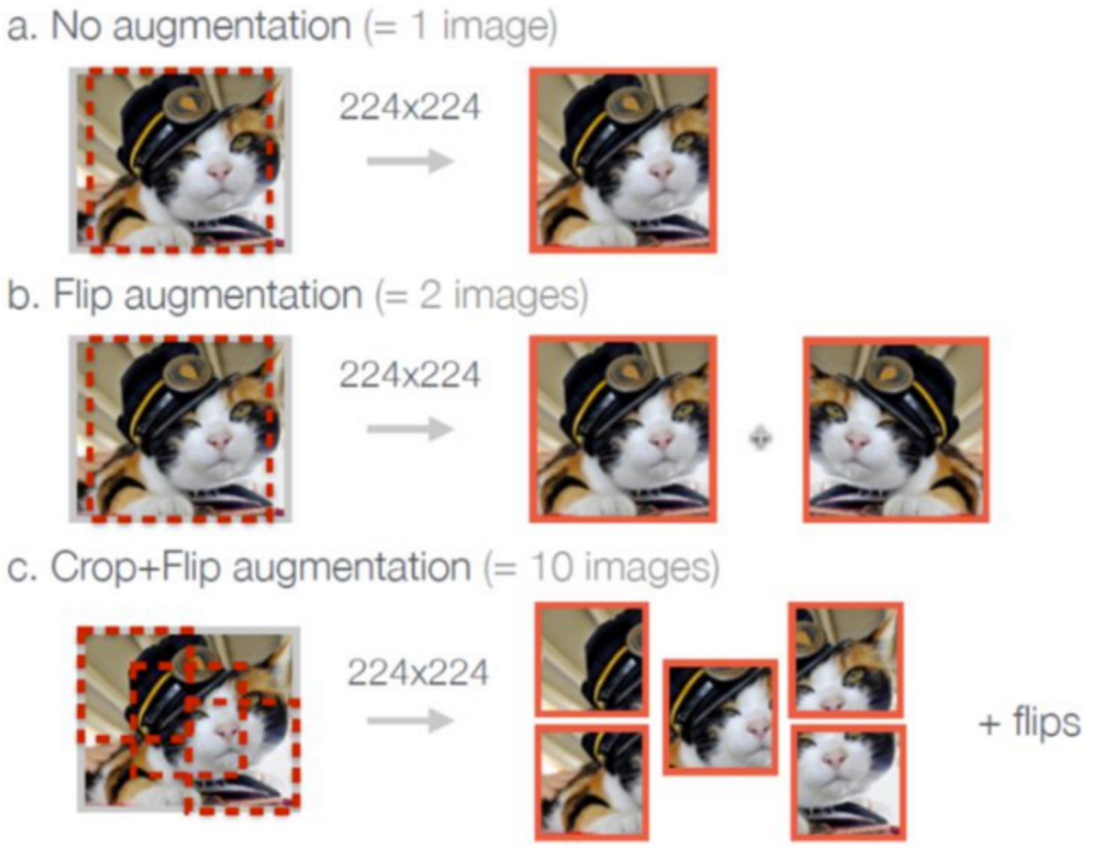
Data Augmentation

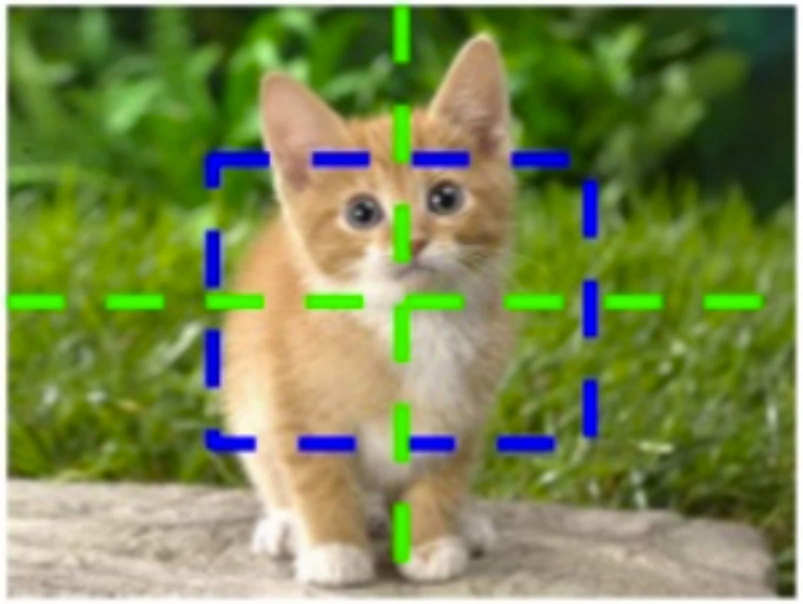
During training:

- Random crops on the original image
- Horizontal reflections

During testing:

- Average prediction of image augmented by the four corner patches and the center patch + flipped image (10 augmentations of the image


a. No augmentation (= 1 image)
224x224
b. Flip augmentation (= 2 images)
224x224
+
c. Crop+Flip augmentation (= 10 images)
224x224
+ flips



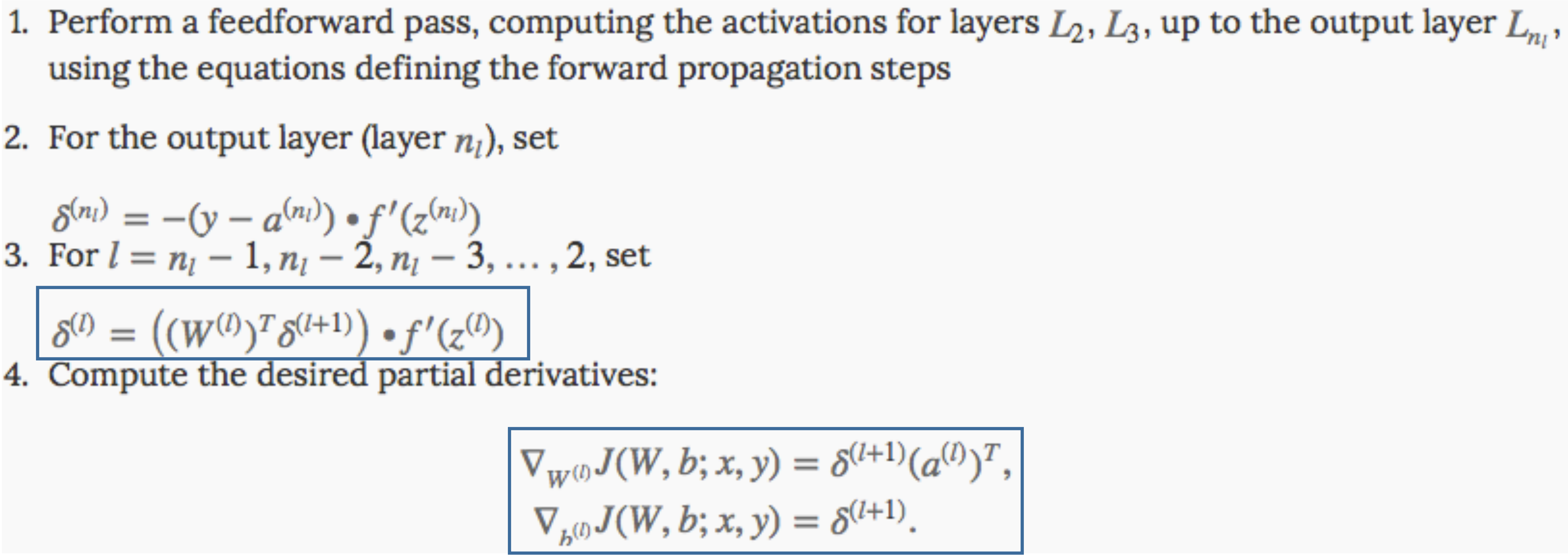
Data augmentation reduces overfitting

Weight Initialization

Interesting Question:

What happens when the weights are initialized to 0? (2 min)

Answer: The gradients in the backward pass will become zero!



1. Perform a feedforward pass, computing the activations for layers L_2, L_3, up to the output layer L_{n_l}, using the equations defining the forward propagation steps

2. For the output layer (layer n_l), set

$$\delta^{(n_l)} = -(y - a^{(n_l)}) \bullet f'(z^{(n_l)})$$

3. For $l = n_l - 1, n_l - 2, n_l - 3, \ldots, 2$, set

$$\boxed{\delta^{(l)} = \left((W^{(l)})^T \delta^{(l+1)}\right) \bullet f'(z^{(l)})}$$

4. Compute the desired partial derivatives:

$$\boxed{\begin{aligned} \nabla_{W^{(l)}} J(W,b;x,y) &= \delta^{(l+1)} (a^{(l)})^T, \\ \nabla_{b^{(l)}} J(W,b;x,y) &= \delta^{(l+1)}. \end{aligned}}$$

Random Initialization

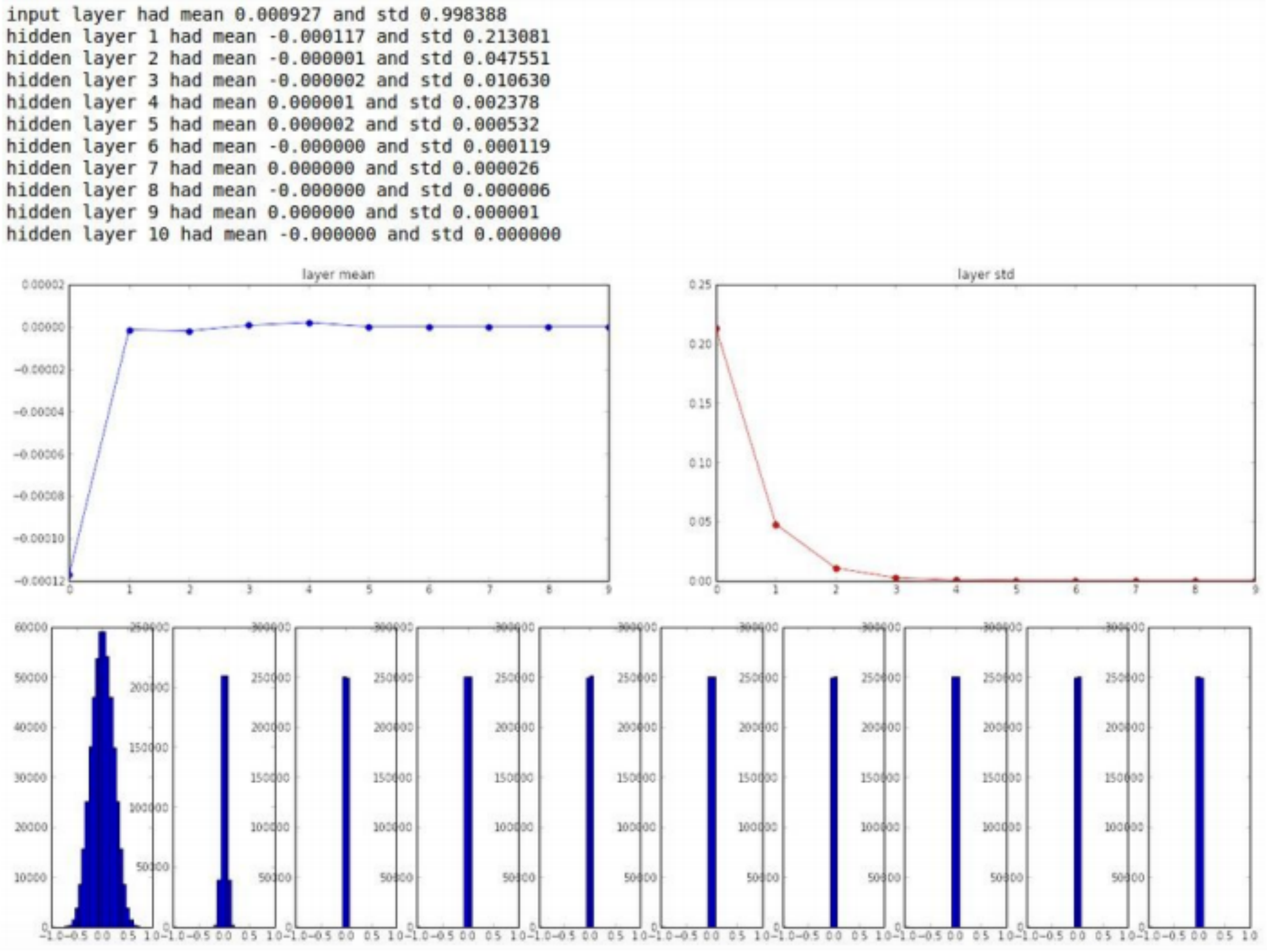
```
W = 0.01 * np.random.randn(D, H)
```

Works fine for small networks, but can lead to non-homogeneous distributions of activations across the layers of a network.

Look at Some Activation Statistics

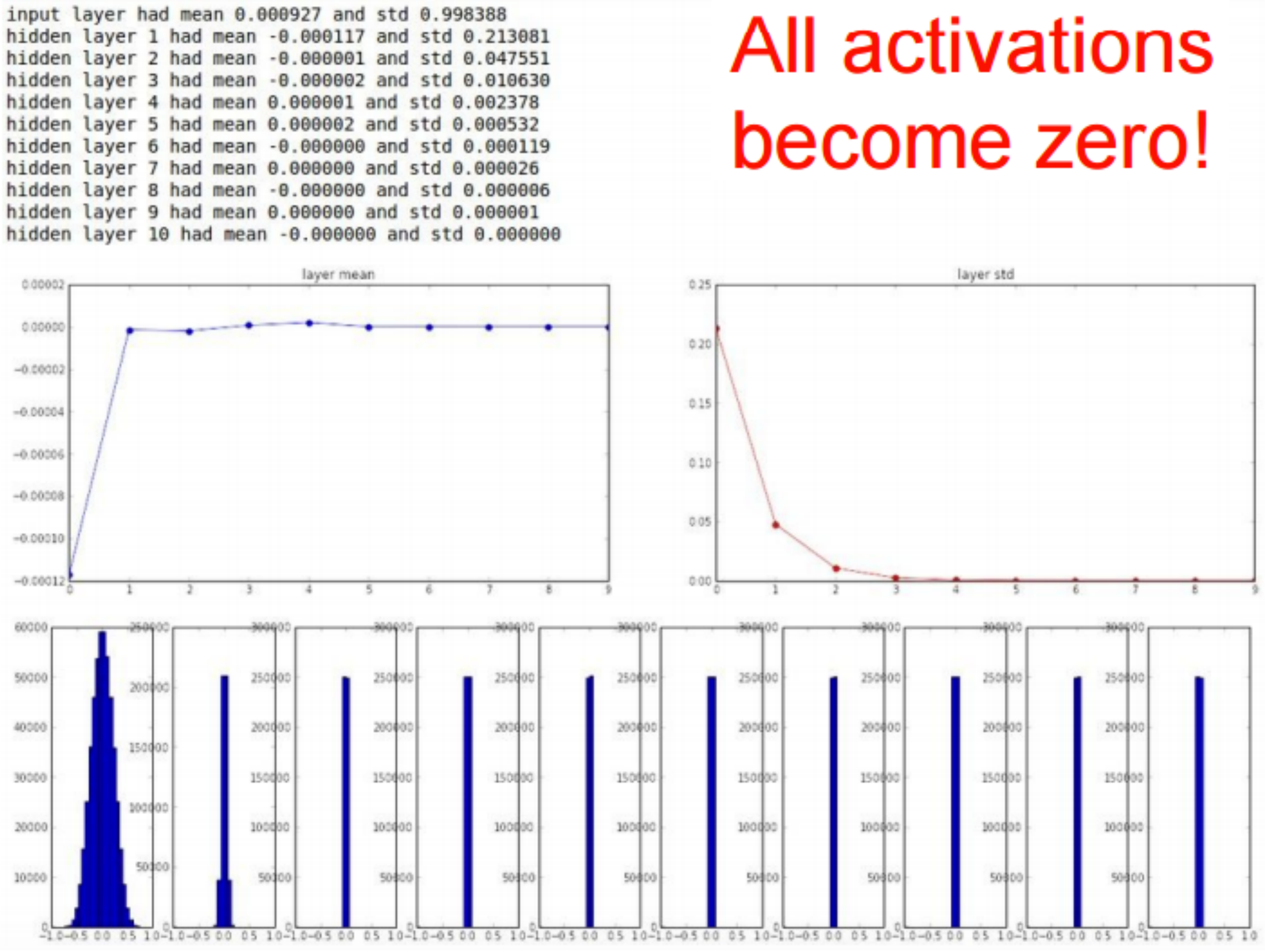
Setup: 10-layer net with 500 neurons on each layer, using tanh nonlinearities, and initializing as described in last slide.

Random Initialization



```
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
```

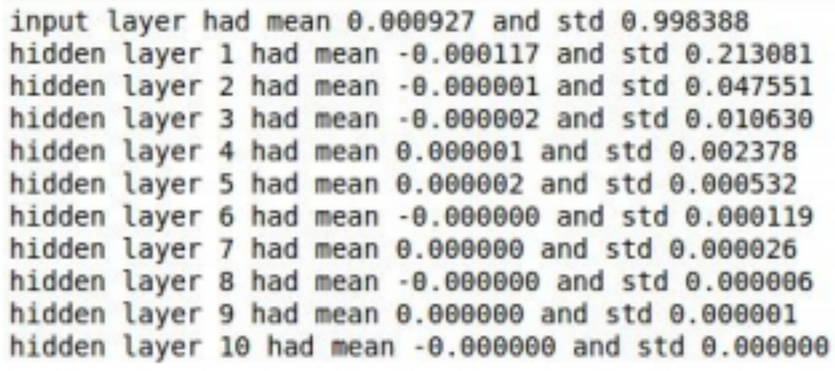
Random Initialization



```
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
```

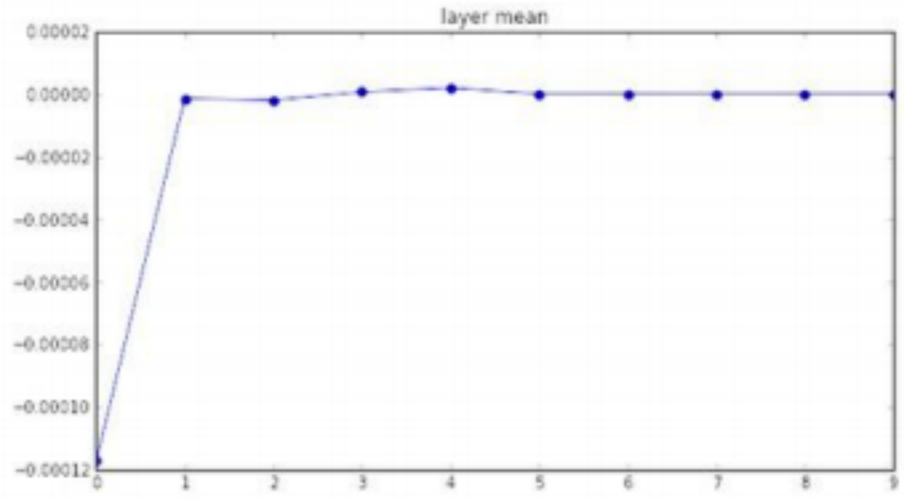
All activations become zero!

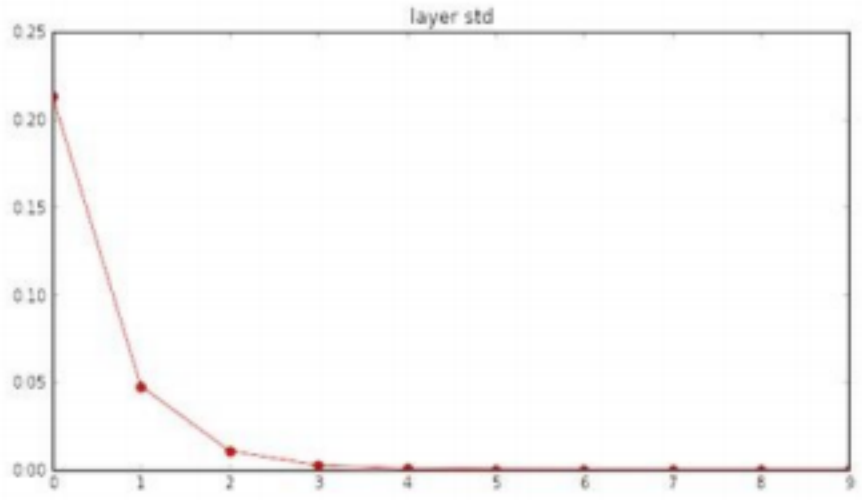
Random Initialization

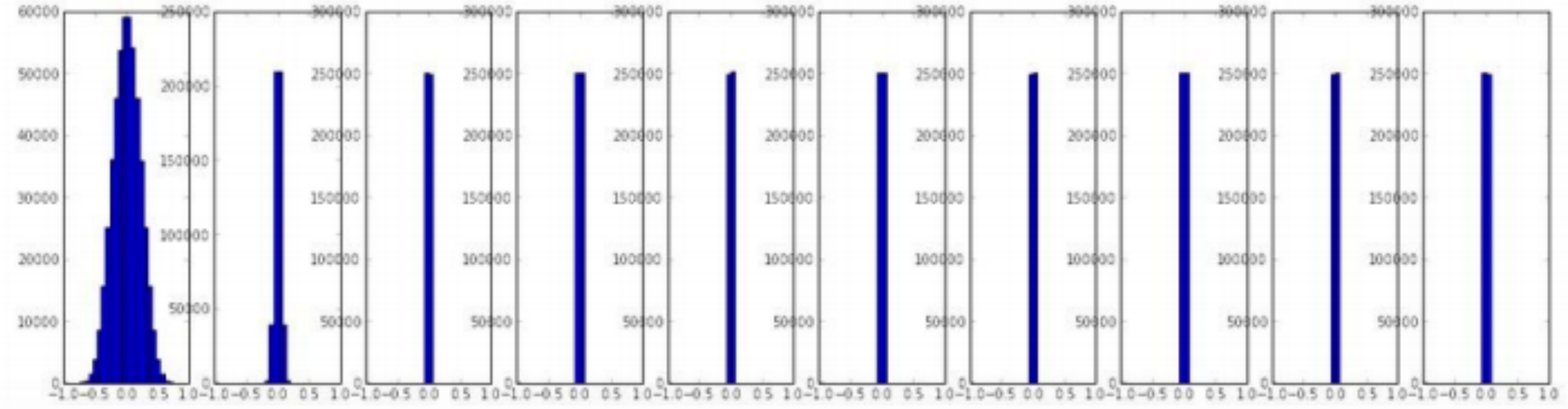


```
input layer had mean 0.000927 and std 0.998388
hidden layer 1 had mean -0.000117 and std 0.213081
hidden layer 2 had mean -0.000001 and std 0.047551
hidden layer 3 had mean -0.000002 and std 0.010630
hidden layer 4 had mean 0.000001 and std 0.002378
hidden layer 5 had mean 0.000002 and std 0.000532
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
```

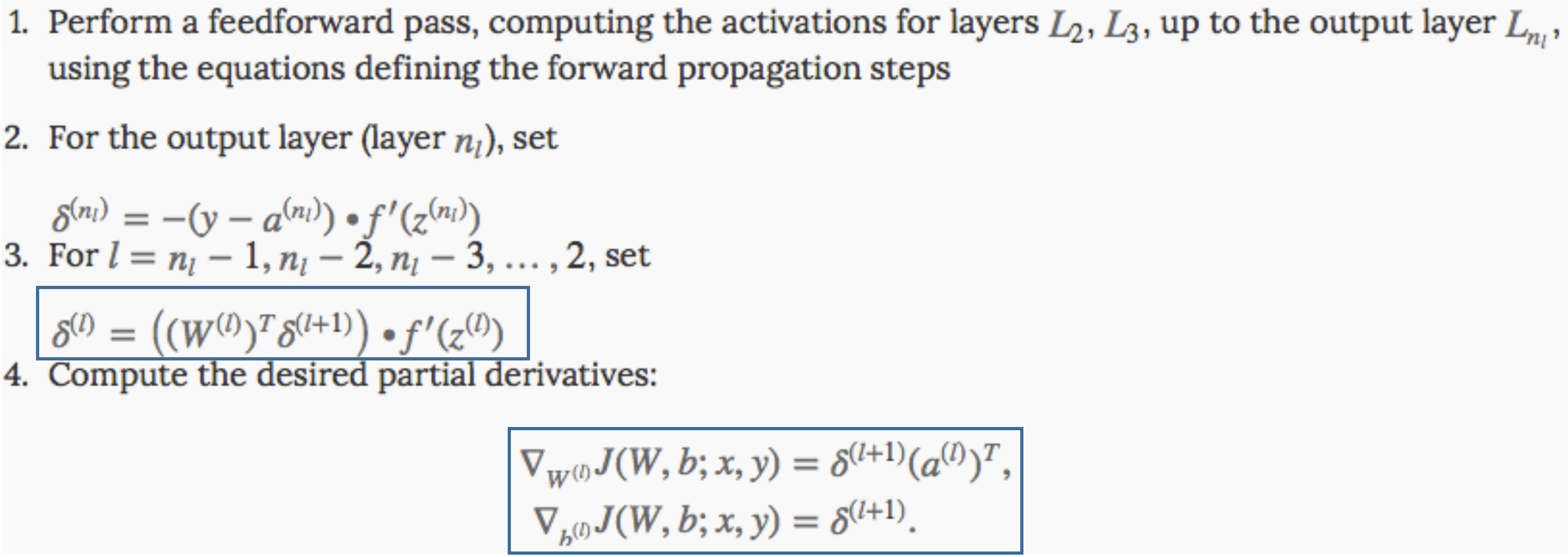
Interesting Question: What will the gradients look like in the backward pass when all activations become zero?







Answer: The gradients in the backward pass will become zero!



1. Perform a feedforward pass, computing the activations for layers L_2, L_3, up to the output layer L_{n_l}, using the equations defining the forward propagation steps

2. For the output layer (layer n_l), set

$$\delta^{(n_l)} = -(y - a^{(n_l)}) \bullet f'(z^{(n_l)})$$

3. For $l = n_l - 1, n_l - 2, n_l - 3, \ldots, 2$, set

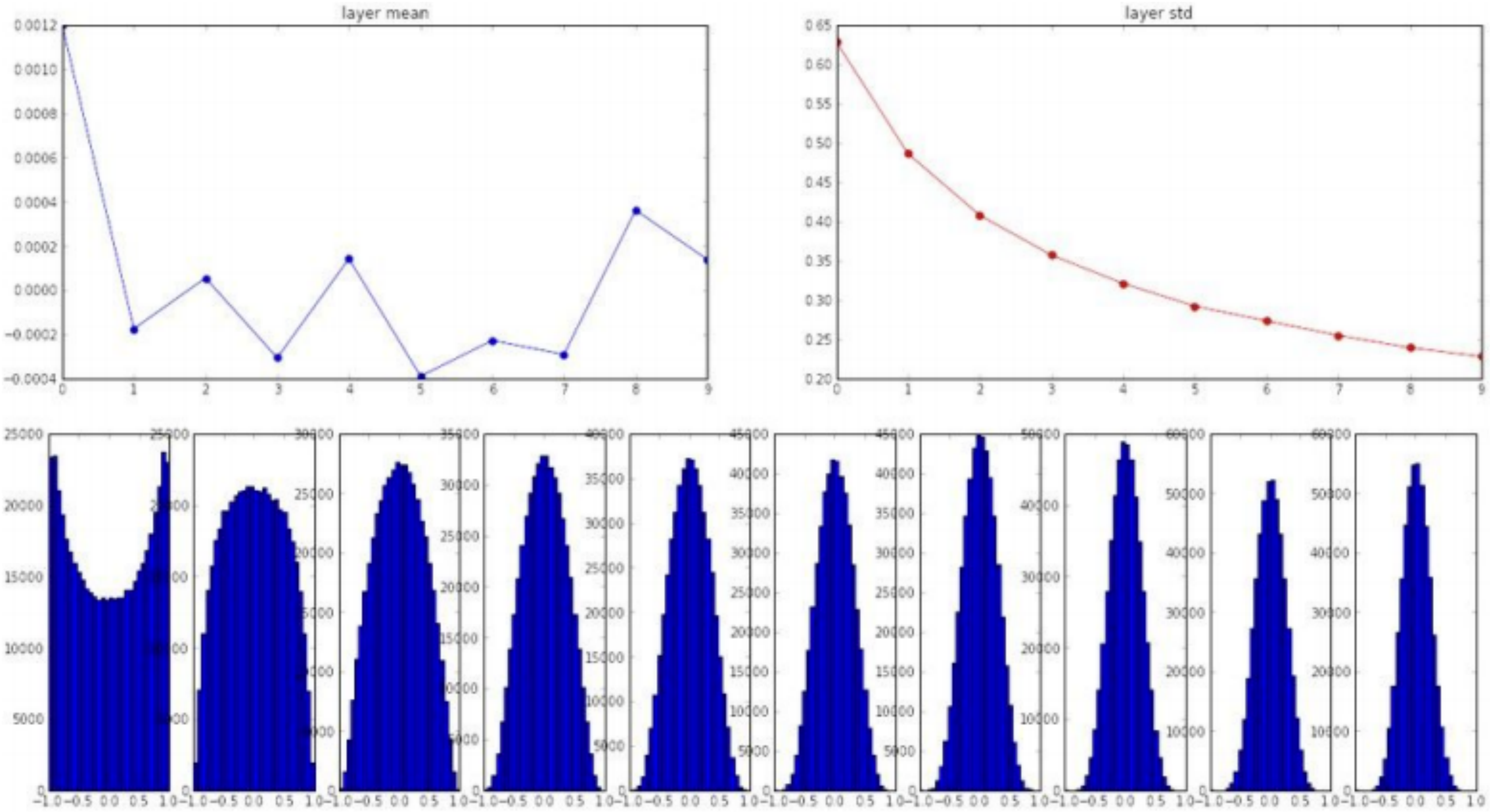
$$\boxed{\delta^{(l)} = \left((W^{(l)})^T \delta^{(l+1)}\right) \bullet f'(z^{(l)})}$$

4. Compute the desired partial derivatives:

$$\boxed{\begin{aligned} \nabla_{W^{(l)}} J(W, b; x, y) &= \delta^{(l+1)} (a^{(l)})^T, \\ \nabla_{b^{(l)}} J(W, b; x, y) &= \delta^{(l+1)}. \end{aligned}}$$

Xavier Initialization

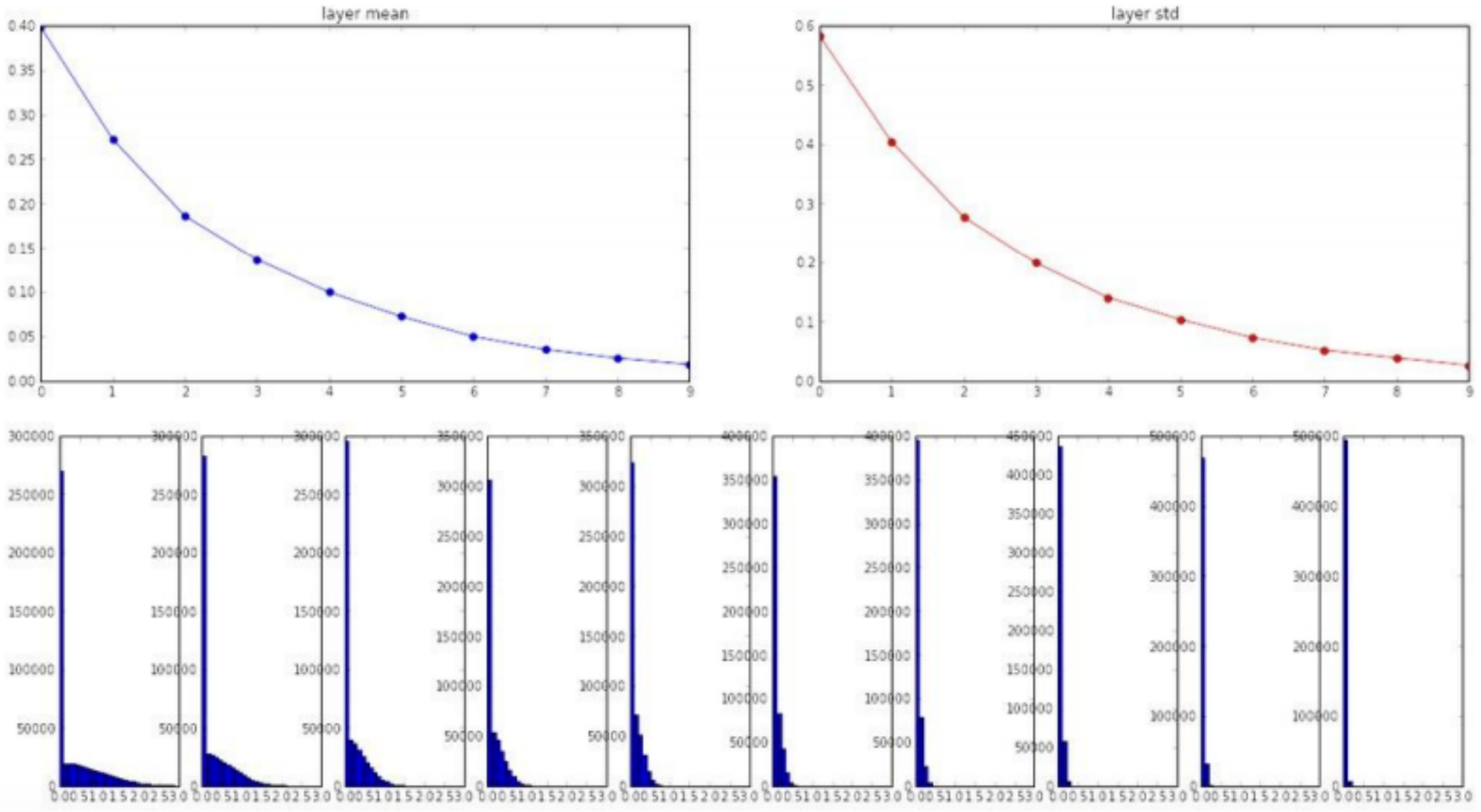
```
W = np.random.randn(fan_in, fan_out) / np.sqrt(fan_in)
```



Reasonable initialization
(Mathematical derivation assumes linear activations)

Xavier Initialization

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



but it breaks when using ReLU non-linearity

More Initialization Techniques

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

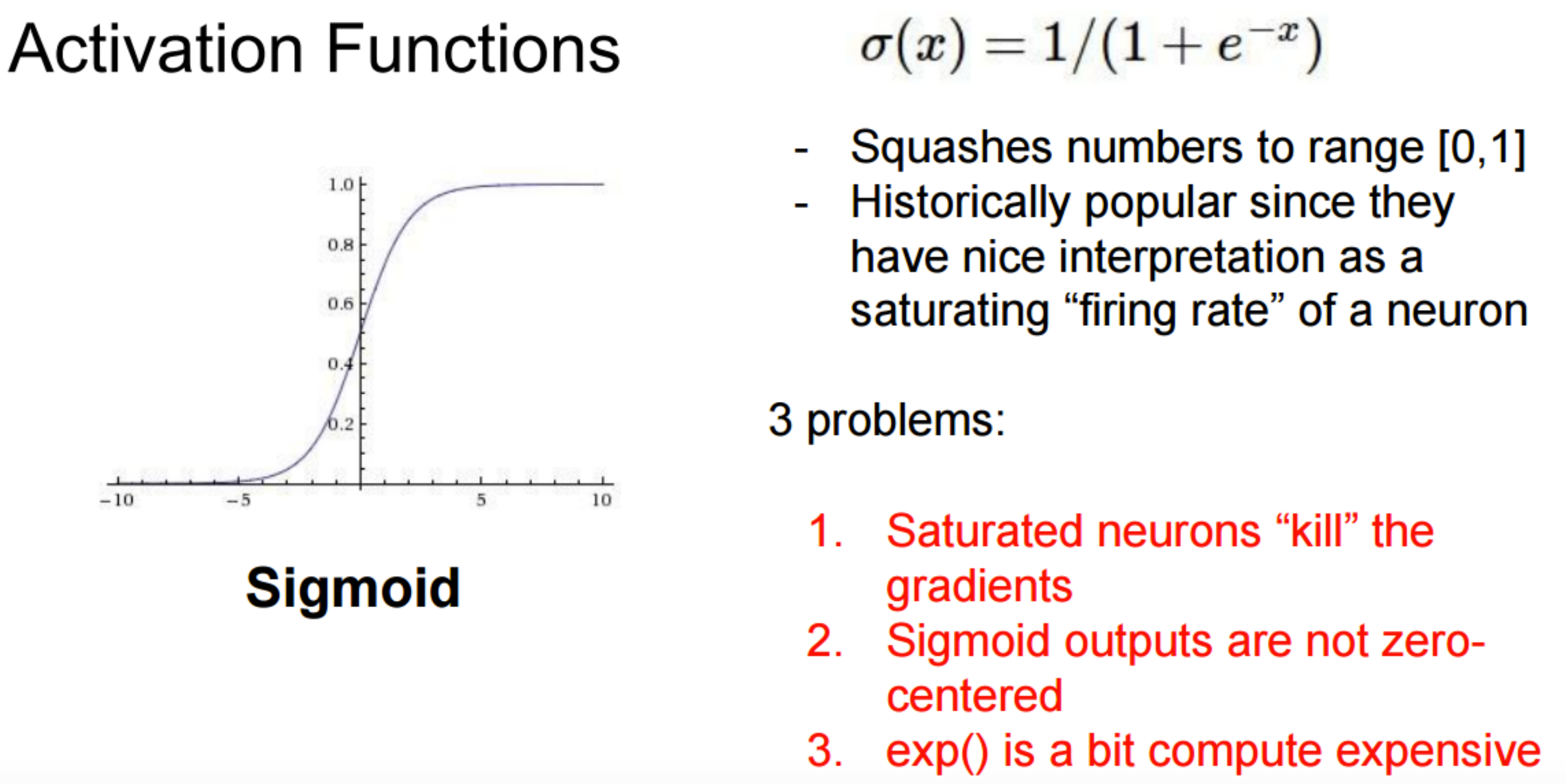
Delving deep into rectifiers: Surpassing human-level performance on ImageNet classification
by He et al., 2015

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

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

Choosing an Activation Function that Helps the Training

Sigmoid



Activation Functions

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

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

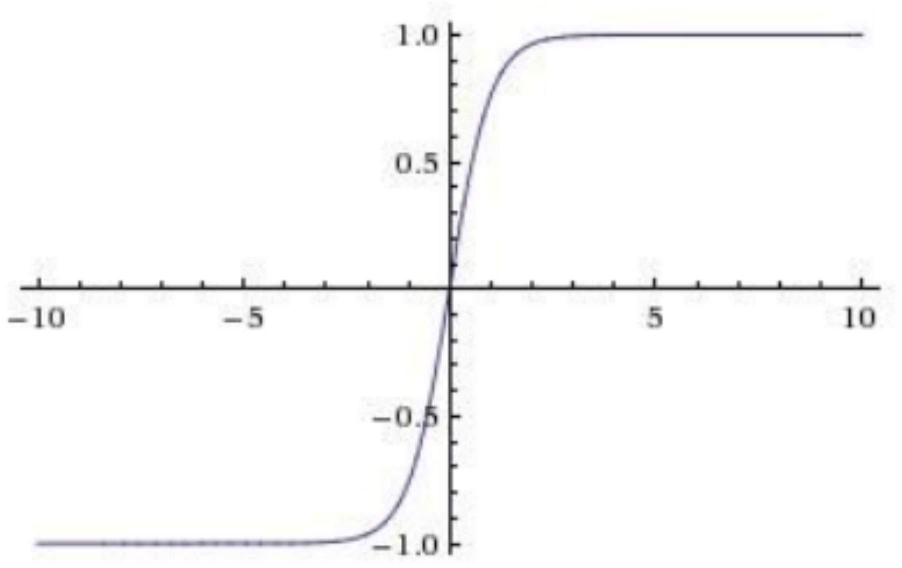
Sigmoid

3 problems:

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

Tanh

Activation Functions

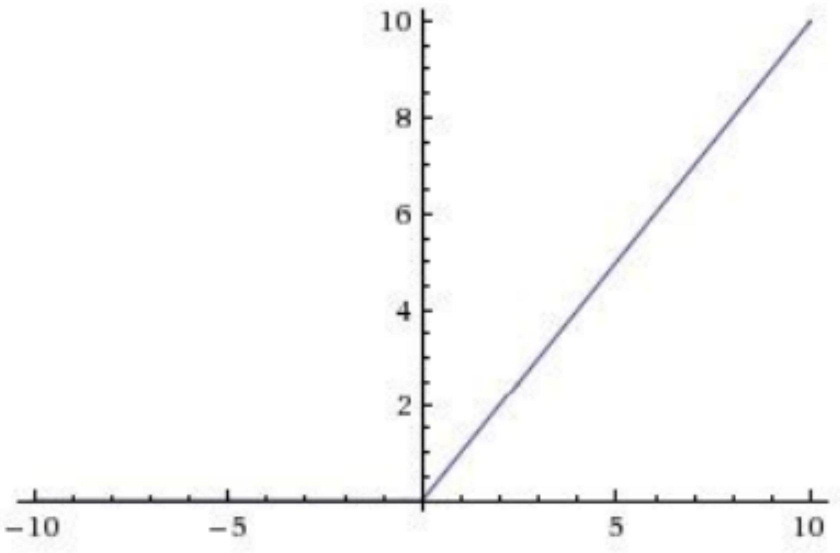


tanh(x)

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

ReLU

Activation Functions



ReLU

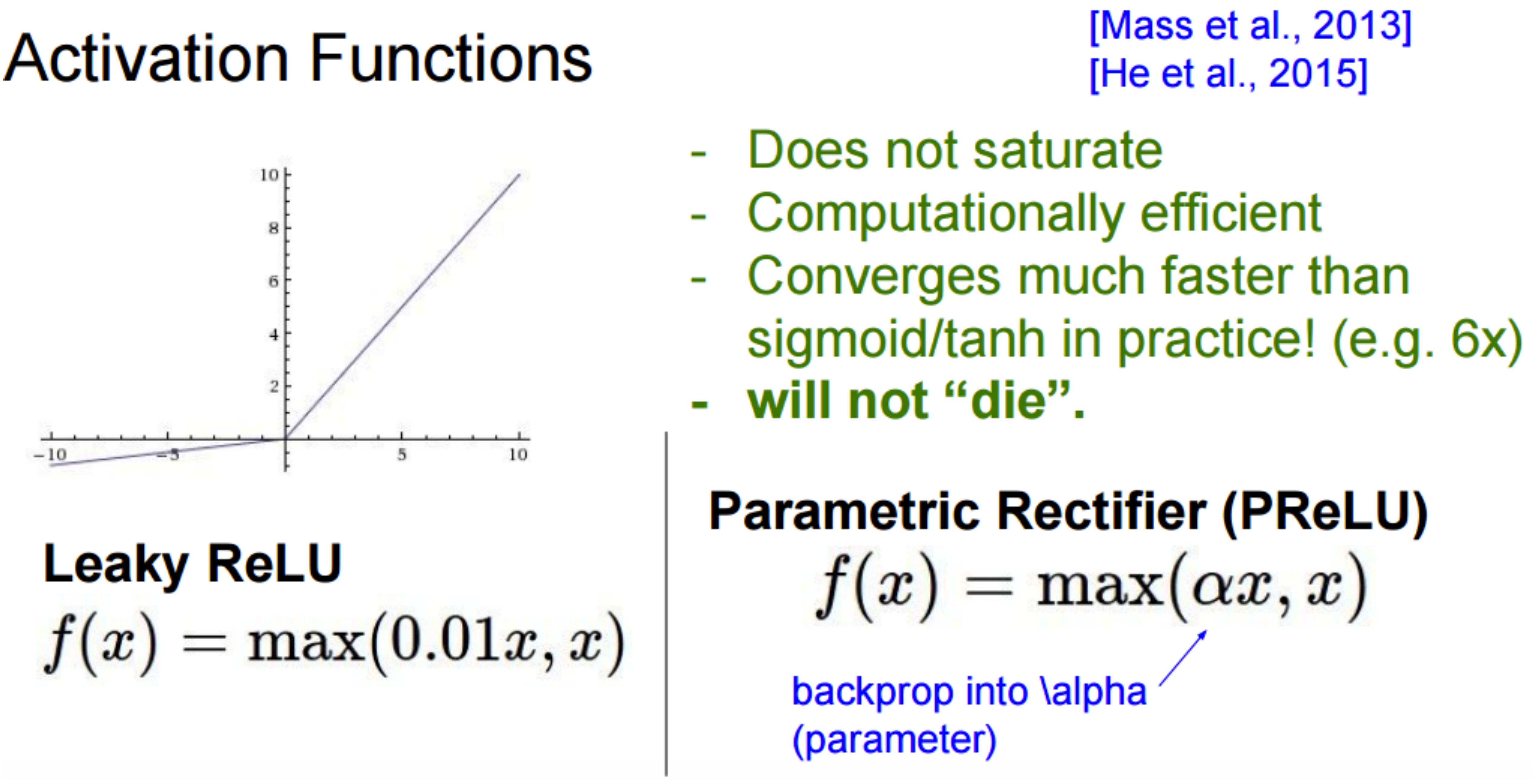
- Computes **f(x) = max(0,x)**

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

- Not zero-centered output
- An annoyance:

“dead” in -region

Leaky ReLU



Activation Functions

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

10
8
6
4
2
−10
−5
5
10

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

Leaky ReLU

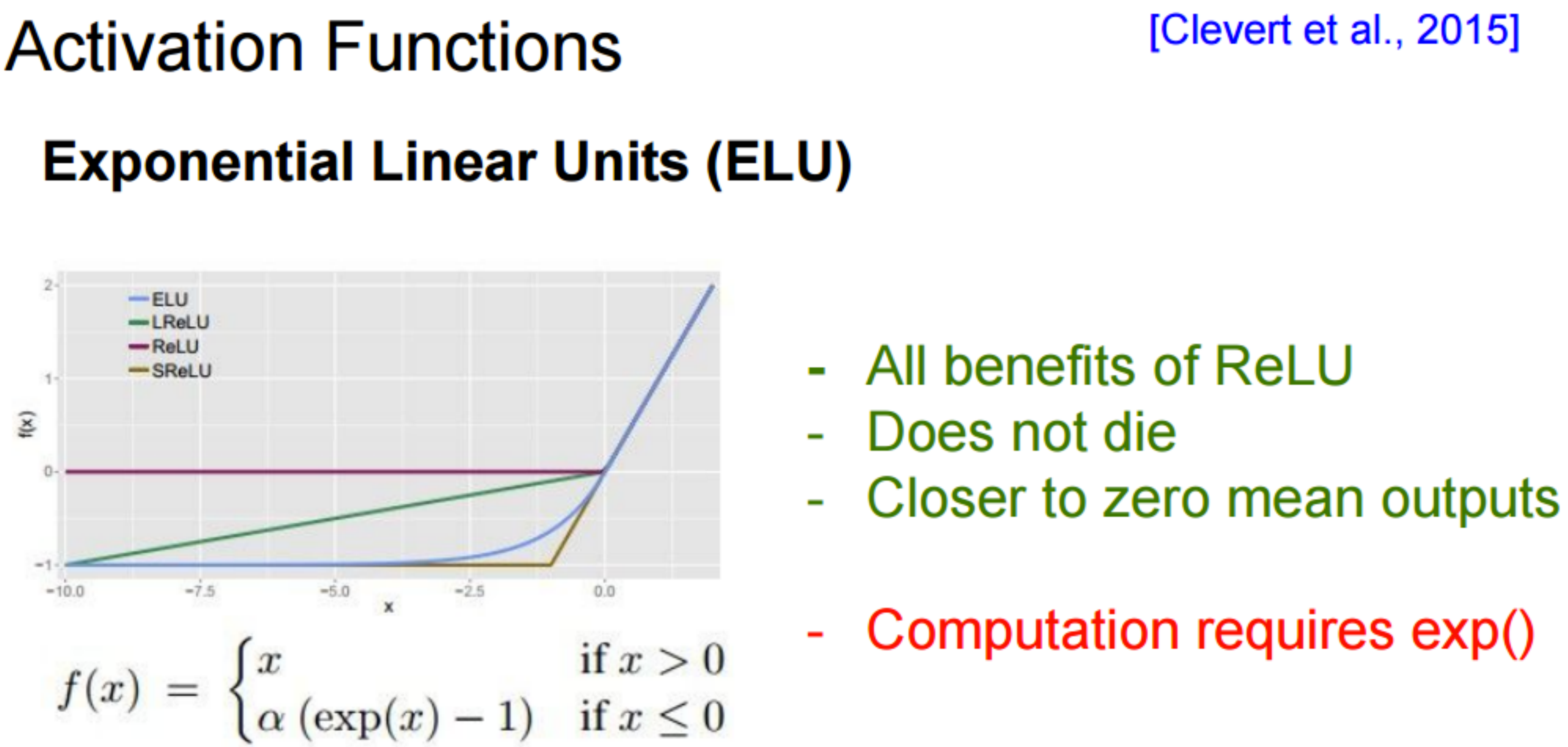
$$f(x) = \max(0.01x, x)$$

Parametric Rectifier (PReLU)

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

backprop into \alpha
(parameter)

Exponential Linear Unit



Activation Functions

[Clevert et al., 2015]

Exponential Linear Units (ELU)

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

- All benefits of ReLU
- Does not die
- Closer to zero mean outputs

- Computation requires exp()

Maxout

[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 :(

In Practice

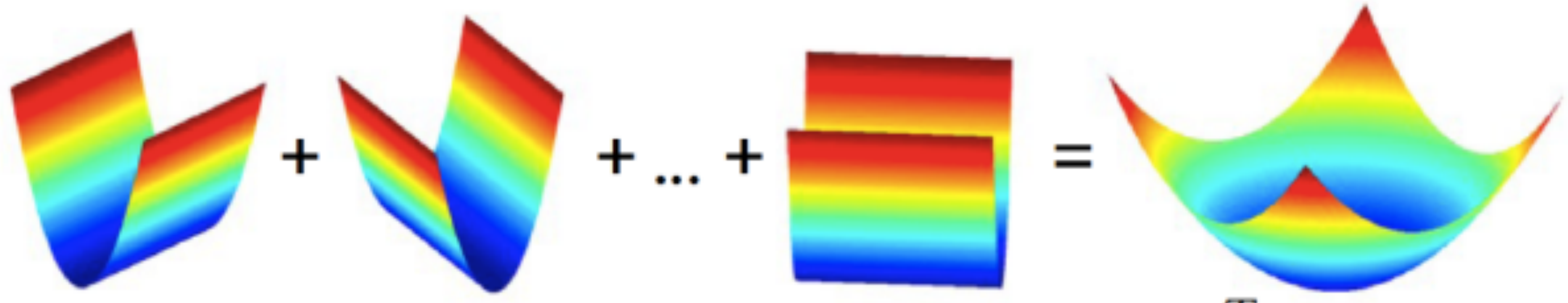
- Use ReLU. Be careful with your learning rates
- Try out Leaky ReLU / Maxout / ELU
- Try out tanh but don’t expect much
- Don’t use sigmoid

Training Algorithms

Stochastic Gradient Descent

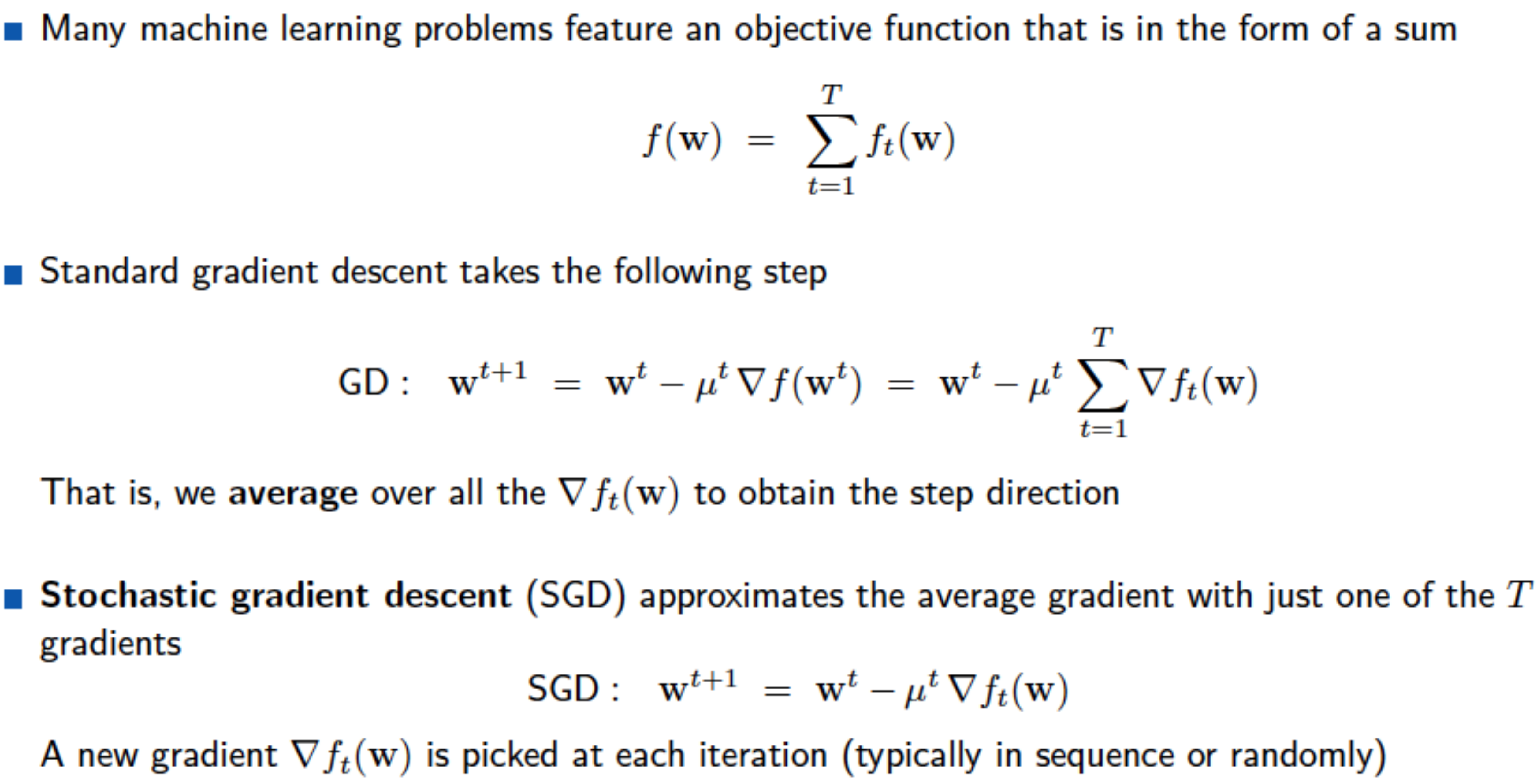
- Many machine learning problems feature an objective function that is in the form of a sum

$$f(\mathbf{w}) = \sum_{t=1}^{T} f_t(\mathbf{w})$$



$$f_1(\mathbf{w}) + f_2(\mathbf{w}) + \ldots + f_T(\mathbf{w}) = \sum_{t=1}^{T} f_t(\mathbf{w})$$

Stochastic Gradient Descent



- Many machine learning problems feature an objective function that is in the form of a sum

$$f(\mathbf{w}) \; = \; \sum_{t=1}^{T} f_t(\mathbf{w})$$

- Standard gradient descent takes the following step

$$\text{GD}: \quad \mathbf{w}^{t+1} \; = \; \mathbf{w}^t - \mu^t \nabla f(\mathbf{w}^t) \; = \; \mathbf{w}^t - \mu^t \sum_{t=1}^{T} \nabla f_t(\mathbf{w})$$

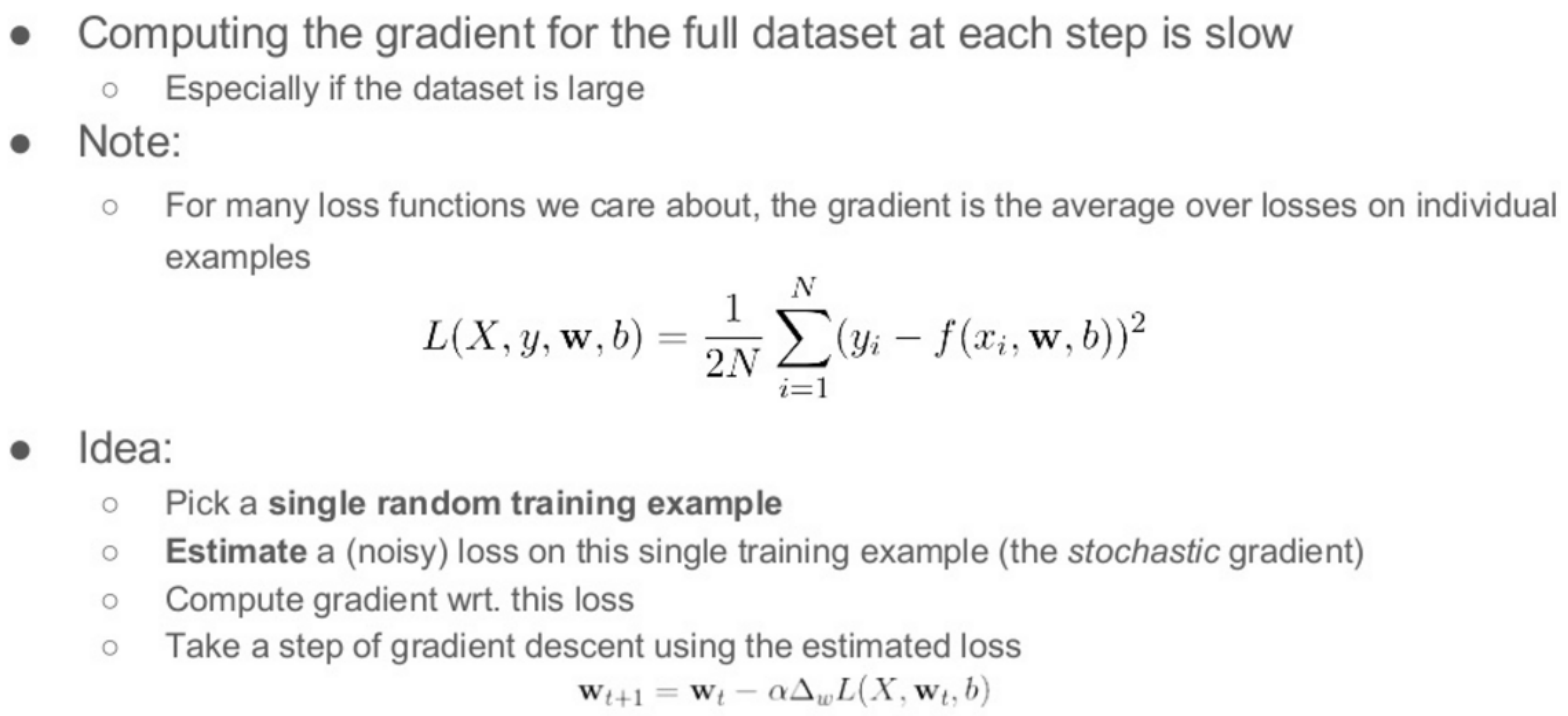
  That is, we **average** over all the $\nabla f_t(\mathrm{w})$ to obtain the step direction

- **Stochastic gradient descent** (SGD) approximates the average gradient with just one of the T gradients

$$\text{SGD}: \quad \mathbf{w}^{t+1} \; = \; \mathbf{w}^t - \mu^t \nabla f_t(\mathbf{w})$$

  A new gradient $\nabla f_t(\mathrm{w})$ is picked at each iteration (typically in sequence or randomly)

Stochastic Gradient Descent for Neural Networks



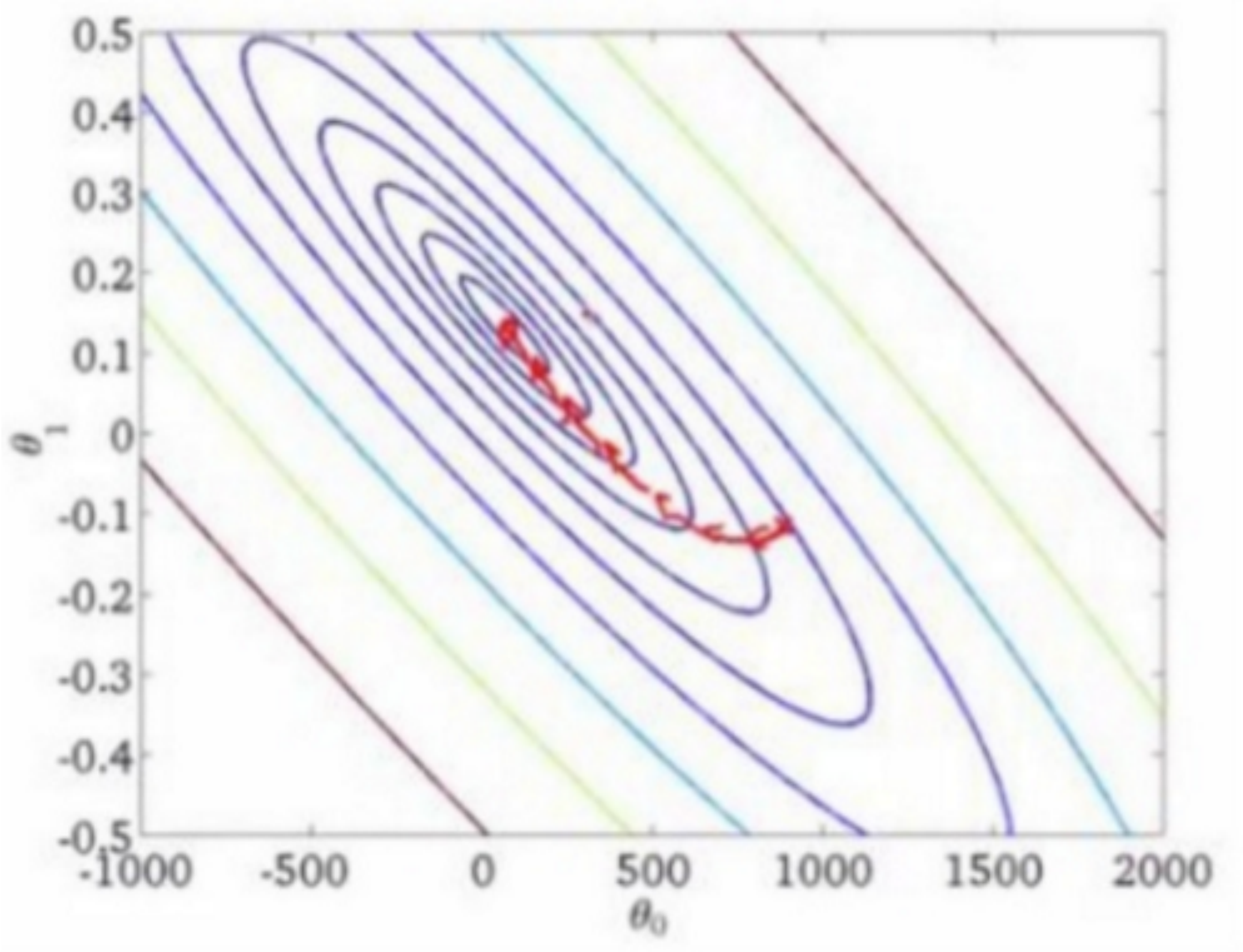
- Computing the gradient for the full dataset at each step is slow
  - Especially if the dataset is large
- Note:
  - For many loss functions we care about, the gradient is the average over losses on individual examples

$$L(X, y, \mathbf{w}, b) = \frac{1}{2N} \sum_{i=1}^{N} (y_i - f(x_i, \mathbf{w}, b))^2$$

- Idea:
  - Pick a **single random training example**
  - **Estimate** a (noisy) loss on this single training example (the *stochastic* gradient)
  - Compute gradient wrt. this loss
  - Take a step of gradient descent using the estimated loss

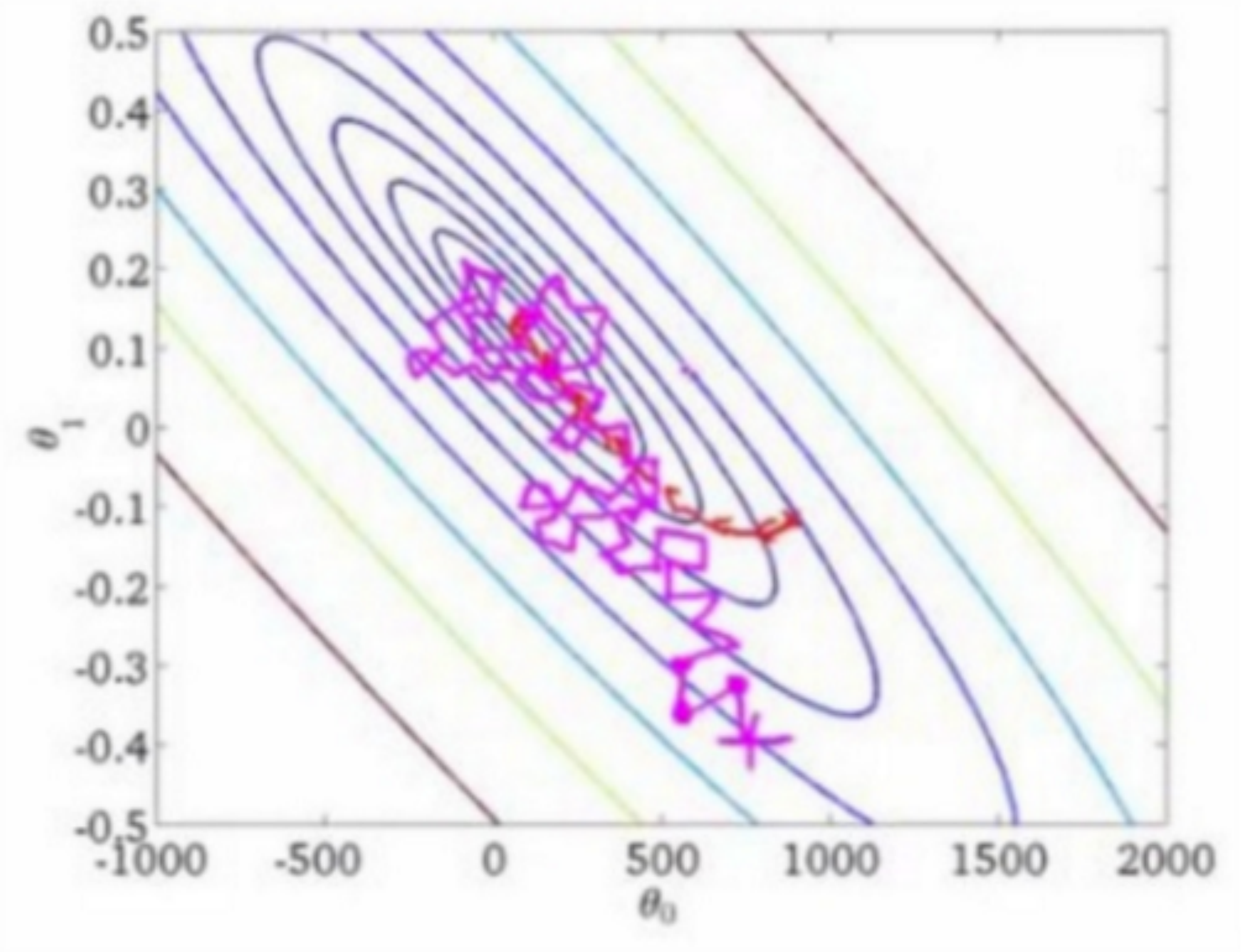
$$\mathbf{w}_{t+1} = \mathbf{w}_t - \alpha \Delta_w L(X, \mathbf{w}_t, b)$$

Batch GD vs Stochastic GD



Batch: gradient

$$x \leftarrow x - \eta \nabla F(x)$$



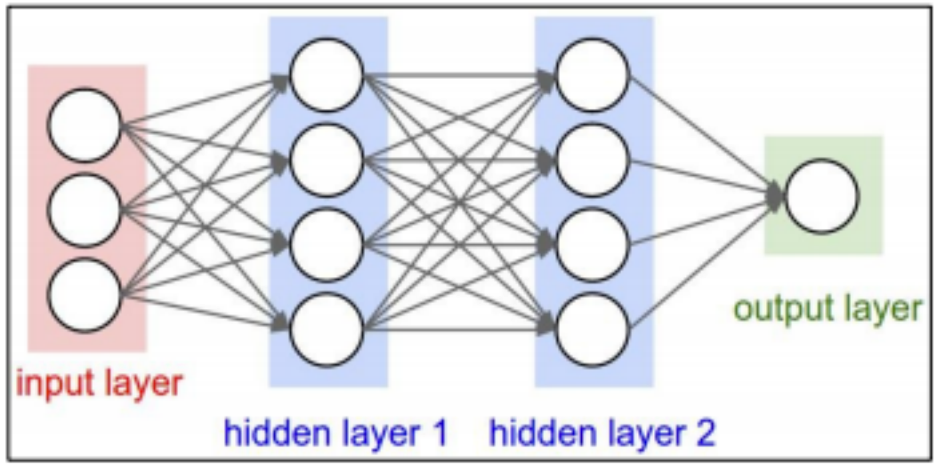
Stochastic: single-example gradient

$$x \leftarrow x - \eta \nabla F_i(x)$$

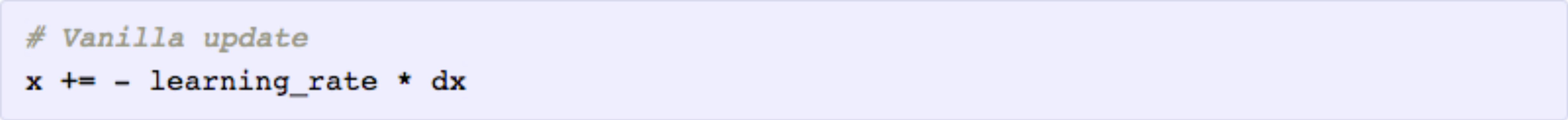
Mini-batch SGD

Loop:

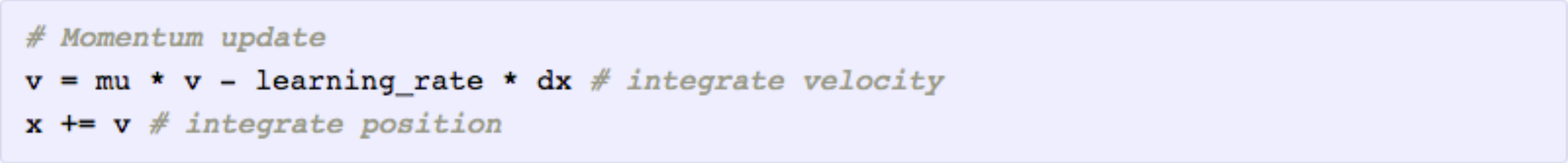
1. **Sample** a batch of data
2. **Forward** prop it through the graph, get loss
3. **Backprop** to calculate the gradients
4. **Update** the parameters using the gradient



Momentum Update



```
# Vanilla update
x += - learning_rate * dx
```

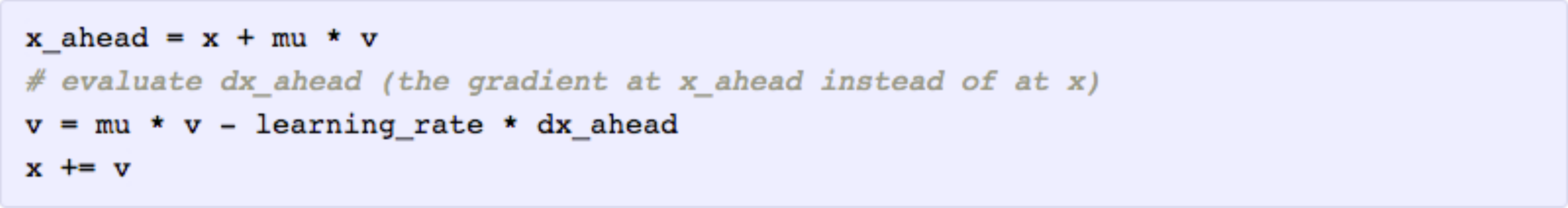


```
# Momentum update
v = mu * v - learning_rate * dx # integrate velocity
x += v # integrate position
```

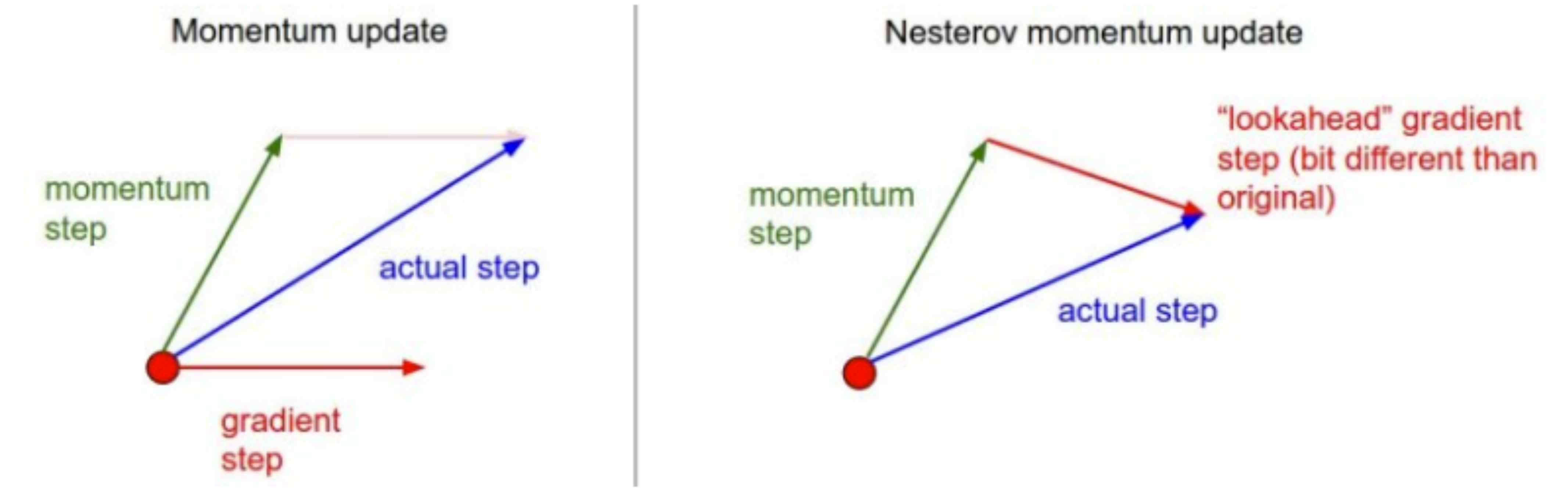
- Physical interpretation as ball rolling down the loss function + friction (mu coefficient).
- mu = usually ~0.5, 0.9, or 0.99 (Sometimes annealed over time, e.g. from 0.5 -> 0.99)

- Allows a velocity to “build up” along shallow directions
- Velocity becomes damped in steep direction due to quickly changing sign

Nesterov Momentum Update

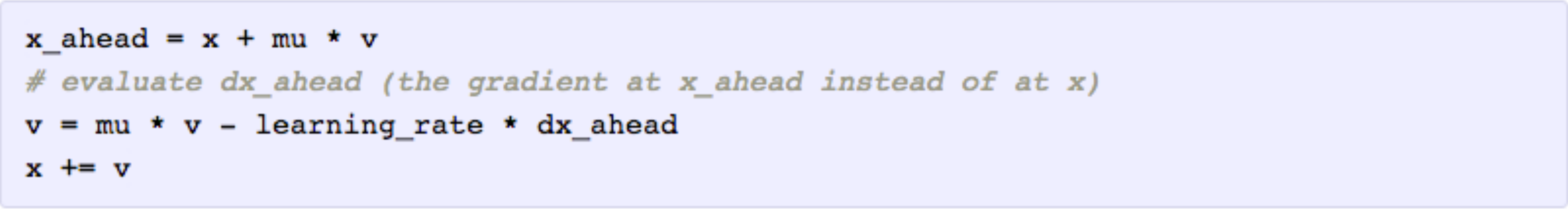


```
x_ahead = x + mu * v
# evaluate dx_ahead (the gradient at x_ahead instead of at x)
v = mu * v - learning_rate * dx_ahead
x += v
```

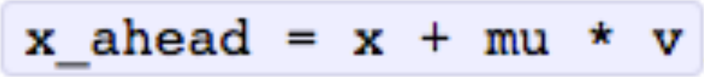


Nesterov momentum. Instead of evaluating gradient at the current position (red circle), we know that our momentum is about to carry us to the tip of the green arrow. With Nesterov momentum we therefore instead evaluate the gradient at this "looked-ahead" position.

Nesterov Momentum Update

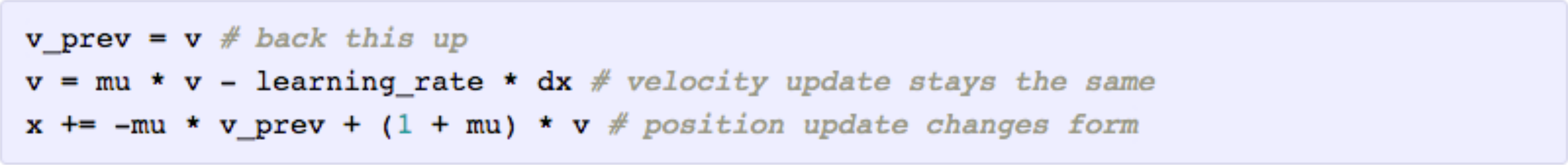


```
x_ahead = x + mu * v
# evaluate dx_ahead (the gradient at x_ahead instead of at x)
v = mu * v - learning_rate * dx_ahead
x += v
```



```
x_ahead = x + mu * v
```

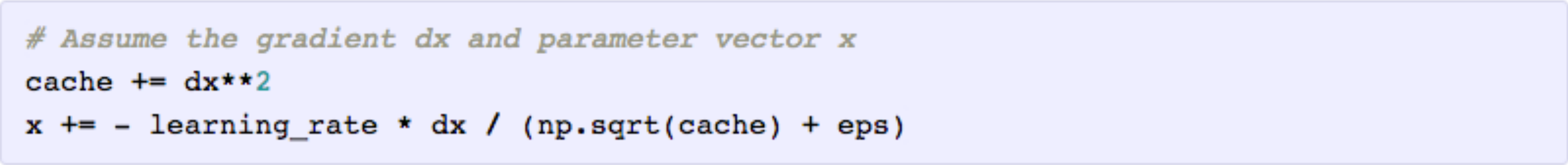
express the update in term
of x_ahead, instead of x



```
v_prev = v # back this up
v = mu * v - learning_rate * dx # velocity update stays the same
x += -mu * v_prev + (1 + mu) * v # position update changes form
```

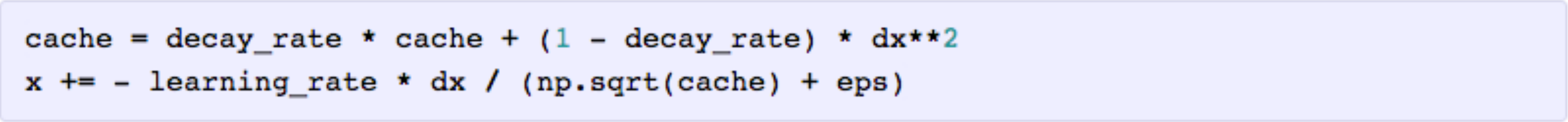
Per-parameter adaptive learning rate methods

Adagrad



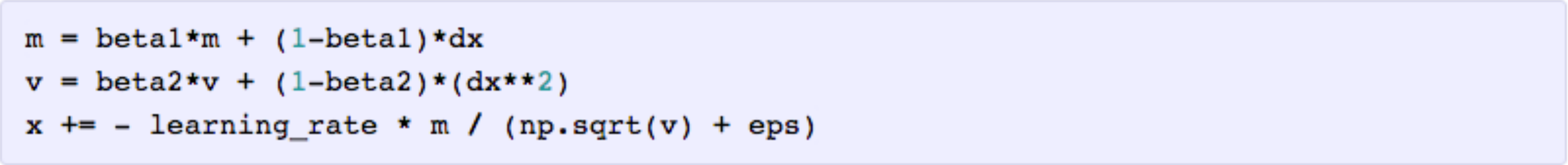
```
# Assume the gradient dx and parameter vector x
cache += dx**2
x += - learning_rate * dx / (np.sqrt(cache) + eps)
```

RMSprop



```
cache = decay_rate * cache + (1 - decay_rate) * dx**2
x += - learning_rate * dx / (np.sqrt(cache) + eps)
```

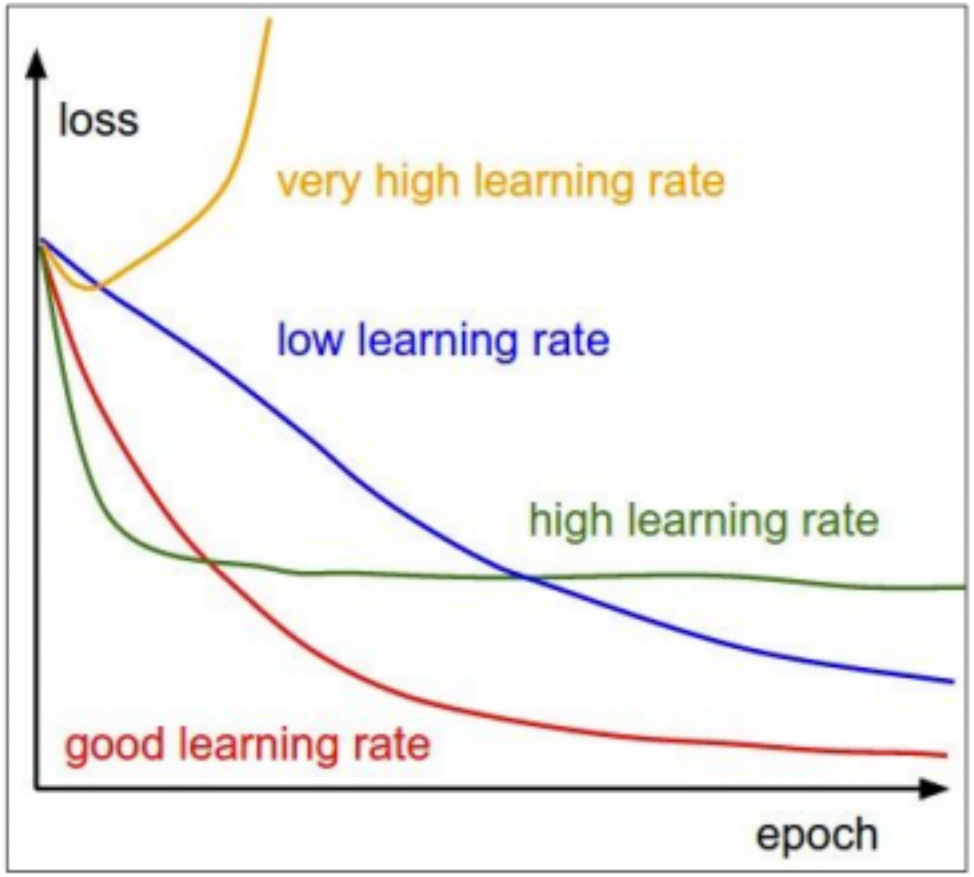
Adam



```
m = beta1*m + (1-beta1)*dx
v = beta2*v + (1-beta2)*(dx**2)
x += - learning_rate * m / (np.sqrt(v) + eps)
```

Annealing the Learning Rates

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:

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

Compare Learning Methods

- http://cs231n.github.io/neural-networks-3/#sgd

In Practice

- Adam is the default choice in most cases

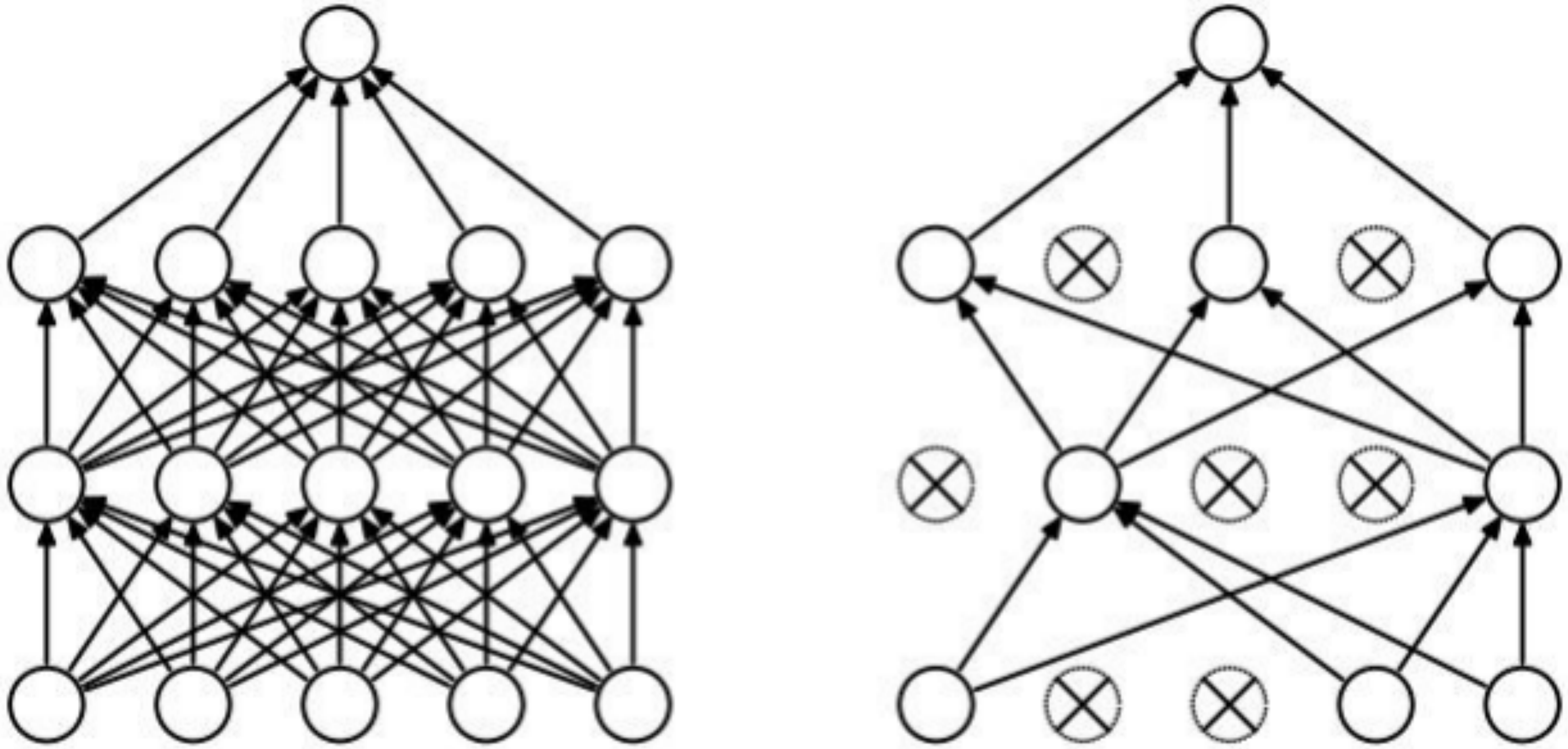
- Instead, SGD variants based on (Nesterov’s) momentum are more standard than second-order methods because they are simpler and scale more easily.

- If you can afford to do full batch updates then try out L-BFGS (Limited-memory version of Broyden–Fletcher–Goldfarb–Shanno (BFGS) algorithm). Don’t forget to disable all sources of noise.

Regularization

DropOut

“randomly set some neurons to zero in the forward pass”



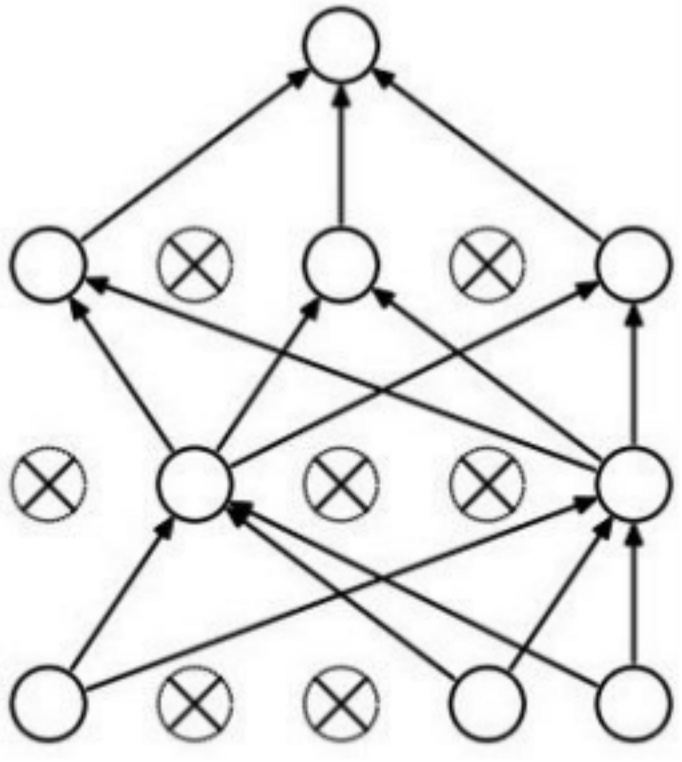
(a) Standard Neural Net

(b) After applying dropout.

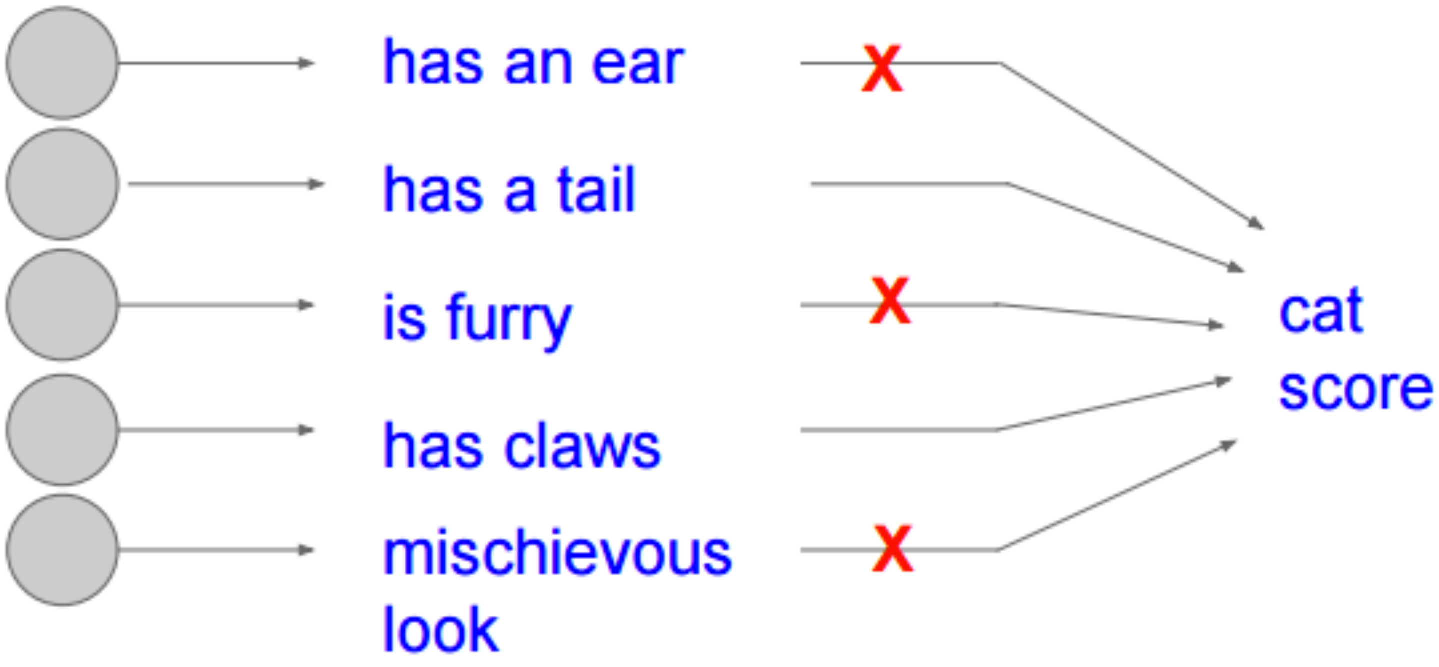
[Srivastava et al., 2014]

DropOut

Waaaait a second…
How could this possibly be a good idea?



Forces the network to have a redundant representation.



has an ear

has a tail

is furry

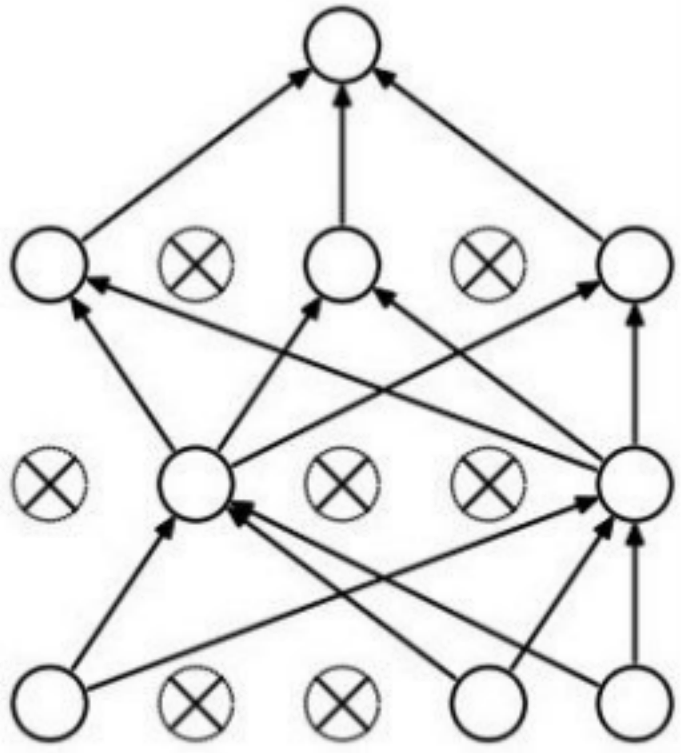
has claws

mischievous look

cat score

DropOut

Waaaait a second…
How could this possibly be a good idea?



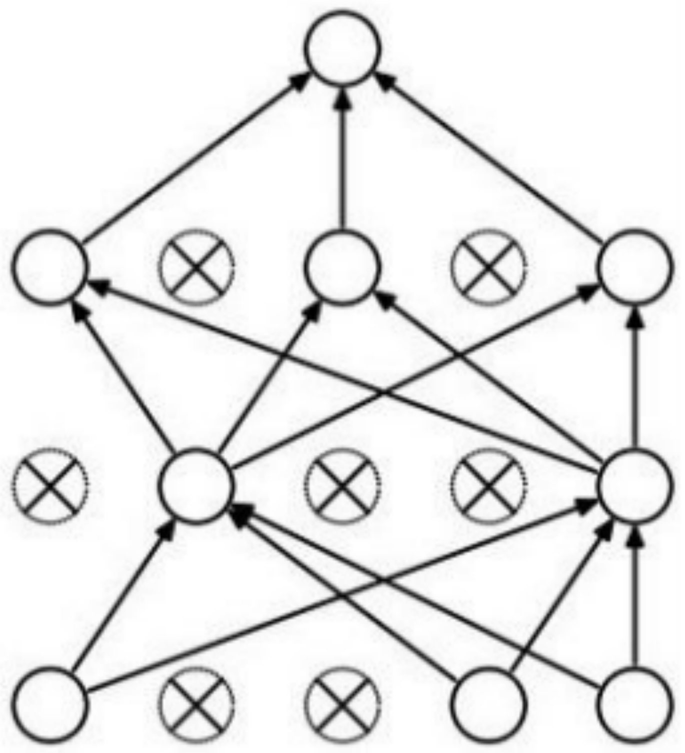
Another interpretation:

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

Each binary mask is one model, gets trained on only ~one datapoint.

DropOut

At test time….



Ideally:
want to integrate out all the noise

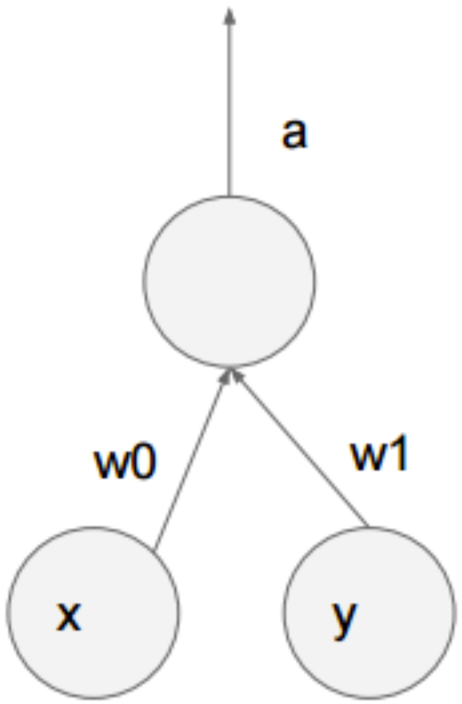
Monte Carlo approximation:
do many forward passes with different dropout masks, average all predictions

DropOut

At test time….

Can in fact do this with a single forward pass! (approximately)

Leave all input neurons turned on (no dropout).



during test: $\mathbf{a = w0*x + w1*y}$

during train:

$$\begin{aligned} E[a] = \tfrac{1}{4} * (&w0*0 + w1*0 \\ &w0*0 + w1*y \\ &w0*x + w1*0 \\ &w0*x + w1*y) \\ = \tfrac{1}{4} * (&2\ w0*x + 2\ w1*y) \\ = \tfrac{1}{2} * (&\mathbf{w0*x + w1*y}) \end{aligned}$$

With p=0.5, using all inputs in the forward pass would inflate the activations by 2x from what the network was “used to” during training! => Have to compensate by scaling the activations back down by ½

Normalization

Batch Normalization

[Ioffe and Szegedy, 2015]

Normalize:

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

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

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

- 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]

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:

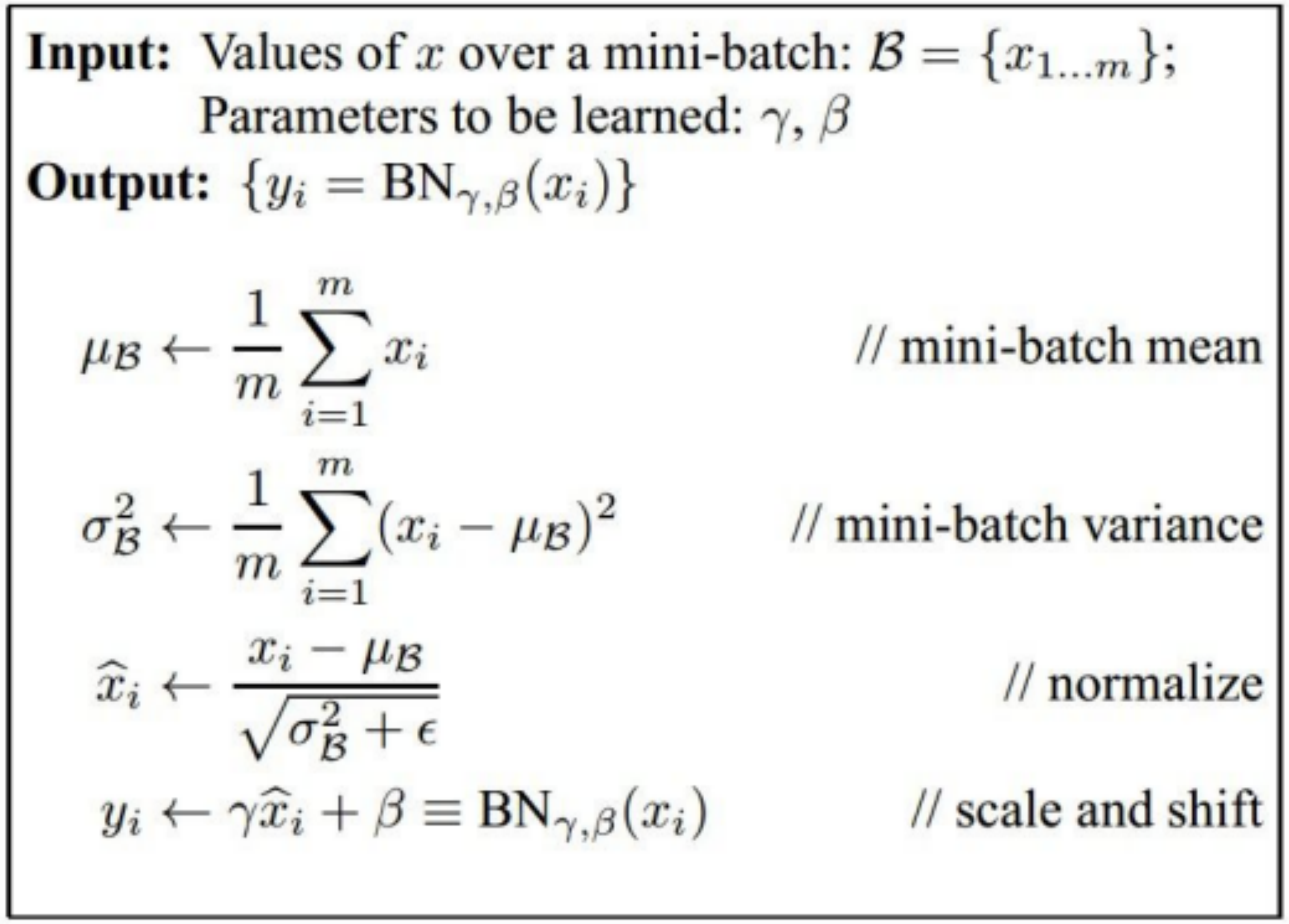
$$\gamma^{(k)} = \sqrt{\mathrm{Var}[x^{(k)}]}$$

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

to recover the identity mapping.

Batch Normalization

[Ioffe and Szegedy, 2015]



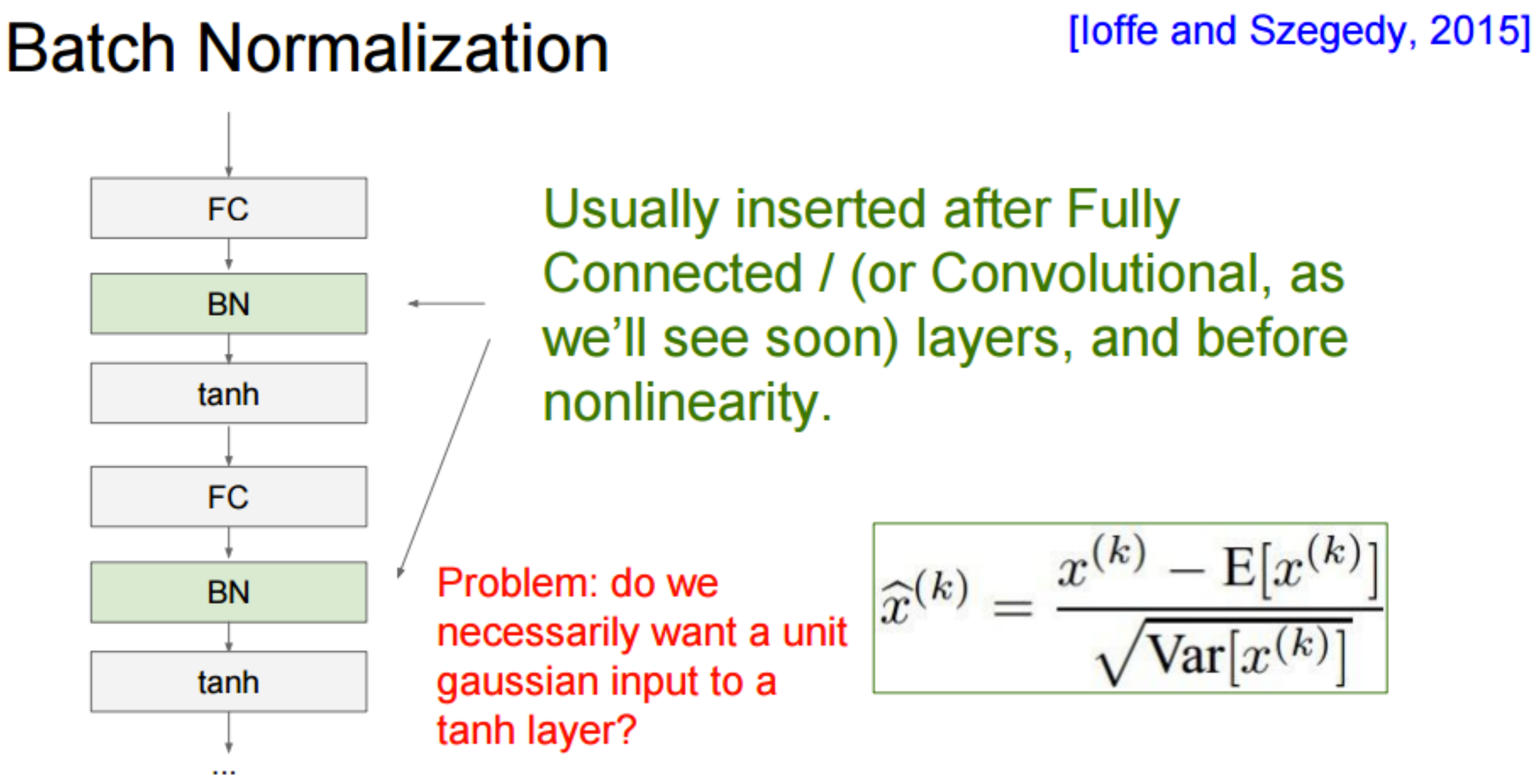
```
Input: Values of $x$ over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
       Parameters to be learned: $\gamma$, $\beta$
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

    $\mu_\mathcal{B} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i$                          // mini-batch mean
    $\sigma_\mathcal{B}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_\mathcal{B})^2$        // mini-batch variance
    $\widehat{x}_i \leftarrow \frac{x_i - \mu_\mathcal{B}}{\sqrt{\sigma_\mathcal{B}^2 + \epsilon}}$                  // normalize
    $y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i)$      // 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)

Batch Normalization



Batch Normalization

[Ioffe and Szegedy, 2015]

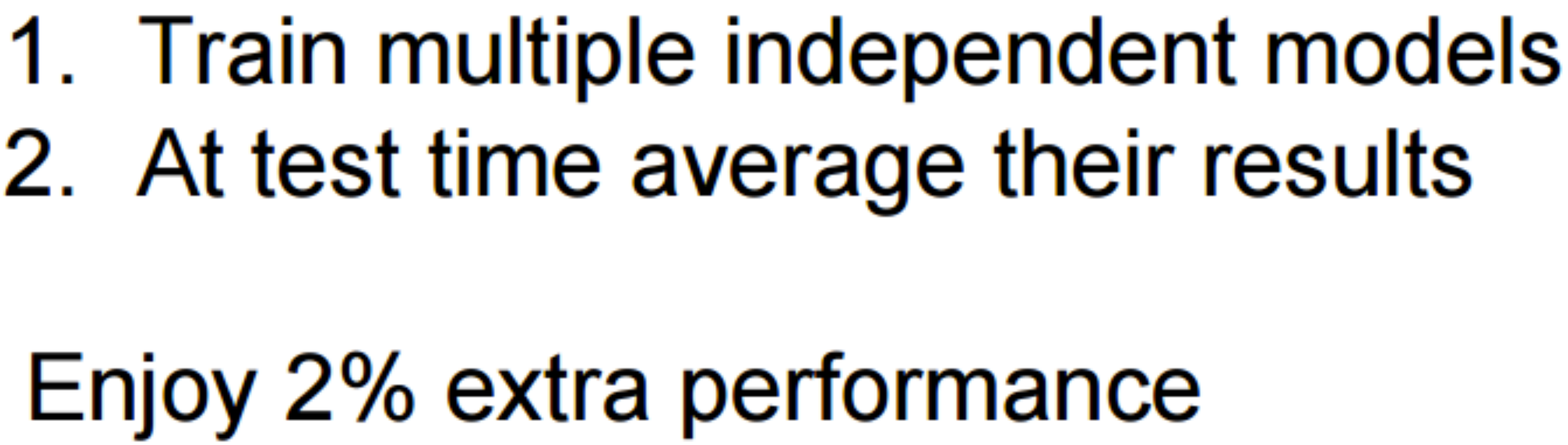
Usually inserted after Fully Connected / (or Convolutional, as we'll see soon) layers, and before nonlinearity.

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

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

Ensembles

Model Ensembles



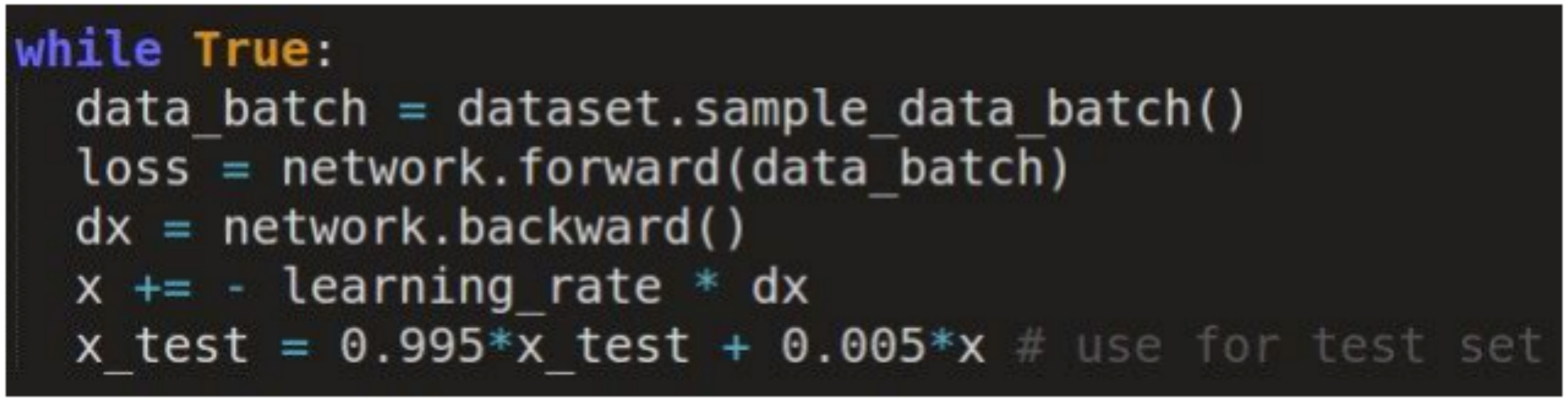
1. Train multiple independent models
2. At test time average their results

Enjoy 2% extra performance

Model Ensembles

Fun Tips/Tricks:

- can also get a small boost from averaging multiple model checkpoints of a single model.
- keep track of (and use at test time) a running average parameter vector:



```
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
```

Hyperparameter Optimization

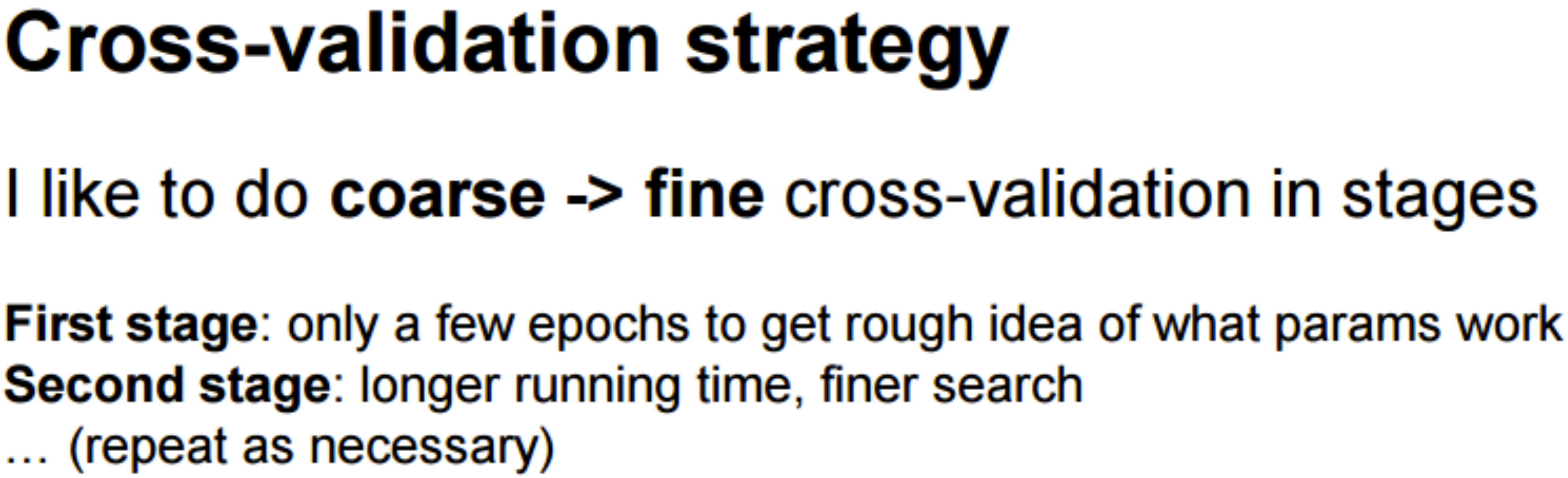
Hyperparameter Optimization

Hyperparameters to play with:

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

neural networks practitioner
music = loss function

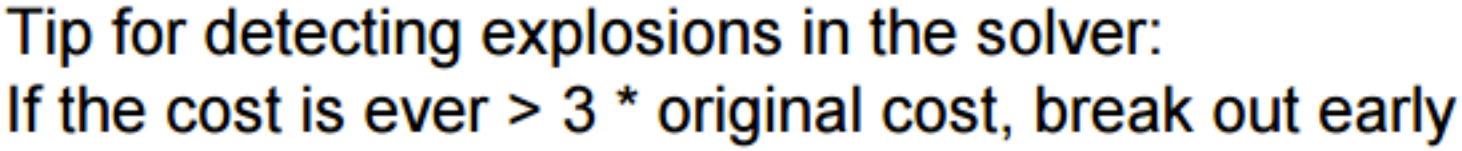
Hyperparameter Optimization



Cross-validation strategy

I like to do **coarse -> fine** cross-validation in stages

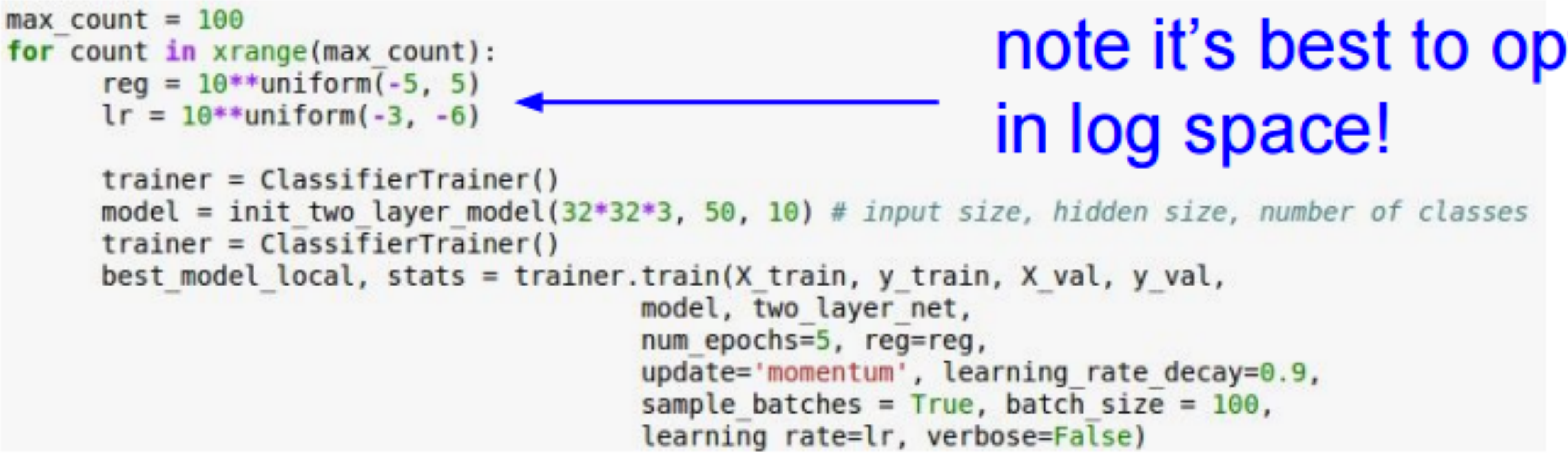
First stage: only a few epochs to get rough idea of what params work
Second stage: longer running time, finer search
… (repeat as necessary)



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

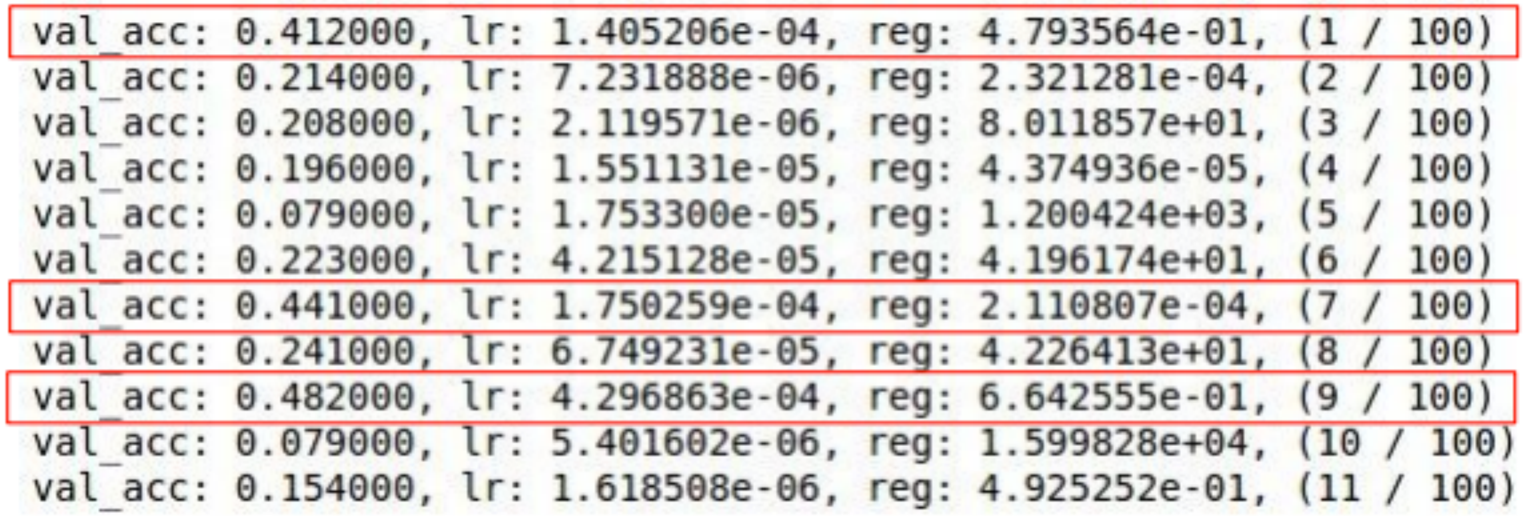
Hyperparameter Optimization

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!

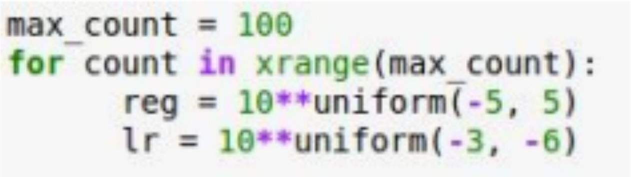


```
val_acc: 0.412000, lr: 1.405206e-04, reg: 4.793564e-01, (1 / 100)
val_acc: 0.214000, lr: 7.231888e-06, reg: 2.321281e-04, (2 / 100)
val_acc: 0.208000, lr: 2.119571e-06, reg: 8.011857e+01, (3 / 100)
val_acc: 0.196000, lr: 1.551131e-05, reg: 4.374936e-05, (4 / 100)
val_acc: 0.079000, lr: 1.753300e-05, 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

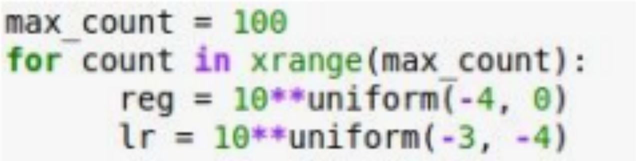
Hyperparameter Optimization

Now run finer search...

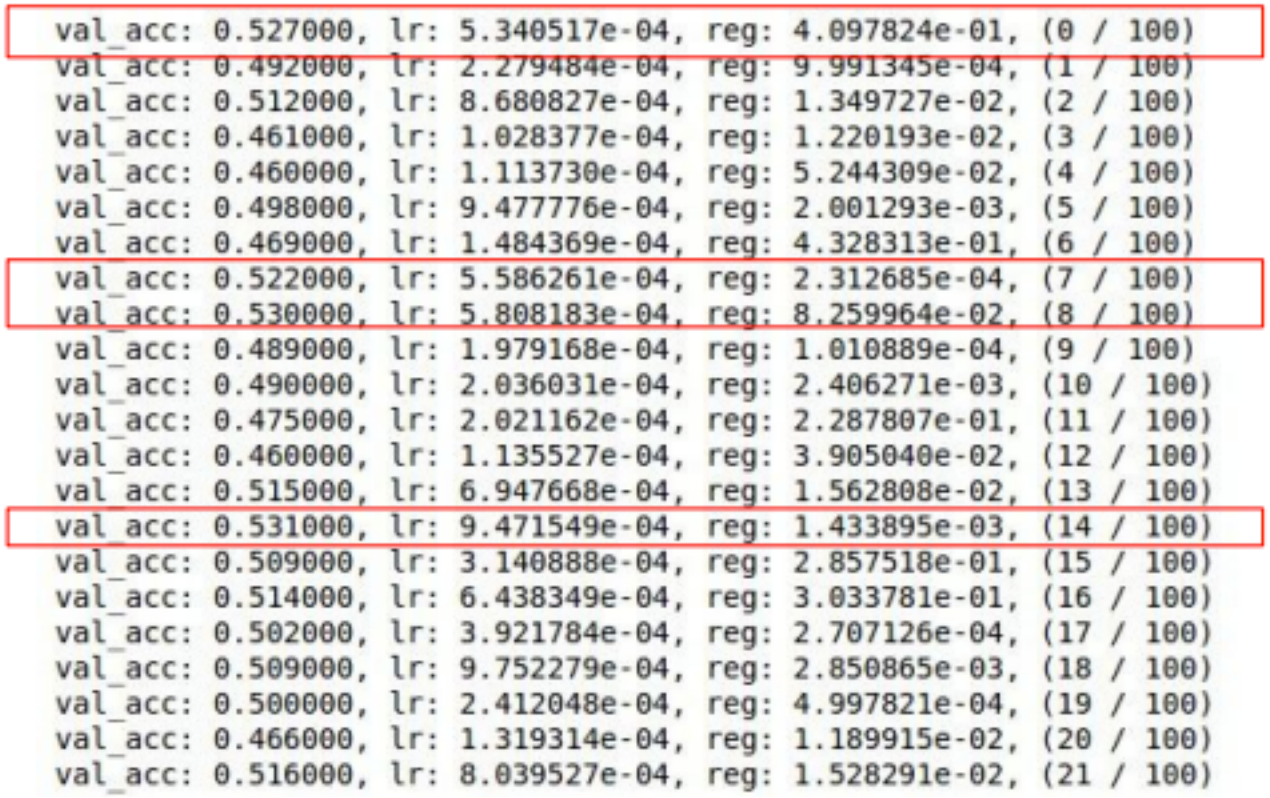


```
max_count = 100
for count in xrange(max_count):
      reg = 10**uniform(-5, 5)
      lr = 10**uniform(-3, -6)
```

adjust range



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

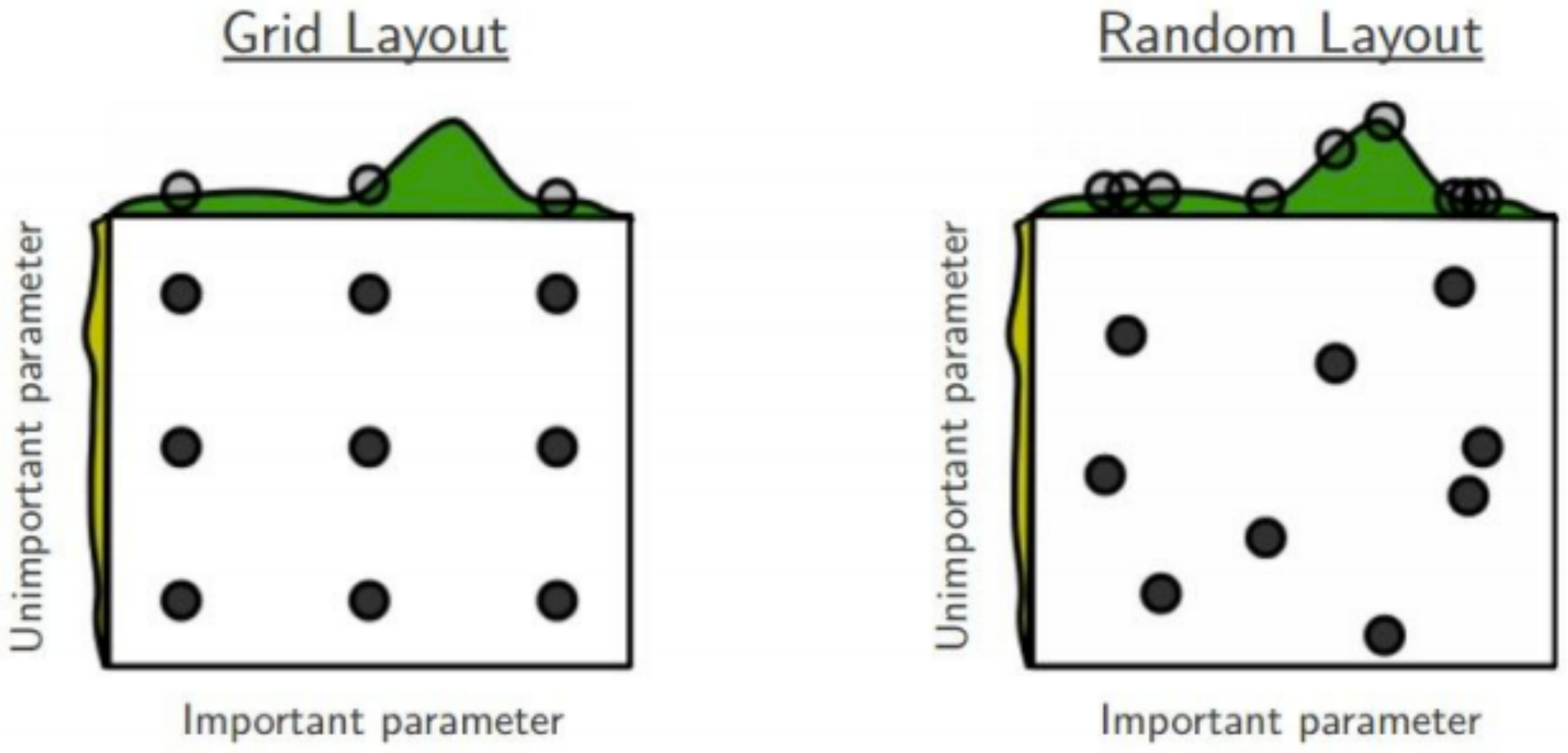


```
val_acc: 0.527000, lr: 5.340517e-04, reg: 4.097824e-01, (0 / 100)
val_acc: 0.492000, lr: 2.279484e-04, reg: 9.991345e-04, (1 / 100)
val_acc: 0.512000, lr: 8.680827e-04, reg: 1.349727e-02, (2 / 100)
val_acc: 0.461000, lr: 1.028377e-04, reg: 1.220193e-02, (3 / 100)
val_acc: 0.460000, lr: 1.113730e-04, 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.

Hyperparamter Optimization

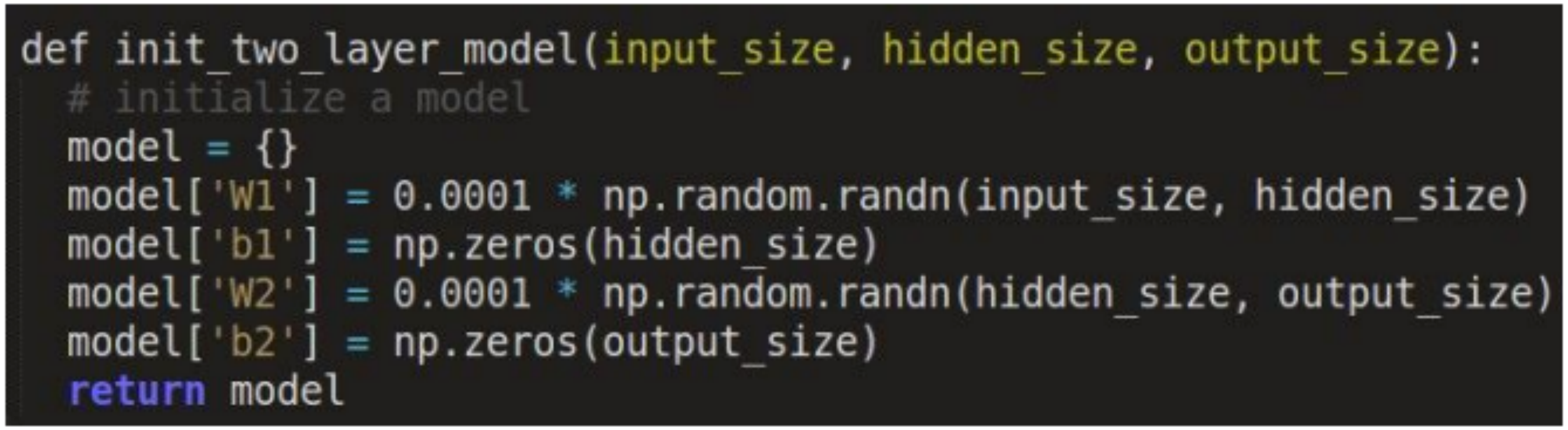
Random Search vs. Grid Search



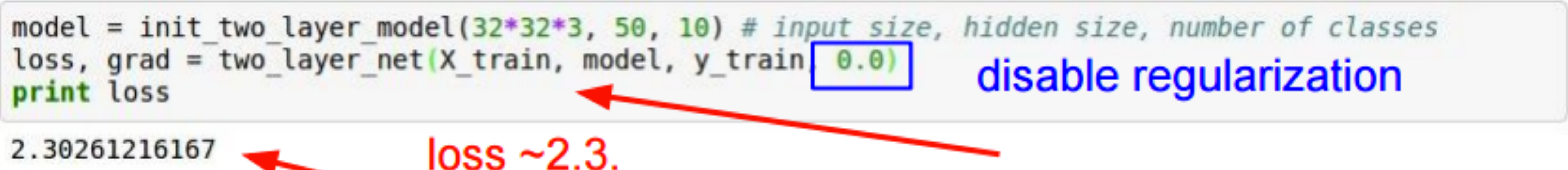
Random Search for Hyper-Parameter Optimization
Bergstra and Bengio, 2012

Monitoring the Learning Process

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
```

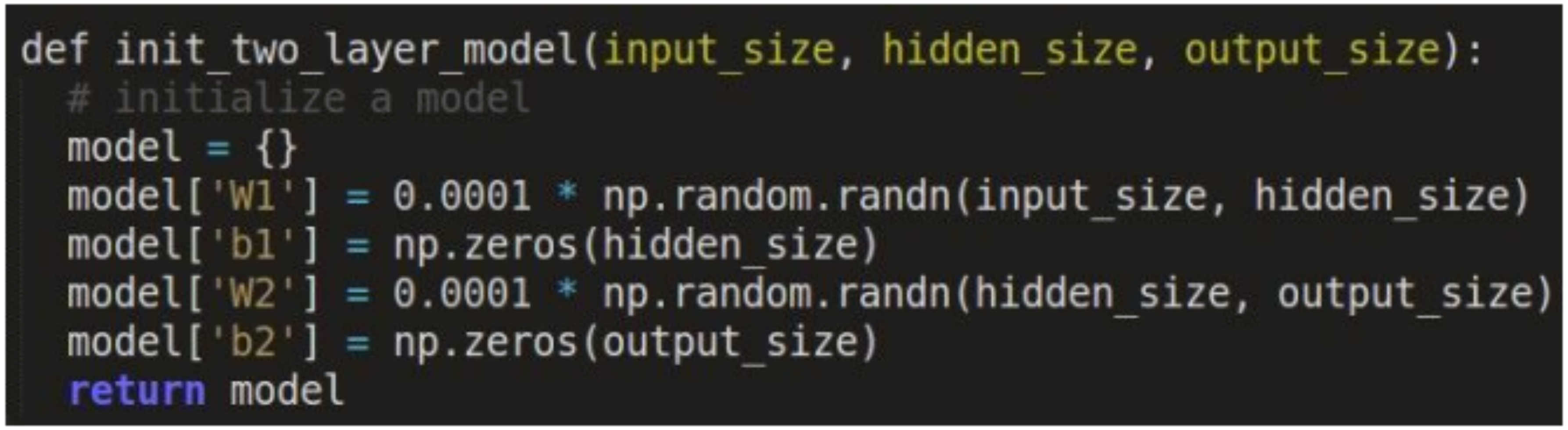
```
2.30261216167
```

disable regularization

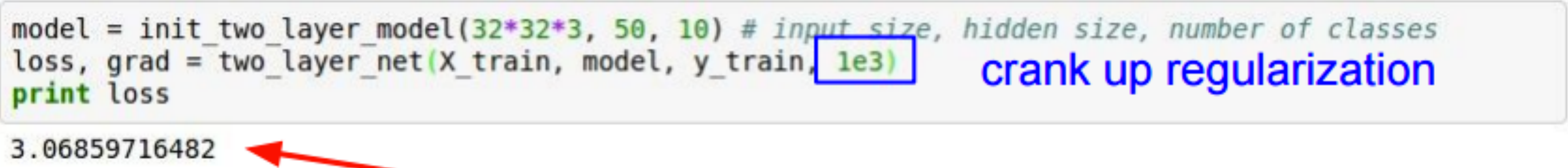
loss ~2.3.
“correct “ for
10 classes

returns the loss and the
gradient for all parameters

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
```



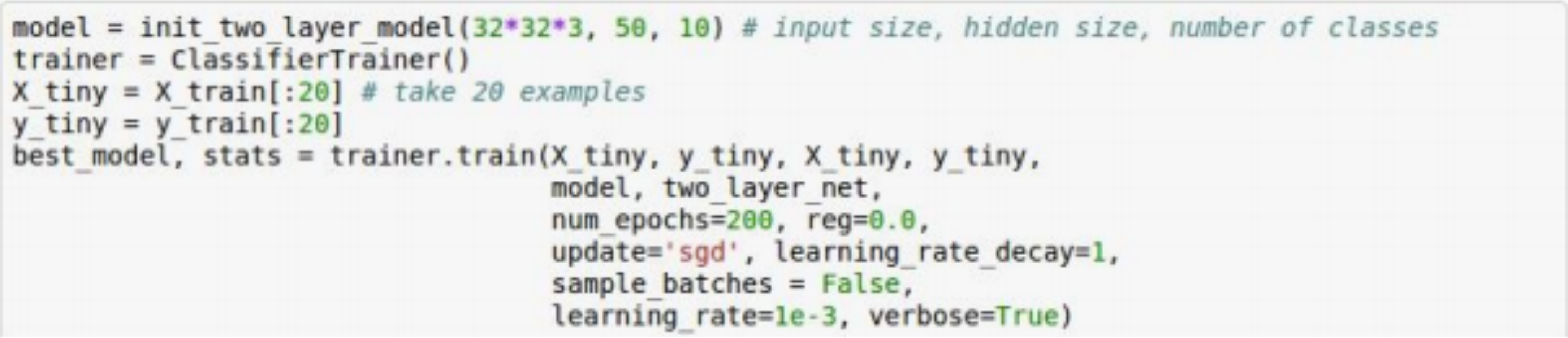
```
model = init_two_layer_model(32*32*3, 50, 10) # input_size, hidden_size, number of classes
loss, grad = two_layer_net(X_train, model, y_train, 1e3)
print loss
```

crank up regularization

```
3.06859716482
```

loss went up, good. (sanity check)

Overfit Very Small Portion of the Training Data



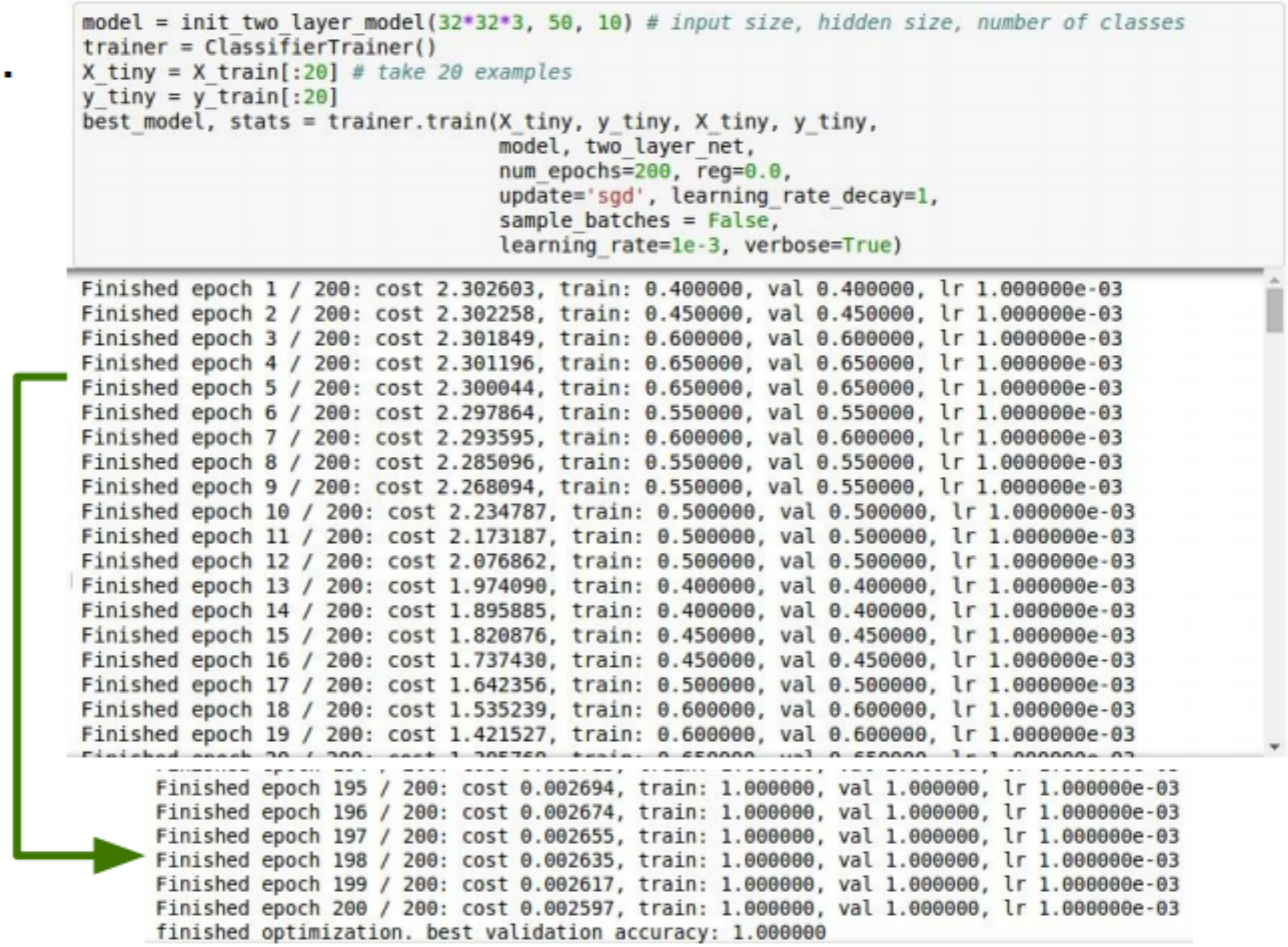
```
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
trainer = ClassifierTrainer()
X_tiny = X_train[:20] # take 20 examples
y_tiny = y_train[:20]
best_model, stats = trainer.train(X_tiny, y_tiny, X_tiny, y_tiny,
                                  model, two_layer_net,
                                  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’

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
```

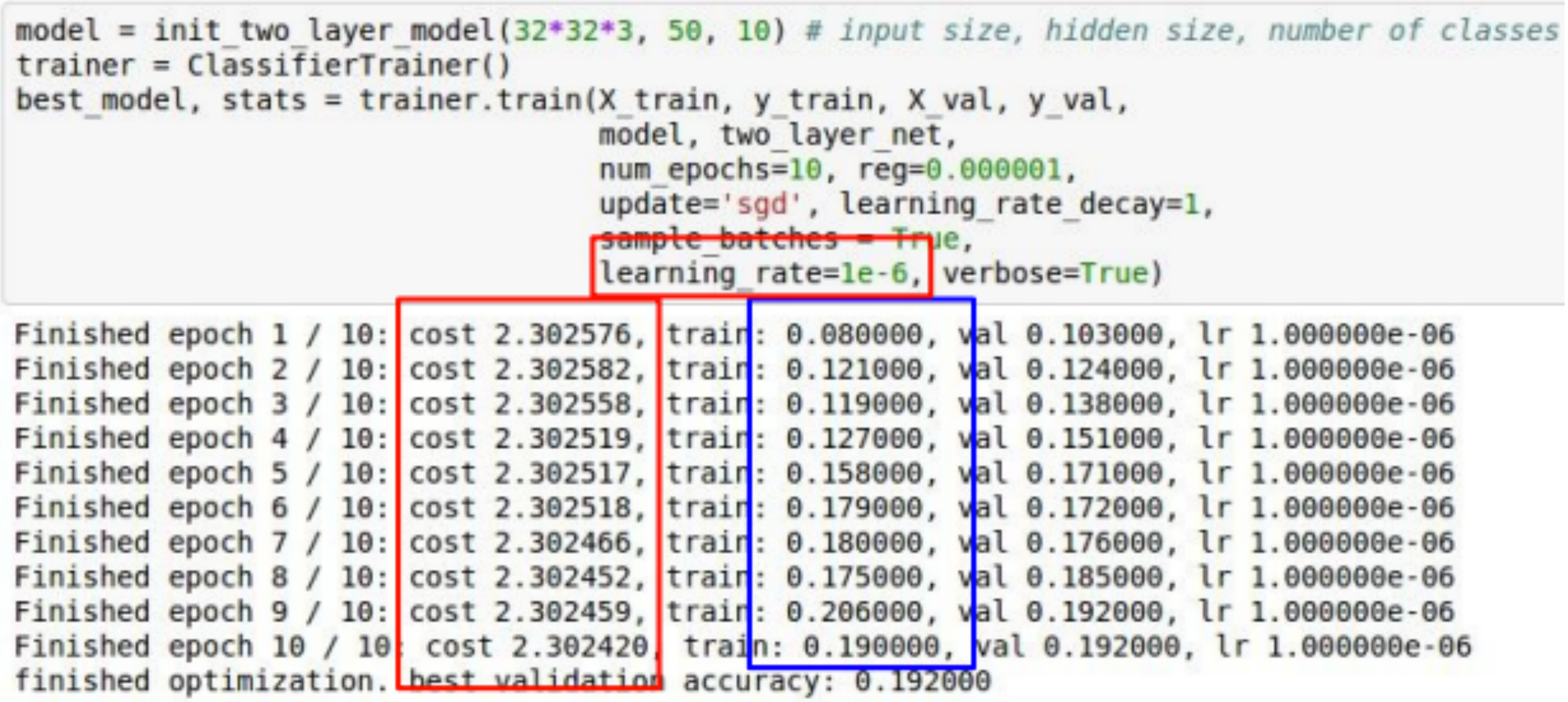
```
Finished epoch 195 / 200: cost 0.002694, train: 1.000000, val 1.000000, lr 1.000000e-03
Finished epoch 196 / 200: cost 0.002674, train: 1.000000, val 1.000000, lr 1.000000e-03
Finished epoch 197 / 200: cost 0.002655, train: 1.000000, val 1.000000, lr 1.000000e-03
Finished epoch 198 / 200: cost 0.002635, 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
```

Very small loss, train accuracy 1.00, nice!

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



```
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
```

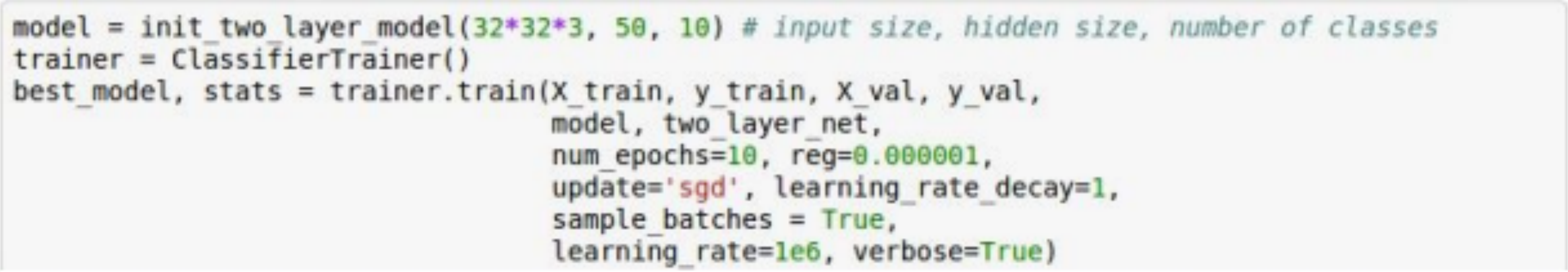
Loss barely changing: Learning rate is probably too low

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

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



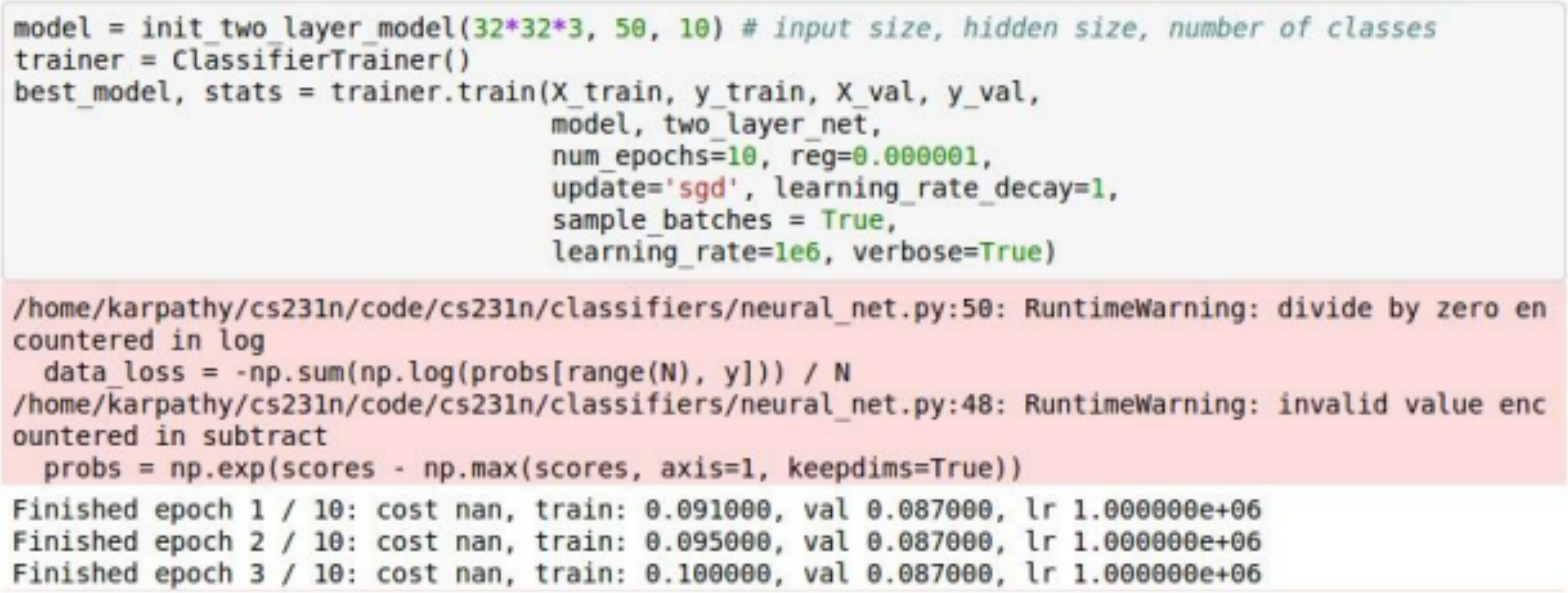
```
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
trainer = ClassifierTrainer()
best_model, stats = trainer.train(X_train, y_train, X_val, y_val,
                                  model, two_layer_net,
                                  num_epochs=10, reg=0.000001,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

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



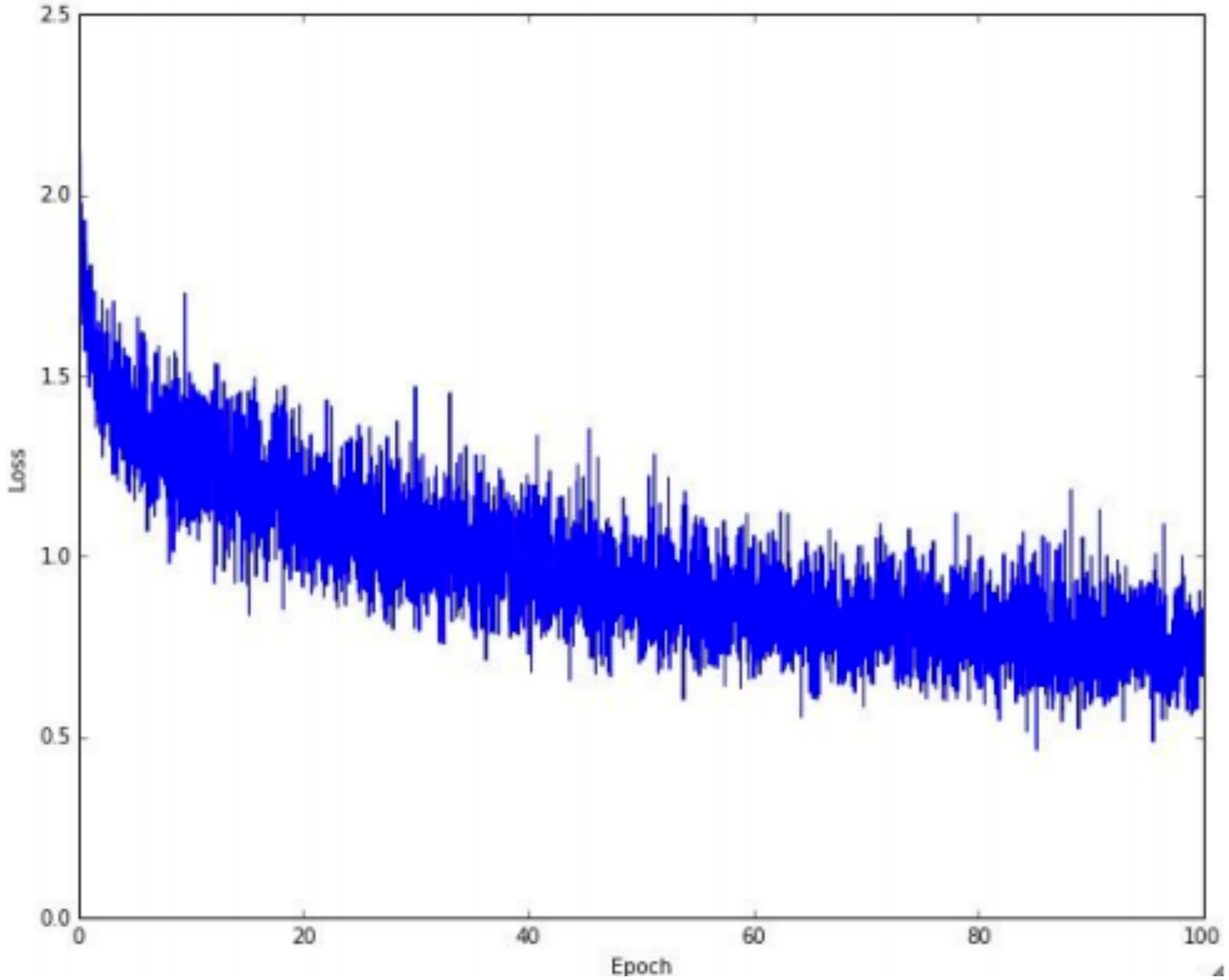
```
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
trainer = ClassifierTrainer()
best_model, stats = trainer.train(X_train, y_train, X_val, y_val,
                                  model, two_layer_net,
                                  num_epochs=10, reg=0.000001,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

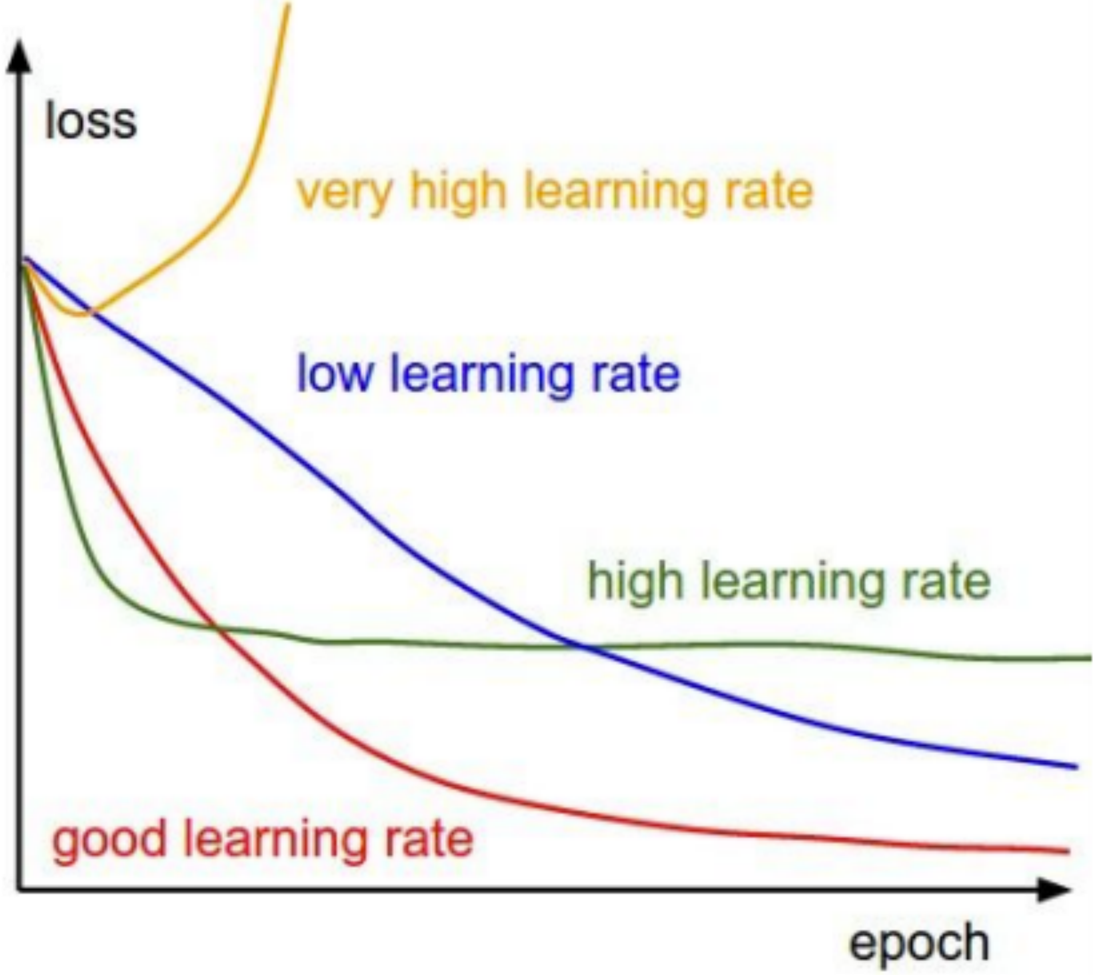
```
/home/karpathy/cs231n/code/cs231n/classifiers/neural_net.py:50: RuntimeWarning: divide by zero en
countered in log
  data_loss = -np.sum(np.log(probs[range(N), y])) / N
/home/karpathy/cs231n/code/cs231n/classifiers/neural_net.py:48: RuntimeWarning: invalid value enc
ountered in subtract
  probs = np.exp(scores - np.max(scores, axis=1, keepdims=True))
```

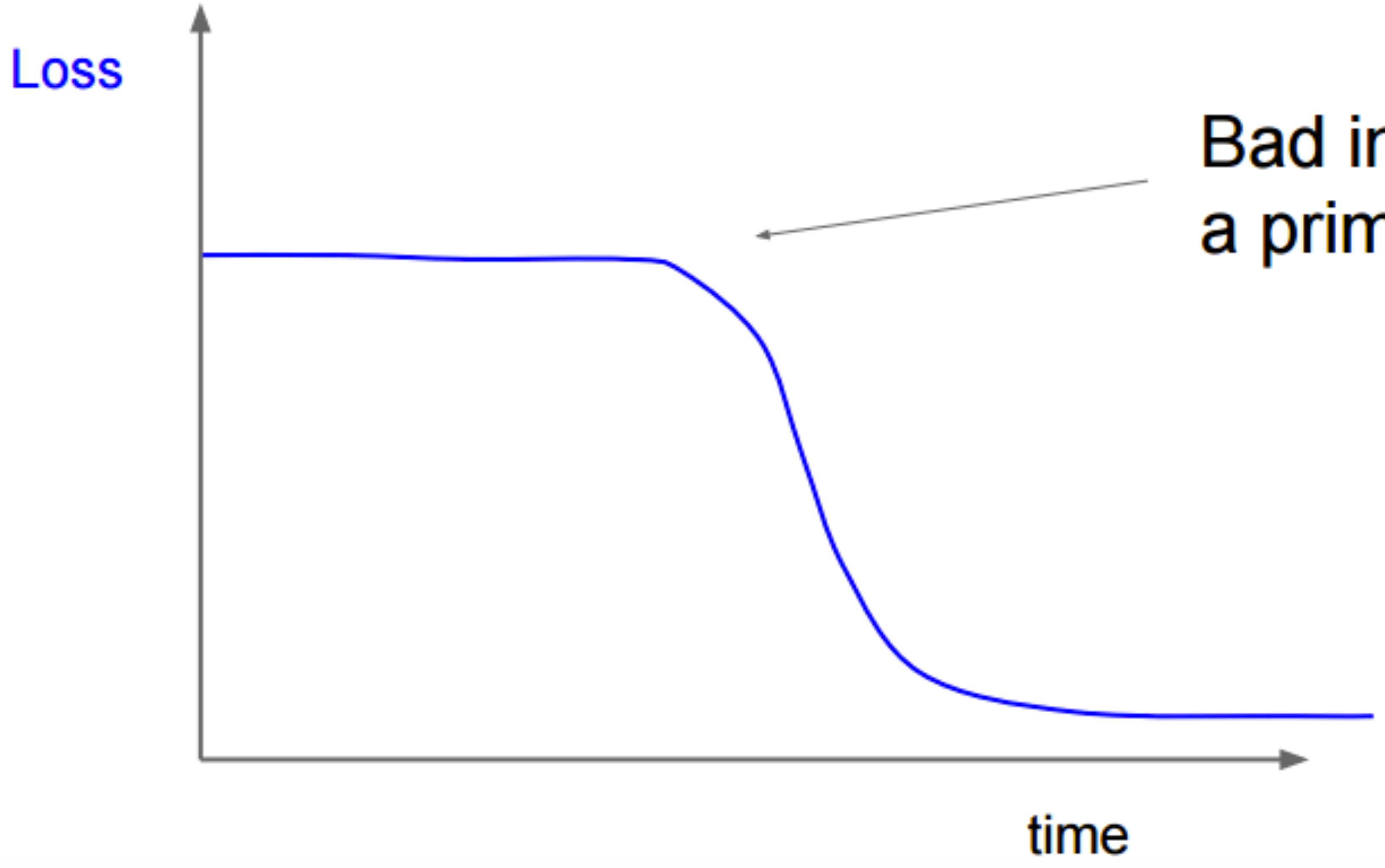
```
Finished epoch 1 / 10: cost nan, train: 0.091000, val 0.087000, lr 1.000000e+06
Finished epoch 2 / 10: cost nan, train: 0.095000, val 0.087000, lr 1.000000e+06
Finished epoch 3 / 10: cost nan, train: 0.100000, val 0.087000, lr 1.000000e+06
```

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

Monitor and visualize the loss curve

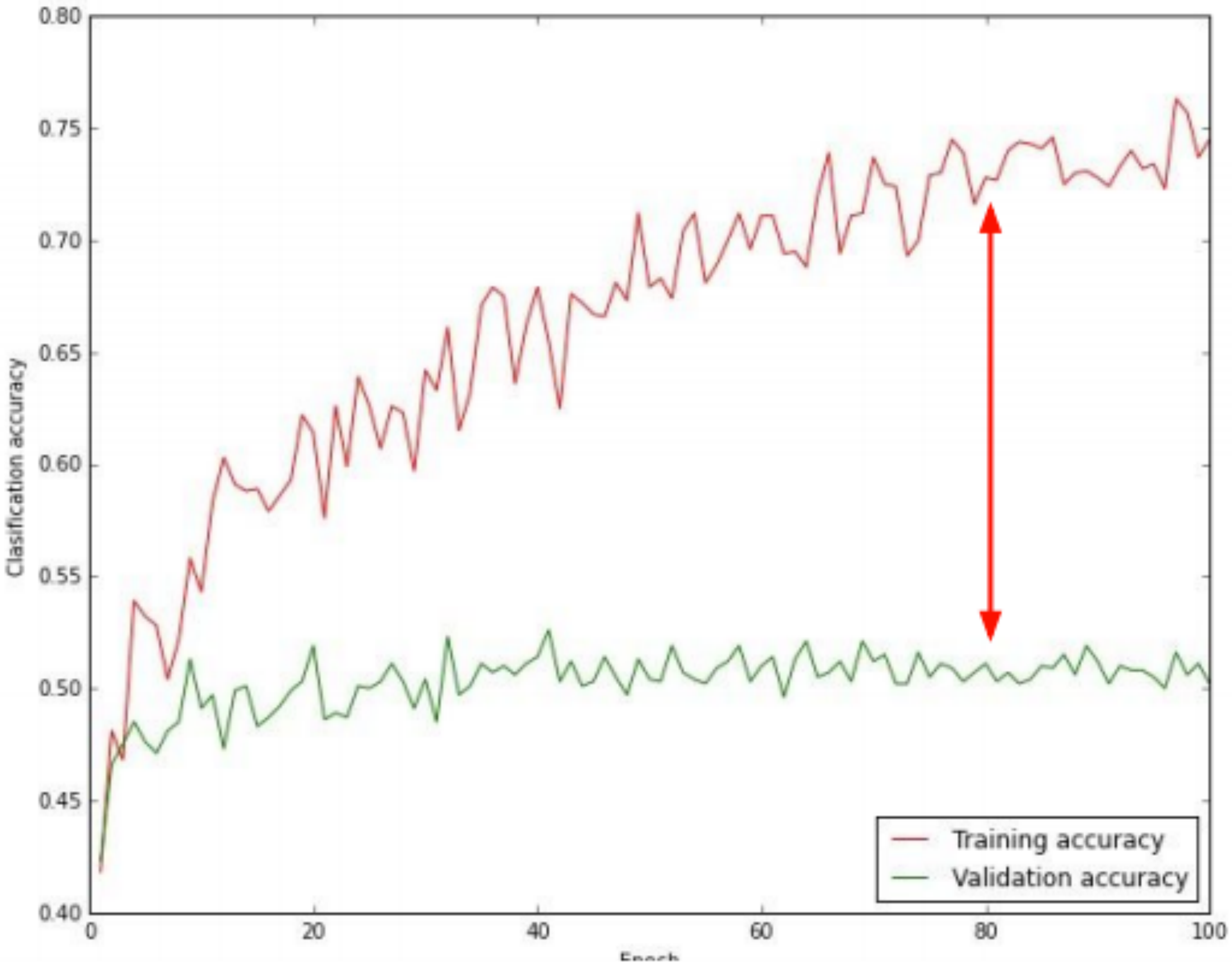






Bad initialization a prime suspect

Monitor and visualize the accuracy:



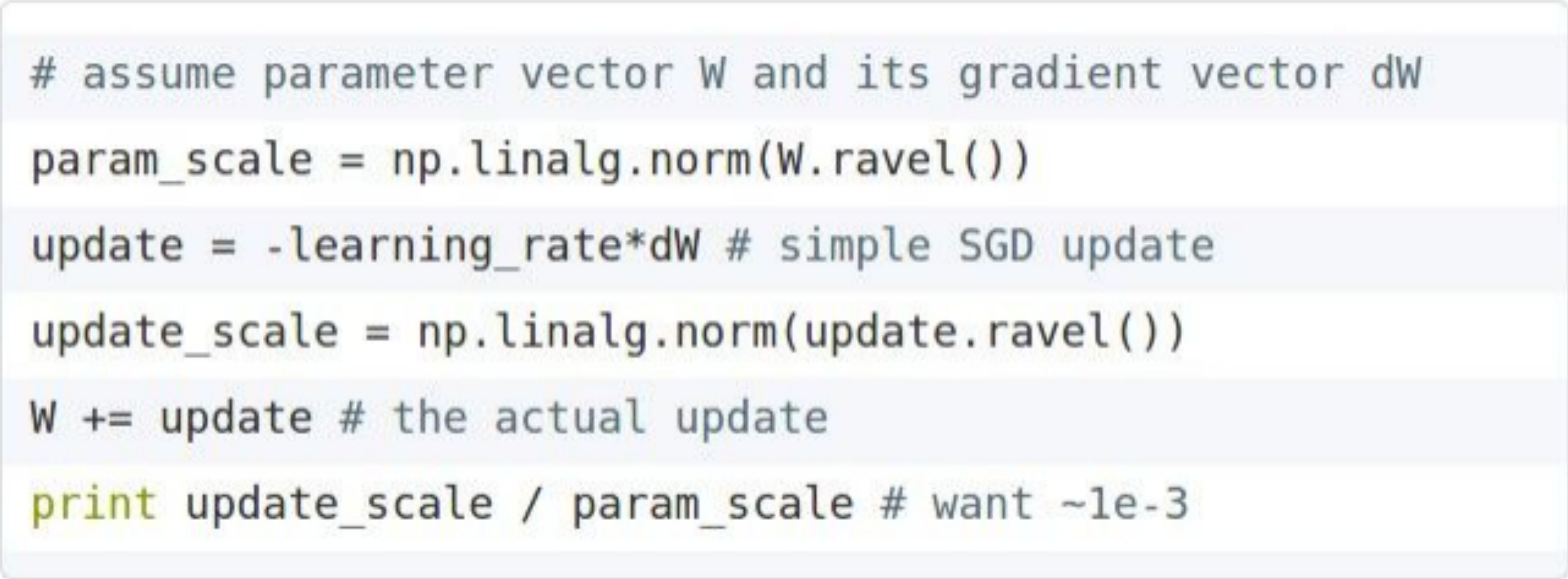
big gap = overfitting

=> increase regularization strength?

no gap

=> increase model capacity?

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