Getting Neural Nets to Train

Designing, Visualizing and Understanding Deep Neural Networks

CS W182/282A

Instructor: Sergey Levine
UC Berkeley

This lecture

Help! My network doesn't train

If you follow everything I described in the previous lectures…

And you implement everything correctly…

And you train everything for a long time…

There is a good chance it **still** won't work

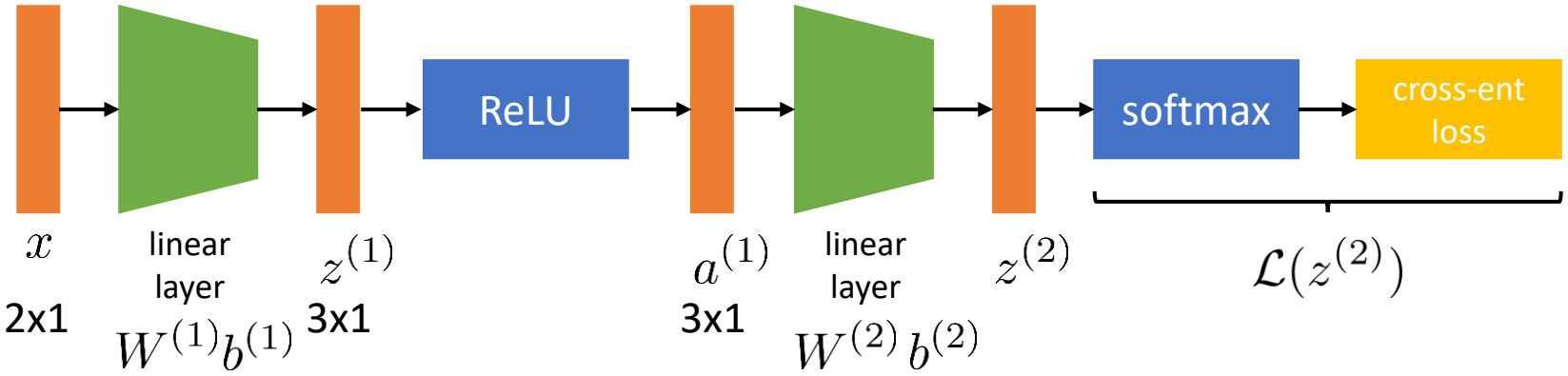
Neural networks are messy

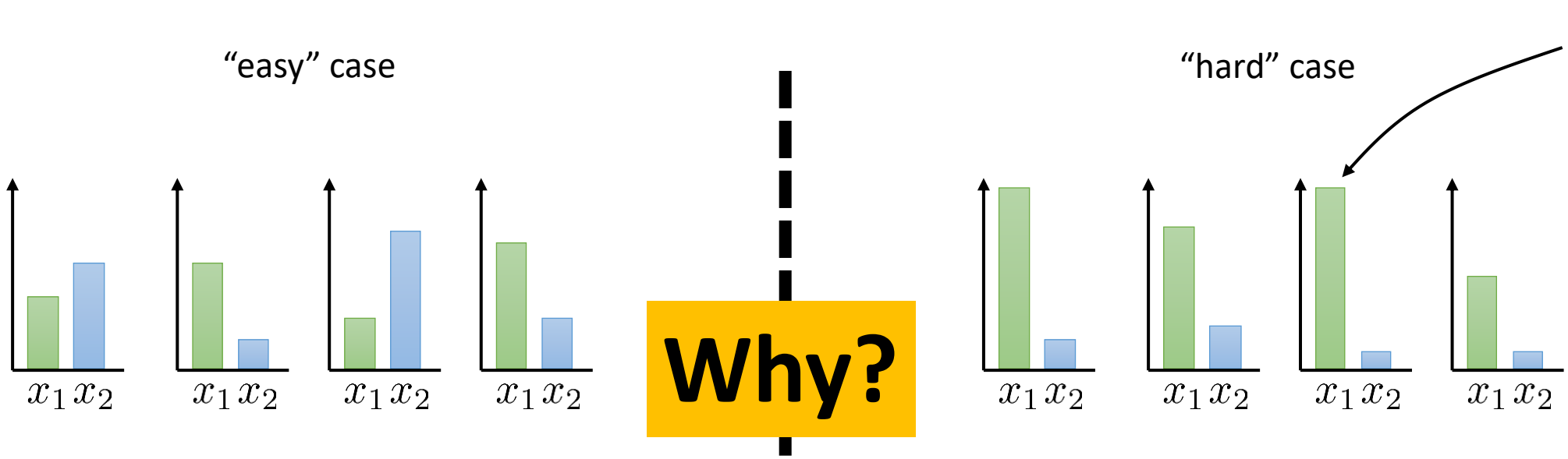
They require lots of "tricks" to train well

We'll discuss these tricks today

- Normalizing inputs and outputs
- Normalizing **activations** (batch normalization)
- Initialization of weight matrices & bias vectors
- Gradient clipping
- Best practices for hyperparameter optimization
- Ensembling, dropout

The dangers of **big** inputs, activations, and outputs

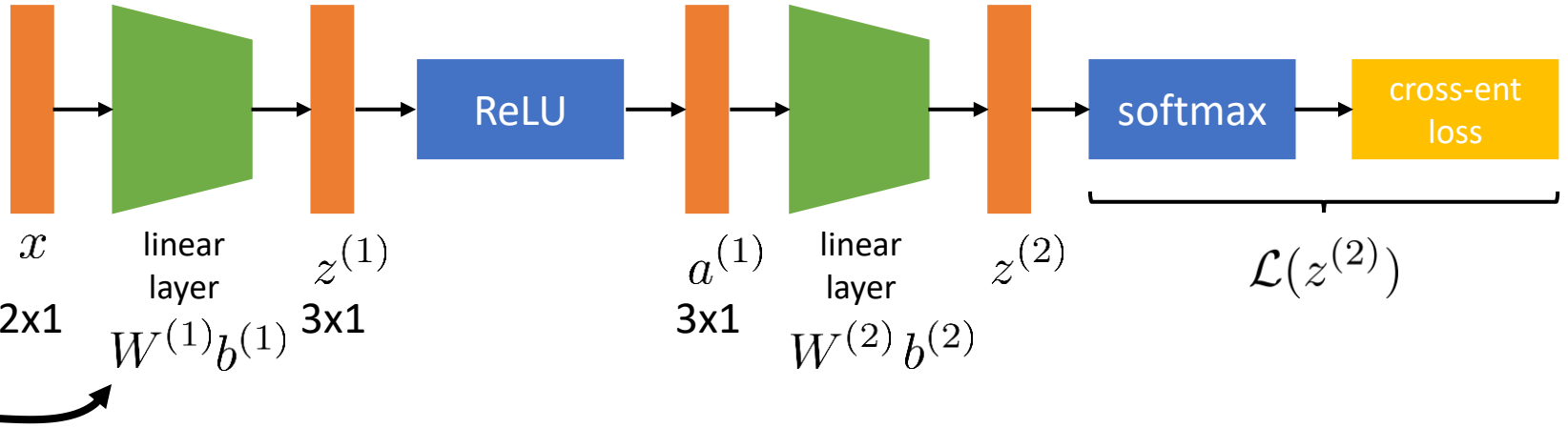




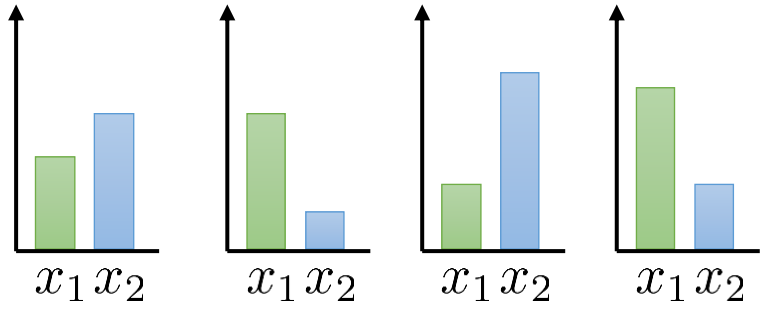
The dangers of **big** inputs, activations, and outputs

gradient will get scaled by big *x* values

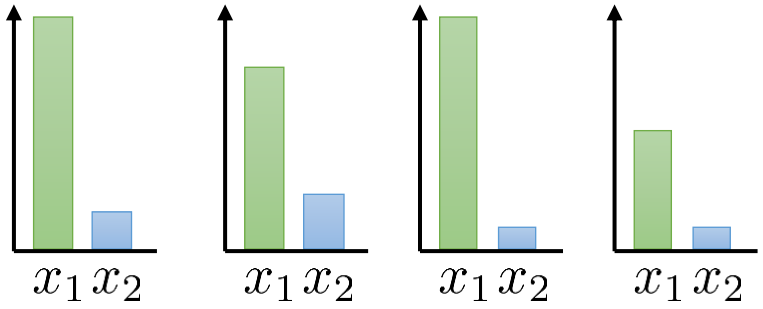
$$\frac{d\mathcal{L}}{dW^{(1)}} = \frac{dz^{(1)}}{dW^{(1)}} \overbrace{\frac{d\mathcal{L}}{dz^{(1)}}}^{\delta} = \delta x^T$$



"easy" case

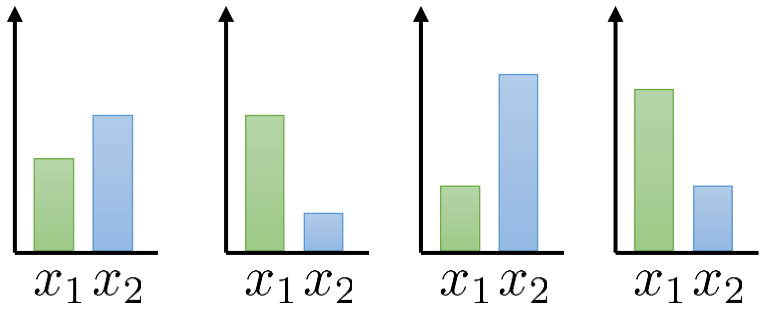


"hard" case

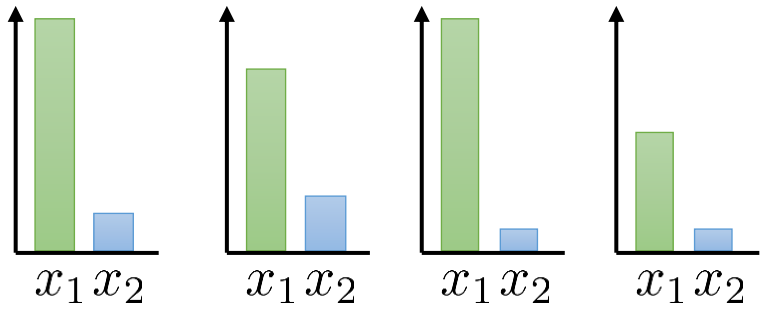


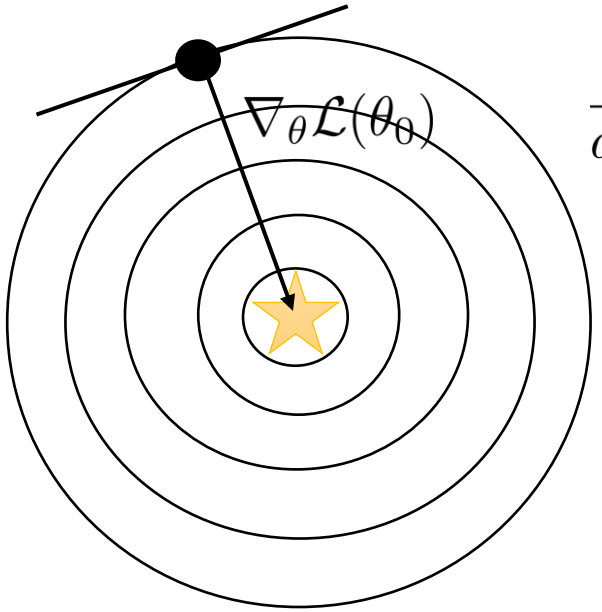
The dangers of **big** inputs, activations, and outputs

"easy" case

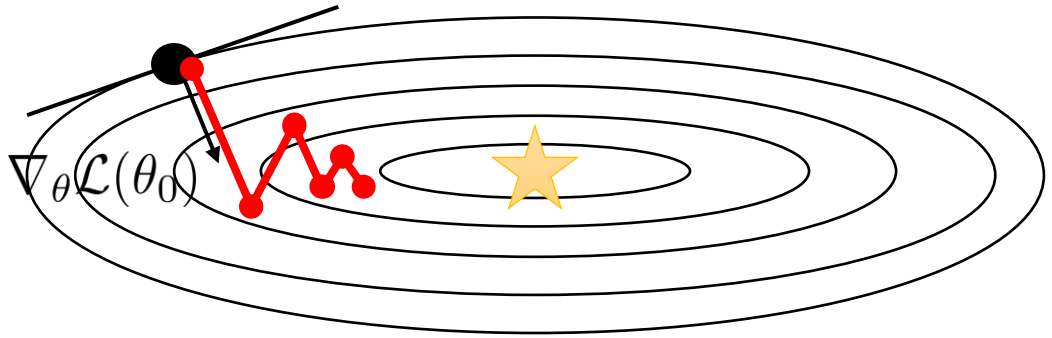


"hard" case





$$\frac{d\mathcal{L}}{dW^{(1)}} = \frac{dz^{(1)}}{dW^{(1)}} \frac{d\mathcal{L}}{dz^{(1)}} = \delta x^T$$



The dangers of **big** inputs, activations, and outputs

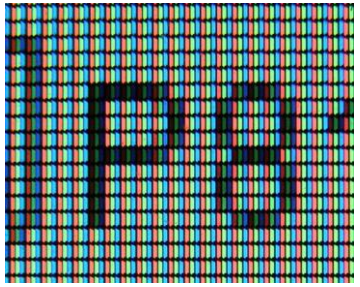
In **general**…

we really want all entries in x to be roughly on the same scale

sometimes not a problem:

images: all pixels are roughly in $[0, 1]$ or $\{0, \ldots, 255\}$

discrete inputs (e.g., NLP): all inputs are one-hot (zero or one)



"good"	"bad"
0	0
1	0
0	0
⋮	⋮
0	0
0	1
0	0

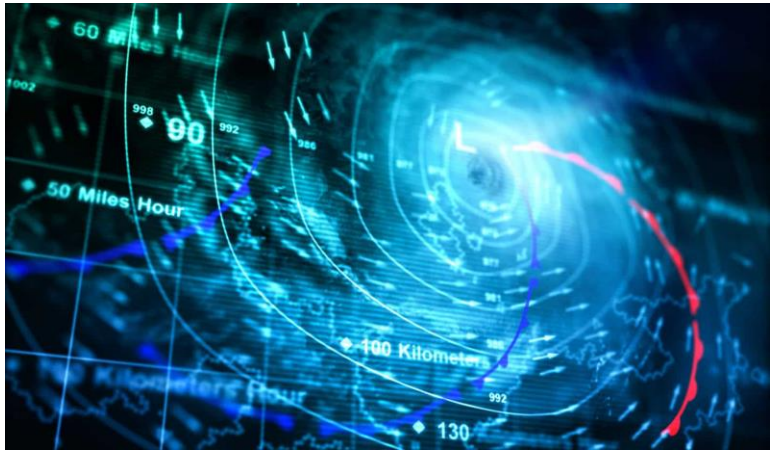
sometimes a **huge** problem:

forecasting the weather?

temperature: somewhere 40-100?

humidity: somewhere 0.3 - 0.6?

etc.



The dangers of **big** inputs, activations, and outputs

What can we do?

and outputs! (if doing regression)

…and activations??

Standardization: transform inputs so they have $\mu = 0$, $\sigma = 1$

To make $\mu = 0$: $\bar{x}_i = x_i - E[x]$ $\quad E[x] \approx \frac{1}{N}\sum_{i=1}^{N} x_i$

all operations are per-dimension

To *also* make $\sigma = 1$: $\bar{x}_i = \frac{x_i - E[x]}{\sqrt{E[(x_i - E[x])^2]}}$ ← standard deviation

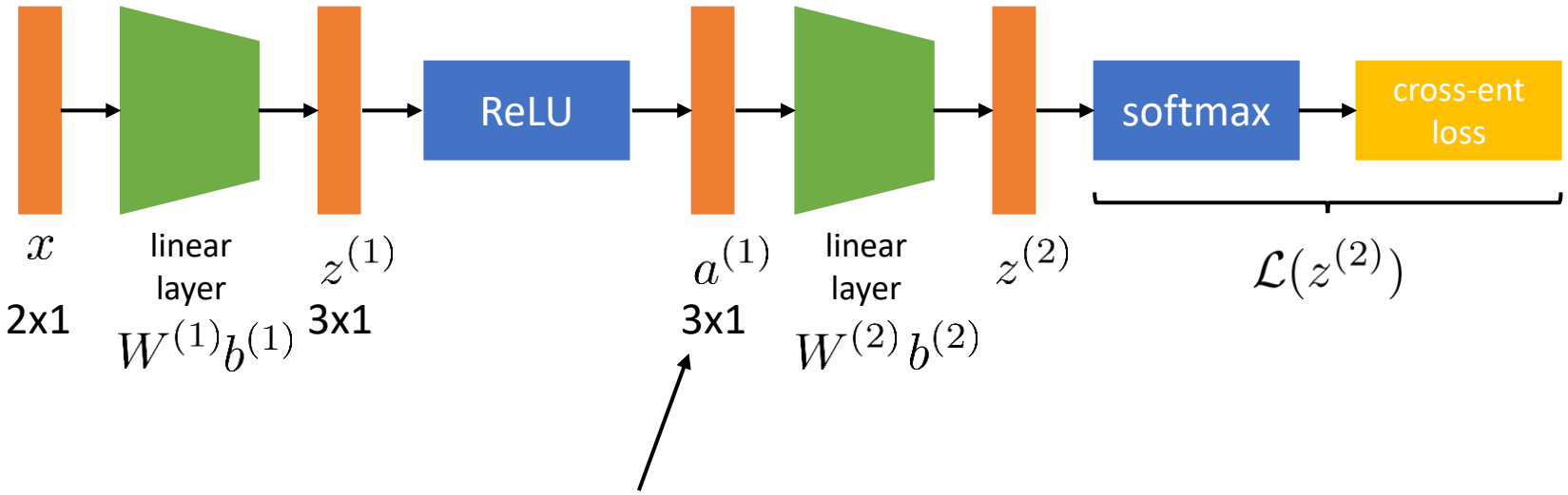
Standardizing activations?

What can we do?

and outputs! (if doing regression)

…and activations??

Standardization: transform inputs so they have $\mu = 0$, $\sigma = 1$



what if we start getting really different scales for each dimension **here**?

can we just standardize these activations too?

basically yes, but now the mean and standard deviation changes during training…

Standardizing activations?

Basic idea:

$$z_i^{(1)} = W^{(1)} x_i + b^{(1)}$$

$$a_i^{(1)} = \text{ReLU}(z_i^{(1)})$$

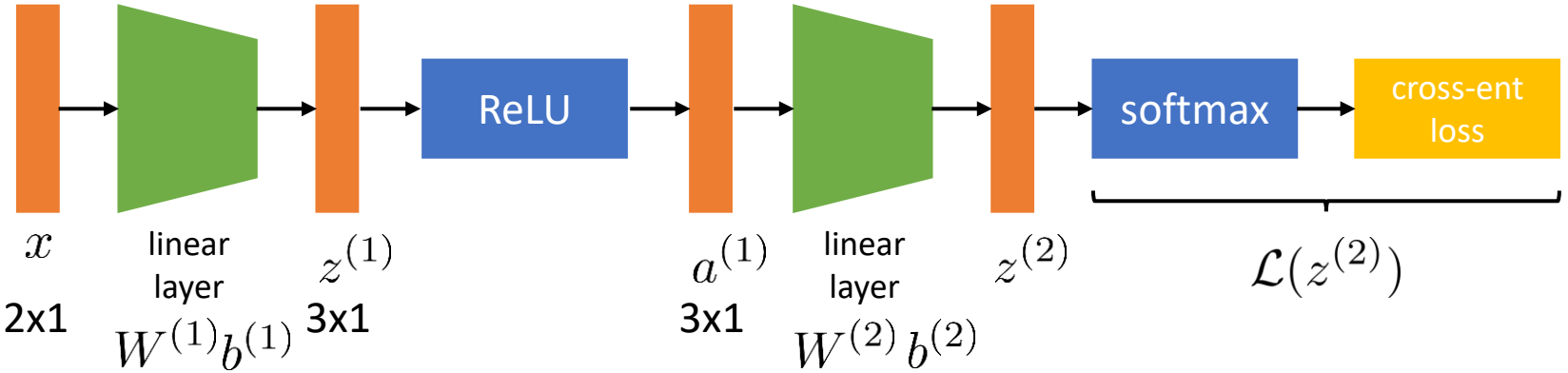
$$\mu^{(1)} = \frac{1}{N} \sum_{i=1}^{N} a_i^{(1)}$$

$$\sigma^{(1)} = \sqrt{\frac{1}{N} \sum_{i=1}^{N} (a_i^{(1)} - \mu^{(1)})^2}$$

$$\bar{a}_i^{(1)} = \frac{a_i^{(1)} - \mu^{(1)}}{\sigma^{(1)}}$$

$$z_i^{(2)} = W^{(2)} \bar{a}_i^{(1)} + b^{(2)}$$

etc…



these depend on $W^{(1)}$ and $b^{(1)}$
need to recompute them every gradient step!

This seems very expensive, since we don't want to evaluate all points in the dataset every gradient step

Batch normalization (basic version)

Basic idea:

$$z_i^{(1)} = W^{(1)} x_i + b^{(1)}$$

$$a_i^{(1)} = \mathrm{ReLU}(z_i^{(1)})$$

$$\mu^{(1)} = \frac{1}{N}\sum_{i=1}^{N} a_i^{(1)}$$

$$\sigma^{(1)} = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(a_i^{(1)} - \mu^{(1)})^2}$$

$$\bar{a}_i^{(1)} = \frac{a_i^{(1)} - \mu^{(1)}}{\sigma^{(1)}}$$

$$z_i^{(2)} = W^{(2)} \bar{a}_i^{(1)} + b^{(2)}$$

etc…

This seems very expensive, since we don't want to evaluate all points in the dataset every gradient step

$$\mu^{(1)} \approx \frac{1}{B}\sum_{j=1}^{B} a_{i_j}^{(1)}$$

$$\sigma^{(1)} = \sqrt{\frac{1}{B}\sum_{j=1}^{B}(a_{i_j}^{(1)} - \mu^{(1)})^2}$$

compute mean and std only over the current **batch**

Batch normalization (real version)

Basic idea:

$$z_i^{(1)} = W^{(1)} x_i + b^{(1)}$$

$$a_i^{(1)} = \mathrm{ReLU}(z_i^{(1)})$$

$$\mu^{(1)} = \frac{1}{N}\sum_{i=1}^{N} a_i^{(1)}$$

$$\sigma^{(1)} = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(a_i^{(1)} - \mu^{(1)})^2}$$

$$\bar{a}_i^{(1)} = \frac{a_i^{(1)} - \mu^{(1)}}{\sigma^{(1)}}$$

$$z_i^{(2)} = W^{(2)} \bar{a}_i^{(1)} + b^{(2)}$$

etc…

This seems very expensive, since we don't want to evaluate all points in the dataset every gradient step

$$\mu^{(1)} \approx \frac{1}{B}\sum_{j=1}^{B} a_{i_j}^{(1)}$$

$$\sigma^{(1)} \approx \sqrt{\frac{1}{B}\sum_{j=1}^{B}(a_{i_j}^{(1)} - \mu^{(1)})^2}$$

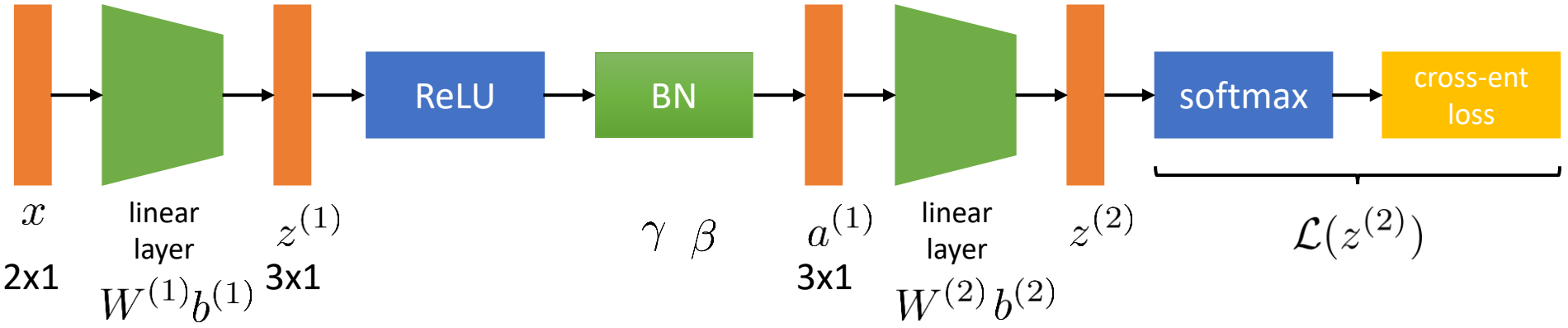
compute mean and std only over the current **batch**

$$\bar{a}_i^{(1)} = \frac{a_i^{(1)} - \mu^{(1)}}{\sigma^{(1)}}\gamma + \beta$$

learnable scale and bias

same dim as $\bar{a}_i^{(1)}$

Batch normalization “layer”



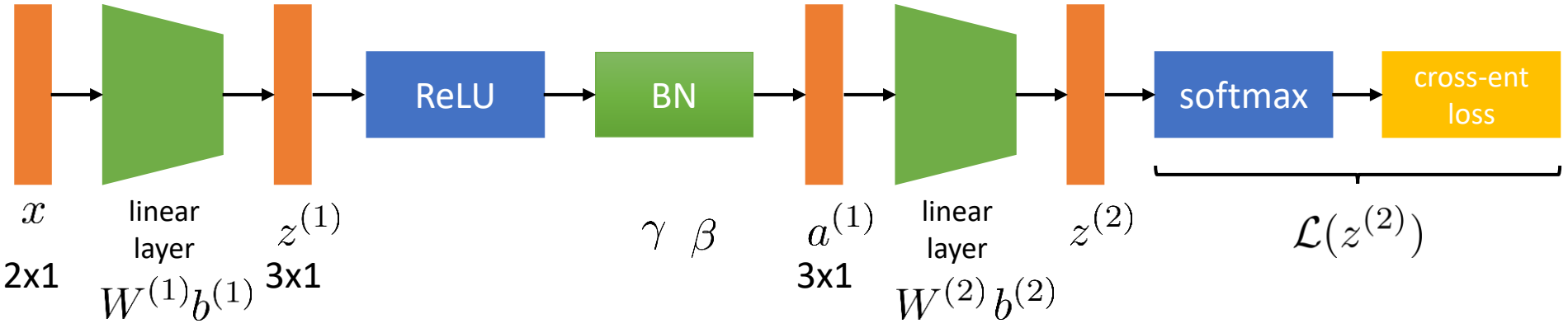
$$\mu^{(1)} \approx \frac{1}{B}\sum_{j=1}^{B} a_{i_j}^{(1)} \qquad \sigma^{(1)} \approx \sqrt{\frac{1}{B}\sum_{j=1}^{B}(a_{i_j}^{(1)} - \mu^{(1)})^2} \qquad \bar{a}_i^{(1)} = \frac{a_i^{(1)} - \mu^{(1)}}{\sigma^{(1)}}\gamma + \beta$$

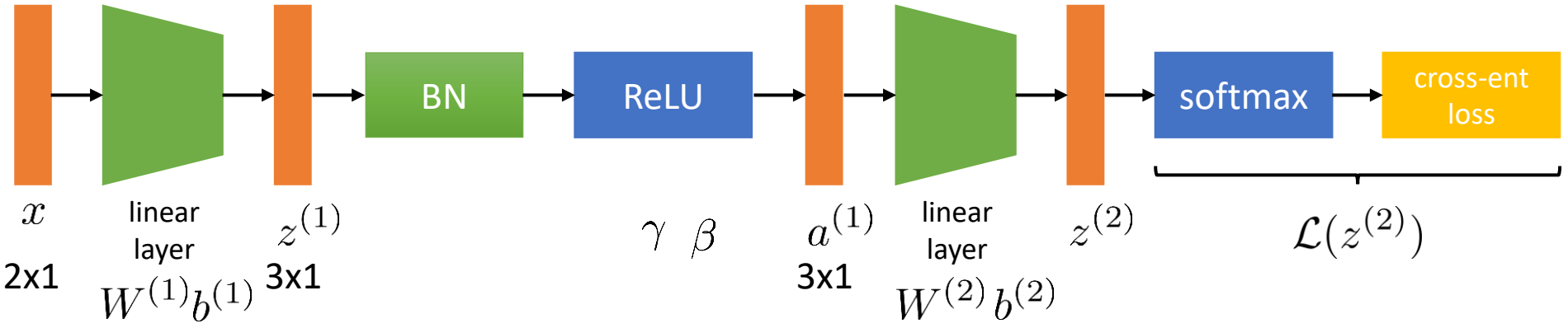
How to train?

Just use backpropagation!

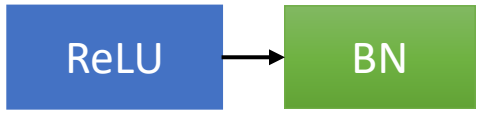
Exercise: figure out the derivatives w.r.t. parameters and input!

Where to put back normalization?

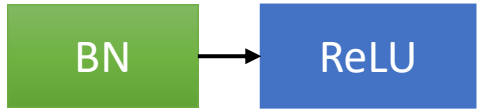




Where to put back normalization?



- Scale and bias seemingly should be subsumed by next linear layer?
- All ReLU outputs are positive

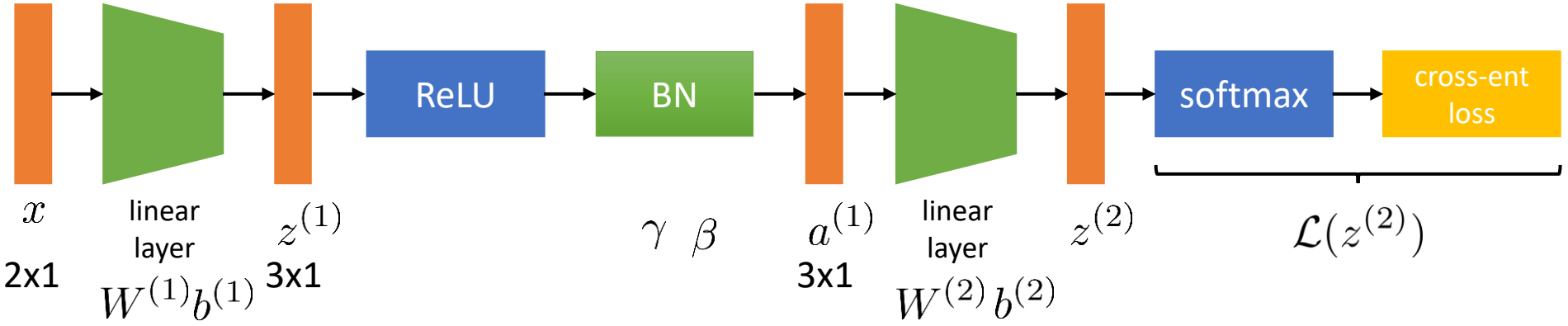


- The “classic” version
- Just appears to be a transformation on the preceding linear layer?

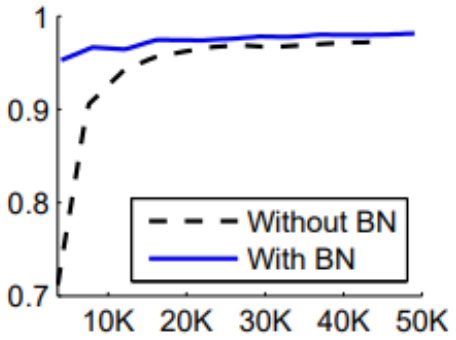
No one seems to agree on what the right way to do it is, try a few options and see what works (but both often work)

A few considerations about batch norm

$$\mu^{(1)} \approx \frac{1}{B}\sum_{j=1}^{B} a_{i_j}^{(1)} \qquad \sigma^{(1)} \approx \sqrt{\frac{1}{B}\sum_{j=1}^{B}(a_{i_j}^{(1)} - \mu^{(1)})^2} \qquad \bar{a}_i^{(1)} = \frac{a_i^{(1)} - \mu^{(1)}}{\sigma^{(1)}}\gamma + \beta$$

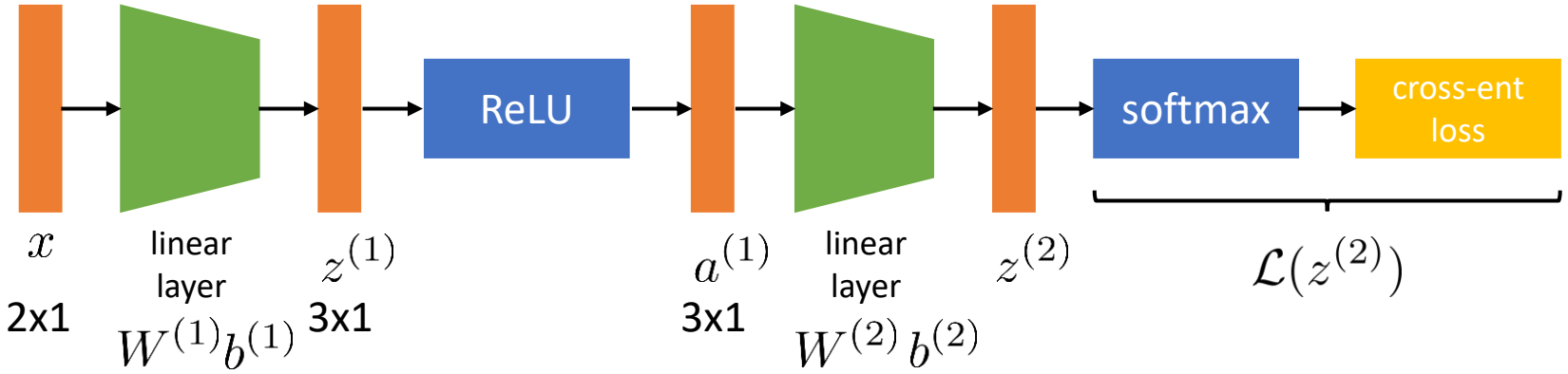


- Often we can use a **larger** learning rate with batch norm
- Models with batch norm can train **much** faster
- Generally requires less regularization (e.g., doesn't need dropout)
- Very good idea in many cases



Weight initialization

General themes



- We want the overall **scale** of activations in the network not to be **too big** or **too small** for our initial (randomized) weights, so that the gradients propagate well
- **Basic initialization methods:** ensure that activations are on a reasonable scale, and the scale of activations doesn't grow or shrink in later layers as we increase the number of layers
- **More advanced initialization methods:** try to do something about eigenvalues of Jacobians

$$\frac{d\mathcal{L}}{dW^{(1)}} = \frac{dz^{(1)}}{dW^{(1)}} \frac{da^{(1)}}{dz^{(1)}} \frac{dz^{(2)}}{da^{(1)}} \frac{d\mathcal{L}}{dz^{(2)}}$$

$$\frac{d\mathcal{L}}{dW^{(1)}} = J_1 J_2 J_3 \dots J_n \frac{d\mathcal{L}}{dz^{(n)}}$$

If we multiply many many numbers together, what will we get?

If most of the numbers are < 1, we get 0

If most of the numbers are > 1, we get infinity

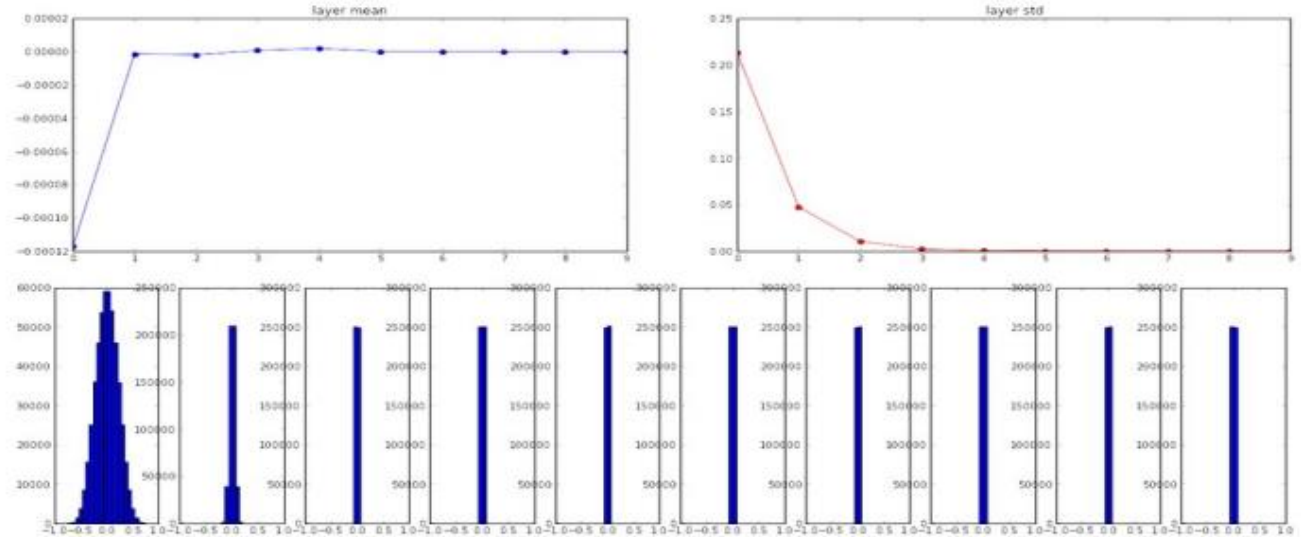
We only get a reasonable answer if the numbers are all close to 1!

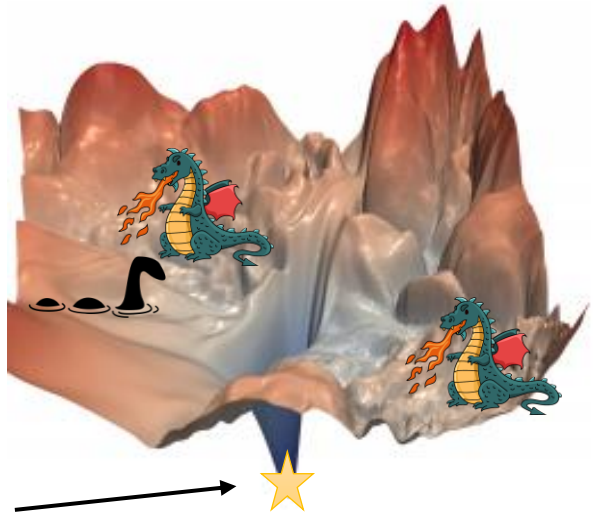
Basic initialization

Simple choice: Gaussian random weights

$$W_{jk}^{(i)} \sim \mathcal{N}(0, 0.0001)$$

```
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
```





Ideally we could just initialize here

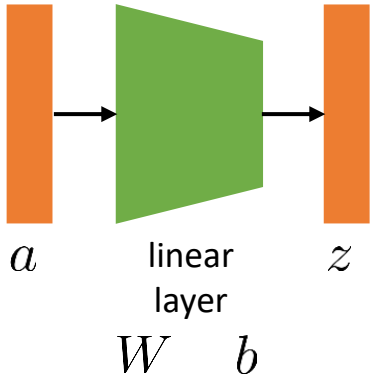
But we have no idea where that is!

Goal is **not** to start at a good solution, but to have well-behaved gradients & activations

Why is this bad? $$\frac{d\mathcal{L}}{dW^{(i)}} = \frac{dz^{(i)}}{dW^{(i)}} \frac{d\mathcal{L}}{dz^{(i)}} = \delta a^{(i-1)^T}$$

Image from: Fei-Fei Li & Andrej Karpathy

Basic initialization



$$W_{ij} \sim \mathcal{N}(0, \sigma_W^2)$$

$$b_i \approx 0$$

reasonable choice!
if we standardize x, then

$$x \sim \mathcal{N}(0, 1)$$

what is (roughly) the magnitude of z_i?

$$z_i = \sum_j W_{ij} a_j + \cancel{b_i} \quad b_i \approx 0$$

assume $a_j \sim \mathcal{N}(0, \sigma_a)$

everything is (roughly) 0-mean

$$E[z_i^2] = \sum_j E[W_{ij}^2] E[a_j^2] = D_a \sigma_W^2 \sigma_a^2$$

dimensionality of a

if $D_a \sigma_W^2 > 1$, magnitude grows with each layer!

if $D_a \sigma_W^2 < 1$, magnitude shrinks with each layer!

what if we choose $\sigma_W^2 = 1/D_a$?

Basic initialization

$$E[z_i^2] = \sum_j E[W_{ij}^2]E[a_j^2] = D_a \sigma_W^2 \sigma_a^2$$

dimensionality of a

if $D_a\sigma_W^2 > 1$, magnitude grows with each layer!

if $D_a\sigma_W^2 < 1$, magnitude shrinks with each layer!

what if we choose $\sigma_W^2 = 1/D_a$?

basic principle: get std of W_{ij} to be about $1/\sqrt{D_a}$ this sometimes referred to as "Xavier initialization"

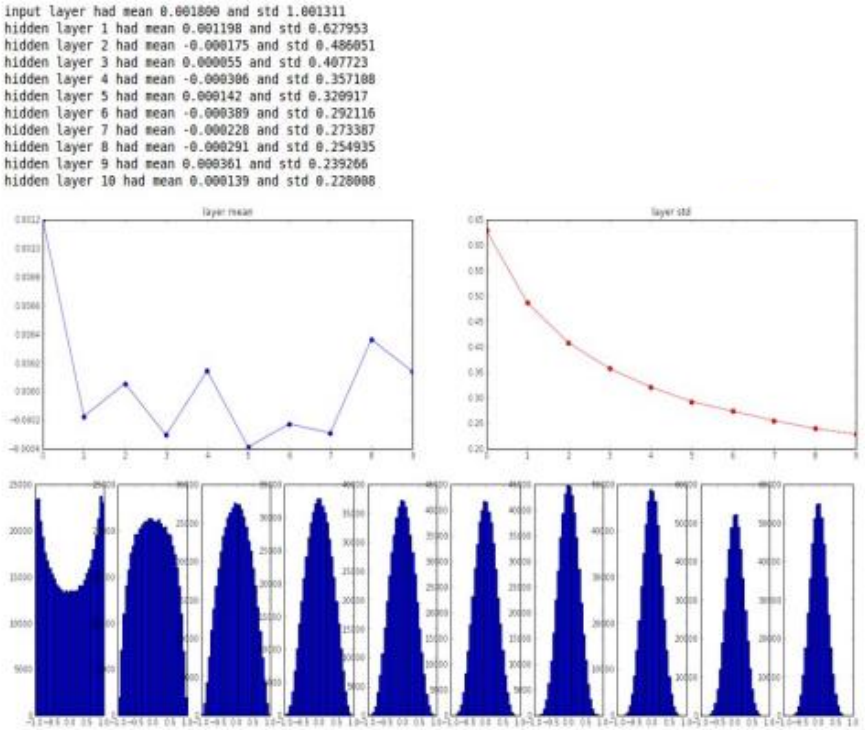


Image from: Fei-Fei Li & Andrej Karpathy

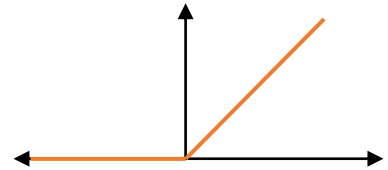
Little detail: ReLUs

$$E[z_i^2] = \sum_j E[W_{ij}^2]E[a_j^2] = D_a \sigma_W^2 \sigma_a^2$$

basic principle: get std of W_{ij} to be about $1/\sqrt{D_a}$

This was all without nonlinearities!

problem: $a_j = \mathrm{ReLU}(z_j)$



"negative half" of 0-mean activations is removed!

variance is cut in half!

might not seem like much…

but it adds up!

```
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 0.148299
hidden layer 6 had mean 0.072234 and std 0.103288
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
```

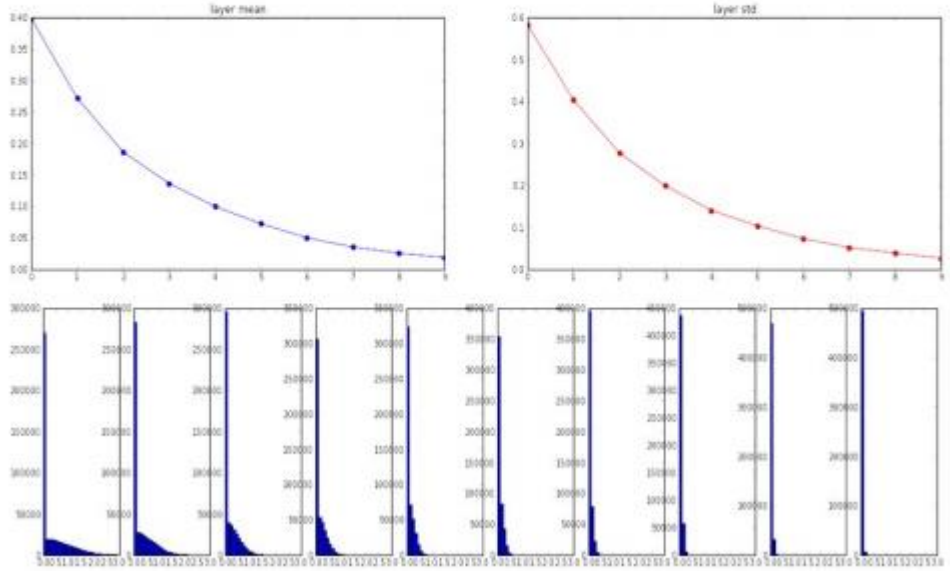


Image from: Fei-Fei Li & Andrej Karpathy

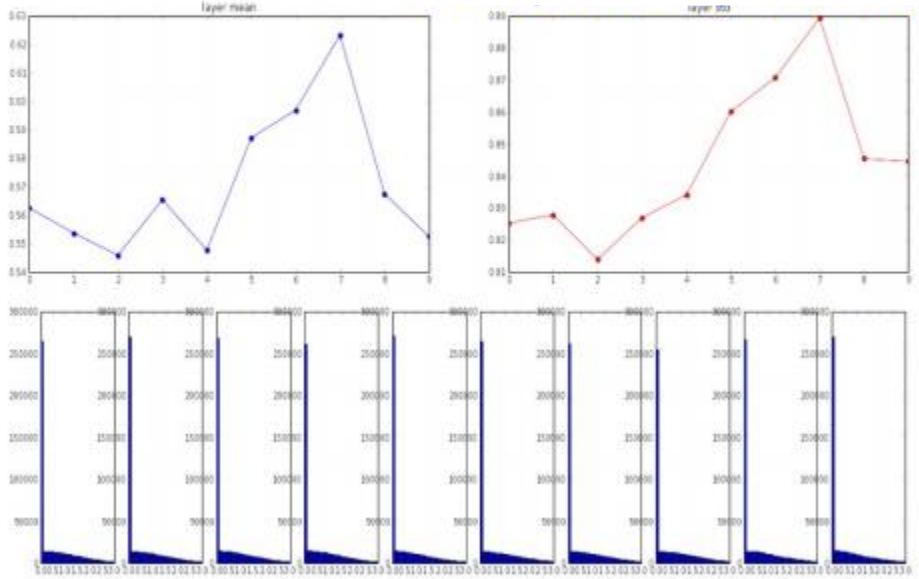
Little detail: ReLUs

$$E[z_i^2] = \sum_j E[W_{ij}^2]E[a_j^2] = D_a \sigma_W^2 \sigma_a^2$$

basic principle: get std of W_{ij} to be about ~~$1/\sqrt{D_a}$~~

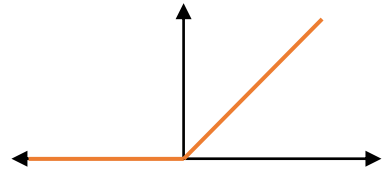
$$1/\sqrt{\tfrac{1}{2}D_a}$$

```
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 0.860835
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
```



This was all without nonlinearities!

problem: $a_j = \mathrm{ReLU}(z_j)$



"negative half" of 0-mean activations is removed!

variance is cut in half!

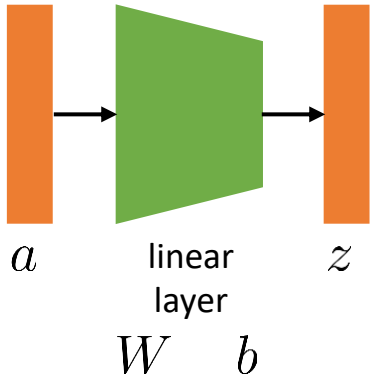
might not seem like much…

proposed by He et al. for ResNet

makes big difference 150+ layers…

Image from: Fei-Fei Li & Andrej Karpathy

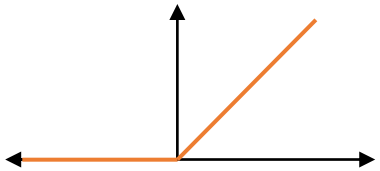
Littler detail: ReLUs & biases



$$W_{ij} \sim \mathcal{N}(0, \sigma_W^2)$$

$$b_i \approx 0$$

problem: $a_j = \mathrm{ReLU}(z_j)$



half of our units (on average) will be “dead”!

often initialize $b_i = 0.1$ (or small constant)

Advanced initialization

$$\frac{d\mathcal{L}}{dW^{(1)}} = \frac{dz^{(1)}}{dW^{(1)}}\frac{da^{(1)}}{dz^{(1)}}\frac{dz^{(2)}}{da^{(1)}}\frac{d\mathcal{L}}{dz^{(2)}}$$

$$\frac{d\mathcal{L}}{dW^{(1)}} = J_1 J_2 J_3 \dots J_n \frac{d\mathcal{L}}{dz^{(n)}}$$

If we multiply many many numbers together, what will we get?

If most of the numbers are < 1, we get 0

If most of the numbers are > 1, we get infinity

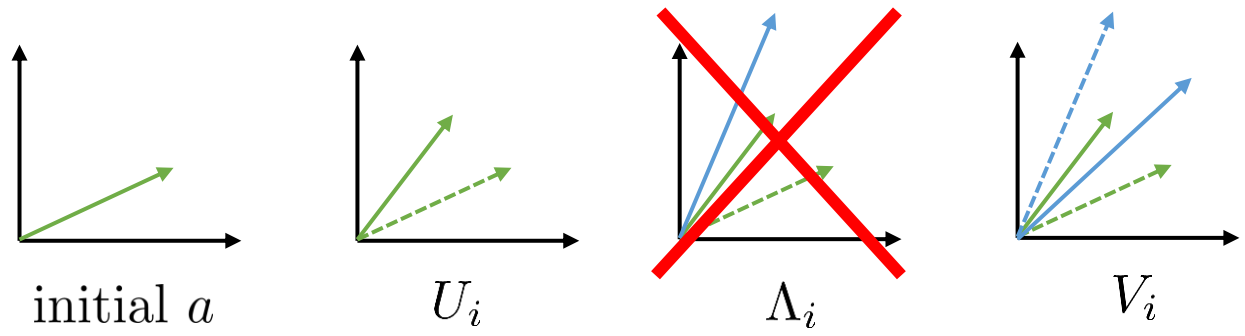
We only get a reasonable answer if the numbers are all close to 1!

for each J_i, we can write: $J_i = U_i \Lambda_i V_i$

e.g., using singular value decomposition

scale-preserving transformations (i.e., orthonormal bases)

diagonal matrix with same eigenvalues as J_i



Advanced initialization

$$\frac{d\mathcal{L}}{dW^{(1)}} = \frac{dz^{(1)}}{dW^{(1)}}\frac{da^{(1)}}{dz^{(1)}}\frac{dz^{(2)}}{da^{(1)}}\frac{d\mathcal{L}}{dz^{(2)}}$$

$$\frac{d\mathcal{L}}{dW^{(1)}} = J_1 J_2 J_3 \dots J_n \frac{d\mathcal{L}}{dz^{(n)}}$$

If we multiply many many numbers together, what will we get?

If most of the numbers are < 1, we get 0

If most of the numbers are > 1, we get infinity

We only get a reasonable answer if the numbers are all close to 1!

for each $W^{(i)}$, we can write: $W^{(i)} = U^{(i)}\Lambda^{(i)}V^{(i)}$ e.g., using singular value decomposition

$W^{(i)} \leftarrow U^{(i)}V^{(i)}$ just need to force this to be identity matrix

even simpler:

```
a = get_rng().normal(0.0, 1.0, flat_shape)
u, _, v = np.linalg.svd(a, full_matrices=False)
# pick the one with the correct shape
q = u if u.shape == flat_shape else v
```

arbitrary random matrix (doesn't really matter how)

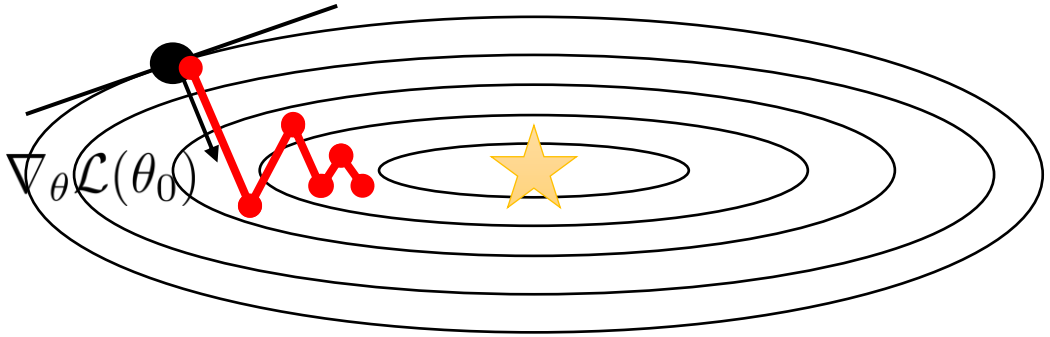
needed if non-square

guaranteed orthonormal

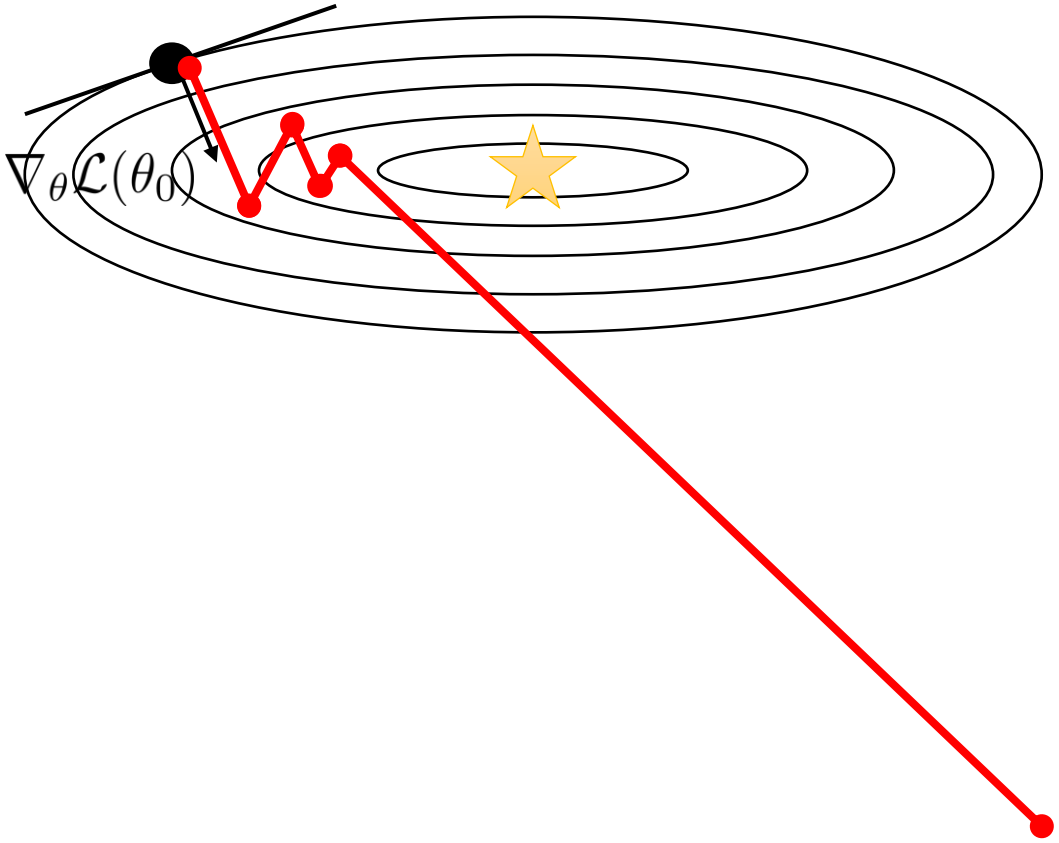
https://github.com/Lasagne/Lasagne

Last bit: Gradient clipping

what we hope happens:



what actually happens:
because deep learning, that's why



- Took a step that was too big in the wrong place
- Something got divided by something small (e.g., in batch norm, softmax, etc.)
- Just got really unlucky

Clipping the monster gradients

per-element clipping:

$$\bar{g}_i \leftarrow \max(\min(g_i, c_i), -c_i)$$

norm clipping:

$$\bar{g}_i \leftarrow g\frac{\min(||g||, c)}{||g||}$$

how to choose c?

run a few epochs (assuming it doesn't explode)
see what "healthy" magnitudes look like

Ensembles & dropout

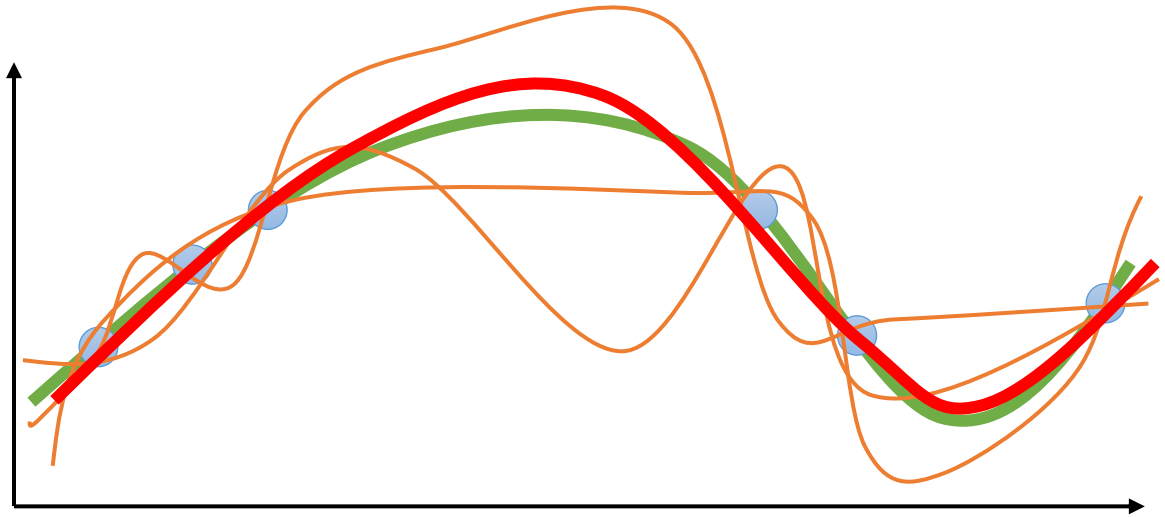
What if my model makes a mistake?

Problem: neural networks have many parameters, often have high variance

Not **nearly** as high as we would expect from basic learning theory
(i.e., overfitting is usually **not** catastrophic), but still…

Interesting idea: when we have multiple high-variance learners, maybe they'll **agree** on the right answer, but **disagree** on the wrong answer

Said another way: there are many more ways to be wrong than to be right



Ensembles in **theory**

$$\text{Variance} = E_{\mathcal{D}\sim p(\mathcal{D})}[||f_{\mathcal{D}}(x) - \bar{f}(x)||^2]$$

$$\bar{f}(x) = E_{\mathcal{D}\sim p(\mathcal{D})}[f_{\mathcal{D}}(x)] \approx \frac{1}{M}\sum_{i=1}^{M} f_{\mathcal{D}_j}(x)$$

can we actually estimate this thing?

where do we get **M** different datasets??

Can we **cook up** multiple independent datasets from a single one?

Simple approach: just chop a big dataset into M ~~non overlapping~~ overlapping but **independently sampled** parts

turns out we actually don't need this!

$\mathcal{D} = \{(x_i, y_i)\}$

for each $\mathcal{D}_j$ pick N indices randomly in $\{1, ..., N\}$ $i_{j,1}, ...i_{j,N}$

$\mathcal{D}_j = \{(x_{i_{j,1}}, y_{i_{j,1}})(x_{i_{j,2}}, y_{i_{j,2}}), ..., (x_{i_{j,N}}, y_{i_{j,N}})\}$

Ensembles in **theory**

$\mathcal{D} = \{(x_i, y_i)\}$

for each $\mathcal{D}_j$ pick N indices randomly in $\{1, ..., N\}$ $i_{j,1}, ...i_{j,N}$

$\mathcal{D}_j = \{(x_{i_{j,1}}, y_{i_{j,1}})(x_{i_{j,2}}, y_{i_{j,2}}), ..., (x_{i_{j,N}}, y_{i_{j,N}})\}$

This is called resampling **with replacement**

$\mathcal{D}$ x_1 x_2 x_3

$\mathcal{D}_1$ x_2 x_3 x_3  2  3  3

$\mathcal{D}_2$ x_3 x_1 x_3  3  1  3

train separate models on each $\mathcal{D}_j$

Ensembles in **theory**

$\mathcal{D} = \{(x_i, y_i)\}$

for each $\mathcal{D}_j$ pick N indices randomly in $\{1, ..., N\}$ $i_{j,1}, ...i_{j,N}$

$\mathcal{D}_j = \{(x_{i_{j,1}}, y_{i_{j,1}})(x_{i_{j,2}}, y_{i_{j,2}}), ..., (x_{i_{j,N}}, y_{i_{j,N}})\}$

train separate models on each $\mathcal{D}_j$

$p_{\theta_1}(y|x), ..., p_{\theta_M}(y|x)$

how do we predict?

principled approach: average the probabilities:

$$p(y|x) = \frac{1}{M} \sum_{j=1}^{M} p_{\theta_j}(y|x)$$

simple approach: majority vote

Ensembles in **practice**

There is already a lot of randomness in neural network training

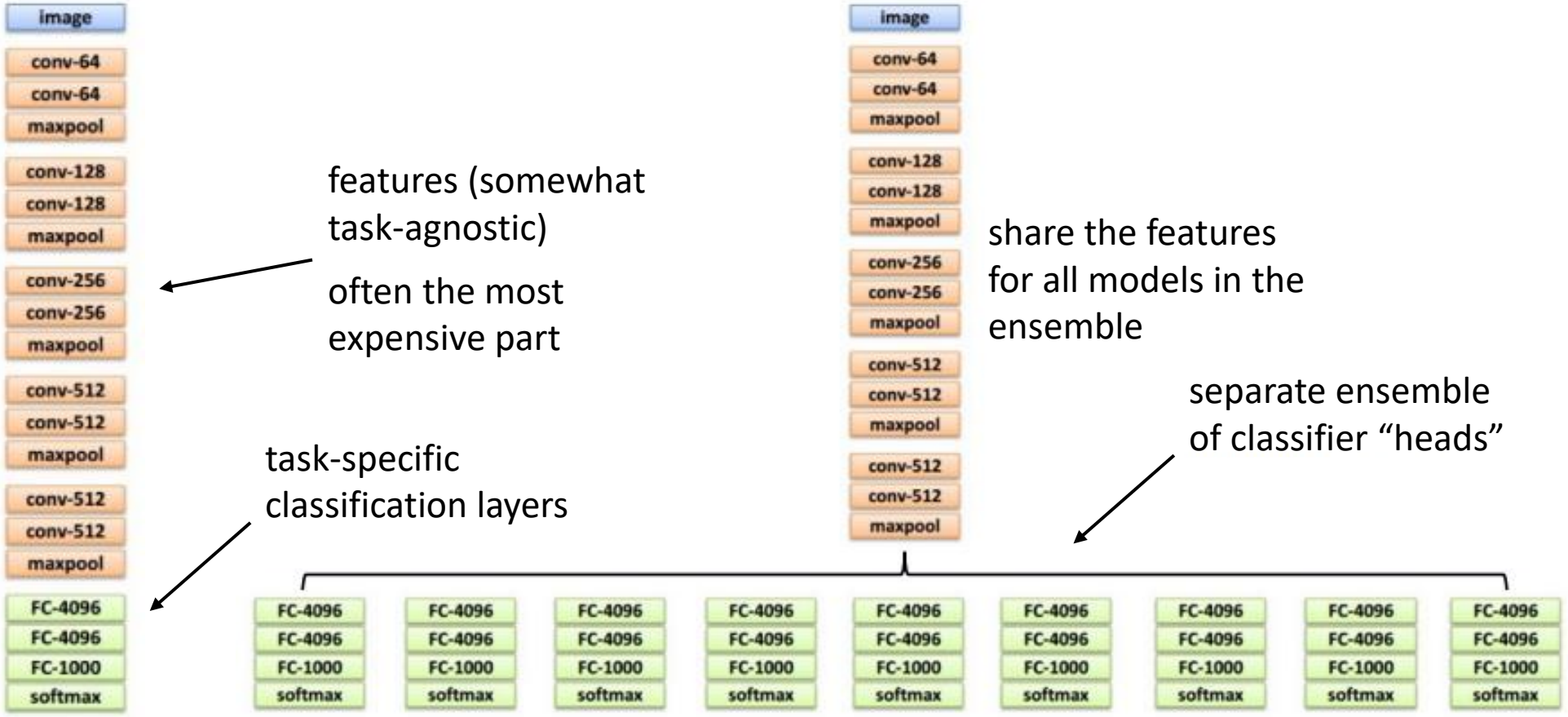
- Random initialization
- Random minibatch shuffling
- Stochastic gradient descent

In practice we get much of the same benefit **without** resampling

$$\text{train } M \text{ models } p_{\theta_j}(y|x) \text{ on the same } \mathcal{D}$$

$$p(y|x) = \frac{1}{M}\sum_{j=1}^{M} p_{\theta_j}(y|x) \qquad \text{or majority vote}$$

Even **faster** ensembles



Even faster**er** ensembles

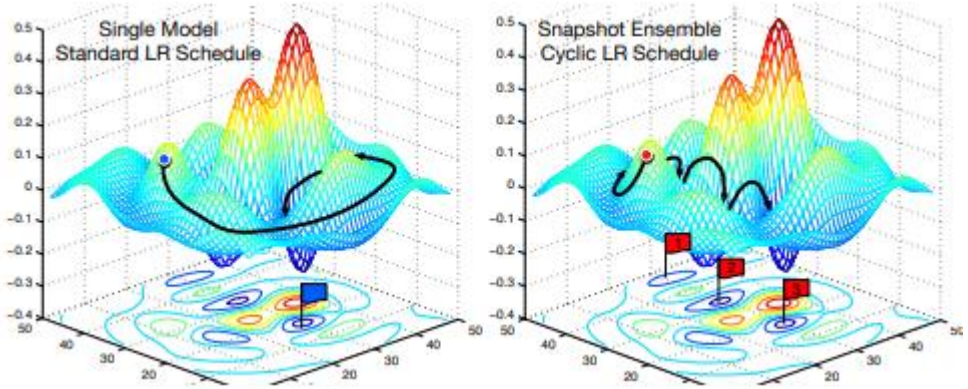
snapshot ensembles:

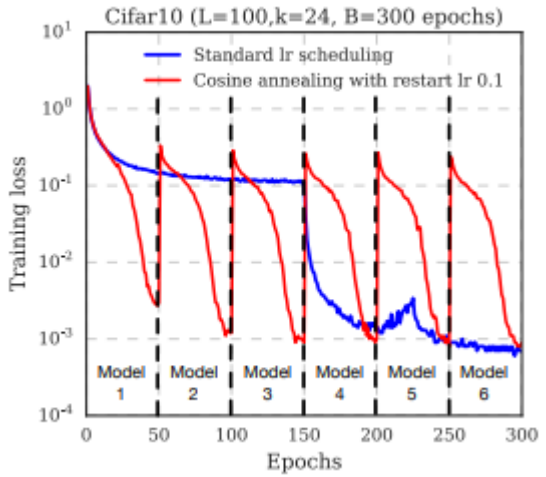
save out parameter snapshots over the course of SGD optimization, use each snapshot as a model in the ensemble

advantage: don't need to have a bunch of separate training runs

…but need to set things up carefully so that the snapshots are actually different

combining predictions: could average probabilities or vote, or **just average the parameter vectors together**





Huang et al., Snapshot Ensembles: Train 1, Get M For Free

Some comparisons

Model	Prediction method	Test Accuracy
Baseline (10 epochs)	Single model	0.837
True ensemble of 10 models	Average predictions	0.855
True ensemble of 10 models	Voting	0.851
Snapshots (25) over 10 epochs	Average predictions	0.865
Snapshots (25) over 10 epochs	Voting	0.861
Snapshots (25) over 10 epochs	Parameter averaging	0.864

your mileage may vary

Really really big ensembles?

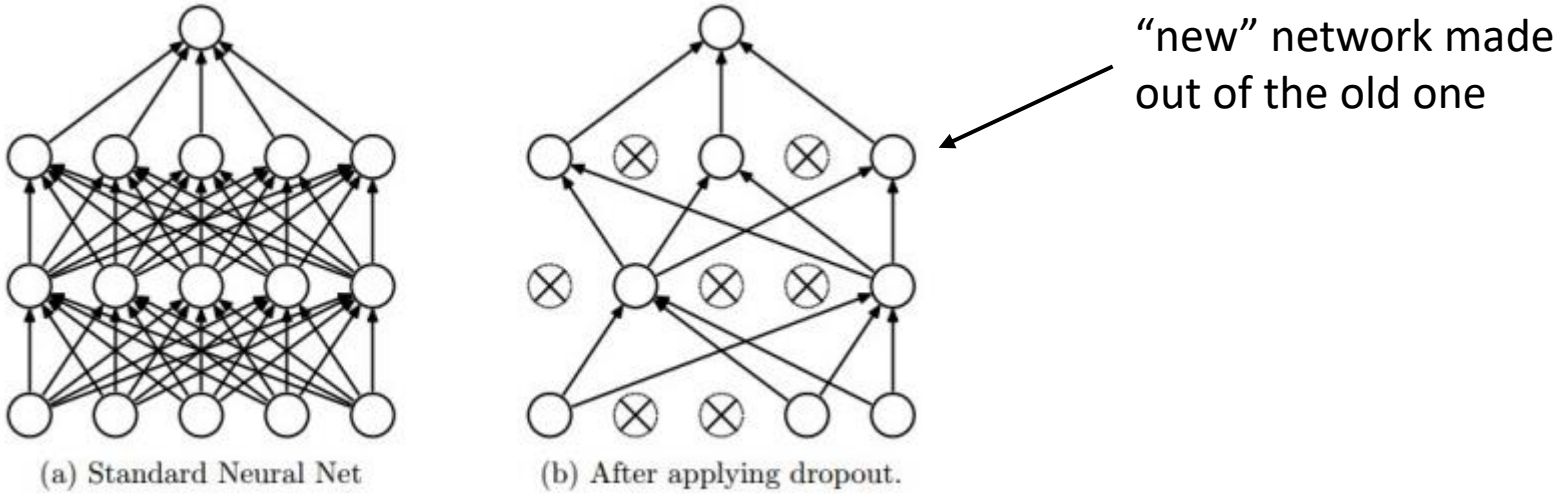
The bigger the ensemble is, the better it works (usually)

But making huge ensembles is expensive

Can we make multiple models **out of a single neural network**?

Dropout

randomly set some activations to zero in the forward pass



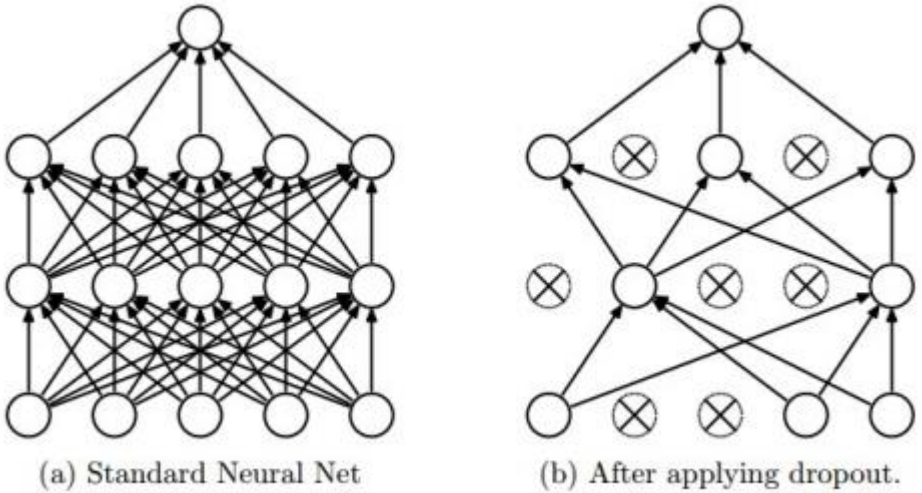
(a) Standard Neural Net

(b) After applying dropout.

"new" network made out of the old one

Dropout

randomly set some activations to zero in the forward pass



(a) Standard Neural Net (b) After applying dropout.

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

def train_step(X):
  """ X contains the data """

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

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

Andrej Karpathy

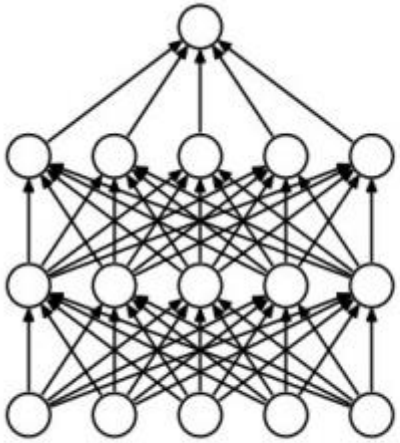
Implementation:

$$\text{for each } a_j^{(i)}, \text{ set it to } a_j^{(i)} m_{ij}$$

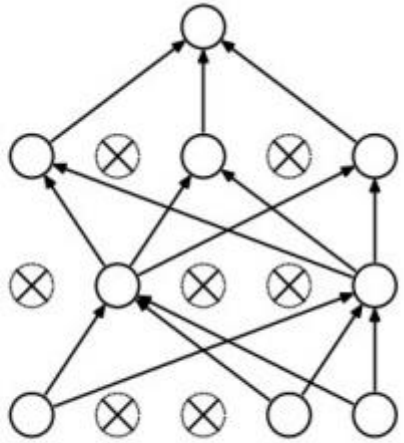
$$m_{ij} \sim \text{Bernoulli}(0.5)$$ 1.0 with probability 50%, 0.0 otherwise

Dropout

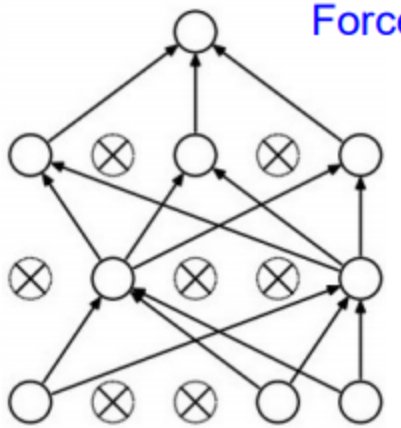
randomly set some activations to zero in the forward pass



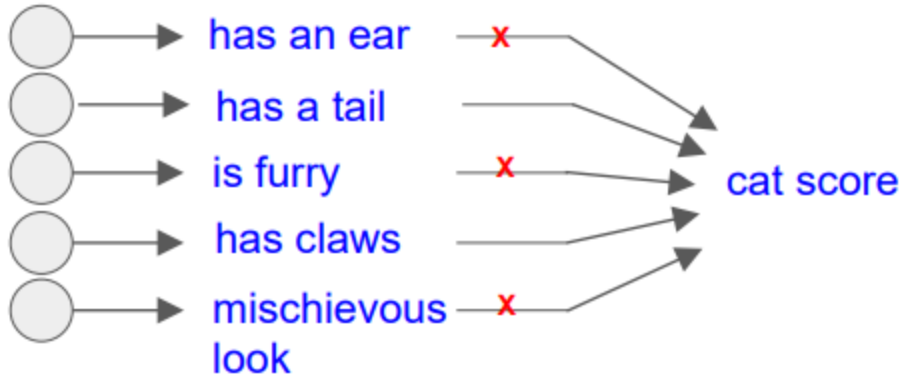
(a) Standard Neural Net



(b) After applying dropout.



Forces the network to have a redundant representation.



How could this possibly work?

Can think of every **dropout mask** as defining a different model

Hence this looks like a **huge** ensemble

How huge?

At test time…

During training:

$$\text{for each } a_j^{(i)}, \text{ set it to } a_j^{(i)} m_{ij}$$

$$m_{ij} \sim \text{Bernoulli}(0.5)$$

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

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

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

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

test time is unchanged!

Andrej Karpathy

At test time: want to combine all the models

could just generate many dropout masks

what if we stop dropping out at test time?

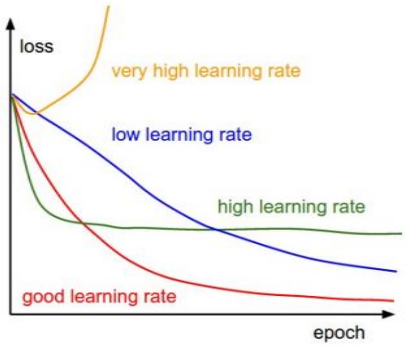
$$\text{before: on average } \tfrac{1}{2} \text{ of dimensions are forced to } 0$$

$$\text{now: none of them are, so } W^{(i)} a^{(i)} \text{ will be } \approx 2\times \text{ bigger}$$

$$\text{solution: } \bar{W}^{(i)} = \tfrac{1}{2} W^{(i)}$$ (divide all weights by 2!)

Hyperparameters

- With all these tricks, we have **a lot** of hyperparameters
- Some of these affect **optimization** (training)
  - Learning rate
  - Momentum
  - Initialization
  - Batch normalization
- Some of these affect **generalization** (validation)
  - Ensembling
  - Dropout
  - Architecture (# and size of layers)
- How do we pick these?
  - Recognize which is which: this can really matter!
    - Bad learning rate, momentum, initialization etc. shows up **very** early on in the training process
    - Effect of architecture usually only apparent after training is done
  - Coarse to fine: start with broad sweep, then zero in
  - Consider random hyperparameter search instead of grid



Example: short (5 epoch) log-space LR & weight decay sweep

```
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!

Andrej Karpathy

```
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

```
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)
```