Introduction to Computer Vision

Lecture 6 - Deep Learning III

Prof. He Wang

Logistics

- Assignment 1: to release on 3/14, due on 3/29 11:59PM (this Saturday)
  - Implementing convolution operation
  - Canny edge detector
  - Harris corner detector
  - Plane fitting using RANSAC

- Some functions are required to be implemented without for loop.

- If 1 day (0 - 24 hours) past the deadline, 15% off
- If 2 day (24 - 48 hours) past the deadline, 30% off
- Zero credit if more than 2 days.

Outline

- Set up the task
- Prepare the data —> Need a labeled dataset.
- Built a model —> construct your neural network
- Decide the fitting/training objective —> Loss function
- Perform fitting —> Training by running optimization
- Testing —> Evaluating on test data

CNN Training

To Train a CNN

Mini-batch SGD

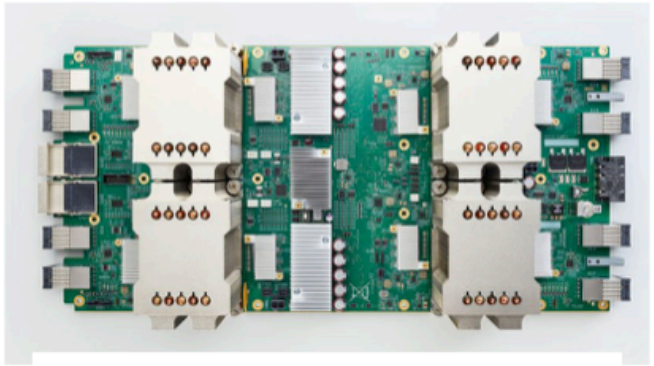
Loop:

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

Slide credit: Stanford CS231N

To Train a CNN

Hardware + Software



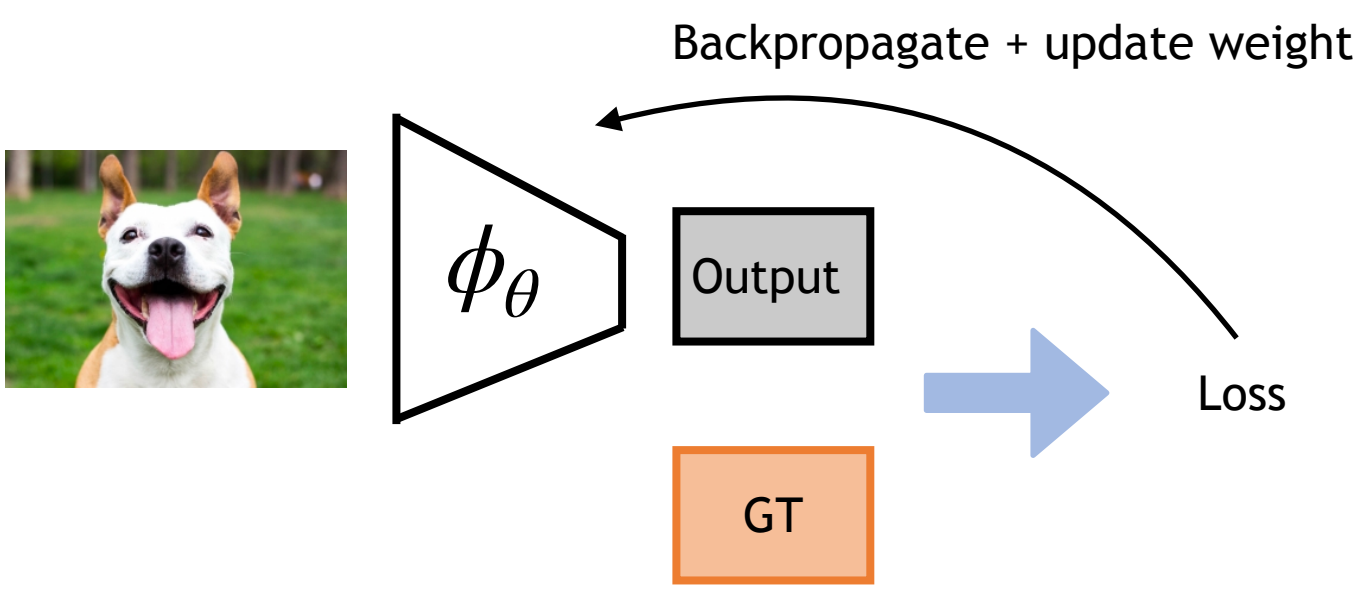
PyTorch

TensorFlow

Slide credit: Stanford CS231N

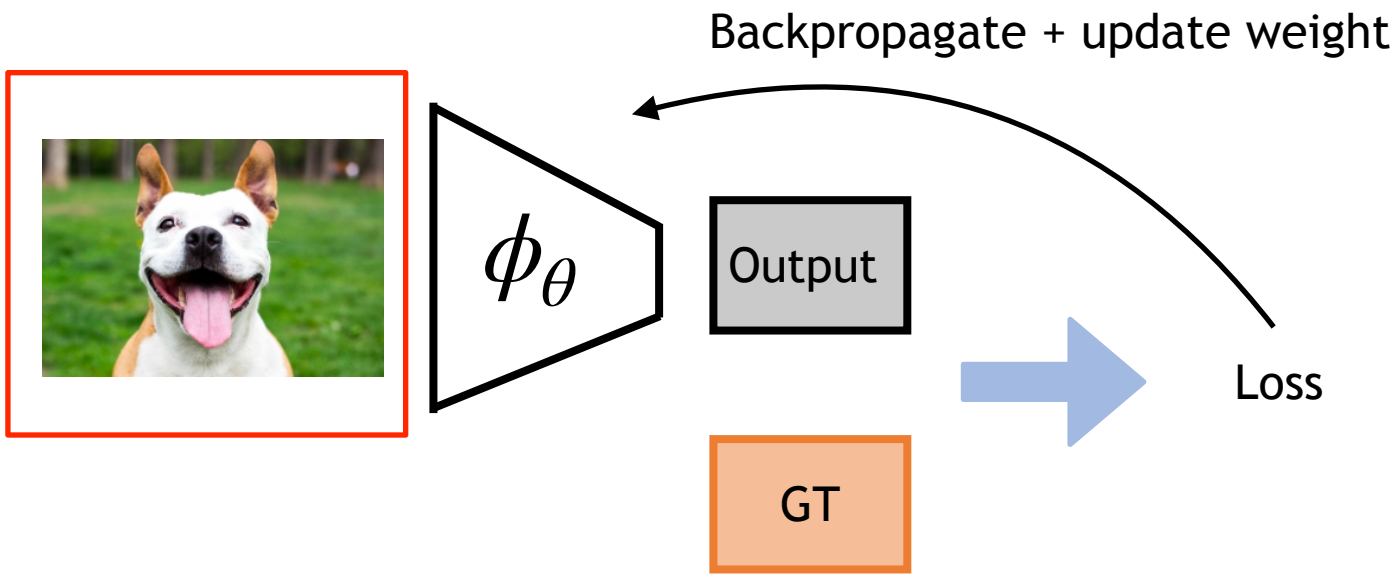
Outline

- Data preparation
- Weight Initialization
- Set a loss function
- Start optimization
  - optimizer?
  - learning rate?

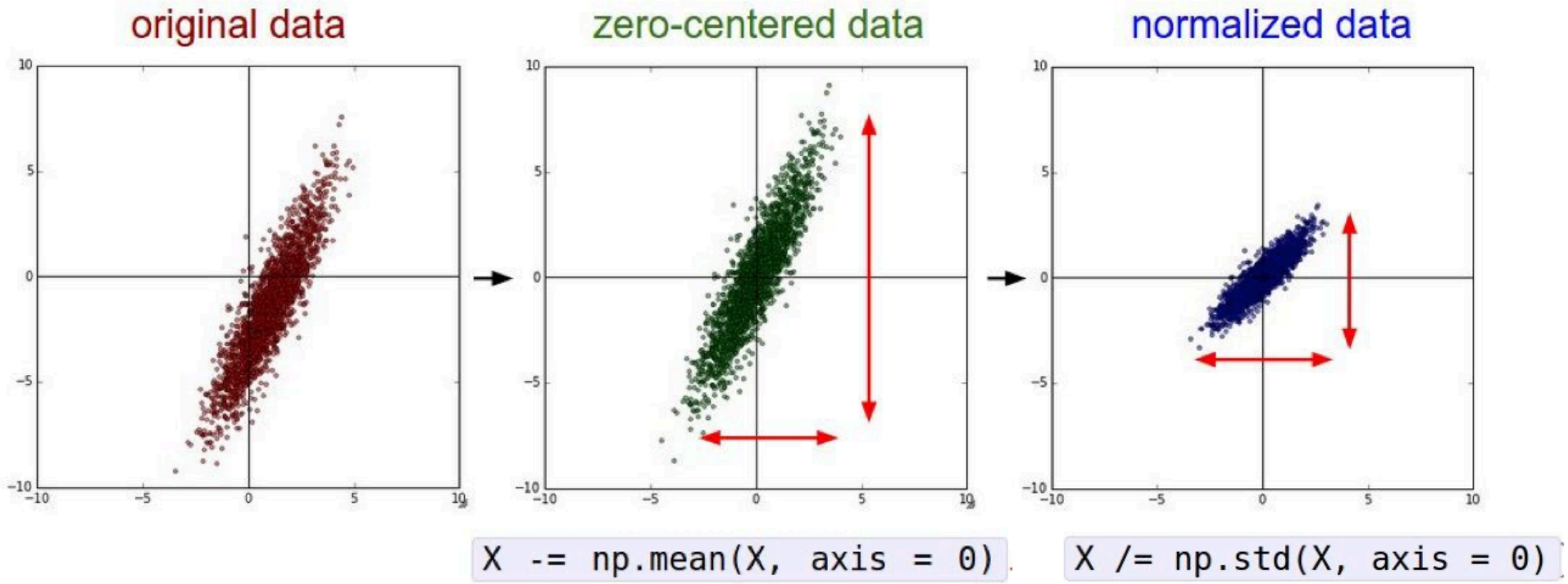


Outline

- Data preparation
- Weight Initialization
- Set a loss function
- Start optimization
  - optimizer?
  - learning rate?



Data Preprocessing



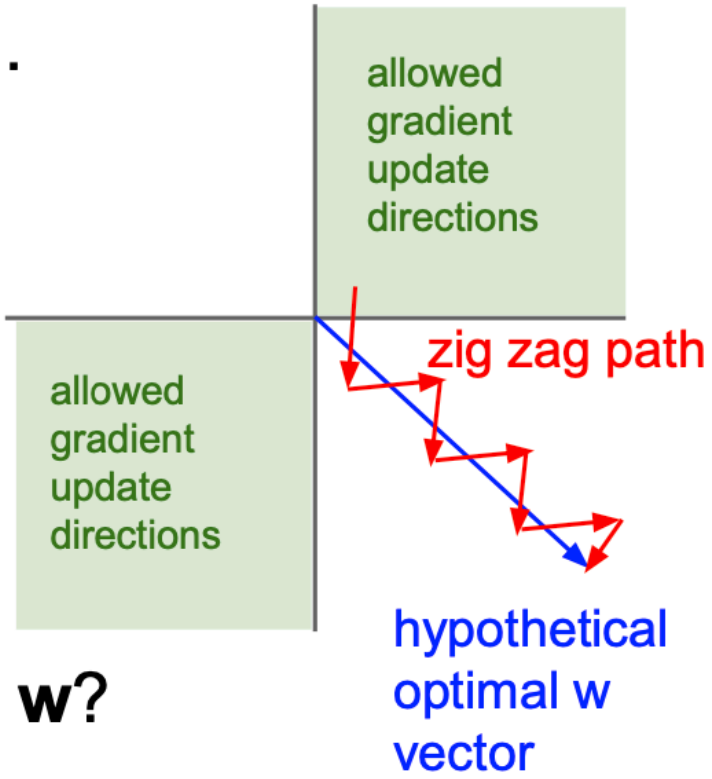
(Assume X [NxD] is data matrix,
each example in a row)

Slide credit: Stanford CS231N

Data Preprocessing

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

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



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

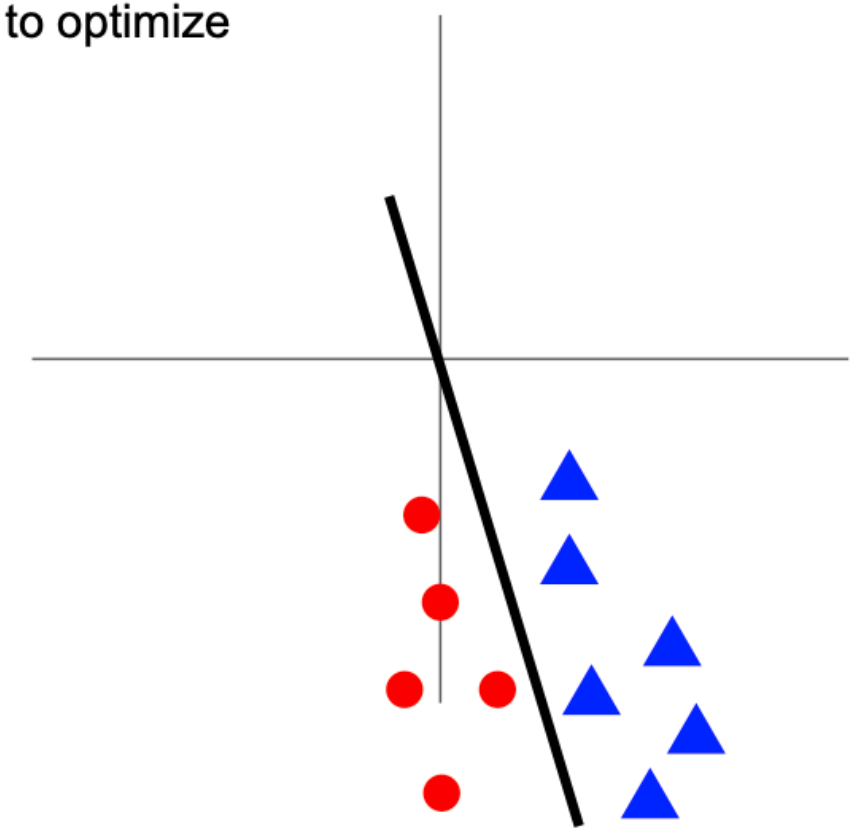
Always all positive or all negative :(

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

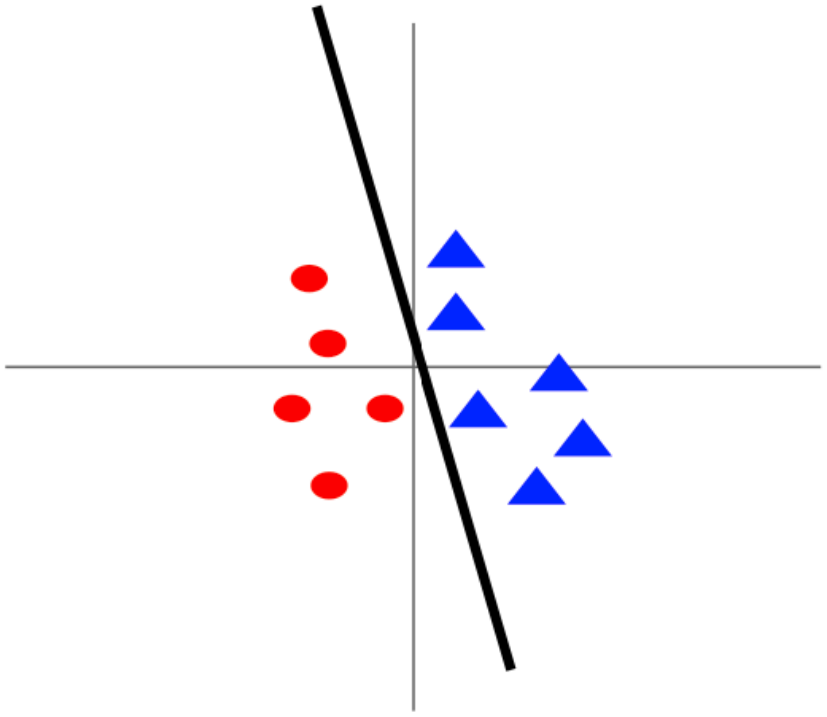
Slide credit: Stanford CS231N

Data Preprocessing

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



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



Slide credit: Stanford CS231N

Summary of Data Preprocessing

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

- Subtract per-channel mean (e.g. VGGNet)
  (mean along each channel = 3 numbers)
- Subtract per-channel mean and
  Divide by per-channel std (e.g. ResNet)
  (mean along each channel = 3 numbers)

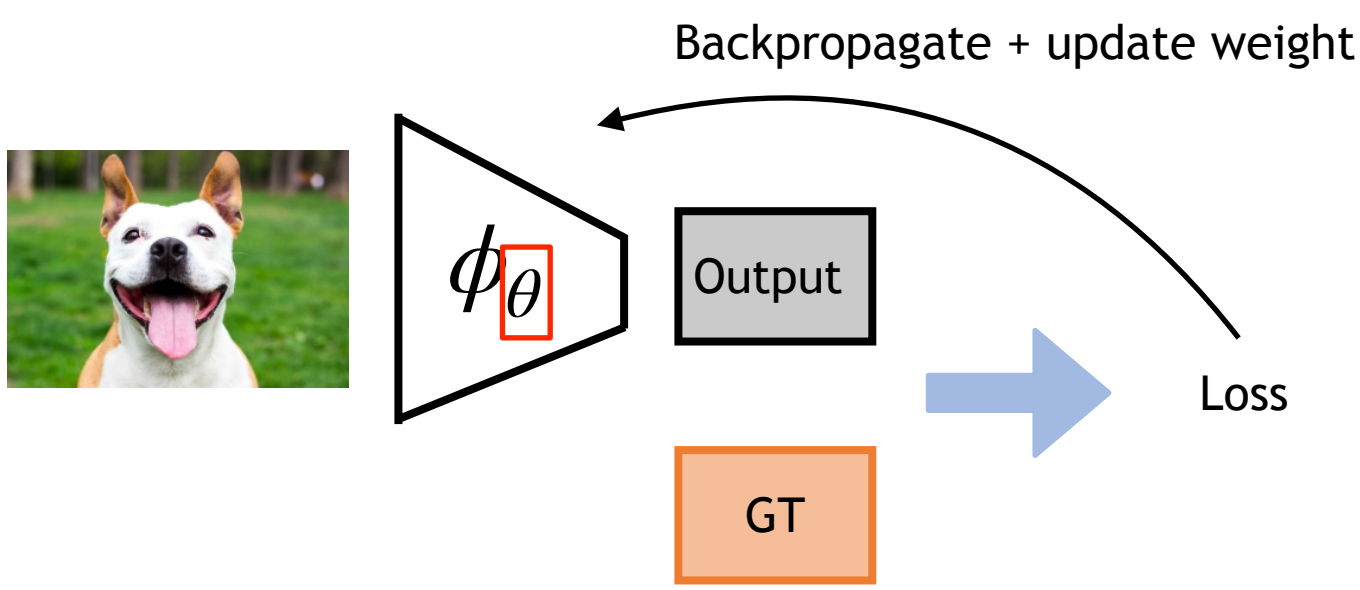
Slide credit: Stanford CS231N

Outline

- Data preparation
- Weight Initialization
- Set a loss function
- Start optimization
  - optimizer?
  - learning rate?

Outline

- Data preparation
- **Weight Initialization**
- Set a loss function
- Start optimization
  - optimizer?
  - learning rate?



Weight Initialization

- First idea: **Small random numbers**

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

```
W = 0.01 * np.random.randn(Din, Dout)
```

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

Slide credit: Stanford CS231N

Weight Initialization

```
dims = [4096] * 7
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = 0.01 * np.random.randn(Din, Dout)
    x = np.tanh(x.dot(W))
    hs.append(x)
```

Forward pass for a 6-layer net with hidden size 4096

What will happen to the activations for the last layer?

Slide credit: Stanford CS231N

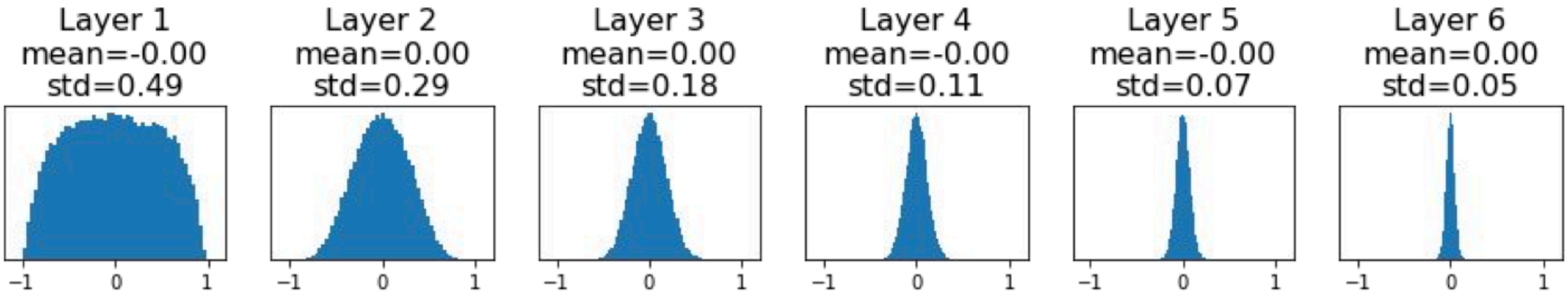
Weight Initialization: Activation Statistics

```
dims = [4096] * 7
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = 0.01 * np.random.randn(Din, Dout)
    x = np.tanh(x.dot(W))
    hs.append(x)
```

Forward pass for a 6-layer net with hidden size 4096

All activations tend to zero for deeper network layers

Q: What do the gradients dL/dW look like?



Slide credit: Stanford CS231N

Weight Initialization: Activation Statistics

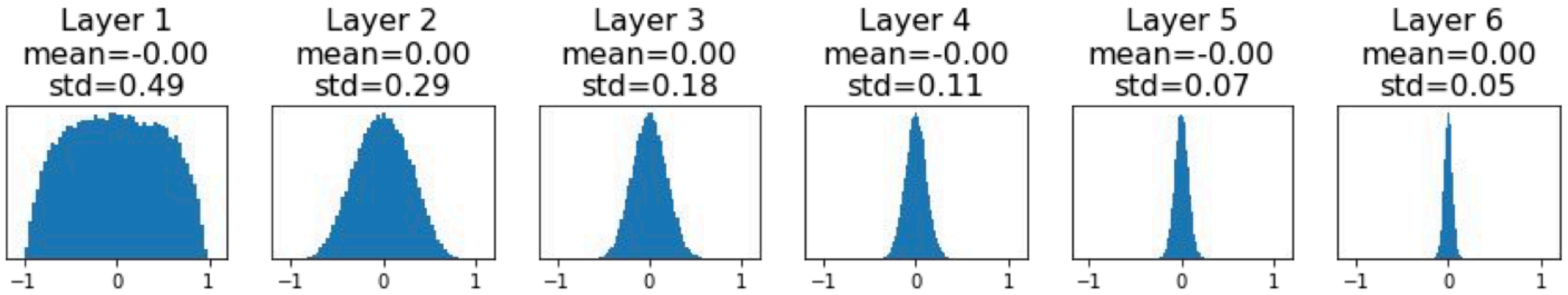
```
dims = [4096] * 7
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = 0.01 * np.random.randn(Din, Dout)
    x = np.tanh(x.dot(W))
    hs.append(x)
```

Forward pass for a 6-layer net with hidden size 4096

All activations tend to zero for deeper network layers

Q: What do the gradients dL/dW look like?

A: All zero, no learning =(



Slide credit: Stanford CS231N

Weight Initialization: Activation Statistics

```
dims = [4096] * 7
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = 0.05 * np.random.randn(Din, Dout)
    x = np.tanh(x.dot(W))
    hs.append(x)
```

Increase std of initial weights from 0.01 to 0.05

What will happen to the activations for the last layer?

Slide credit: Stanford CS231N

Weight Initialization: Activation Statistics

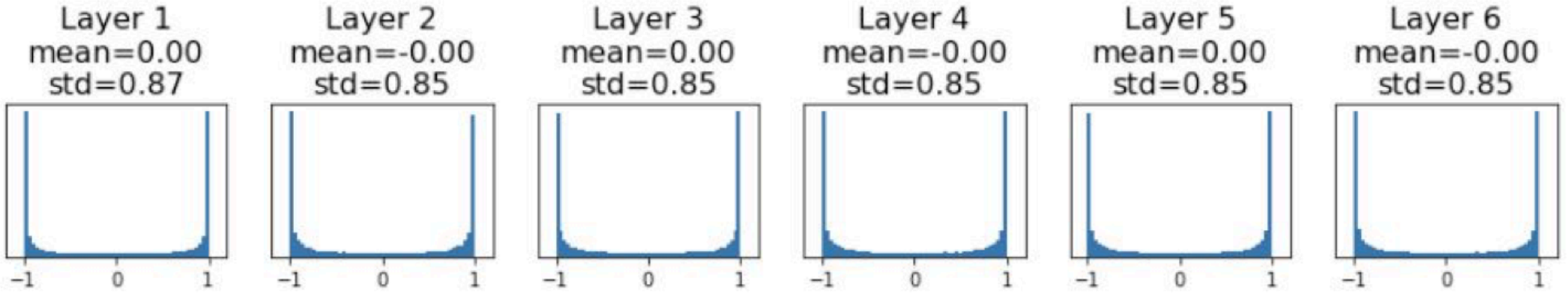
```
dims = [4096] * 7
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = 0.05 * np.random.randn(Din, Dout)
    x = np.tanh(x.dot(W))
    hs.append(x)
```

Increase std of initial weights from 0.01 to 0.05

All activations saturate

Q: What do the gradients look like?

A: Local gradients all zero, no learning =(



Slide credit: Stanford CS231N

Weight Initialization: Xavier Inititalization

```
dims = [4096] * 7
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = np.random.randn(Din, Dout) / np.sqrt(Din)
    x = np.tanh(x.dot(W))
    hs.append(x)
```

"Xavier" initialization:
std = 1/sqrt(Din)

Slide credit: Stanford CS231N

Weight Initialization: Xavier Inititalization

```
dims = [4096] * 7
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = np.random.randn(Din, Dout) / np.sqrt(Din)
    x = np.tanh(x.dot(W))
    hs.append(x)
```

"Xavier" initialization:
std = 1/sqrt(Din)

"Just right": Activations are nicely scaled for all layers!

For conv layers, Din is filter_size^2 * input_channels

Let: $y = x_1w_1+x_2w_2+\ldots+x_{Din}w_{Din}$

Assume: $Var(x_1) = Var(x_2)= \ldots=Var(x_{Din})$

We want: $Var(y) = Var(x_i)$

$$\begin{aligned} Var(y) &= Var(x_1w_1+x_2w_2+\ldots+x_{Din}w_{Din}) \\ &= Din\ Var(x_iw_i) \\ &= Din\ Var(x_i)\ Var(w_i) \end{aligned}$$

[Assume all x_i, w_i are iid]

So, $Var(y) = Var(x_i)$ only when $Var(w_i) = 1/Din$

Glorot and Bengio, "Understanding the difficulty of training deep feedforward neural networks", AISTAT 2010

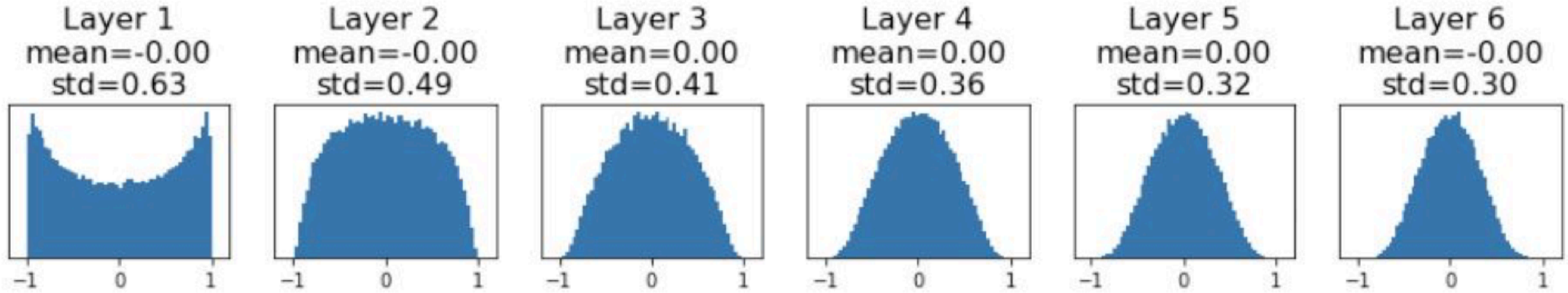
Slide credit: Stanford CS231N

Weight Initialization: Xavier Inititalization

```
dims = [4096] * 7
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = np.random.randn(Din, Dout) / np.sqrt(Din)
    x = np.tanh(x.dot(W))
    hs.append(x)
```

"Xavier" initialization:
std = 1/sqrt(Din)

"Just right": Activations are nicely scaled for all layers!



Glorot and Bengio, "Understanding the difficulty of training deep feedforward neural networks", AISTAT 2010

Slide credit: Stanford CS231N

Weight Initialization: Xavier Inititalization

```
dims = [4096] * 7
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = np.random.randn(Din, Dout) / np.sqrt(Din)
    x = np.maximum(0, x.dot(W))
    hs.append(x)
```

Change from tanh to ReLU

Slide credit: Stanford CS231N

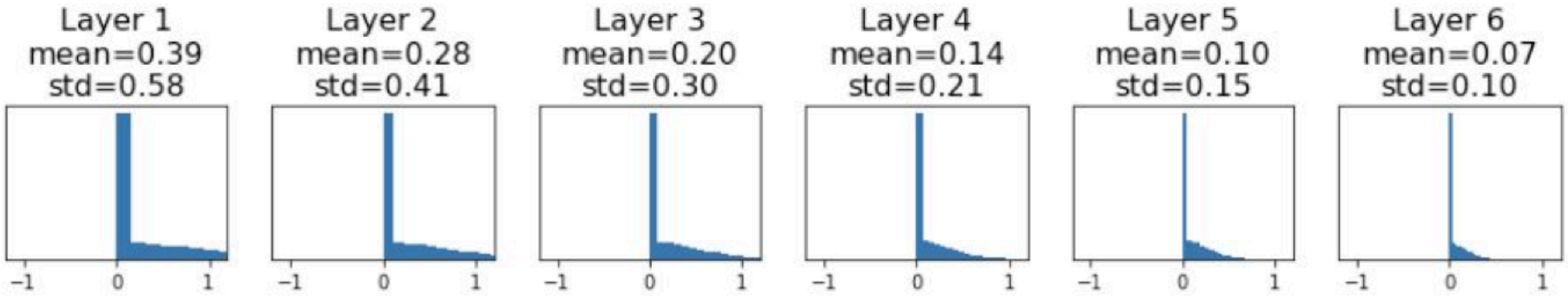
Weight Initialization: Xavier Inititalization

```
dims = [4096] * 7
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = np.random.randn(Din, Dout) / np.sqrt(Din)
    x = np.maximum(0, x.dot(W))
    hs.append(x)
```

Change from tanh to ReLU

Xavier assumes zero centered activation function

Activations collapse to zero again, no learning =(



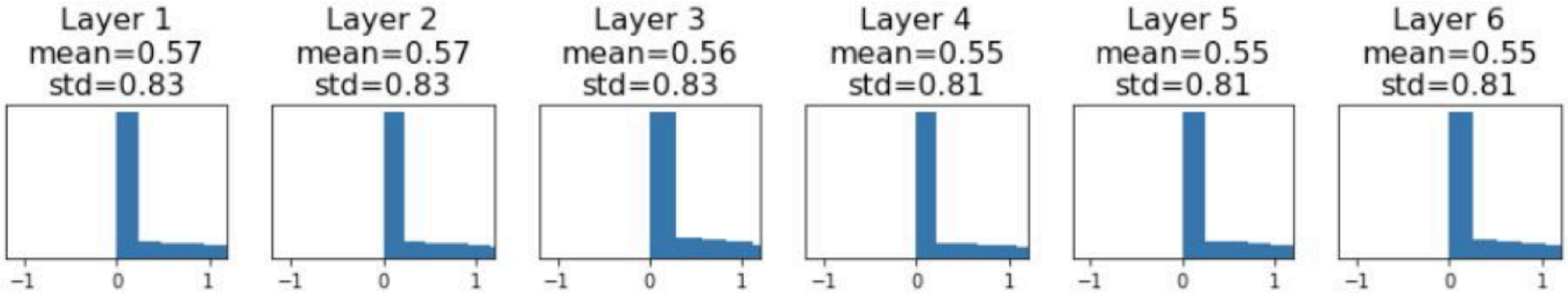
Slide credit: Stanford CS231N

Weight Initialization: He Inititalization

```
dims = [4096] * 7
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = np.random.randn(Din, Dout) * np.sqrt(2/Din)
    x = np.maximum(0, x.dot(W))
    hs.append(x)
```

ReLU correction: std = sqrt(2 / Din)

"Just right": Activations are nicely scaled for all layers!



He et al, "Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification", ICCV 2015

Slide credit: Stanford CS231N

Initialization is still an Active Research Area

Understanding the difficulty of training deep feedforward neural networks
by Glorot and Bengio, 2010

Exact solutions to the nonlinear dynamics of learning in deep linear neural networks by Saxe et al, 2013

Random walk initialization for training very deep feedforward networks by Sussillo and Abbott, 2014

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

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

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

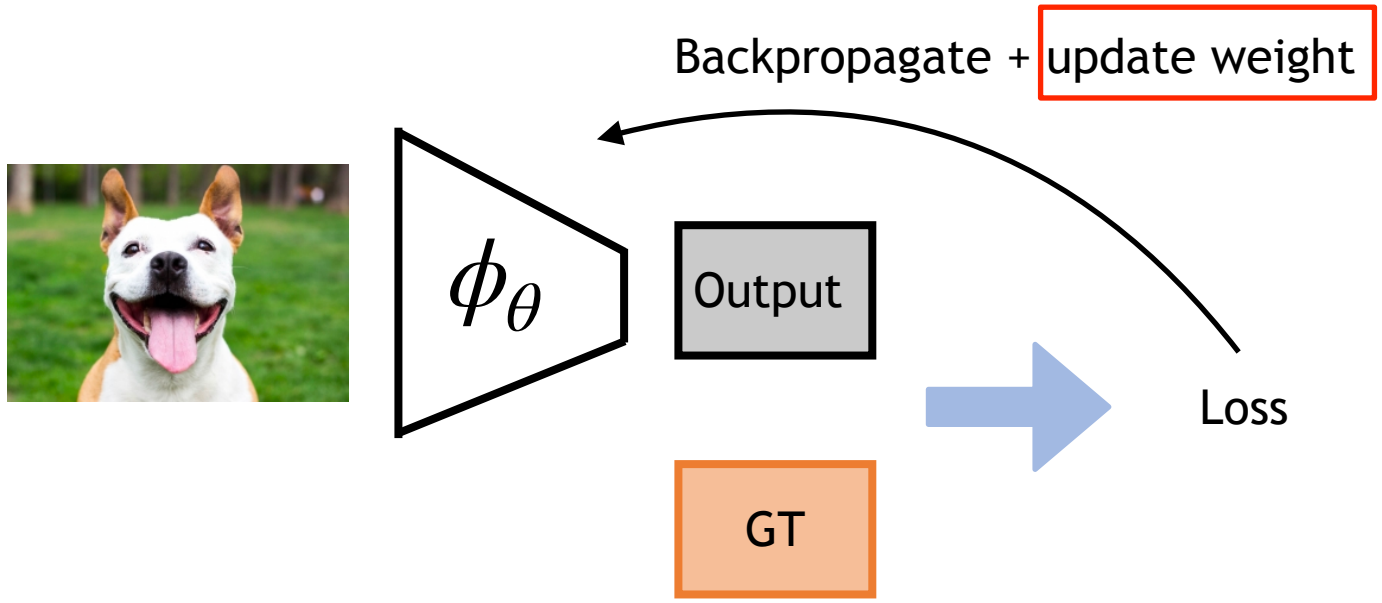
Fixup Initialization: Residual Learning Without Normalization, Zhang et al, 2019

The Lottery Ticket Hypothesis: Finding Sparse, Trainable Neural Networks, Frankle and Carbin, 2019

Slide credit: Stanford CS231N

Outline

- Data preparation
- Weight Initialization
- Set a loss function
- Start optimization
  - optimizer?
  - learning rate?



SGD

Update rule:

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

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

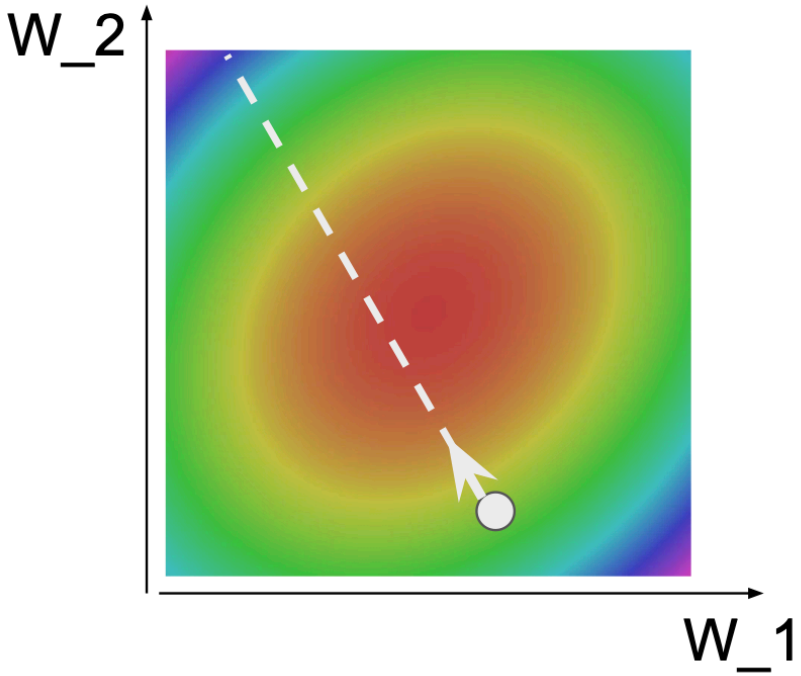
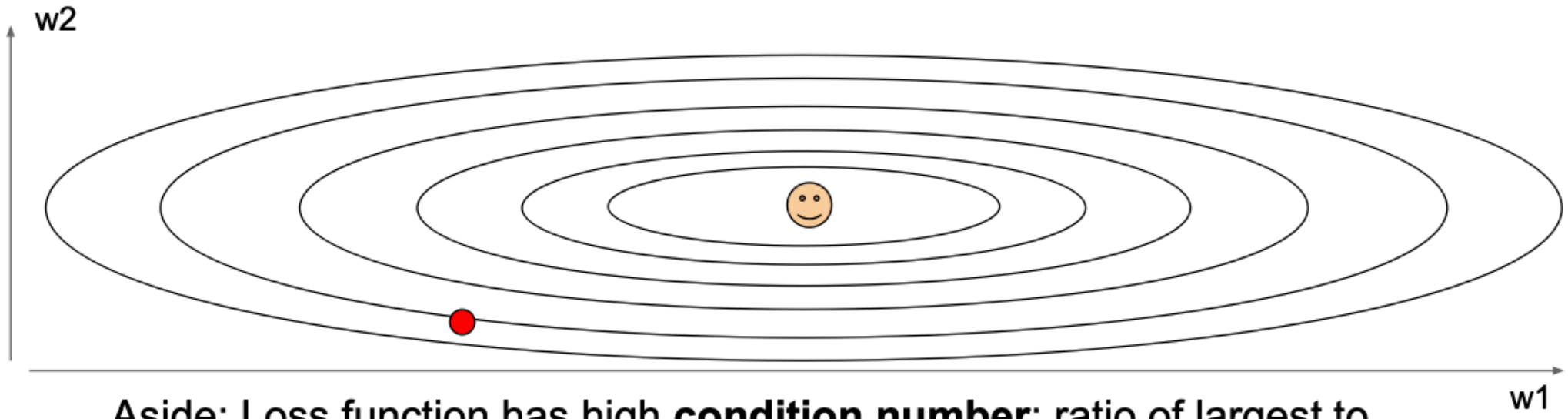


Figure credit: Stanford CS231N

Problems with SGD: #1

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

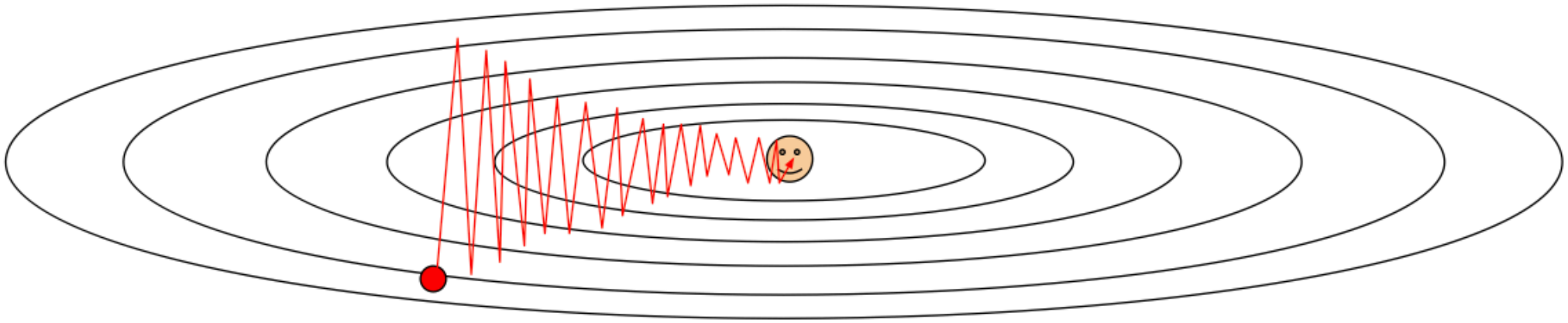


Aside: Loss function has high **condition number**: ratio of largest to smallest singular value of the Hessian matrix is large

Slide credit: Stanford CS231N

Problems with SGD: #1

What if loss changes quickly in one direction and slowly in another?
What does gradient descent do?
Very slow progress along shallow dimension, jitter along steep direction

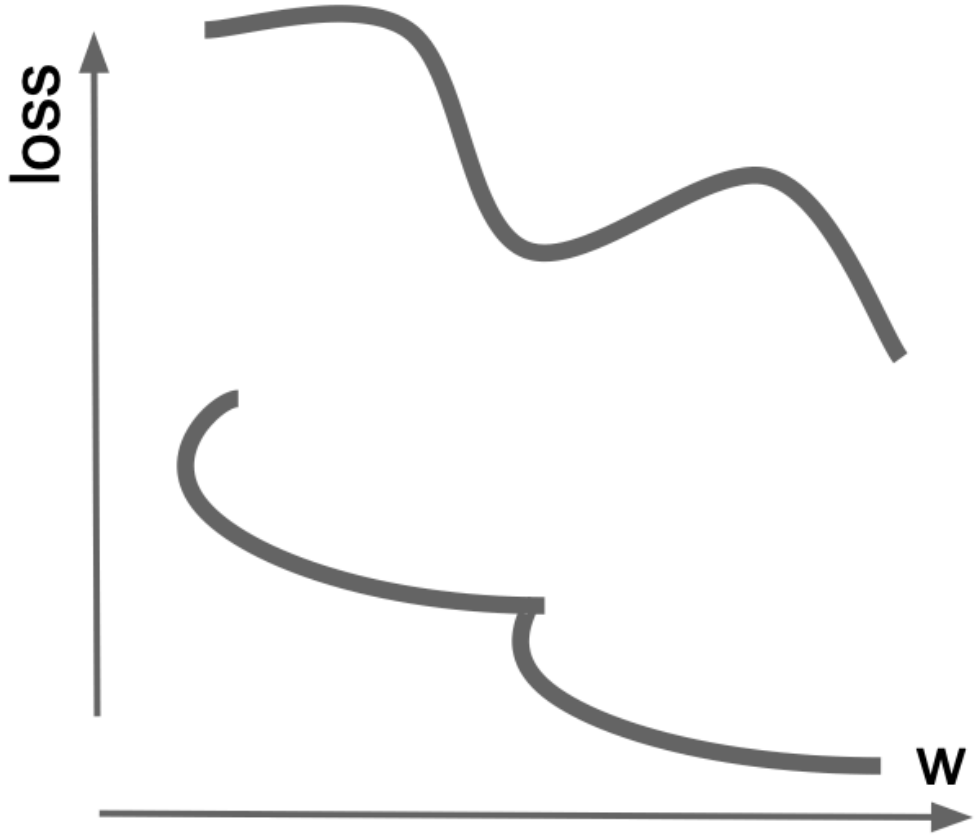


Loss function has high **condition number**: ratio of largest to smallest singular value of the Hessian matrix is large

Slide credit: Stanford CS231N

Problems with SGD: #2

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

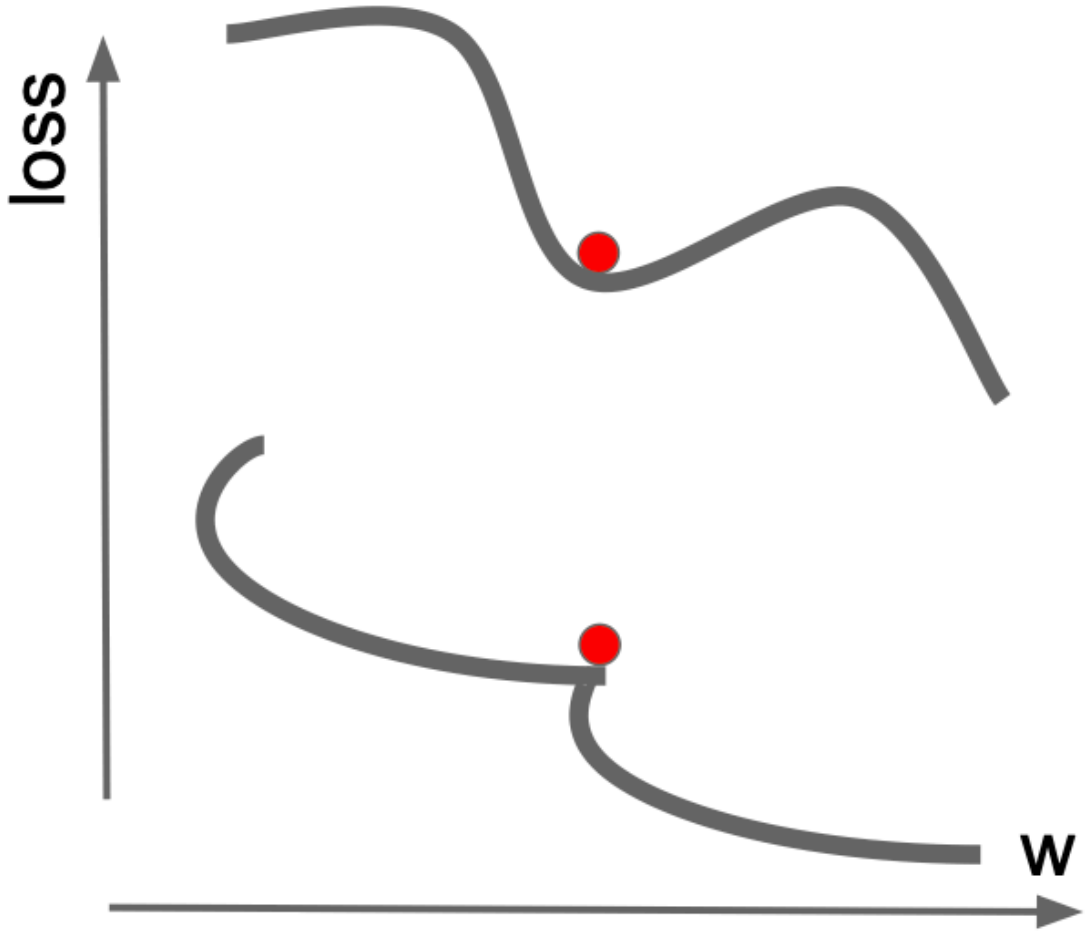


Slide credit: Stanford CS231N

Problems with SGD: #2

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

Zero gradient, gradient descent gets stuck

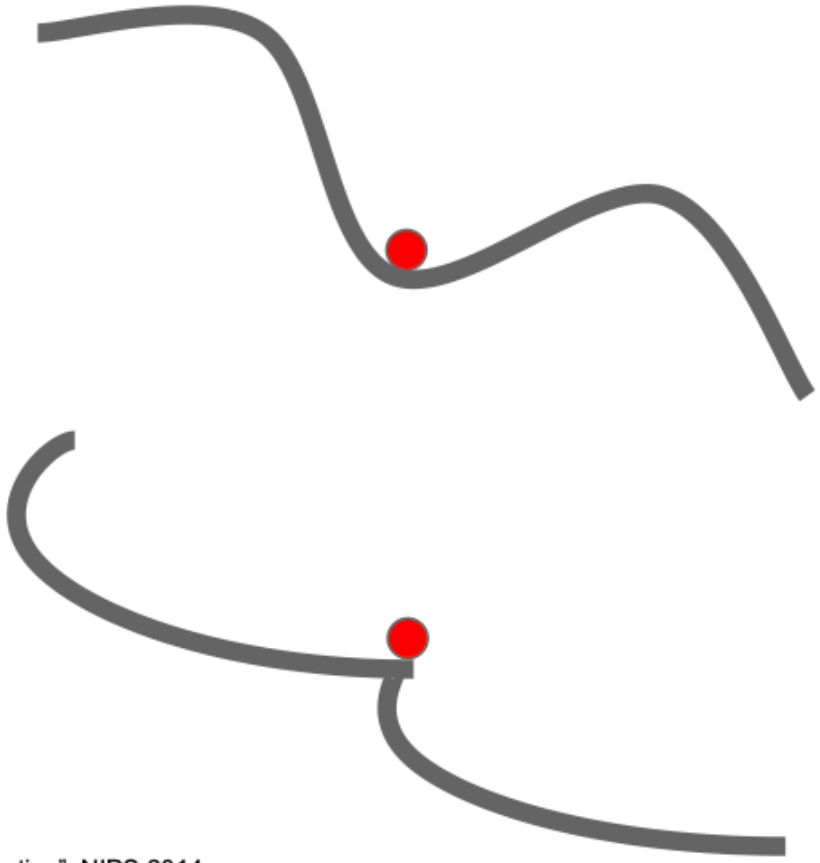


Slide credit: Stanford CS231N

Problems with SGD: #2

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

Saddle points much more common in high dimension



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

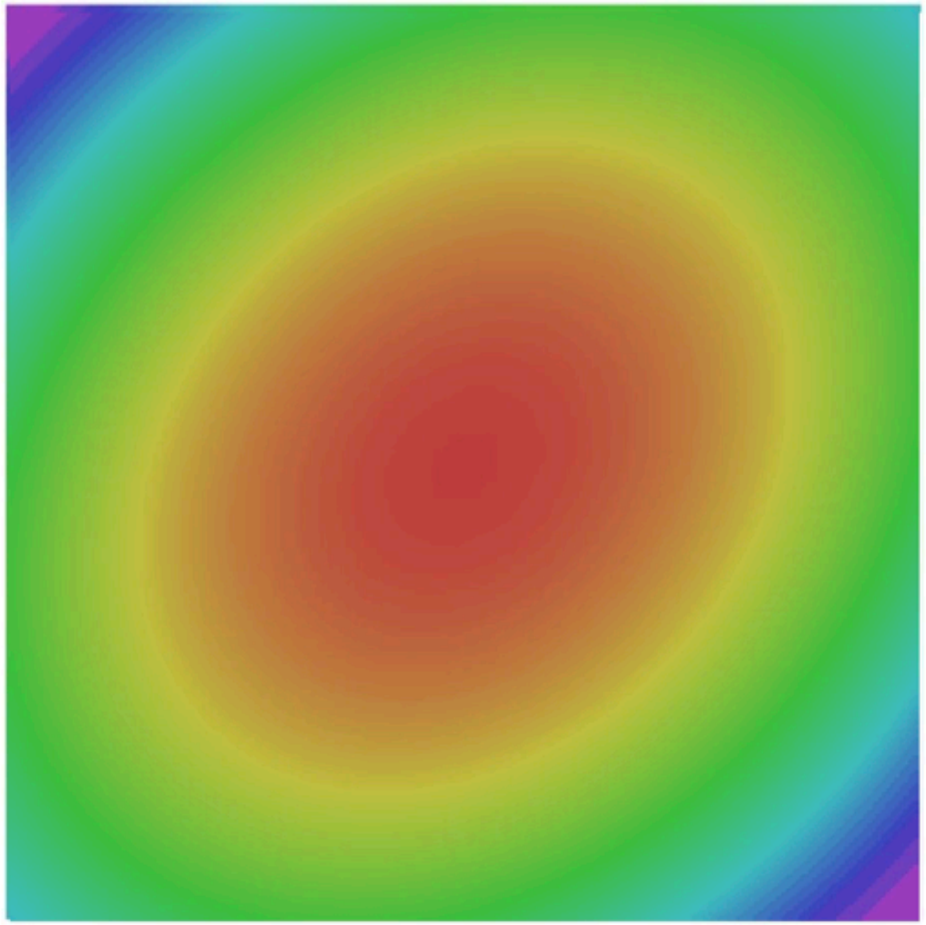
Slide credit: Stanford CS231N

Problems with SGD: #3

Our gradients come from minibatches so they can be noisy!

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

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



Slide credit: Stanford CS231N

SGD + Momentum

continue moving in the general direction as the previous iterations

SGD

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

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

SGD+Momentum

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

$$x_{t+1} = x_t - \alpha v_{t+1}$$

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

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

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

Slide credit: Stanford CS231N

Adam

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

Momentum

Bias correction

AdaGrad / RMSProp

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

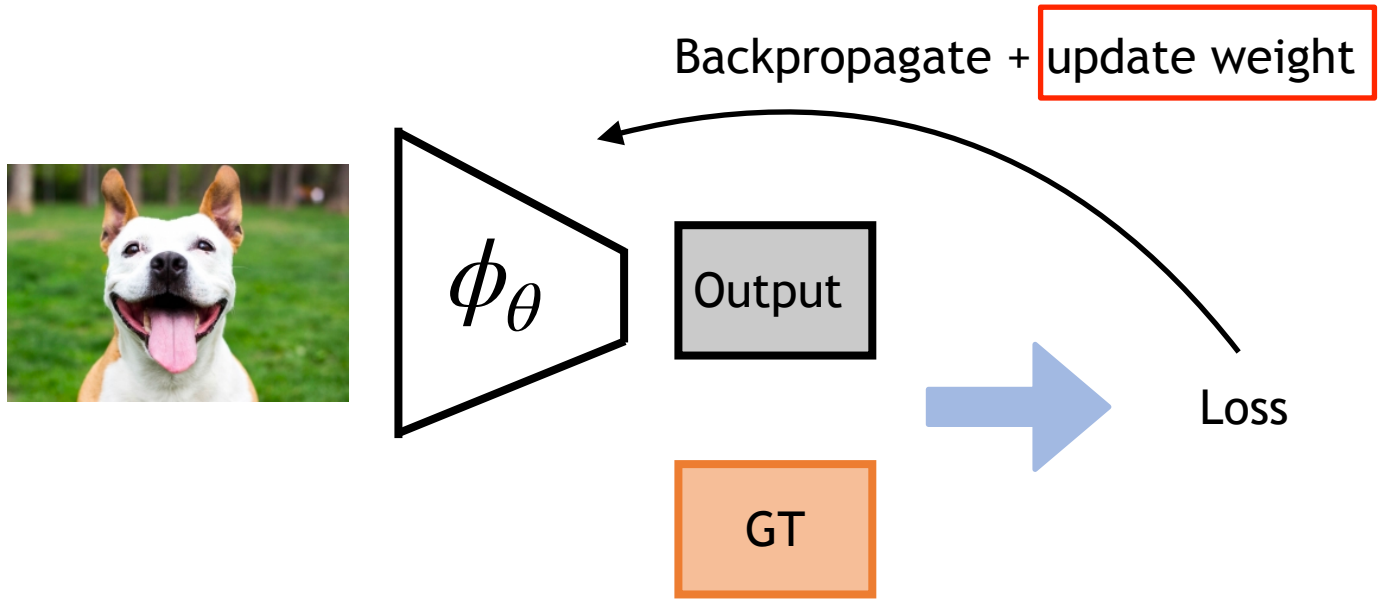
Adam with beta1 = 0.9, beta2 = 0.999, and learning_rate = 1e-3 or 5e-4 is a great starting point for many models!

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

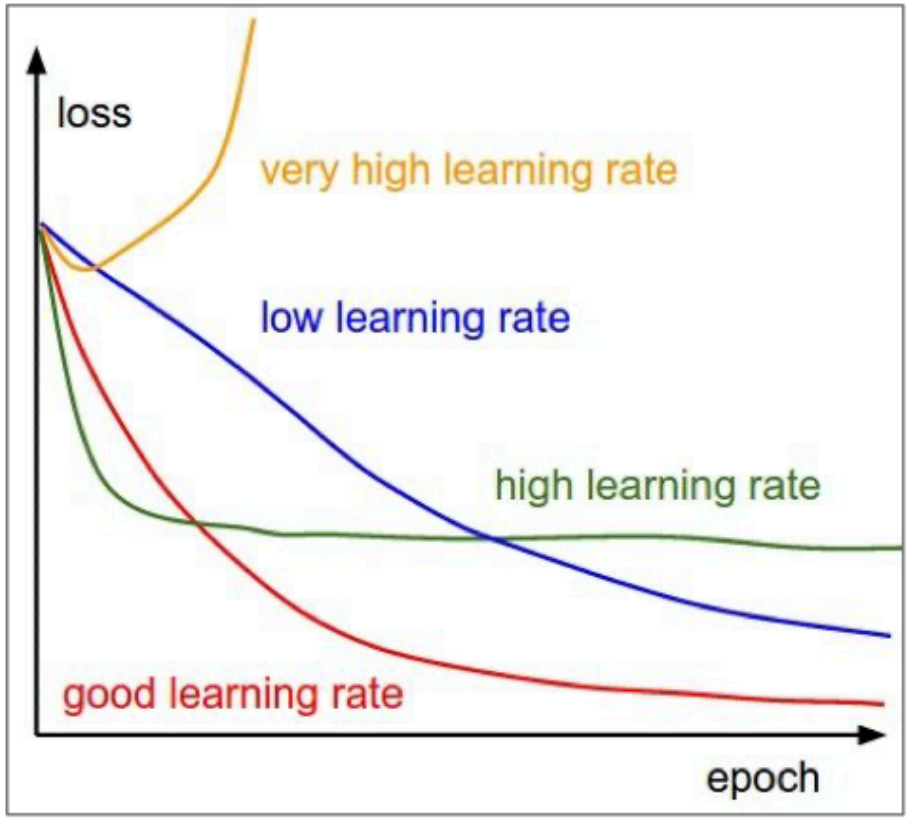
Slide credit: Stanford CS231N

Outline

- Data preparation
- Weight Initialization
- Set a loss function
- Start optimization
  - optimizer?
  - learning rate?



Loss Curves for Different Learning Rates



- An appropriate learning rate for classification: 1e-6 ~ 1e-3
- Low learning rate: undershoot
- High learning rate: overshoot

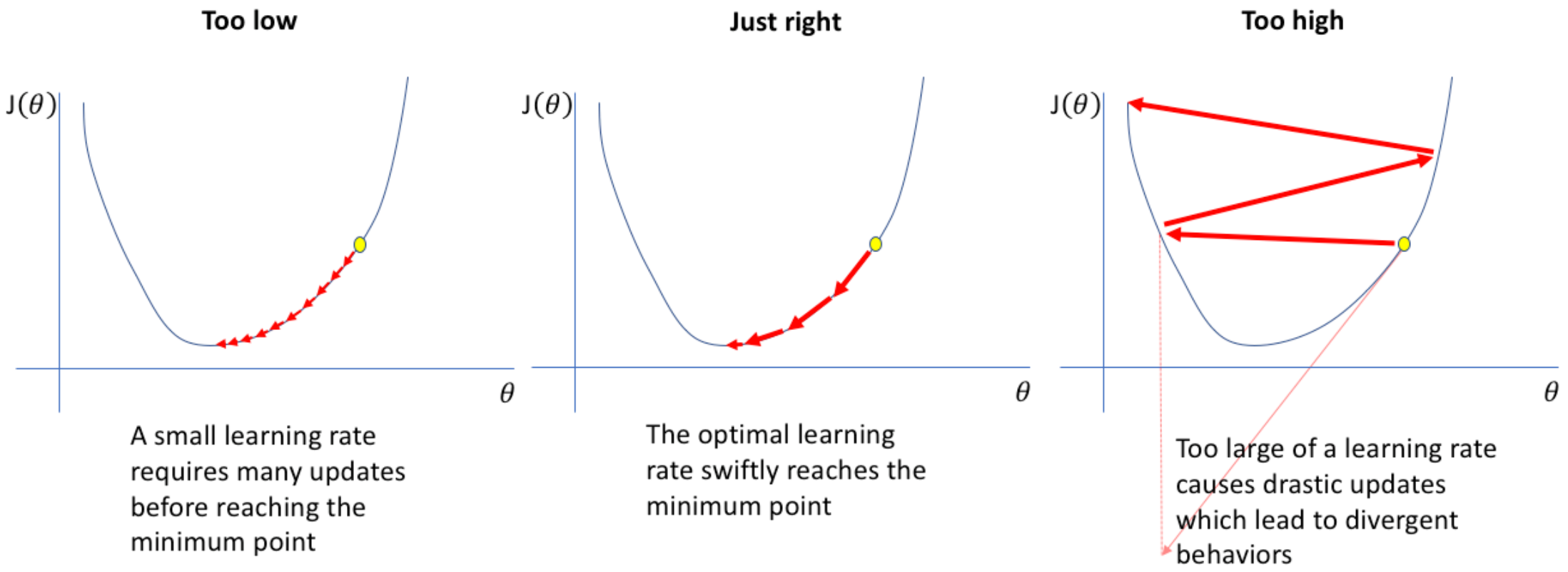
Figure credit: Stanford CS231N.

Iteration and Epoch

- Iteration:
  - One batch (whose size is called batch size)
  - A gradient descent step

- Epoch
  - Contains many iterations that go over the training data for one complete pass
  - After a epoch, plot train curve, evaluate on val, save model…

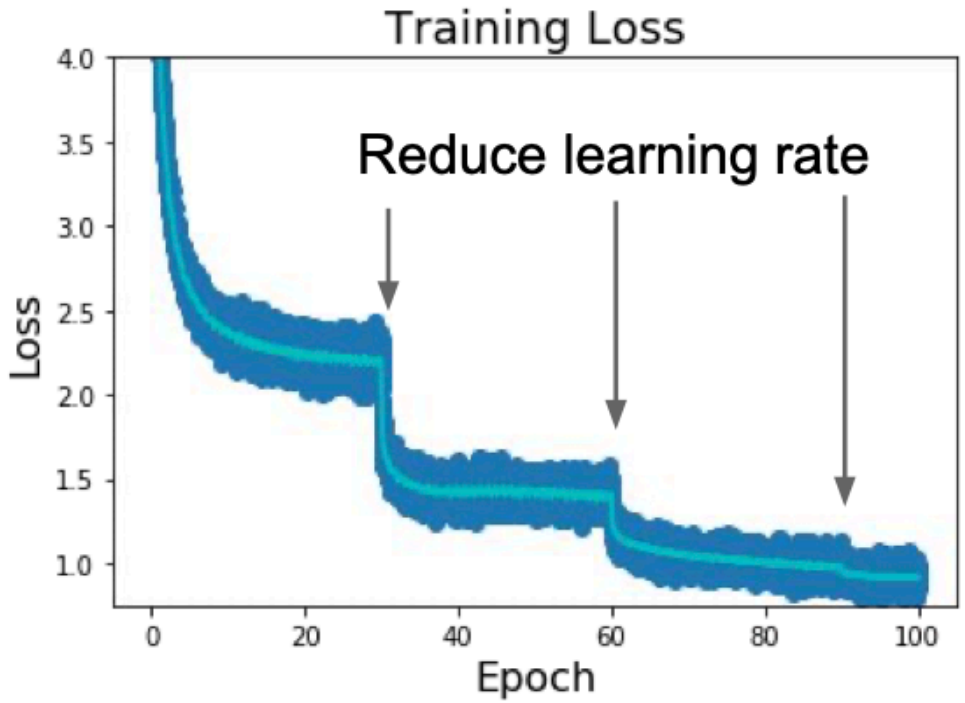
Learning Rate



https://www.jeremyjordan.me/content/images/2018/02/Screen-Shot-2018-02-24-at-11.47.09-AM.png

Learning Rate Schedule

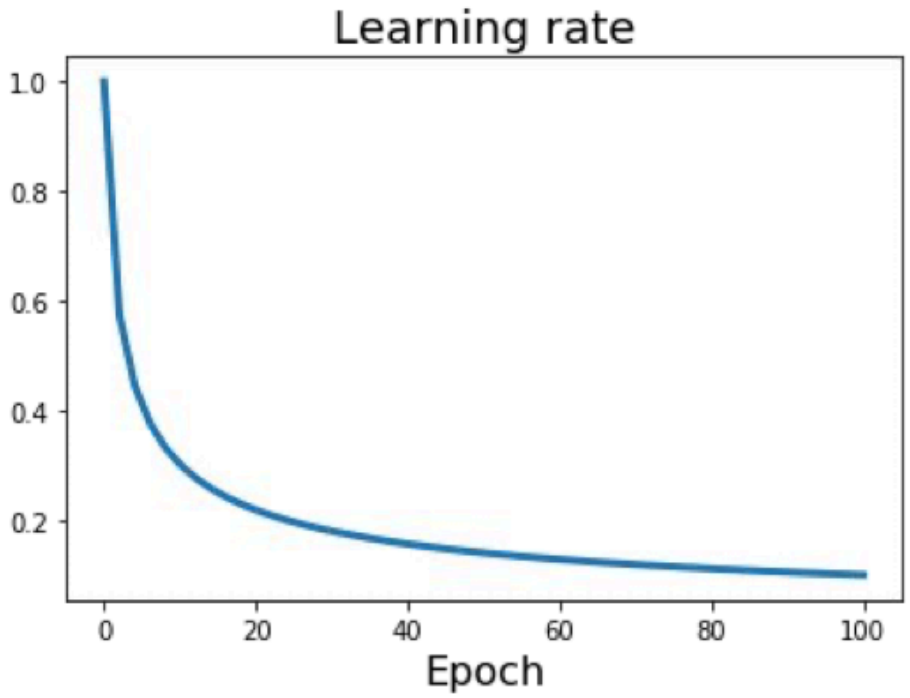
Idea: high learning rate at the beginning, decay it later



Step: Reduce learning rate at a few fixed points. E.g. for ResNets, multiply LR by 0.1 after epochs 30, 60, and 90.

Slide credit: Stanford CS231N

Learning Rate Schedule



Vaswani et al, "Attention is all you need", NIPS 2017

Step: Reduce learning rate at a few fixed points. E.g. for ResNets, multiply LR by 0.1 after epochs 30, 60, and 90.

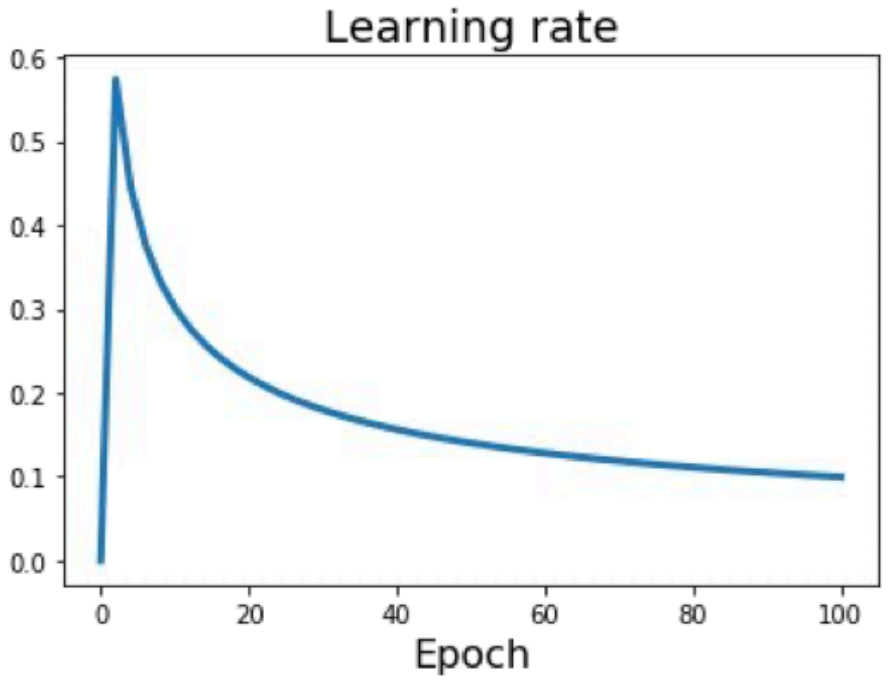
Cosine: $\alpha_t = \frac{1}{2}\alpha_0 \left(1 + \cos(t\pi/T)\right)$

Linear: $\alpha_t = \alpha_0(1 - t/T)$

Inverse sqrt: $\alpha_t = \alpha_0/\sqrt{t}$

α_0 : Initial learning rate
α_t : Learning rate at epoch t
T : Total number of epochs

Learning Rate Schedule: Linear Warmup



High initial learning rates can make loss explode; linearly increasing learning rate from 0 over the first ~5000 iterations can prevent this

Goyal et al, "Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour", arXiv 2017

Batch Size and Learning Rate

- An empirical rule of thumb: if you increase the batch size by N, also scale the initial learning rate by N.

- Why? Suggested reading: visualizing learning rate vs. batch size.

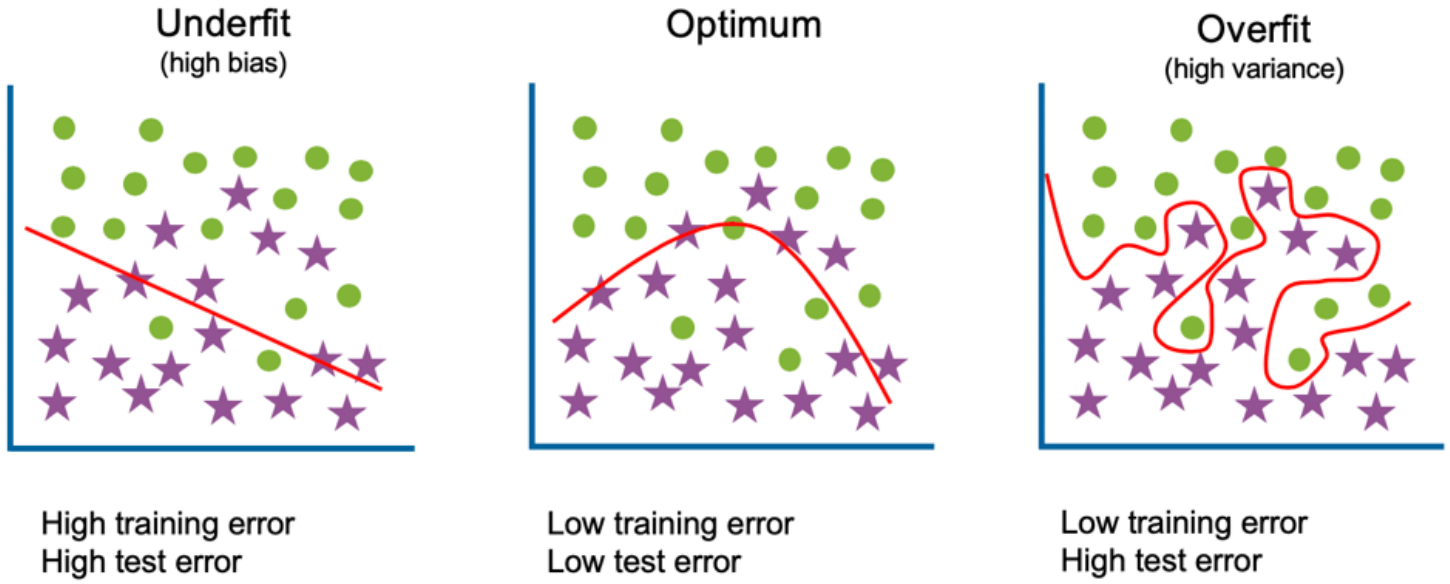
Goyal et al, "Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour", arXiv 2017

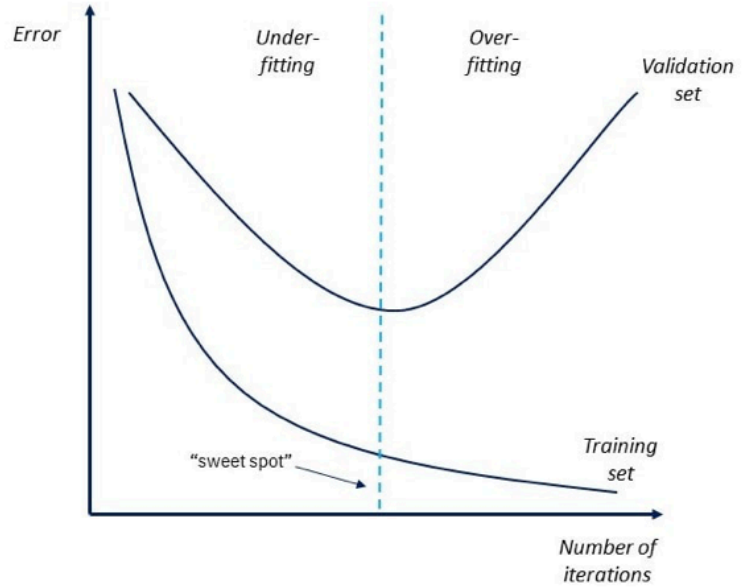
Summary of Learning Rate Schedule

- **Adam** is a good default choices working okay with constant learning rate.
- **SGD + Momentum** can outperform Adam but may require more tuning of LR and schedule
  - Try cosine schedule: very few hyper parameters.
- If you are new to a dataset, use Adam with constant learning rate until you see it converges and then modify the learning rate.

Underfitting & Overfitting

Underfitting and Overfitting



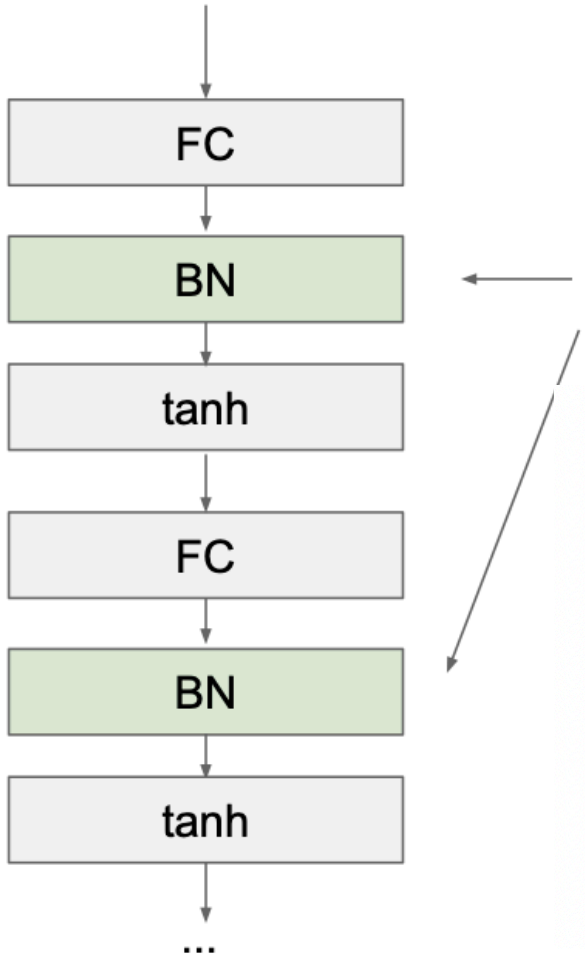


https://www.ibm.com/cloud/learn/overfitting

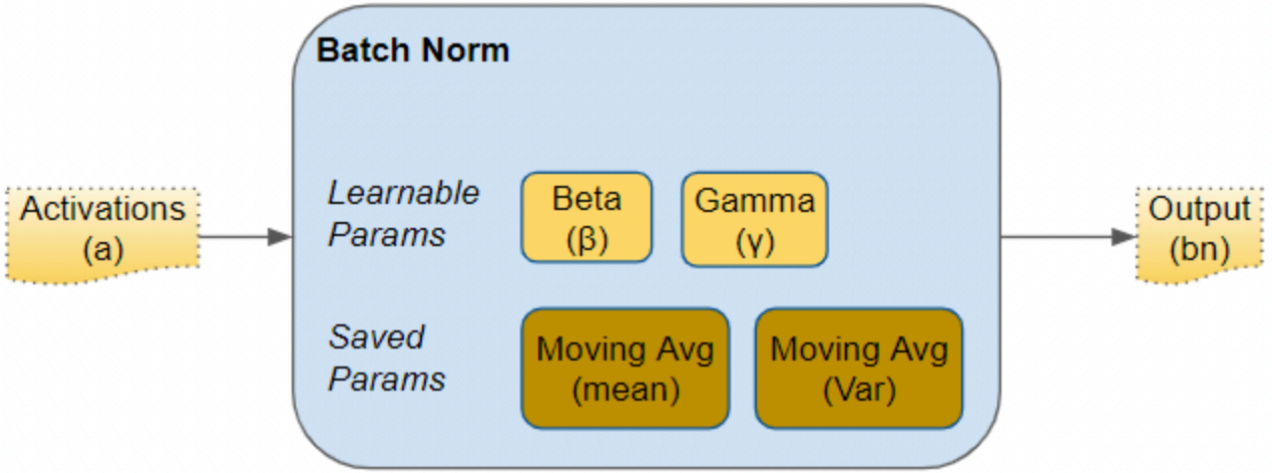
Underfitting

- Underfitting on the train set: usually caused by limited model capacity or unsatisfactory optimization
  - Batch normalization
  - Skip link
- Overfitting on the test set

Batch Normalization



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



https://towardsdatascience.com/batch-norm-explained-visually-how-it-works-and-why-neural-networks-need-it

Batch Normalization: Train Mode

Input: $x : N \times D$

Learnable scale and shift parameters:

$\gamma, \beta : D$

Learning $\gamma = \sigma$,
$\beta = \mu$ will recover the identity function!

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

Per-channel mean, shape is D

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

Per-channel var, shape is D

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

Normalized x, Shape is N x D

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

Output, Shape is N x D

Slide credit: Stanford CS231N

Batch Normalization: Eval Mode (Test-Time)

Input: $x : N \times D$

Learnable scale and shift parameters:

$$\gamma, \beta : D$$

During testing batchnorm becomes a linear operator!
Can be fused with the previous fully-connected or conv layer

$$\mu_{rms} = \text{(Running) average of values seen during training}$$

Per-channel mean, shape is D

$$\mu_{rms} \leftarrow \rho \mu_{rms} + (1 - \rho)\mu_i$$

$$\sigma^2_{rms} = \text{(Running) average of values seen during training}$$

Per-channel var, shape is D

$$\hat{x}_{i,j} = \frac{x_{i,j} - \mu_{rms,j}}{\sqrt{\sigma^2_{rms,j} + \epsilon}}$$

Normalized x, Shape is N x D

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

Output, Shape is N x D

Slide credit: Stanford CS231N

BatchNorm Helps!

- Makes deep networks **much** easier to train!
- Improves gradient flow
- Allows higher learning rates, faster convergence
- Networks become more robust to initialization
- Acts as regularization during training

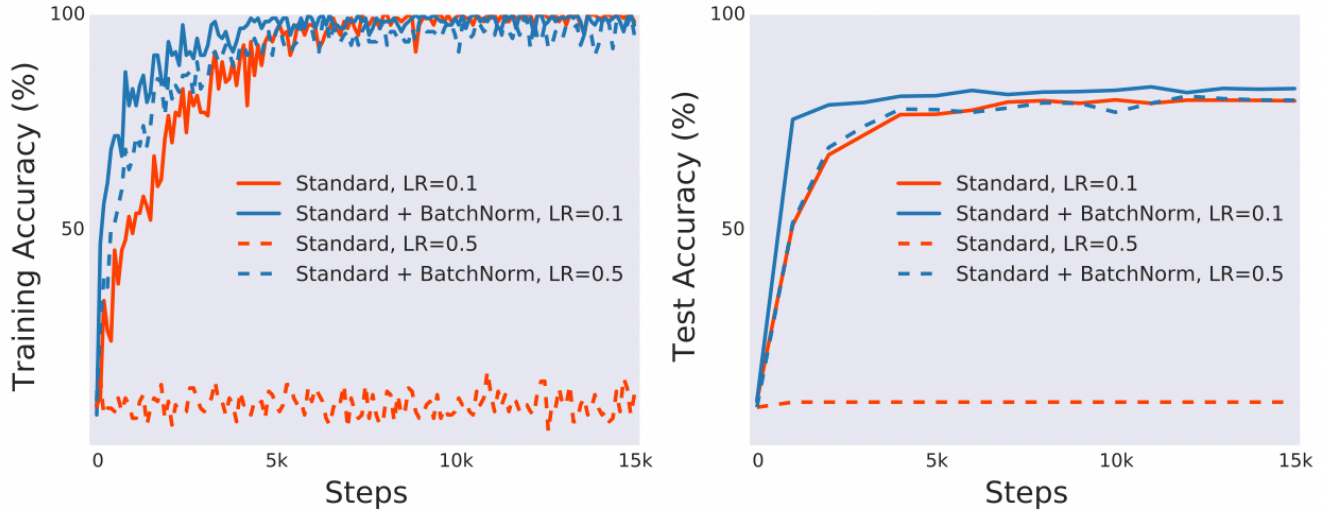


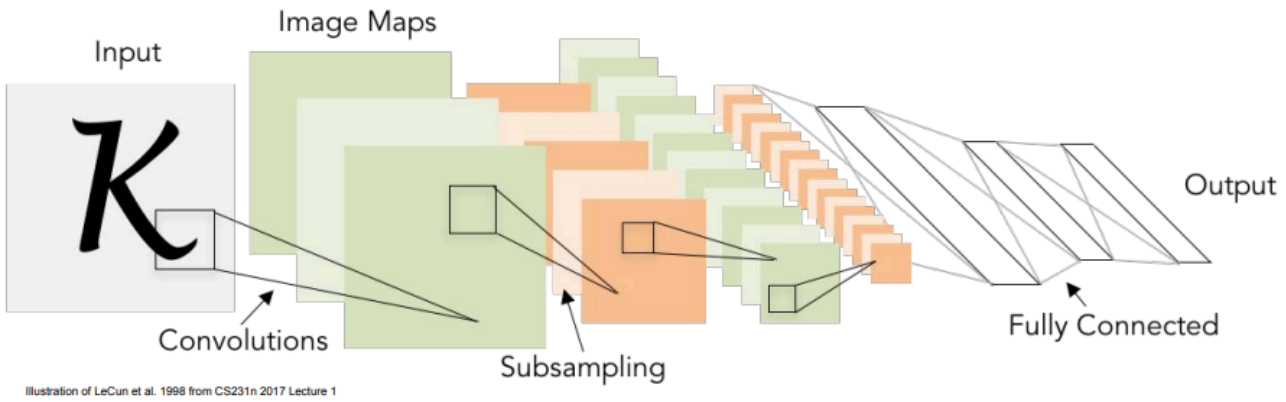
Figure 1: Comparison of (a) training (optimization) and (b) test (generalization) performance of a standard VGG network trained on CIFAR-10 with and without BatchNorm (details in Appendix A).

Stacking More Layers

- ConvNets stack CONV,POOL,FC layers
- Trend towards smaller filters and deeper architectures
- Trend towards getting rid of POOL/FC layers (just CONV)
- Historically architectures looked like

[(Conv-BN-ReLU)*N - POOL?]*m-(FC-BN-RELU)*K-FC-SoftMax

No BN at the last layer.



Slide credit: Stanford CS231N

Why BatchNorm Works?

- Original hypothesis: mitigate the “internal covariate shift”
  - *“Training Deep Neural Networks is complicated by the fact that the distribution of each layer’s inputs changes during training, as the parameters of the previous layers change. This slows down the training by requiring lower learning rates and careful parameter initialization, and makes it notoriously hard to train models with saturating nonlinearities.”*
  - *“We refer to the change in the distributions of internal nodes of a deep network, in the course of training, as Internal Covariate Shift.”*

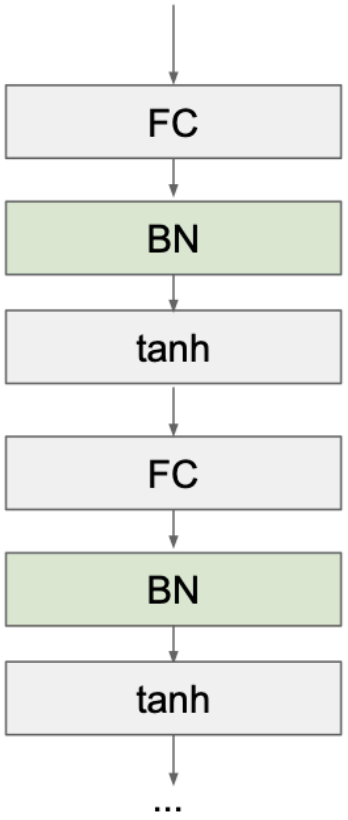
Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift, 2015.

Why BatchNorm Works?

- New findings: BatchNorm smoothens the loss landscape
  - BatchNorm may not reduce the internal covariate shift.
  - Batch normalization is effective because it smooths and, in turn, simplifies the optimization function that is being solved when training the network.
  - *"This ensures, in particular, that the gradients are more predictive and thus allow for use of larger range of learning rates and faster network convergence."*

How Does Batch Normalization Help Optimization? (No, It Is Not About Internal Covariate Shift), 2018.

Pros and Cons of BatchNorm



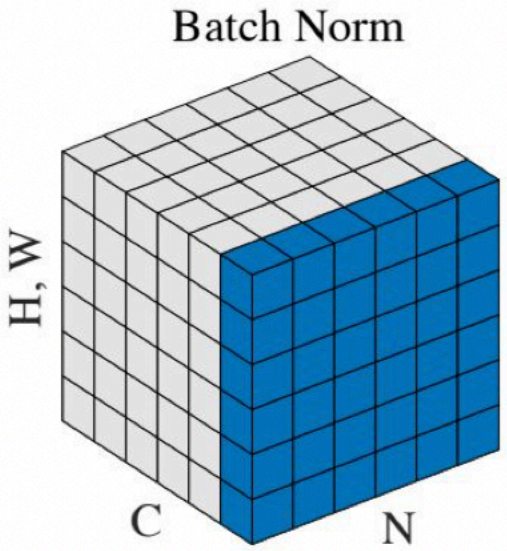
- Makes deep networks **much** easier to train!
- Improves gradient flow
- Allows higher learning rates, faster convergence
- Networks become more robust to initialization
- Acts as regularization during training
- Zero overhead at test-time: can be fused with conv!
- Behaves differently during training and testing: this is a very common source of bugs!

Slide credit: Stanford CS231N

Problems with Batch Normalization

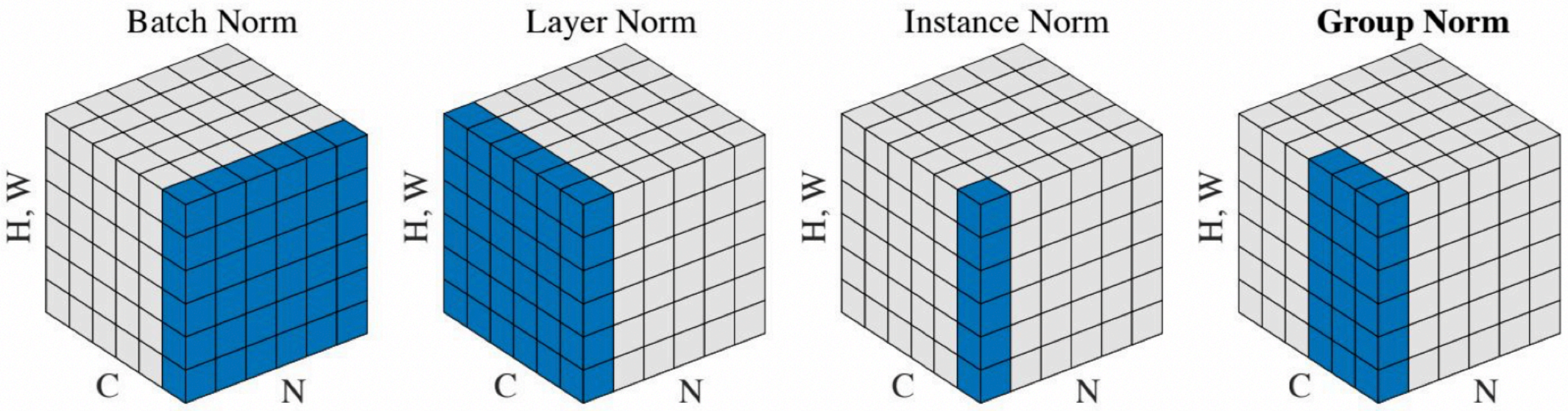
- If batch size in the training time is too small (like 1), then μ, σ in a training batch can be very random.
- There will be a big discrepancy between μ, σ in a training batch and μ_{rms}, σ_{rms} at test time. —> Misaligned objective during training and testing
- May lead to huge performance drop at test time even for training data.

Get Rid of Batch Dimension?



Can we remove the dependence on the batch dimension?
We then don’t have the discrepancy between train and eval modes.

Normalization Techniques



Widely used in NLP!

Style transfer!

Wu and He, “Group Normalization”, ECCV 2018

Group Normalization

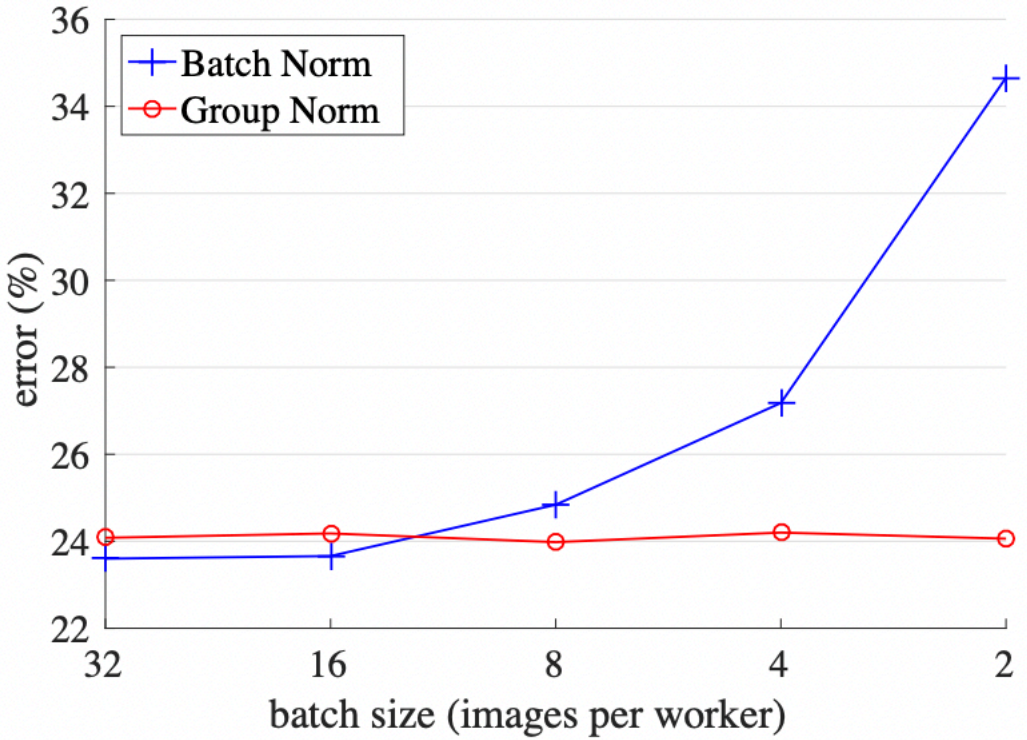


Figure 1. **ImageNet classification error *vs*. batch sizes**. This is a ResNet-50 model trained in the ImageNet training set using 8 workers (GPUs), evaluated in the validation set.

GroupNorm outperforms BatchNorm, when

- Batch size is small
- Instances in a batch are highly correlated

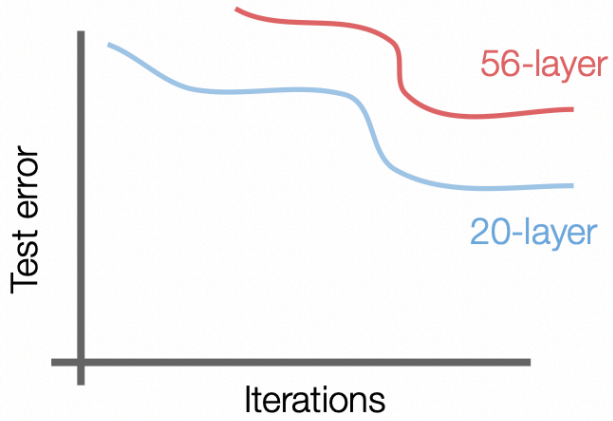
Problems of CNN Training

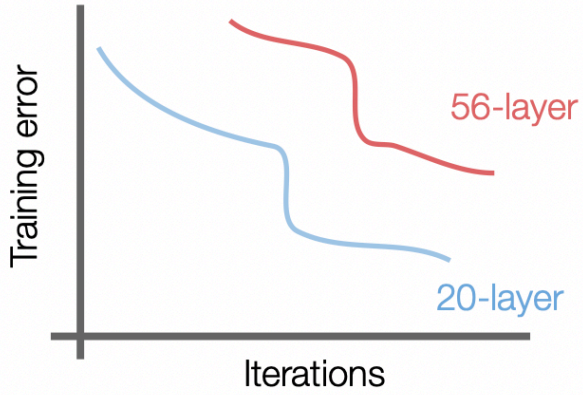
- Underfitting on the train set: usually caused by limited model capacity or unsatisfactory optimization
  - Batch normalization
  - ResNet or Skip links

- Overfitting on the test set

Problems When CNN Gets Really Deep

What happens when we continue stacking deeper layers on a “plain” convolutional neural network?





56-layer model performs worse on both test and training error
-> The deeper model performs worse, but it’s not caused by overfitting!

Slide credit: Stanford CS231N

Problems When CNN Gets Really Deep

Fact: Deep models have more representation power (more parameters) than shallower models.

Hypothesis: the problem is an *optimization* problem, **deeper models are harder to optimize**

Slide credit: Stanford CS231N

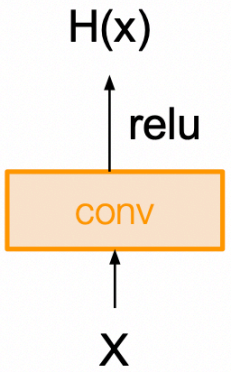
Problems When CNN Gets Really Deep

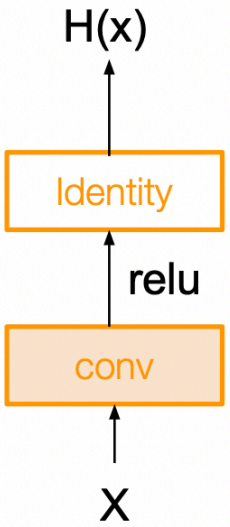
Fact: Deep models have more representation power (more parameters) than shallower models.

Hypothesis: the problem is an *optimization* problem, deeper models are harder to optimize

What should the deeper model learn to be at least as good as the shallower model?

A solution by construction is copying the learned layers from the shallower model and setting additional layers to identity mapping.

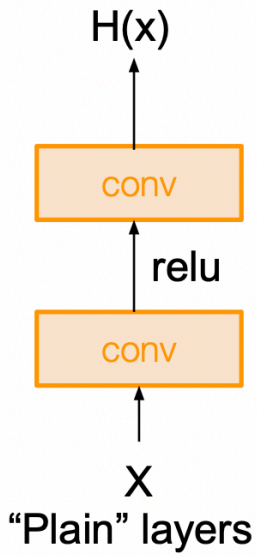




Slide credit: Stanford CS231N

Residual Link

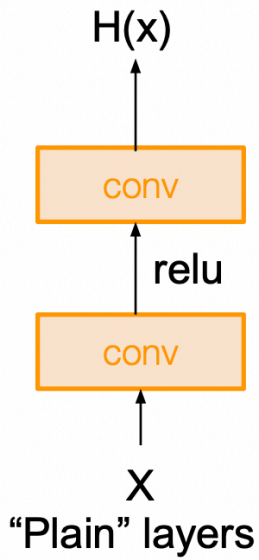
Solution: Use network layers to fit a residual mapping instead of directly trying to fit a desired underlying mapping

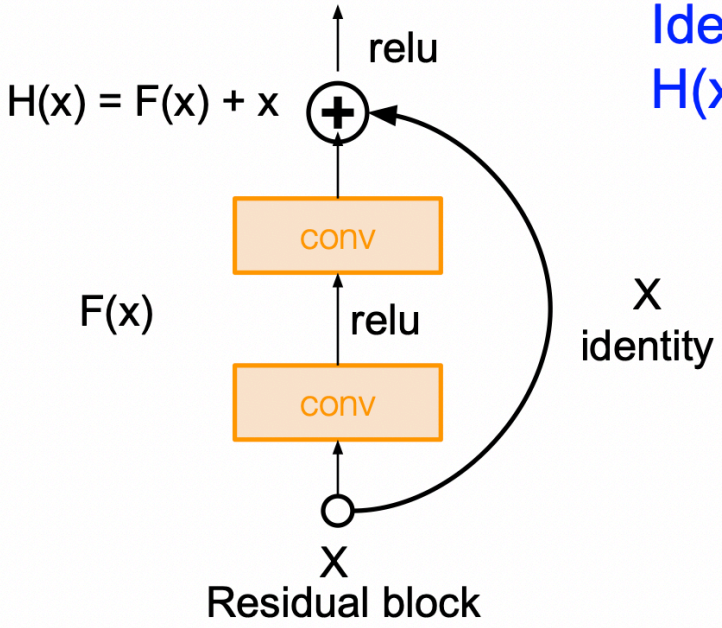


Slide credit: Stanford CS231N

Residual Link

Solution: Use network layers to fit a residual mapping instead of directly trying to fit a desired underlying mapping

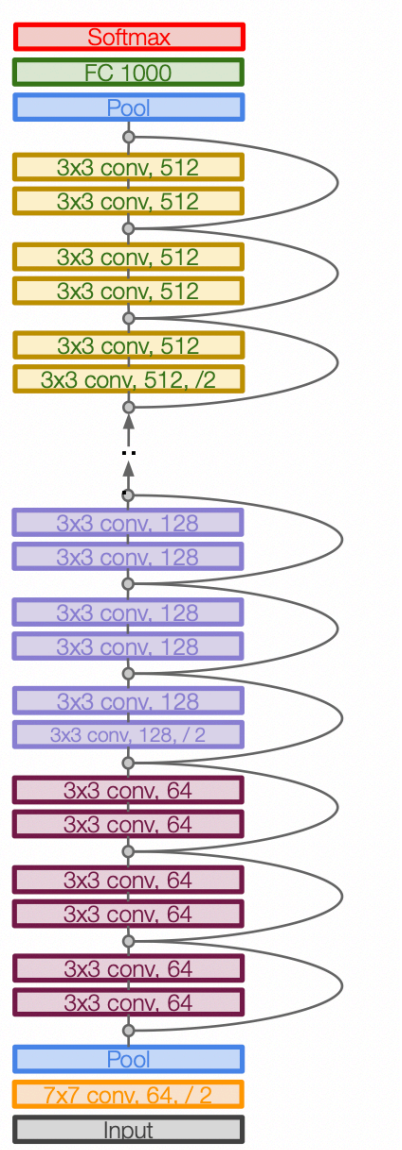




Identity mapping:
H(x) = x if F(x) = 0

Slide credit: Stanford CS231N

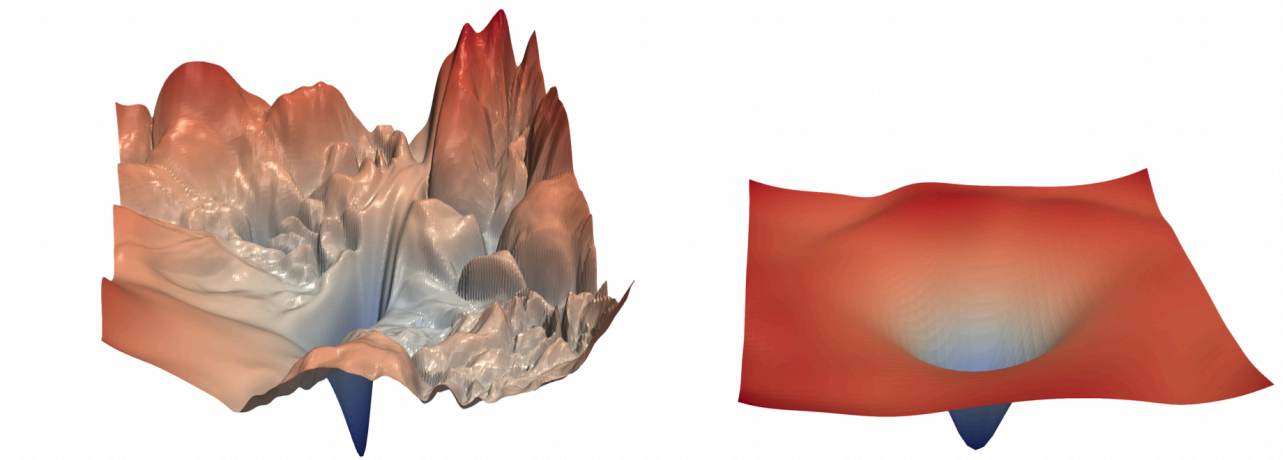
From the Perspective of Gradient BP



Skip links provide bypaths for gradients to backpropagate.

Loss Landscape

- *"When networks become sufficiently deep, neural loss landscapes quickly transition from being nearly convex to being highly chaotic. This transition from convex to chaotic behavior coincides with a dramatic drop in generalization error, and ultimately to a lack of trainability. "*



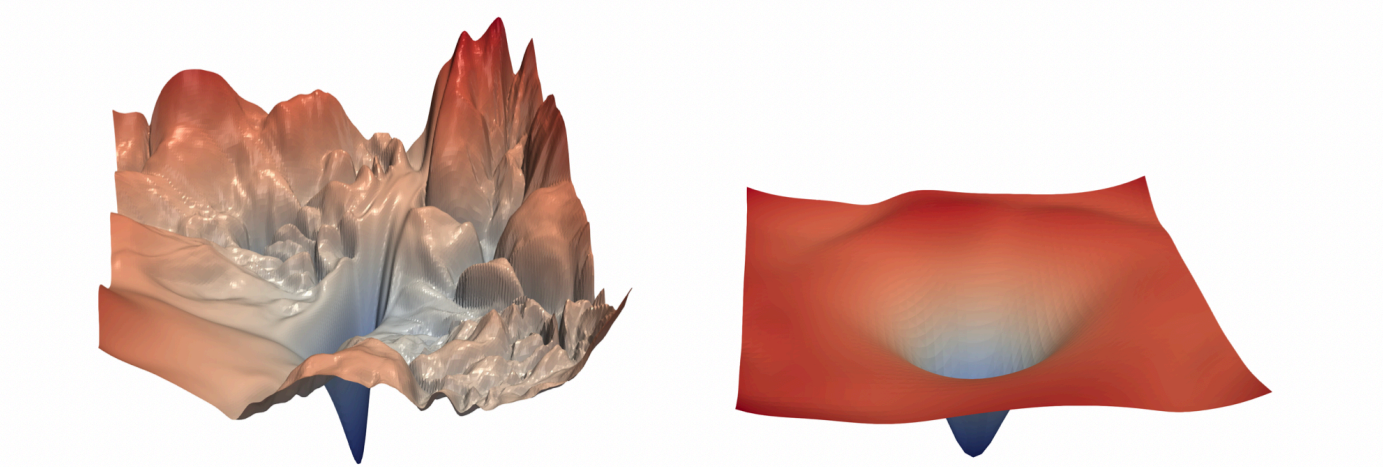
(a) without skip connections (b) with skip connections

Figure 1: The loss surfaces of ResNet-56 with/without skip connections. The proposed filter normalization scheme is used to enable comparisons of sharpness/flatness between the two figures.

Li, Hao, et al. "Visualizing the loss landscape of neural nets." *Advances in neural information processing systems* 31 (2018).

Loss Landscape

- *"skip connections promote flat minimizers and prevent the transition to chaotic behavior, which helps explain why skip connections are necessary for training extremely deep networks. "*



(a) without skip connections (b) with skip connections

Figure 1: The loss surfaces of ResNet-56 with/without skip connections. The proposed filter normalization scheme is used to enable comparisons of sharpness/flatness between the two figures.

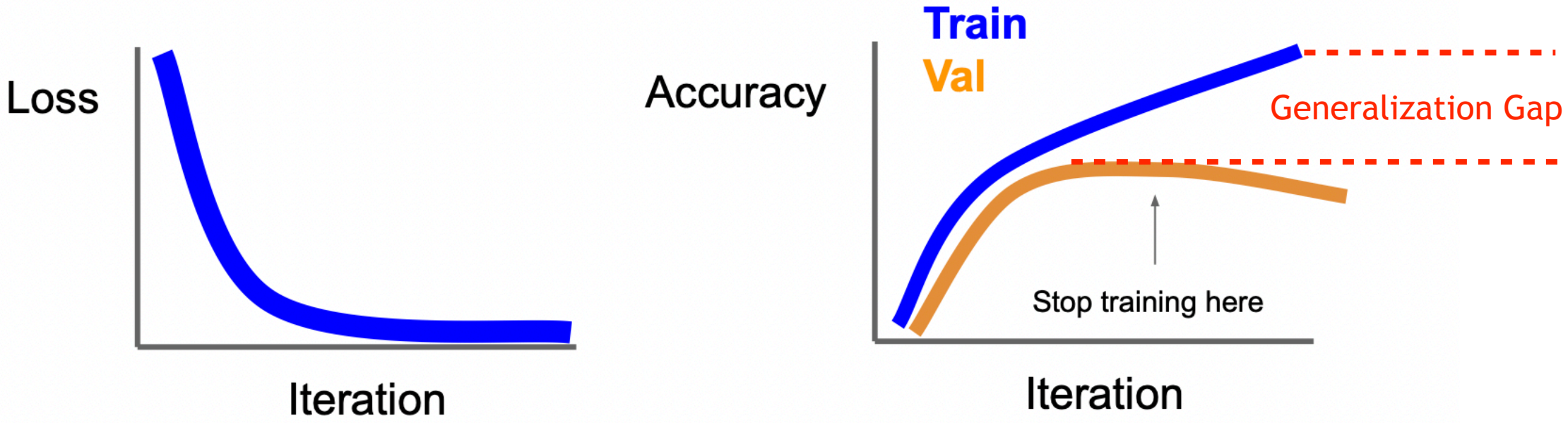
Li, Hao, et al. "Visualizing the loss landscape of neural nets." *Advances in neural information processing systems* 31 (2018).

Overfitting

- Underfitting on the train set: usually caused by limited model capacity or unsatisfactory optimization
  - Batch normalization
  - Skip link

- Overfitting on the test set:
  - Data augmentation
  - Regularization
  - Dropout

The Generalization Gap



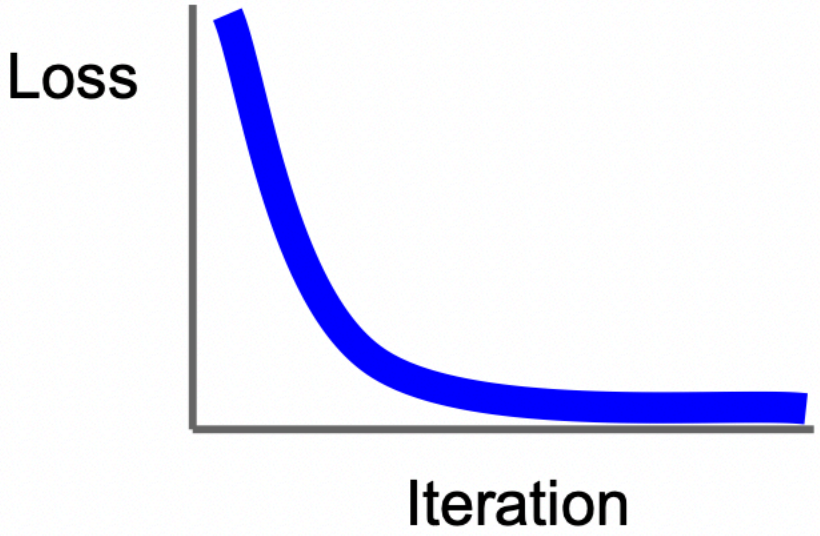
Generalization gap: the difference between a model's performance on training data and its performance on unseen data drawn from the same distribution.

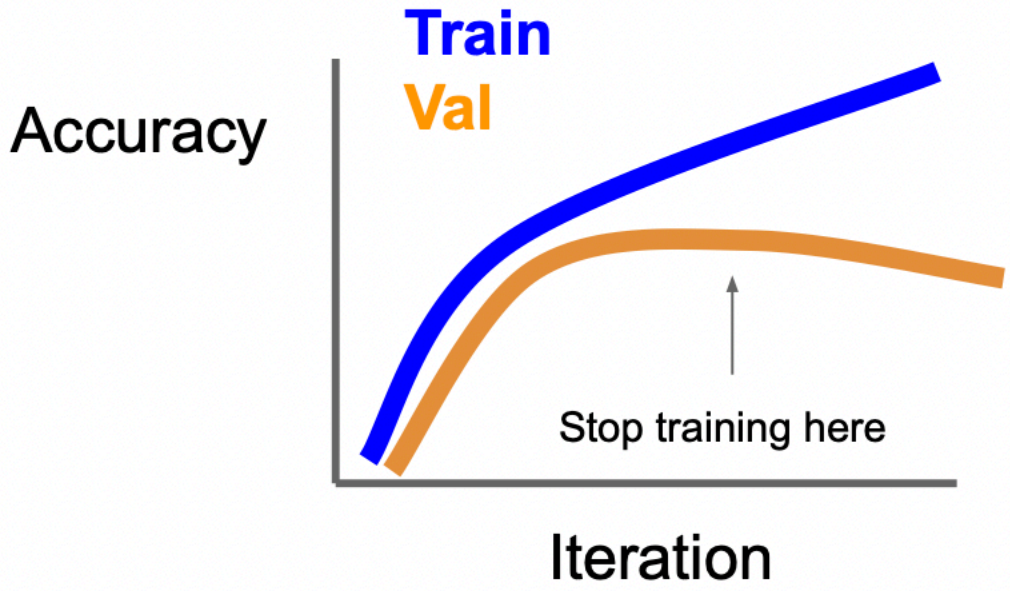
Overfitting

- Underfitting on the train set: usually caused by limited model capacity or unsatisfactory optimization
  - Batch normalization
  - Skip link

- Overfitting on the test set: **usually caused by imbalance between data and model**
  - Data augmentation
  - Regularization
  - Dropout

Early Stopping



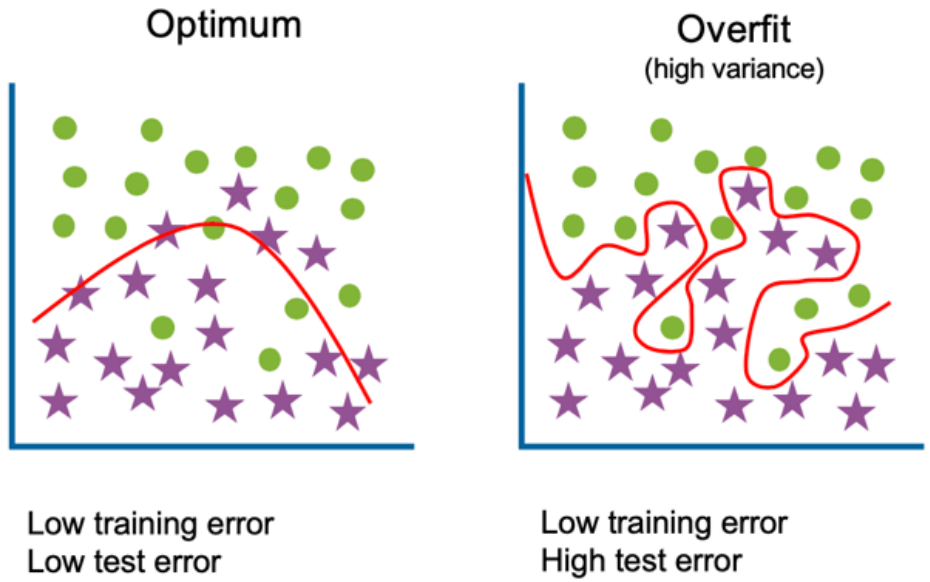


Stop training the model when accuracy on the validation set decreases
Or train for a long time, but always keep track of the model snapshot that worked best on val

Slide credit: Stanford CS231N

The Generalization Gap and Overfitting

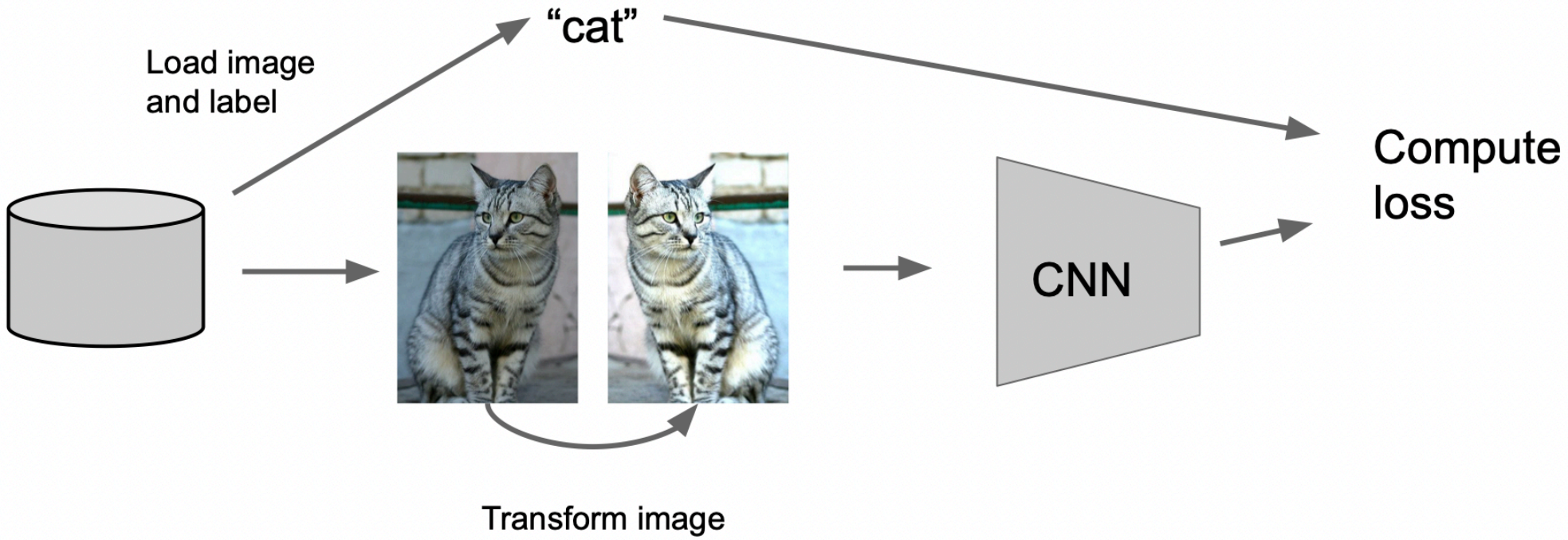
- For an overfitted model, it contains more parameters than can be justified by the data.
- The essence of overfitting is to have unknowingly extracted some of the residual variation (i.e., the noise) as if that variation represented underlying model structure
- To minimize generalization gap, we consider to minimize the mismatch between your model and your data.



From the Perspective of Data

- If your data exhibit sufficient variations, then an appropriate model can't easily overfit.
- To increase the diversity of your data
  - simply collect more data (expensive and time consuming)
  - Data augmentation (free and fast)

Data Augmentation



Data augmentation is a set of techniques to artificially increase the amount of data by gene rating new data points from existing data. This includes making small changes to data or usi ng deep learning models to generate new data points.

Figure credit: Stanford CS231N

Simplest Data Augmentation: Horizontal Flip

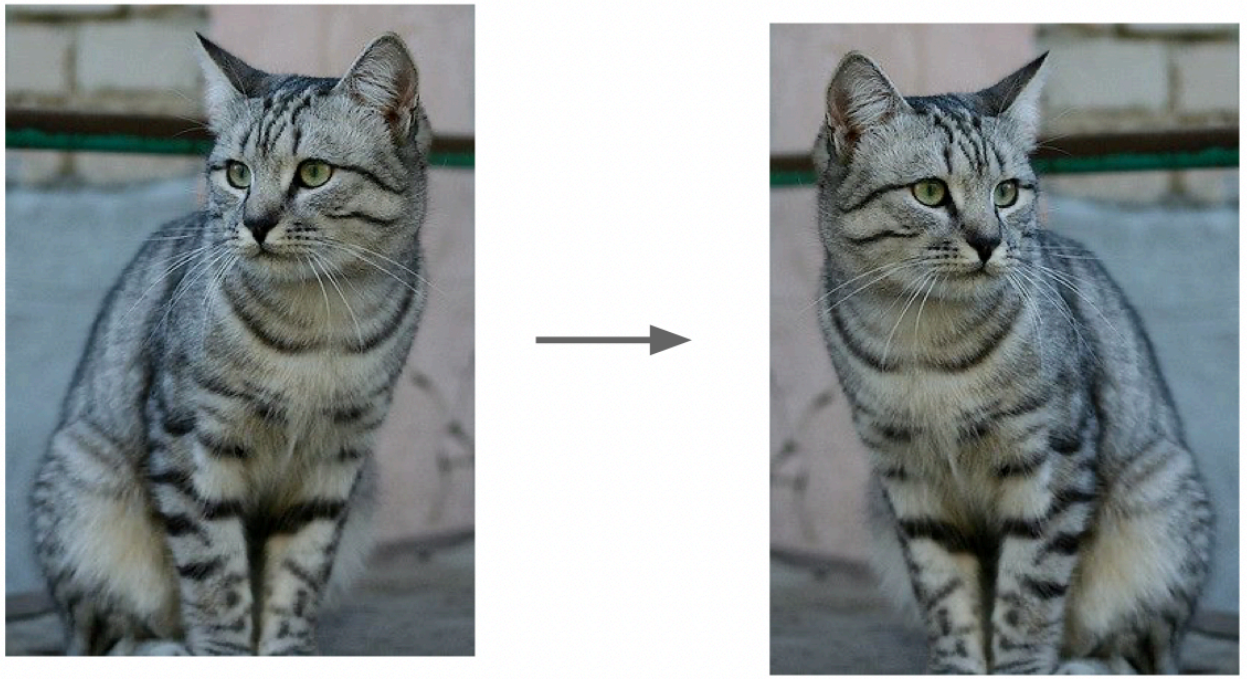
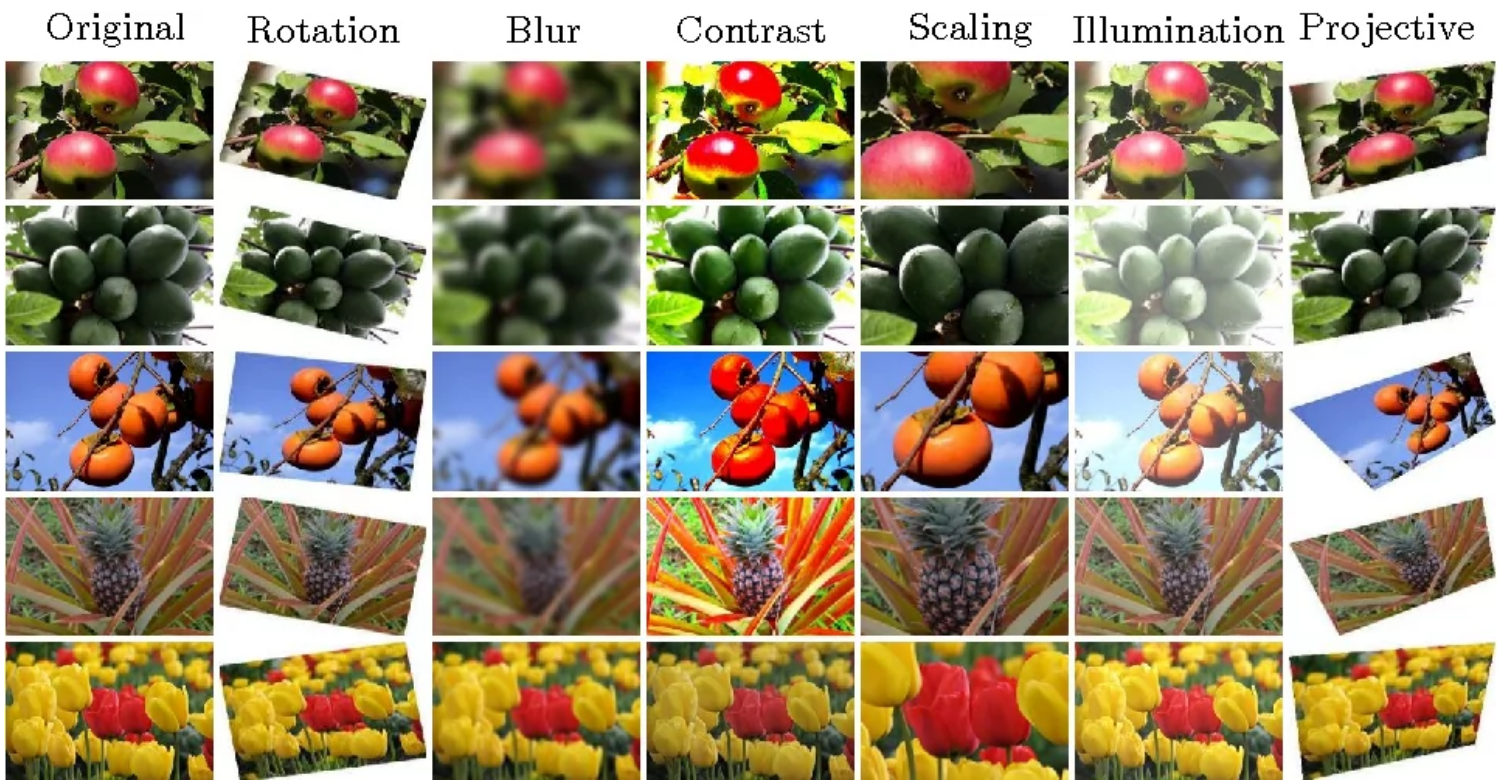


Figure credit: Stanford CS231N

- Data augmentation applies changes to the image while maintaining the label unchanged.
- The thing you cares must be invariant under the transformation of data augmentation.

Data Augmentation Gallery

- **Position augmentation**
  - **Scaling**
  - **Cropping**
  - **Flipping**
  - **Padding**
  - **Rotation**
  - **Translation**
  - **Affine transformation**
- **Color augmentation**
  - **Brightness**
  - **Contrast**
  - **Saturation**
  - **Hue**



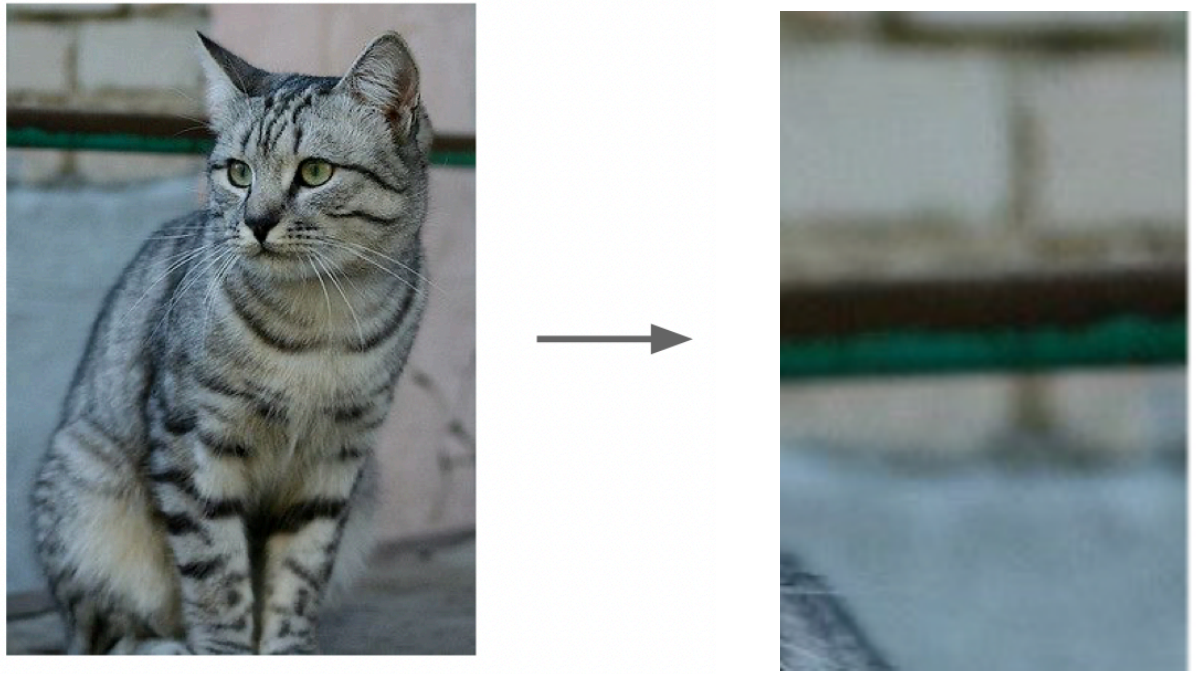
- **Applying GAN/RL for data augmentation**

https://research.aimultiple.com/data-augmentation/

Benefit of Using Data Augmentations

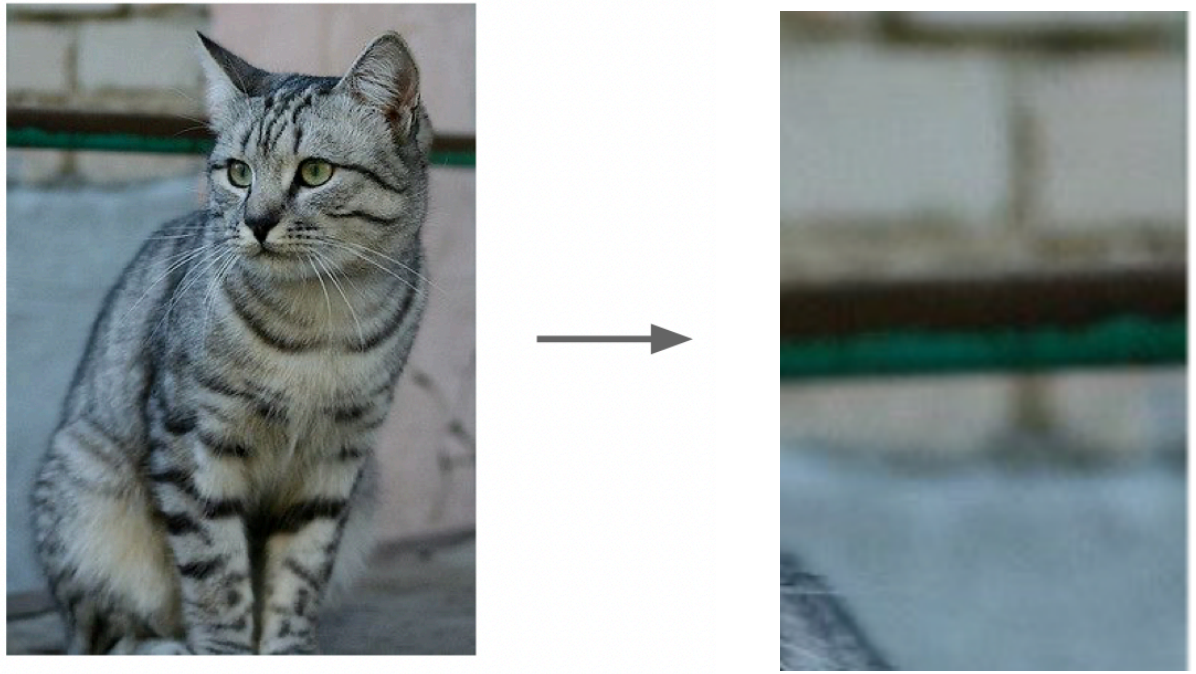
- Improving model prediction accuracy
  - reducing data overfitting and creating variability in data
  - increasing generalization ability of the models
  - helping resolve class imbalance issues in classification

Doing Right Data Augmentations



- The magnitude of DA can’t be too strong. If core information is lost, then model can’t learn.

Doing Right Data Augmentations



- The magnitude of DA can’t be too strong. If core information is lost, then model can’t learn.
- The magnitude of DA shouldn’t be too weak, otherwise no use.
- Decide the magnitude by human or by tuning parameters.

Overfitting

- Underfitting on the train set: usually caused by limited model capacity or unsatisfactory optimization
  - Batch normalization
  - Skip link
- Overfitting on the test set: usually caused by imbalance between data and model
  - Data augmentation
  - Regularization
  - Dropout

Regularization

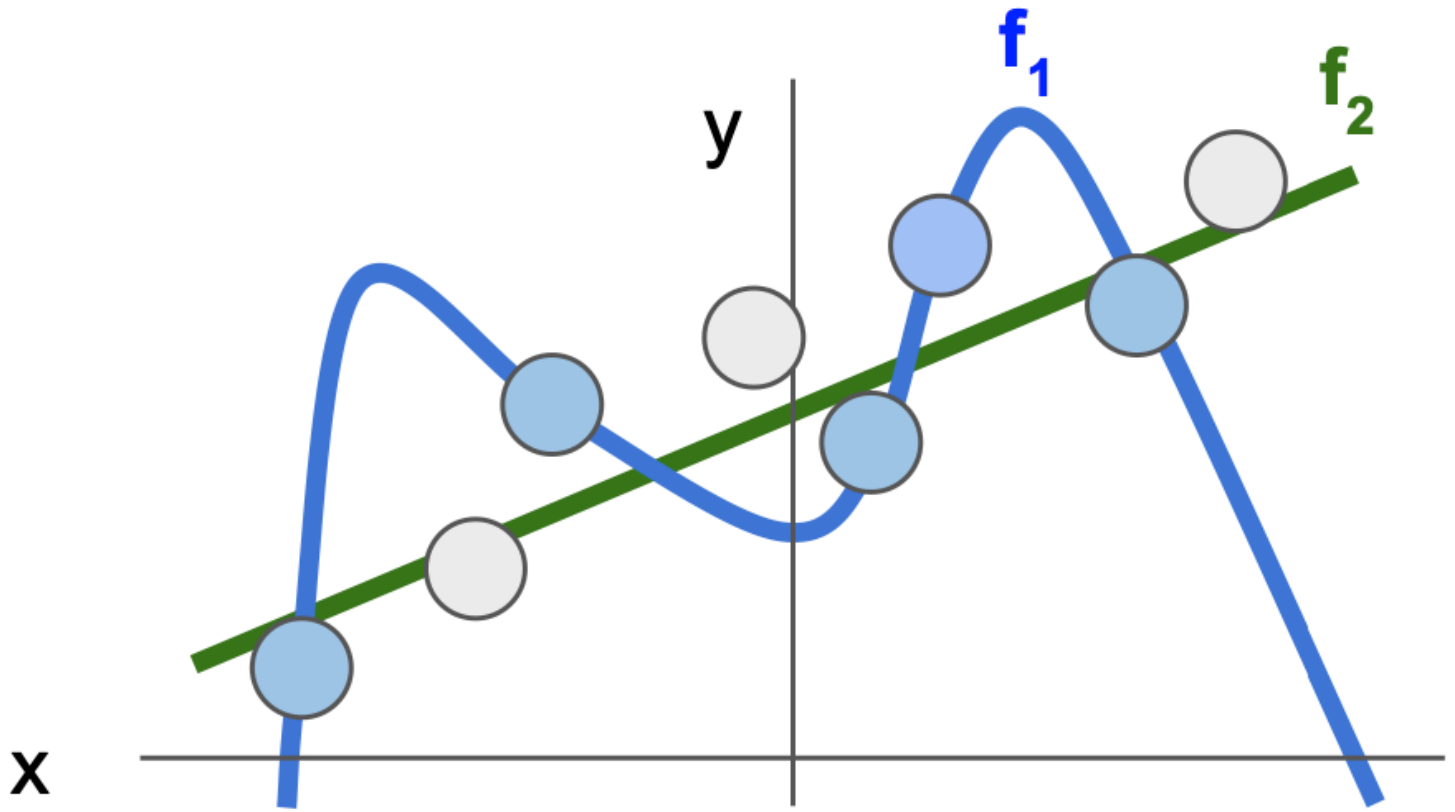
- Avoid the model to be arbitrarily complex

$$L(W) = \underbrace{\frac{1}{N}\sum_{i=1}^{N} L_i(f(x_i, W), y_i)}_{\text{Data loss}} + \underbrace{\lambda R(W)}_{\text{Regularization}}$$

Data loss: Model predictions should match training data

Regularization: Prevent the model from doing *too* well on training data

Regularization: Prefer Simpler Models



Regularization pushes against fitting the data *too* well so we don't fit noise in the data

Regularization

- Avoid the model to be arbitrarily complex

$$L(W) = \underbrace{\frac{1}{N}\sum_{i=1}^{N} L_i(f(x_i, W), y_i)}_{\text{Data loss}} + \underbrace{\lambda R(W)}_{\text{Regularization}}$$

Data loss: Model predictions should match training data

Regularization: Prevent the model from doing *too* well on training data

Occam's Razar: Among multiple competing hypotheses, the simplest is the best,
William of Ockham 1285-1347

Regularization from the Model Perspective

- Avoid the model to be arbitrarily complex

$$\mathscr{L} = \mathscr{L}_{main} + \lambda R(W)$$

In common use:

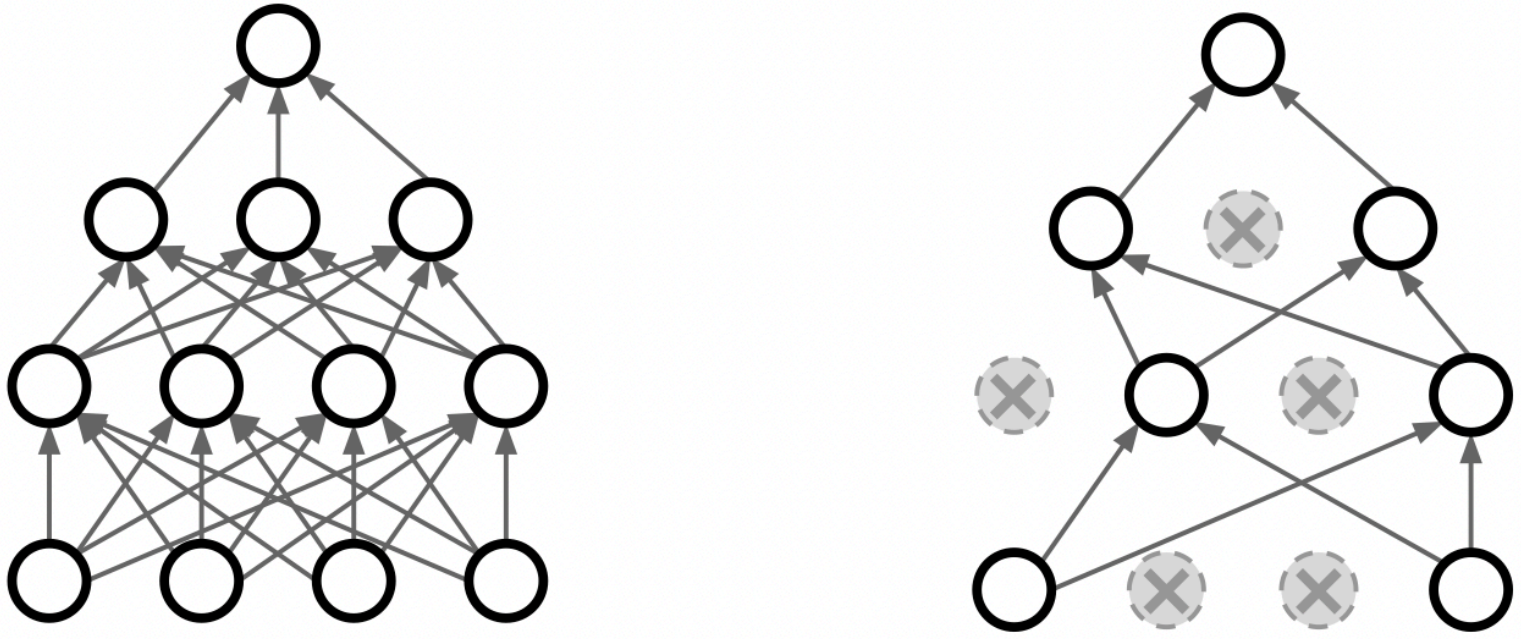
L2 regularization $R(W) = \sum_k \sum_l W_{k,l}^2$ (Weight decay)

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

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

Dropout

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



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

Slide credit: Stanford CS231N

Dropout

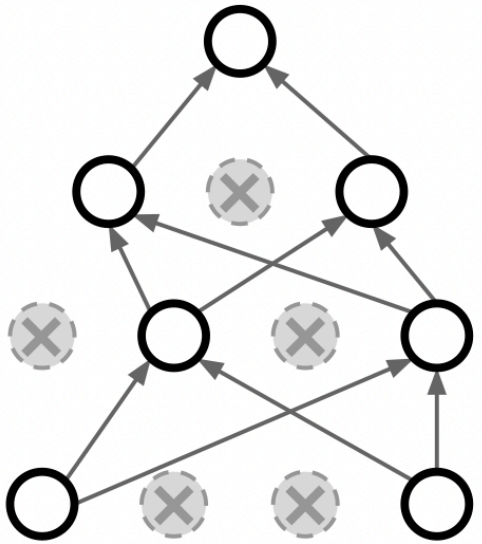
```python
p = 0.5 # probability of keeping a unit active. higher = less dropout

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)
```

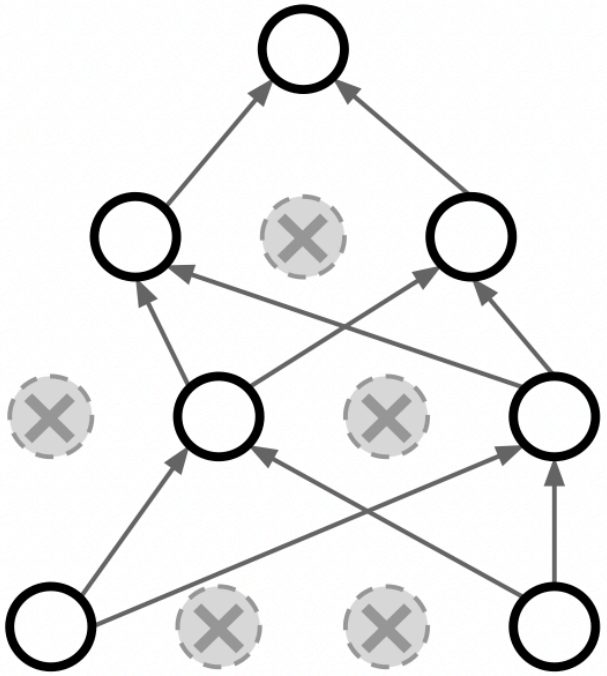
Example forward pass with a 3-layer network using dropout



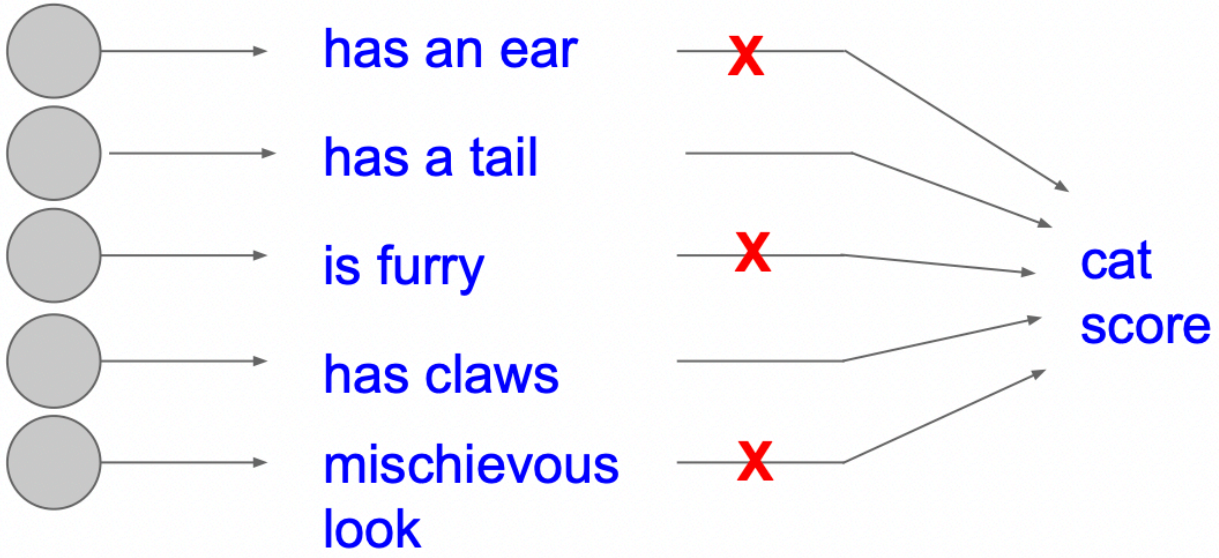
Slide credit: Stanford CS231N

Dropout

How can this possibly be a good idea?



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



Slide credit: Stanford CS231N

Dropout: Test Time

```
""" Vanilla Dropout: Not recommended implementation (see notes below) """

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

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

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

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

def predict(X):
  # ensembled forward pass
  H1 = np.maximum(0, np.dot(W1, X) + b1) * p # NOTE: scale the activations
  H2 = np.maximum(0, np.dot(W2, H1) + b2) * p # NOTE: scale the activations
  out = np.dot(W3, H2) + b3
```

Dropout Summary

drop in train time

scale at test time

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

Slide credit: Stanford CS231N

BatchNorm as a Regularization

- BatchNorm forces the output before activations to follow a certain Gaussian distribution, which limits the capacity of a model ==> Regularization
- BatchNorm thus helps alleviate overfitting.
- With BatchNorm, people may not need dropout.

Summary of Mitigating Overfitting

- Principle:
  - to balance the **data** variability and the **model** capacity

- Techniques:
  - Data augmentation (from the data perspective)
  - BatchNorm (from the data perspective)
  - Regularization (from the model perspective)
  - Dropout (from the model perspective)
  - …

(Always good to use)

(used only for large FC layers)

Next week: Lecture 7,
Deep Learning IV