

CS 4644-DL / 7643-A: LECTURE 10

DANFEI XU

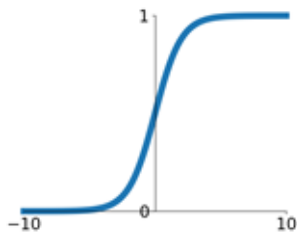
Topics:

- Training Neural Networks (Part 2)

Activation Functions

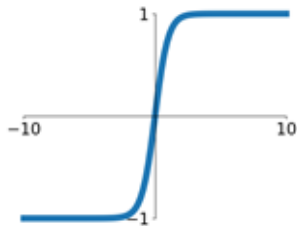
Sigmoid

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



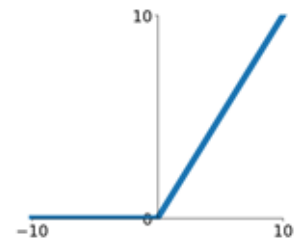
tanh

$$\tanh(x)$$



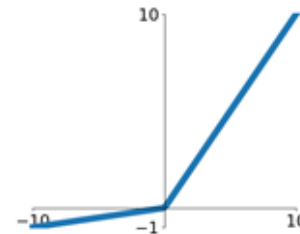
ReLU

$$\max(0, x)$$



Leaky ReLU

$$\max(0.1x, x)$$

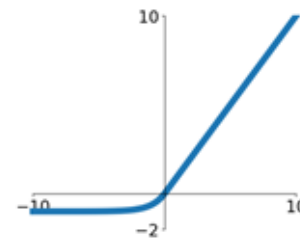


Maxout

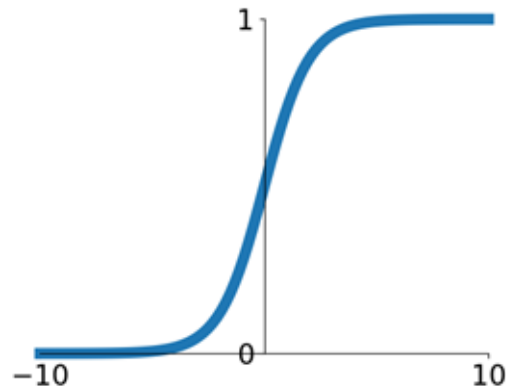
$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

ELU

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



Activation Functions



Sigmoid

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

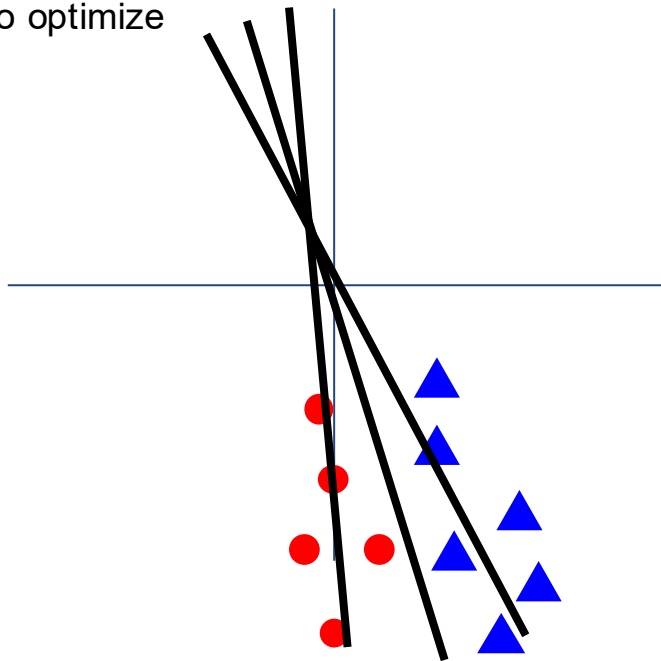
- Squashes numbers to range [0,1]
- Historically popular since they have nice interpretation as a saturating “firing rate” of a neuron

2 problems:

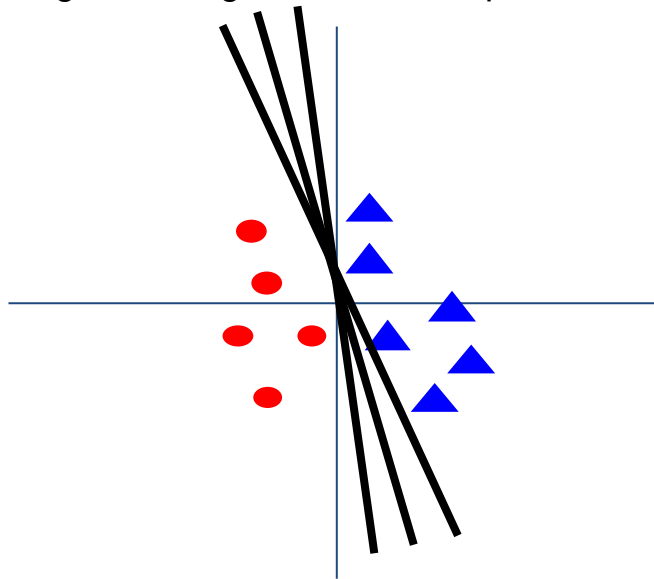
1. **Saturated neurons “kill” the gradients**
2. **exp() is a bit compute expensive**

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



Batch Normalization

[Ioffe and Szegedy, 2015]

Input: $x : N \times D$
Learnable scale and shift parameters:

$$\gamma, \beta : \mathbb{R}^D$$

We want to give the model a chance to **adjust batchnorm** if the default is not optimal.

Learning $\gamma = \sigma$ and $\beta = \mu$ will recover the original input batch!

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

Per-batch mean,
shape is D

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

Per-batch var,
shape is D

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

Normalized x,
Shape is N x D

$$y_{i,j} = \underline{\gamma}_j \hat{x}_{i,j} + \underline{\beta}_j$$

Output,
Shape is N x D

Initialize $\gamma = 1, \beta = 0$

Batch Normalization: Test-Time

Input: $x : N \times D$
Learnable scale and shift parameters:

$$\gamma, \beta: \mathbb{R}^D$$

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

$$\mu_j = \text{(Moving) average of values seen during training}$$

Per-batch mean, shape is D

$$\sigma_j^2 = \text{(Moving) average of values seen during training}$$

Per-batch var, shape is D

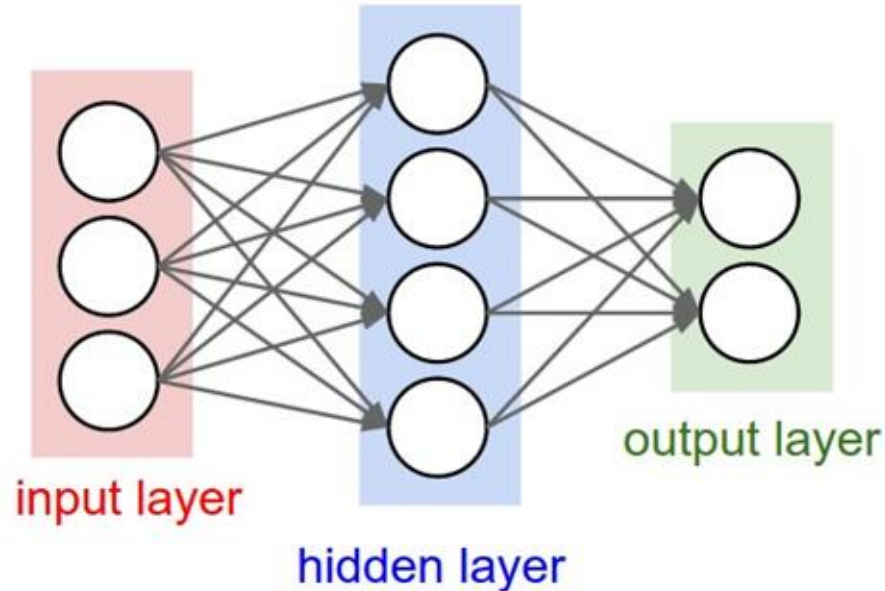
$$\hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \epsilon}}$$

Normalized x,
Shape is N x D

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

Output,
Shape is N x D

Weight initialization: goal is to **maximize diversity with bounded variance** of layer output throughout the network, at least at the beginning of the training



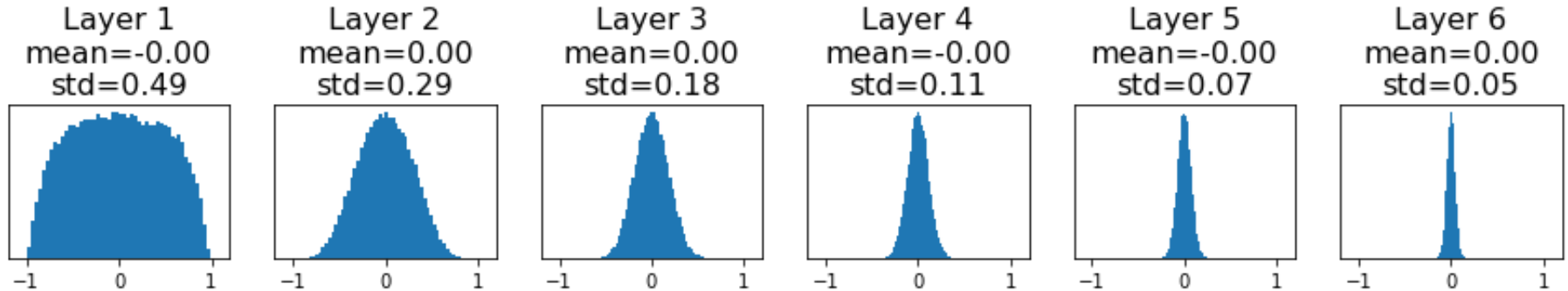
Weight Initialization: Activation statistics

```
dims = [4096] * 7      Forward pass for a 6-layer
hs = []               net with hidden size 4096
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)
```

All activations tend to zero for deeper network layers

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

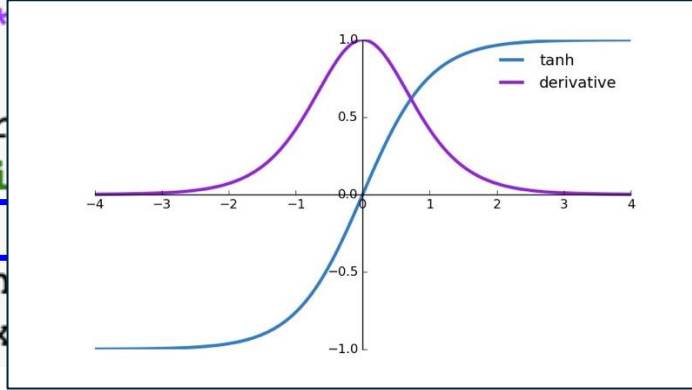
A: Very small, slow learning



Visualize distribution of activations

Weight Initialization: Activation statistics

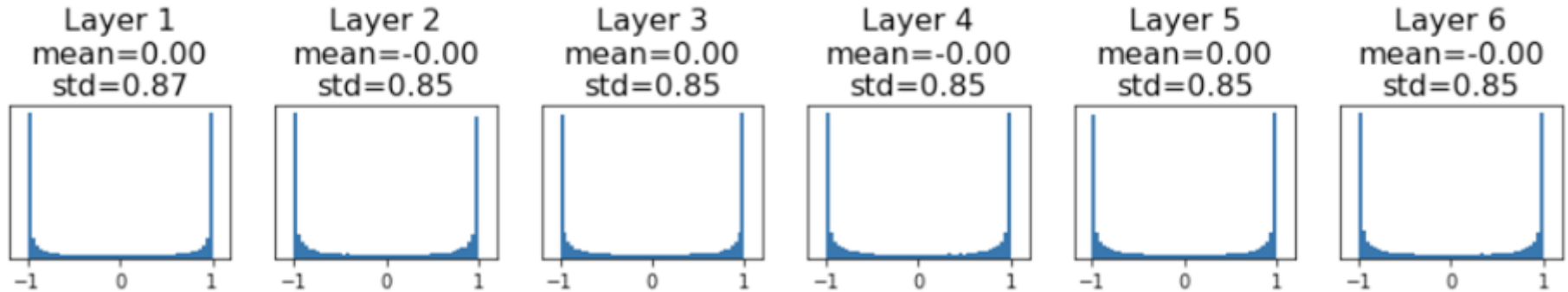
```
dims = [4096] *  
hs = []  
x = np.random.r  
for Din, Dout i  
    W = 0.05 *  
    x = np.tanh  
    hs.append(x
```



All activations saturate

Q: What do the gradients look like?

A: For tanh, large value $\rightarrow$ small gradient



Visualize distribution of activations

Weight Initialization: Activation statistics

```
dims = [4096] * 7    Increase std of initial
hs = []              weights from 0.01 to 0.05
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)
```

All activations saturate

Q: What do the gradients look like?

More generally, *gradient explosion* (high $w \rightarrow$ high output $\rightarrow$ high gradient).

Visualize distribution of activations

Weight Initialization: “Xavier” Initialization

```
dims = [4096] * 7          “Xavier” initialization:
hs = []                    std = 1/sqrt(Din)
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)
```

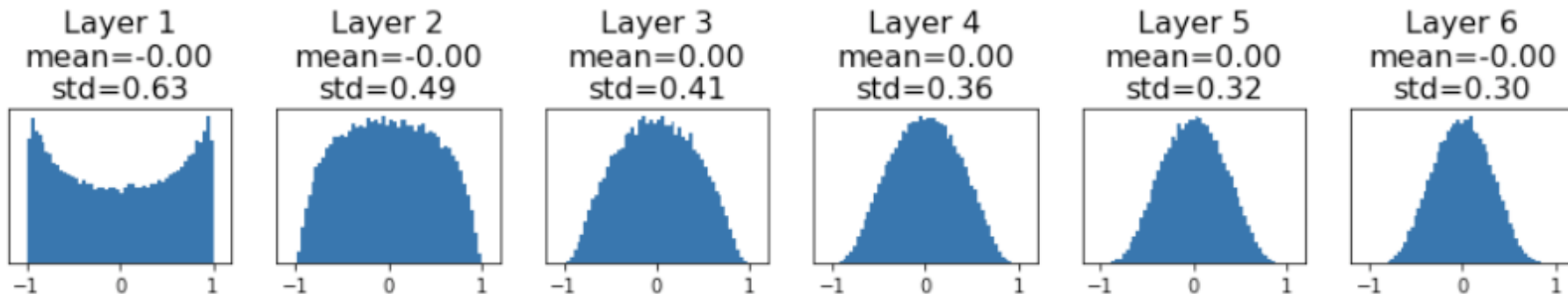
Assume each input contribute similarly to output
more number of weights needs -> small weight multiplier

Weight Initialization: “Xavier” Initialization

```
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:
 $\text{std} = 1/\sqrt{D_{\text{in}}}$

“Just right”: Activations are nicely scaled for all layers!



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

Visualize distribution of activations

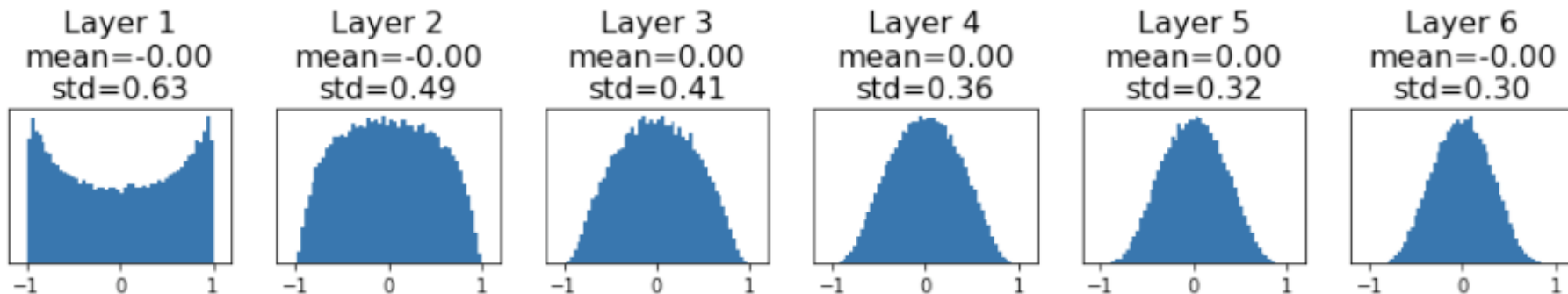
Weight Initialization: “Xavier” Initialization

```
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 = $1/\sqrt{D_{in}}$

“Just right”: Activations are nicely scaled for all layers!

For conv layers, D_{in} is $\text{filter_size}^2 * \text{input_channels}$



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

Visualize distribution of activations

Weight Initialization: “Xavier” Initialization

```
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 = $1/\sqrt{D_{in}}$

“Just right”: Activations are nicely scaled for all layers!

For conv layers, D_{in} is $\text{filter_size}^2 * \text{input_channels}$

Let: $y = x_1 w_1 + x_2 w_2 + \dots + x_{D_{in}} w_{D_{in}}$

Weight Initialization: “Xavier” Initialization

```
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 = $1/\sqrt{D_{in}}$

“Just right”: Activations are nicely scaled for all layers!

For conv layers, D_{in} is $\text{filter_size}^2 * \text{input_channels}$

Let: $y = x_1 w_1 + x_2 w_2 + \dots + x_{D_{in}} w_{D_{in}}$

Assume: $\text{Var}(x_1) = \text{Var}(x_2) = \dots = \text{Var}(x_{D_{in}})$

Weight Initialization: “Xavier” Initialization

```
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 = $1/\sqrt{D_{in}}$

“Just right”: Activations are nicely scaled for all layers!

For conv layers, D_{in} is $\text{filter_size}^2 * \text{input_channels}$

Let: $y = x_1 w_1 + x_2 w_2 + \dots + x_{D_{in}} w_{D_{in}}$

Assume: $\text{Var}(x_1) = \text{Var}(x_2) = \dots = \text{Var}(x_{D_{in}})$

We want: $\text{Var}(y) = \text{Var}(x_i)$

Weight Initialization: “Xavier” Initialization

```
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 = $1/\sqrt{D_{in}}$

“Just right”: Activations are nicely scaled for all layers!

For conv layers, D_{in} is $\text{filter_size}^2 * \text{input_channels}$

Let: $y = x_1 w_1 + x_2 w_2 + \dots + x_{D_{in}} w_{D_{in}}$

Assume: $\text{Var}(x_1) = \text{Var}(x_2) = \dots = \text{Var}(x_{D_{in}})$

We want: $\text{Var}(y) = \text{Var}(x_i)$

$\text{Var}(y) = \text{Var}(x_1 w_1 + x_2 w_2 + \dots + x_{D_{in}} w_{D_{in}})$
[substituting value of y]

Weight Initialization: “Xavier” Initialization

```
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 = $1/\sqrt{D_{in}}$

“Just right”: Activations are nicely scaled for all layers!

For conv layers, D_{in} is $\text{filter_size}^2 * \text{input_channels}$

Let: $y = x_1 w_1 + x_2 w_2 + \dots + x_{D_{in}} w_{D_{in}}$

Assume: $\text{Var}(x_1) = \text{Var}(x_2) = \dots = \text{Var}(x_{D_{in}})$

We want: $\text{Var}(y) = \text{Var}(x_i)$

$$\text{Var}(y) = \text{Var}(x_1 w_1 + x_2 w_2 + \dots + x_{D_{in}} w_{D_{in}})$$
$$= \sum \text{Var}(x_i w_i) = D_{in} \text{Var}(x_i w_i)$$

[Assume all x_i, w_i are iid] $\sigma_{X+Y}^2 = \sigma_X^2 + \sigma_Y^2$

Weight Initialization: “Xavier” Initialization

```
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 = $1/\sqrt{D_{in}}$

“Just right”: Activations are nicely scaled for all layers!

For conv layers, D_{in} is $\text{filter_size}^2 * \text{input_channels}$

Let: $y = x_1 w_1 + x_2 w_2 + \dots + x_{D_{in}} w_{D_{in}}$

Assume: $\text{Var}(x_1) = \text{Var}(x_2) = \dots = \text{Var}(x_{D_{in}})$

We want: $\text{Var}(y) = \text{Var}(x_i)$

$$\begin{aligned}\text{Var}(y) &= \text{Var}(x_1 w_1 + x_2 w_2 + \dots + x_{D_{in}} w_{D_{in}}) \\ &= D_{in} \text{Var}(x_i w_i) \\ &= D_{in} \text{Var}(x_i) \text{Var}(w_i)\end{aligned}$$

[Assume all x_i, w_i are zero mean]

$$\begin{aligned}\text{Var}(XY) &= E(X^2 Y^2) - (E(XY))^2 = \text{Var}(X)\text{Var}(Y) + \cancel{\text{Var}(X)(E(Y))^2} \\ &\quad + \cancel{\text{Var}(Y)(E(X))^2}\end{aligned}$$

Weight Initialization: “Xavier” Initialization

```
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 = $1/\sqrt{D_{in}}$

“Just right”: Activations are nicely scaled for all layers!

For conv layers, D_{in} is $\text{filter_size}^2 * \text{input_channels}$

Let: $y = x_1 w_1 + x_2 w_2 + \dots + x_{D_{in}} w_{D_{in}}$

Assume: $\text{Var}(x_1) = \text{Var}(x_2) = \dots = \text{Var}(x_{D_{in}})$

We want: $\text{Var}(y) = \text{Var}(x_i)$

$$\begin{aligned}\text{Var}(y) &= \text{Var}(x_1 w_1 + x_2 w_2 + \dots + x_{D_{in}} w_{D_{in}}) \\ &= D_{in} \text{Var}(x_i w_i) \\ &= D_{in} \text{Var}(x_i) \text{Var}(w_i) \\ &[\text{Assume all } x_i, w_i \text{ are iid}]\end{aligned}$$

So, $\text{Var}(y) = \text{Var}(x_i)$ only when $\text{Var}(w_i) = 1/D_{in}$

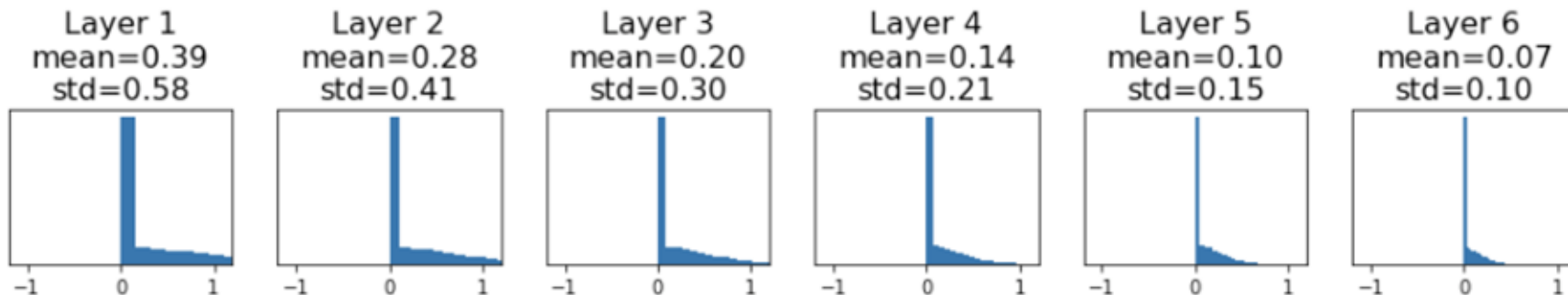
Weight Initialization: What about ReLU?

```
dims = [4096] * 7      Change from tanh to ReLU
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)
```

Weight Initialization: What about ReLU?

```
dims = [4096] * 7      Change from tanh to ReLU
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)
```

Xavier assumes zero centered activation function



Visualize distribution of activations

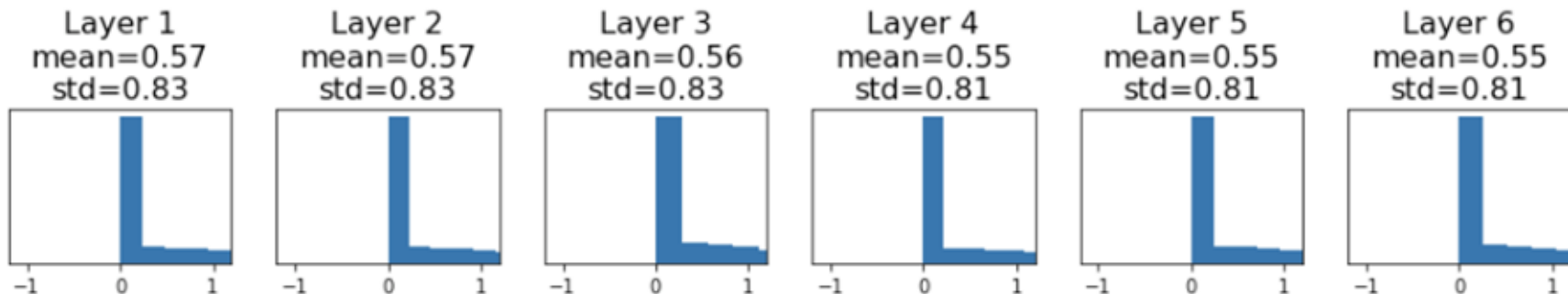
Weight Initialization: Kaiming / MSRA Initialization

```
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: $\text{std} = \sqrt{2 / \text{Din}}$

Issue: Half of the activation get killed.

Solution: make the non-zero output variance twice as large as input



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

Visualize distribution of activations

Proper initialization is still an active area of research...

Understanding the difficulty of training deep feedforward neural networks

by Glorot and Bengio, 2010

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

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

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

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

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

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

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

(Fancier) Optimizers

Optimization: (Stochastic) Gradient Descent

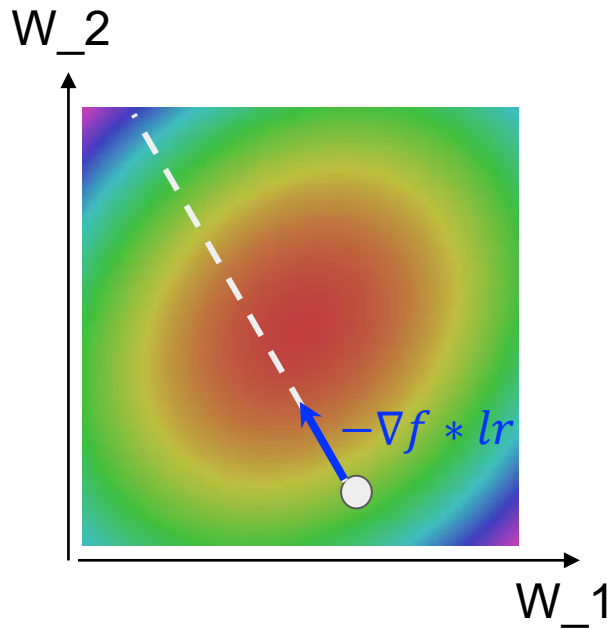
(Batch) Gradient Descent

```
While True:  
    loss = model.compute_loss(dataset)  
    loss.backward()  
    model.weights -= model.weights.grad * lr
```

Dataset may be really large (millions of images)!

Minibatch (Stochastic) Gradient Descent

```
While True:  
    minibatch = sample(dataset, batch_size)  
    loss = model.compute_loss(minibatch)  
    loss.backward()  
    model.weights -= model.weights.grad * lr
```



Optimization: (Stochastic) Gradient Descent

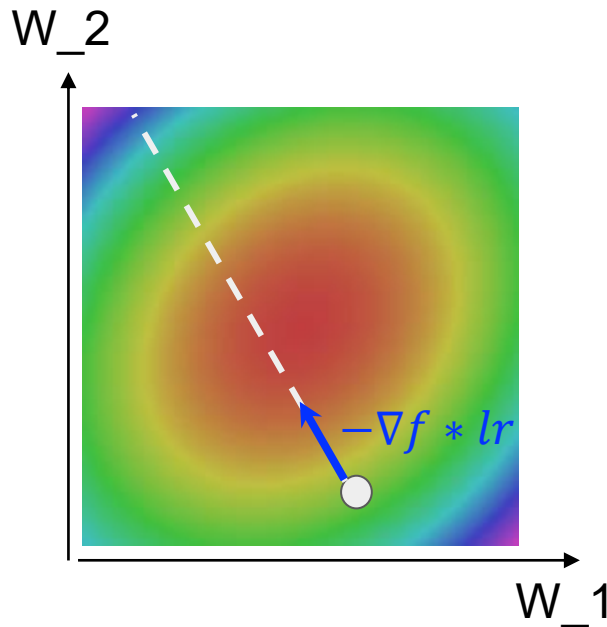
Minibatch (Stochastic) Gradient Descent

While True:

```
    minibatch = sample(dataset, batch_size)
    loss = model.compute_loss(minibatch)
    loss.backward()
    model.weights -= model.weights.grad * lr
```

Reasons to prefer SGD over GD for Deep Learning:

- More computationally-tractable
- GD doesn't guarantee optimality for non-convex functions anyways



Optimization: (Stochastic) Gradient Descent

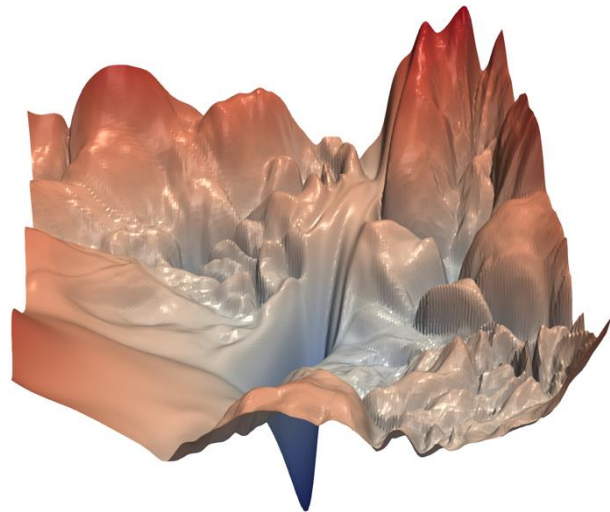
Minibatch (Stochastic) Gradient Descent

While True:

```
    minibatch = sample(dataset, batch_size)
    loss = model.compute_loss(minibatch)
    loss.backward()
    model.weights -= model.weights.grad * lr
```

Reasons to prefer SGD over GD for Deep Learning:

- More computationally-tractable
- GD doesn't guarantee optimality for non-convex functions anyways
- SGD usually has faster convergence in wall-clock time, even if you can run GD



Loss landscape for DNN

<https://www.cs.umd.edu/~tomg/projects/landscapes/>

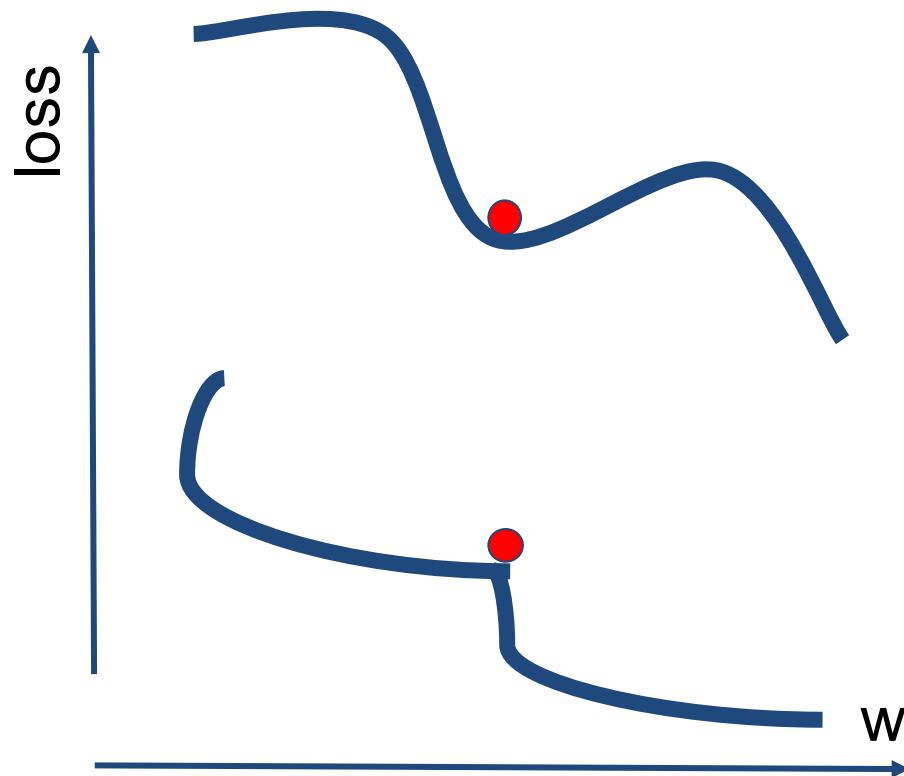
Optimization: Problem #1 with SGD

- Stochastic minibatch gives a **noisy estimate of the true gradient direction**. Approximates dataset gradient if batch is *large enough*.
- Very problematic when the batch size is small (e.g., due to compute resource limit).
- Poorly-selected learning rate makes the oscillation worse (overshoot)



Optimization: Problem #2 with SGD

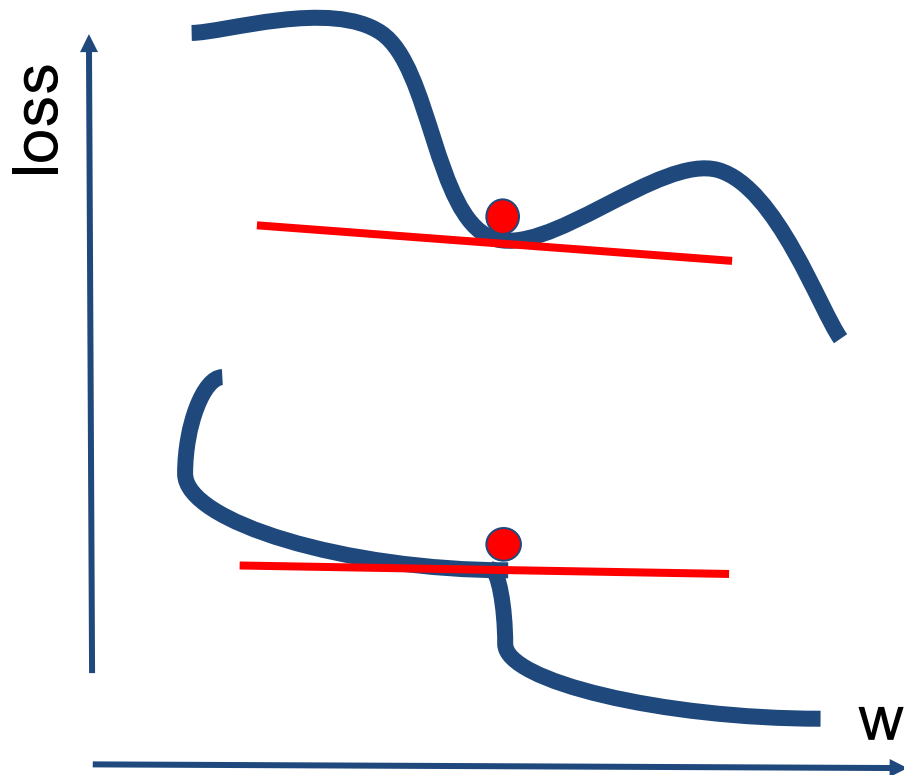
What if the loss function has a **local minima** or **saddle point**?



Optimization: Problem #2 with SGD

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

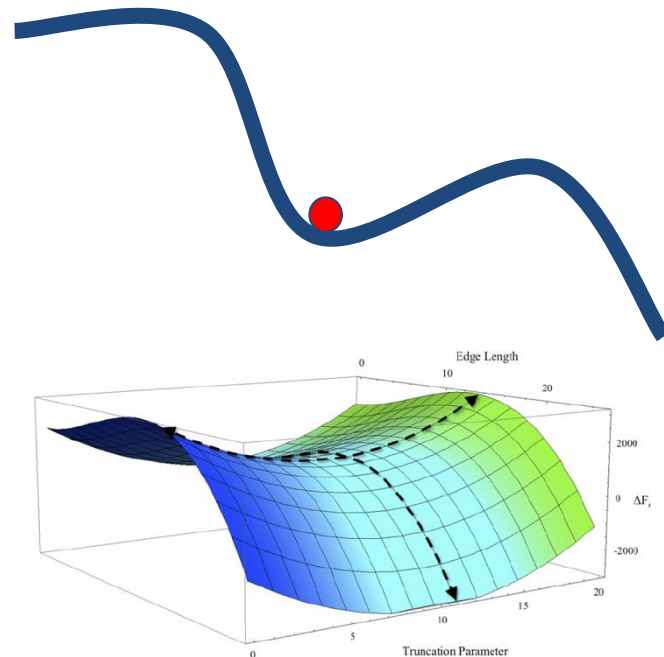
Zero gradient,
gradient descent
gets stuck



Optimization: Problem #2 with SGD

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

Saddle points are much more common in high dimension



<https://blog.paperspace.com/intro-to-optimization-in-deep-learning-gradient-descent/>

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

SGD + Momentum

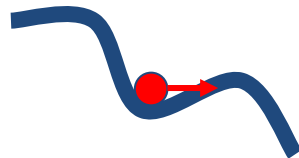
Intuitions:

- Think of a ball (set of parameters) moving in space (loss landscape), with momentum keeping it going in a direction.
- Individual gradient step may be noisy, the general trend accumulated over a few steps will point to the right direction.
- Momentum can “push” the ball over saddle points or local minima.

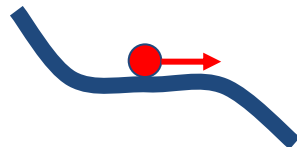
Noisy gradients



Local Minima



Saddle points



SGD: the simple two line update code

SGD

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

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

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/momentum” as a running mean of gradients
- Rho gives “friction”; typically rho=0.9 or 0.99

SGD + Momentum:

alternative equivalent formulation

SGD+Momentum

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

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

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

SGD+Momentum

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

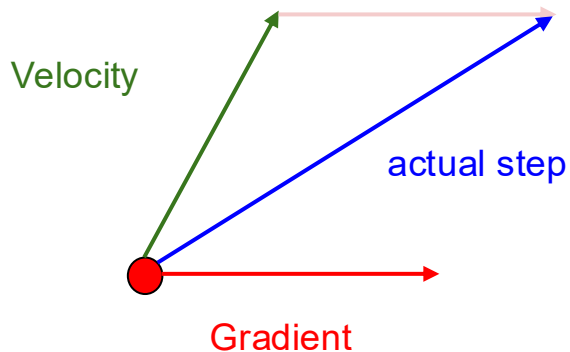
$$x_{t+1} = x_t - \alpha v_{t+1}$$

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

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

SGD+Momentum

Momentum update:



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

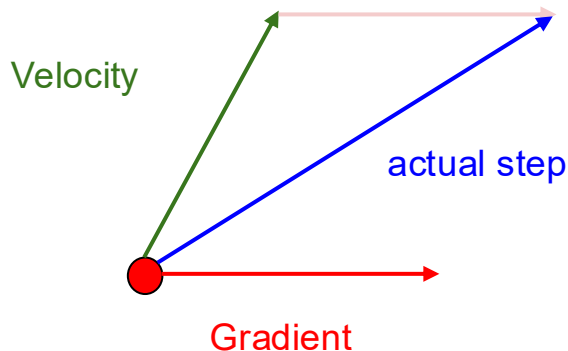
Nesterov, "A method of solving a convex programming problem with convergence rate $O(1/k^2)$ ", 1983

Nesterov, "Introductory lectures on convex optimization: a basic course", 2004

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

Nesterov Momentum

Momentum update:



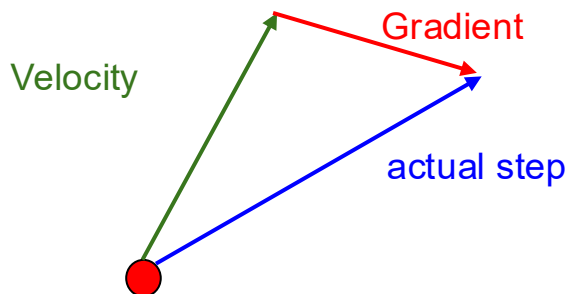
Combine gradient at current point with velocity to get step used to update weights

Nesterov, "A method of solving a convex programming problem with convergence rate $O(1/k^2)$ ", 1983

Nesterov, "Introductory lectures on convex optimization: a basic course", 2004

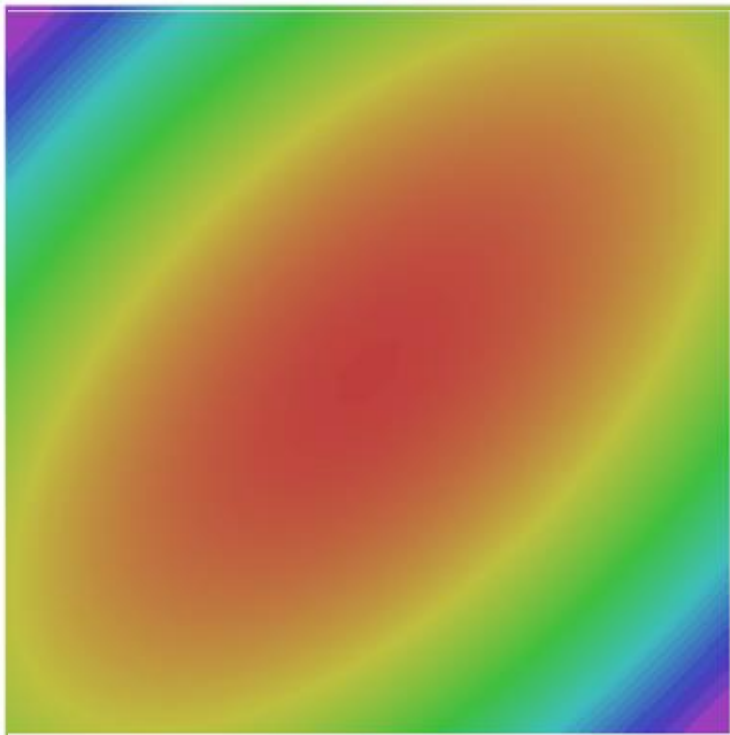
Sutskever et al, "On the importance of initialization and momentum in deep learning", ICML 2013

Nesterov Momentum



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

Nesterov Momentum



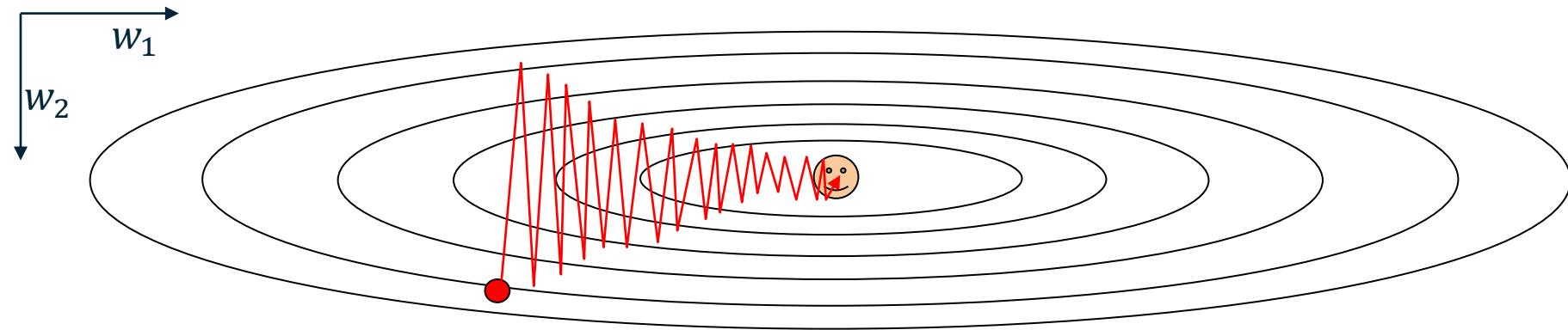
- SGD
- SGD+Momentum
- Nesterov

Optimization: Problem #3 with SGD

What if loss **changes quickly** in one direction and slowly in another?

What does gradient descent do?

Very slow progress along shallow dimension, jitter along steep direction



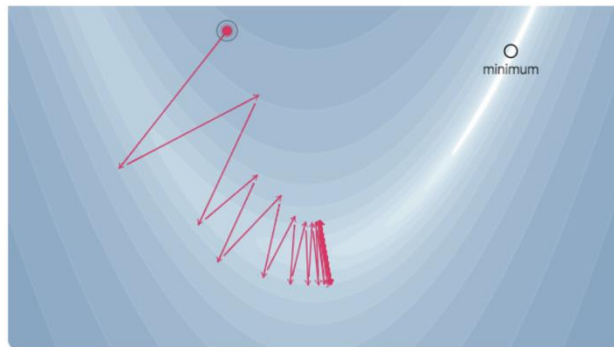
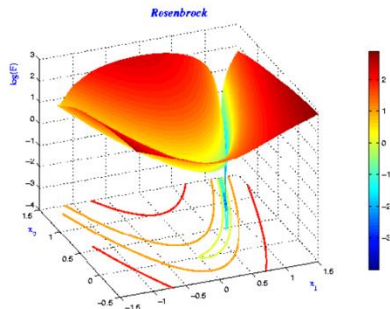
Assume each contour line has the same loss
(iso-loss contour)

Optimization: Problem #3 with SGD

What if loss changes quickly in one direction and slowly in another?

Very slow progress along shallow dimension, jitter along steep direction

Long, narrow ravines:



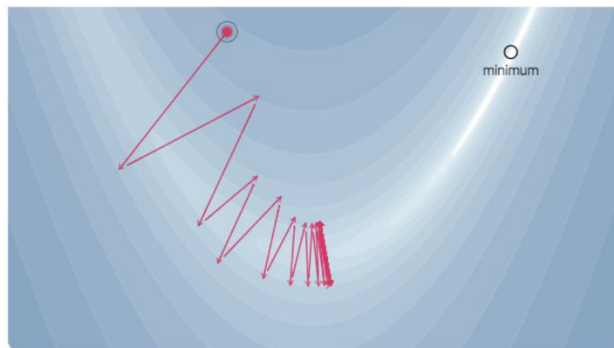
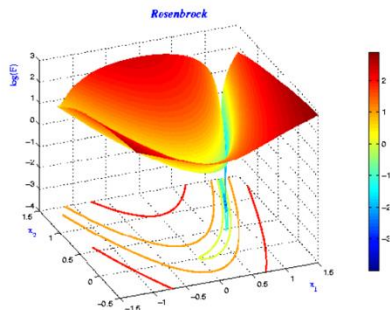
https://www.cs.toronto.edu/~rgrosse/courses/csc421_2019/slides/lec07.pdf

Optimization: Problem #3 with SGD

What if loss changes quickly in one direction and slowly in another?

Very slow progress along shallow dimension, jitter along steep direction

Long, narrow ravines:



https://www.cs.toronto.edu/~rgrosse/courses/csc421_2019/slides/lec07.pdf

Loss function has high **condition number**: ratio of largest to smallest eigen value ($\lambda_{max}/\lambda_{min}$) of the Hessian matrix of a loss function is large

Small condition number in loss Hessian -> circular contour

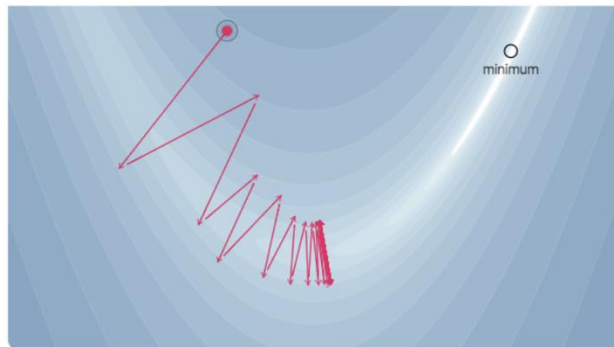
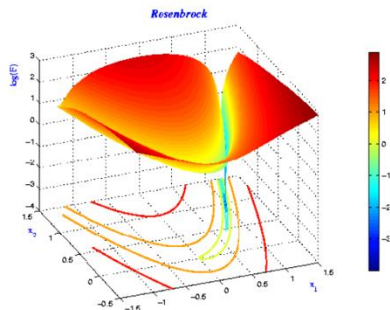
Large condition number in loss Hessian -> skewed contour

Optimization: Problem #3 with SGD

What if loss changes quickly in one direction and slowly in another?

Very slow progress along shallow dimension, jitter along steep direction

Long, narrow ravines:



https://www.cs.toronto.edu/~rgrosse/courses/csc421_2019/slides/lec07.pdf

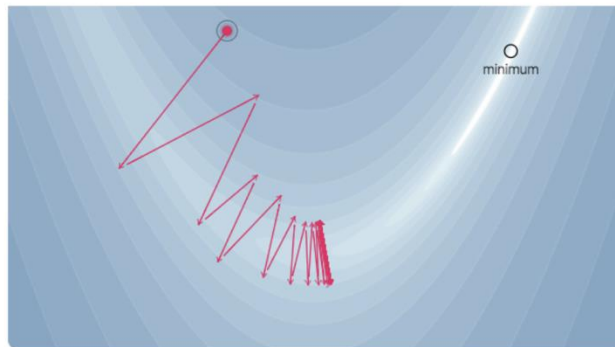
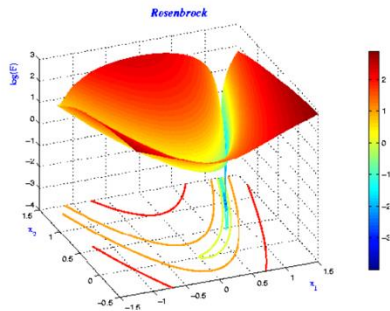
Q: What's the simplest solution to this problem?

Optimization: Problem #3 with SGD

What if loss changes quickly in one direction and slowly in another?

Very slow progress along shallow dimension, jitter along steep direction

Long, narrow ravines:



https://www.cs.toronto.edu/~rgrosse/courses/csc421_2019/slides/lec07.pdf

Q: What's the simplest solution to this problem?

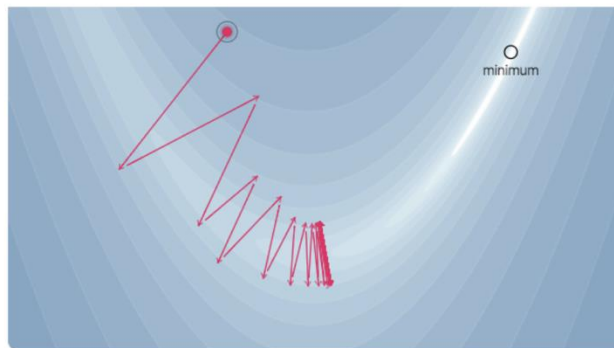
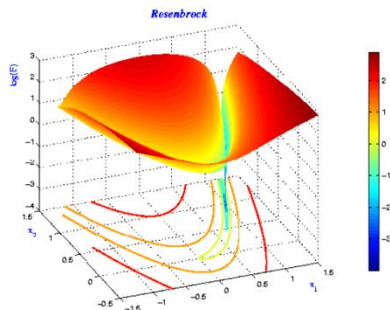
A: Pick a different learning rate for each direction! E.g., lower the learning rate for fast-changing directions

Optimization: Problem #3 with SGD

What if loss changes quickly in one direction and slowly in another?

Very slow progress along shallow dimension, jitter along steep direction

Long, narrow ravines:



https://www.cs.toronto.edu/~rgrosse/courses/csc421_2019/slides/lec07.pdf

Q: What's the simplest solution to this problem?

A: Pick a different learning rate for each direction! E.g., lower the learning rate for fast-changing directions

But manually picking an optimal LR for each weight dimension seems hard ...

AdaGrad

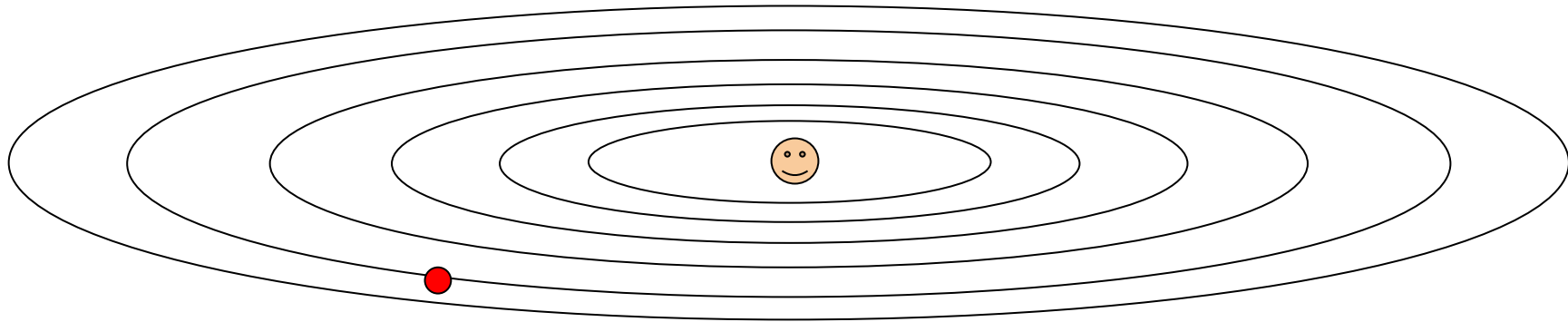
```
grad_squared = 0
while True:
    dx = compute_gradient(x)
    grad_squared += dx * dx
    x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```

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

“Per-parameter learning rates”
or “adaptive learning rates”

AdaGrad

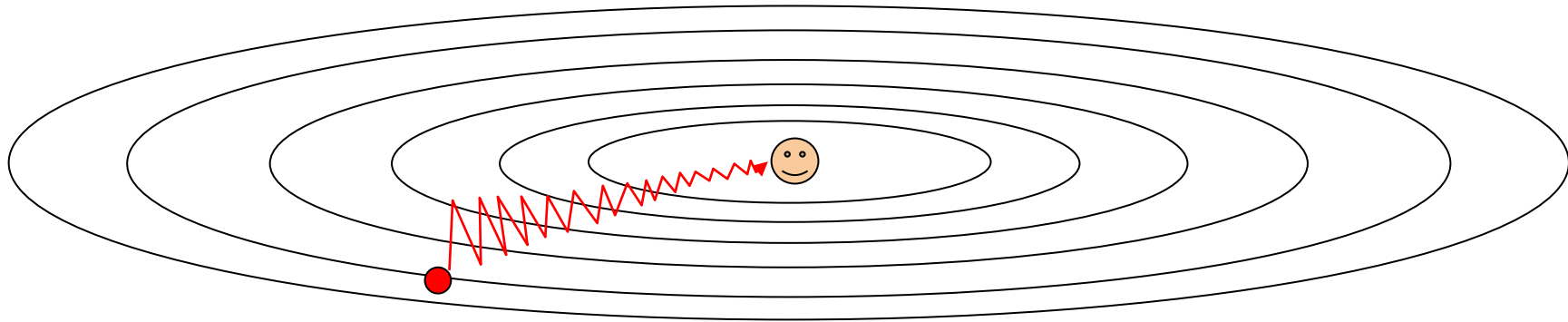
```
grad_squared = 0
while True:
    dx = compute_gradient(x)
    grad_squared += dx * dx
    x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```



Q: What happens with AdaGrad?

AdaGrad

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



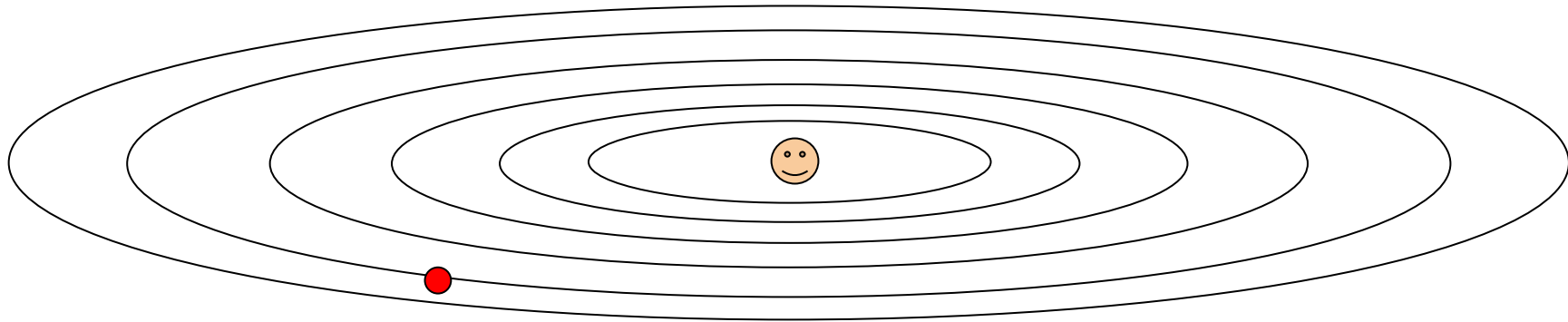
Q: What happens with AdaGrad?

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



AdaGrad

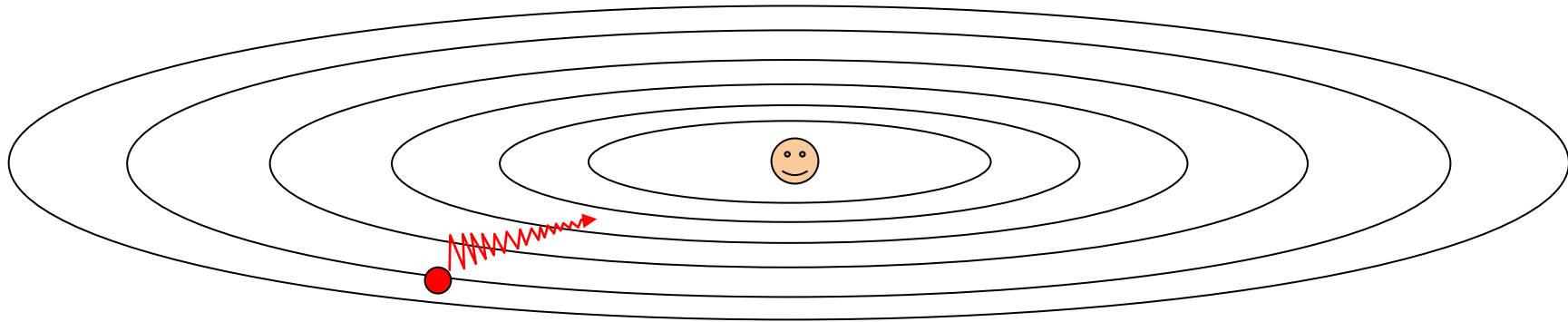
```
grad_squared = 0
while True:
    dx = compute_gradient(x)
    grad_squared += dx * dx
    x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```



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

AdaGrad

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



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

Reduces to zero 😞

RMSProp: “Leaky AdaGrad”

AdaGrad

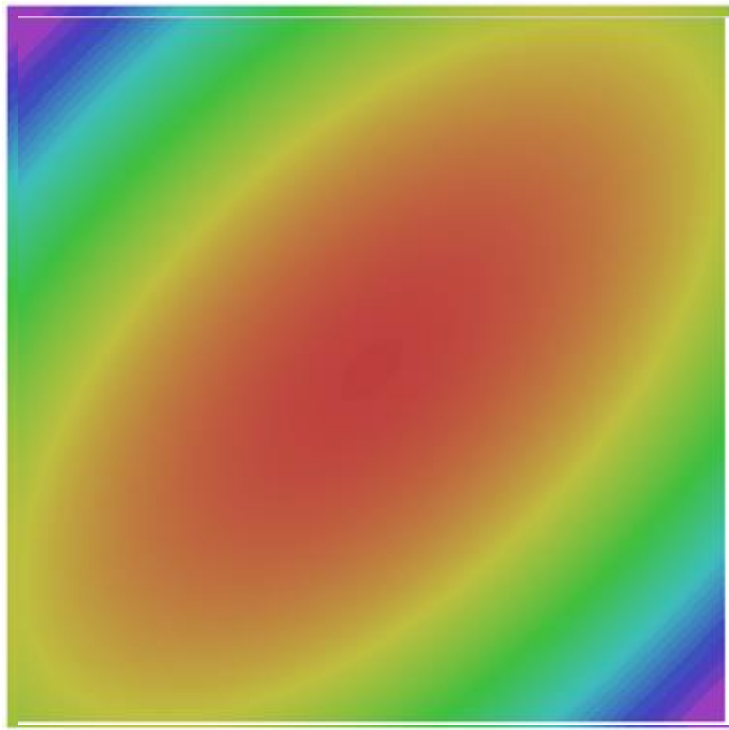
```
grad_squared = 0
while True:
    dx = compute_gradient(x)
    grad_squared += dx * dx
    x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```



RMSProp

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

RMSProp



- SGD
- SGD+Momentum
- RMSProp
- AdaGrad
(stuck due to decaying lr)

Adam (almost)

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

Typical hyperparams: beta1=0.9, beta2=0.999

Adam (almost)

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

Momentum

AdaGrad / RMSProp

Typical hyperparams: beta1=0.9, beta2=0.999

Sort of like RMSProp with momentum

Q: What happens at first timestep?

Adam (almost)

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

Momentum

AdaGrad / RMSProp

Typical hyperparams: beta1=0.9, beta2=0.999

Small -> divide by small number -> bad initial step

Q: What happens at first timestep?

Adam (full form)

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

Momentum

Bias correction

AdaGrad / RMSProp

Typical hyperparams: beta1=0.9, beta2=0.999

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

Adam (full form)

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

Momentum

Bias correction

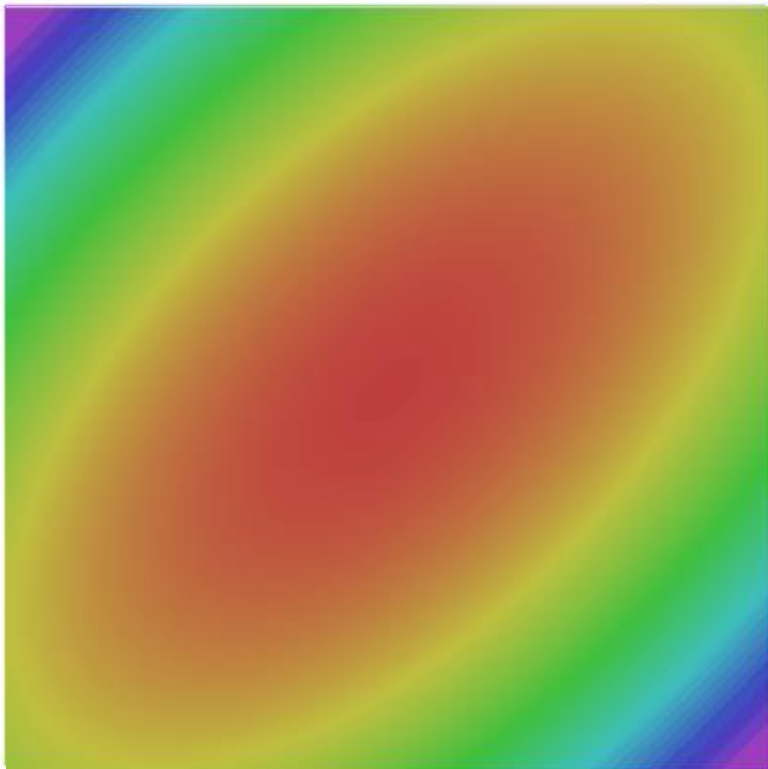
AdaGrad / RMSProp

Typical hyperparams: $\beta_1=0.9$, $\beta_2=0.999$

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

Adam with $\beta_1 = 0.9$, $\beta_2 = 0.999$, and $\text{learning_rate} = 1e-3$ or $5e-4$ is a great starting point for many models!

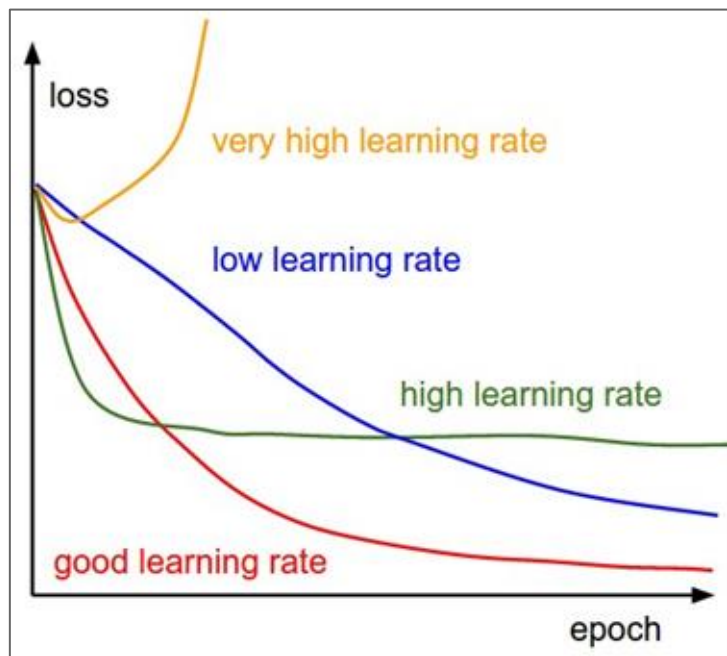
Adam



- SGD
- SGD+Momentum
- RMSProp
- Adam

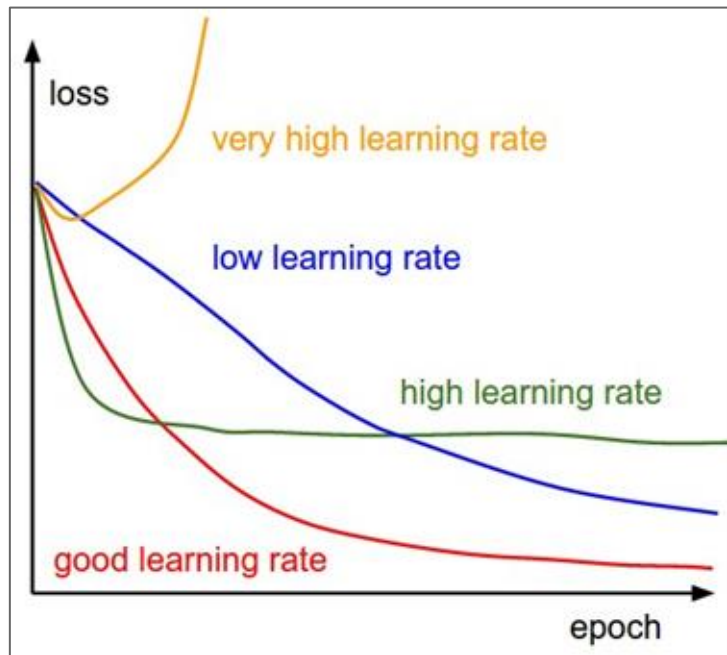
Learning rate schedules

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



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

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

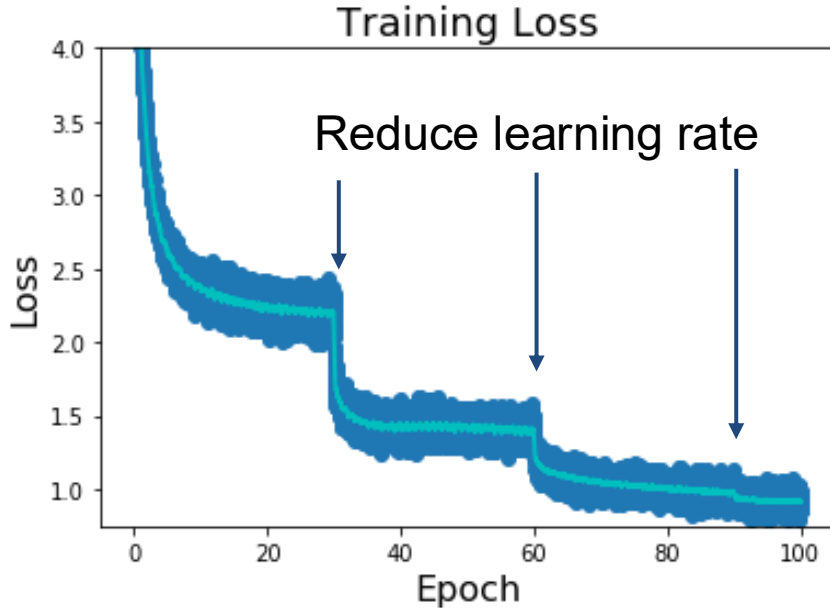


Q: Which one of these learning rates is best to use?

A: In reality, all of these are good learning rates.

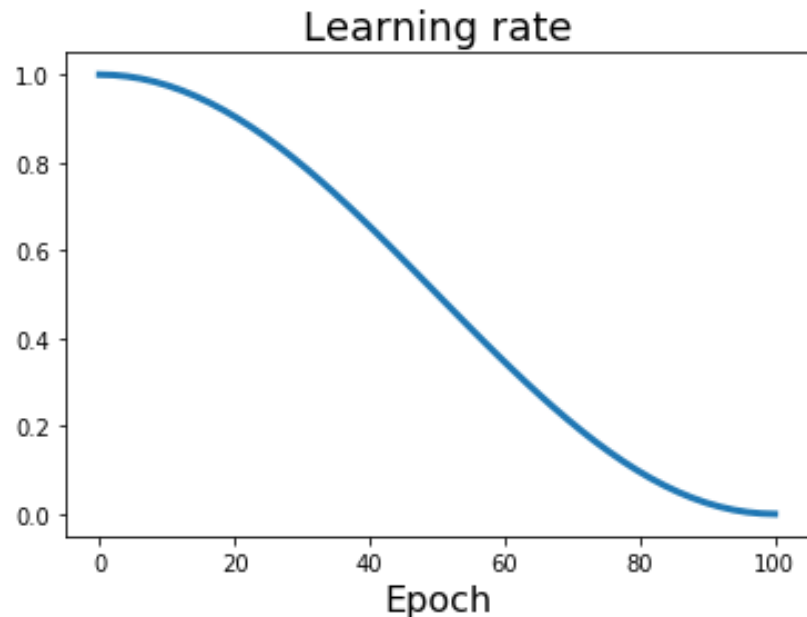
Need finer adjustment closer to convergence, so we want to reduce learning rate over time to keep making progress.

Learning rate decays over time



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

Learning Rate Decay



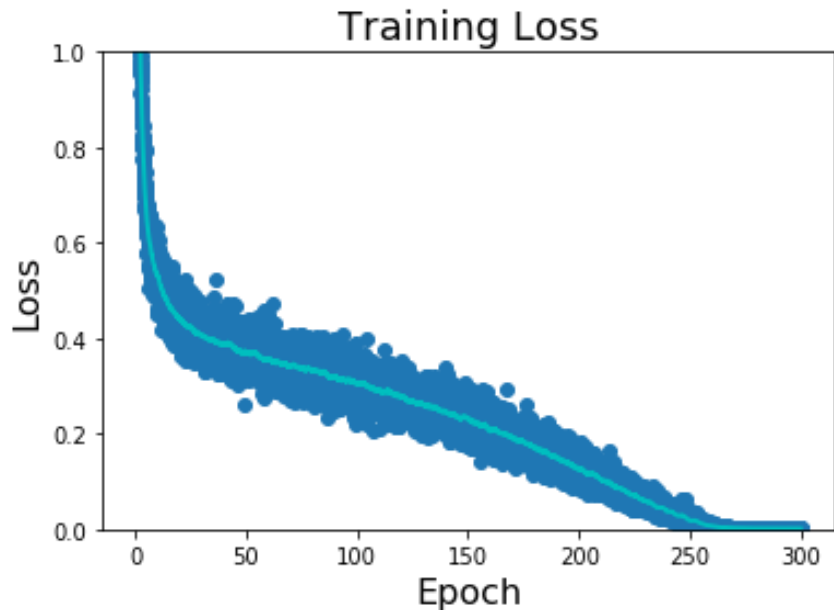
Step: Reduce learning rate at a few fixed points. E.g. for ResNets, multiply LR by 0.1 after epochs 30, 60, and 90.

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

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

Loshchilov and Hutter, “SGDR: Stochastic Gradient Descent with Warm Restarts”, ICLR 2017
Radford et al, “Improving Language Understanding by Generative Pre-Training”, 2018
Feichtenhofer et al, “SlowFast Networks for Video Recognition”, arXiv 2018
Child et al, “Generating Long Sequences with Sparse Transformers”, arXiv 2019

Learning Rate Decay



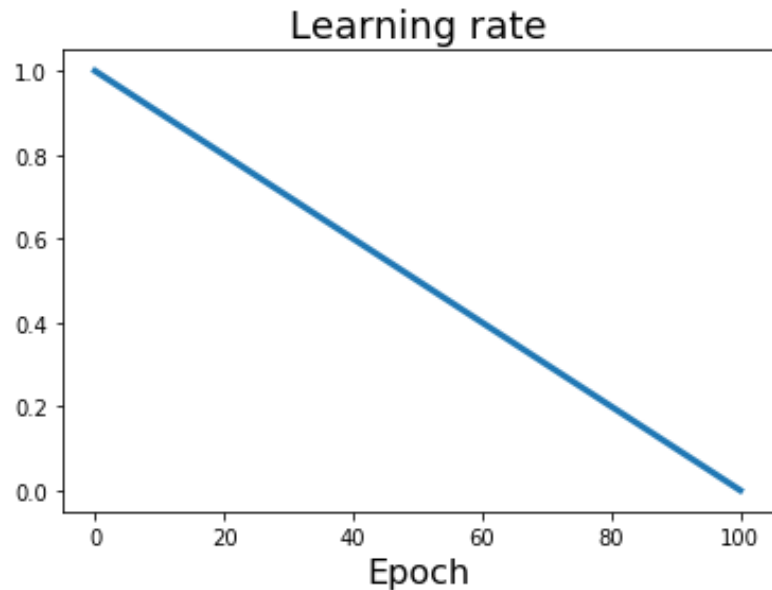
Step: Reduce learning rate at a few fixed points. E.g. for ResNets, multiply LR by 0.1 after epochs 30, 60, and 90.

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

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

Loshchilov and Hutter, “SGDR: Stochastic Gradient Descent with Warm Restarts”, ICLR 2017
Radford et al, “Improving Language Understanding by Generative Pre-Training”, 2018
Feichtenhofer et al, “SlowFast Networks for Video Recognition”, arXiv 2018
Child et al, “Generating Long Sequences with Sparse Transformers”, arXiv 2019

Learning Rate Decay



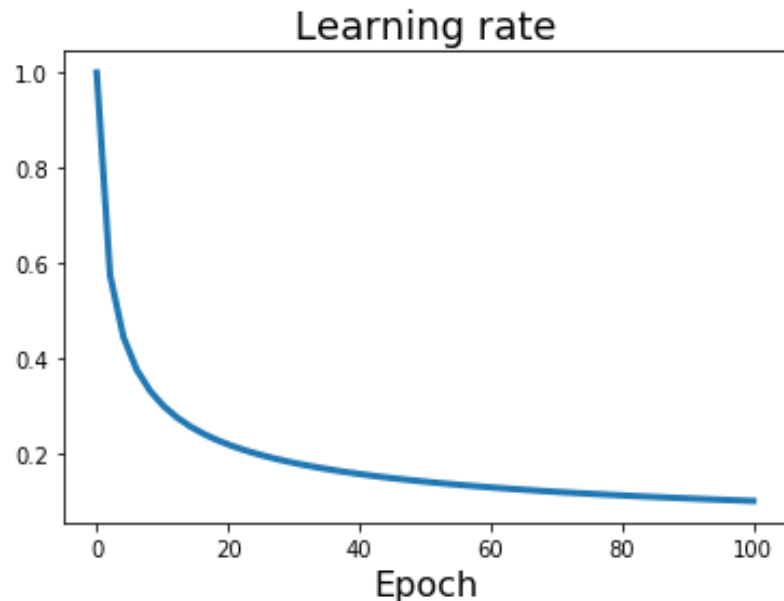
Step: Reduce learning rate at a few fixed points. E.g. for ResNets, multiply LR by 0.1 after epochs 30, 60, and 90.

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

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

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

Learning Rate Decay



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

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

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

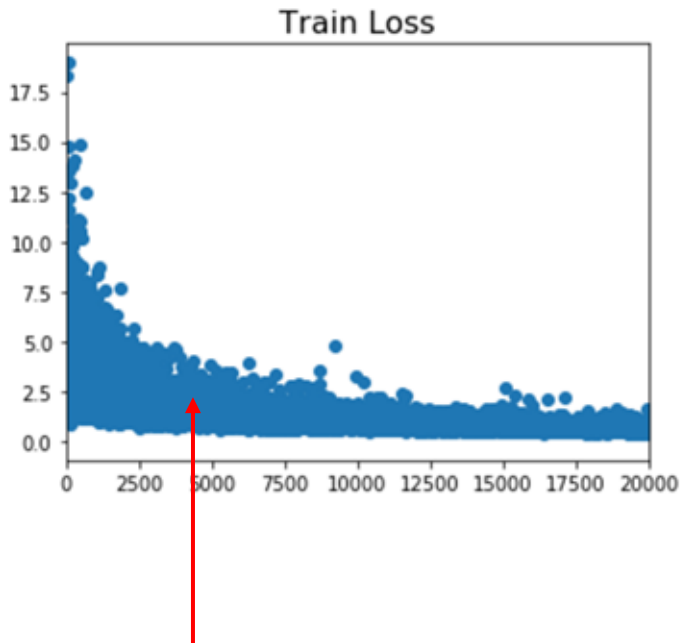
α_0 : Initial learning rate

α_t : Learning rate at epoch t

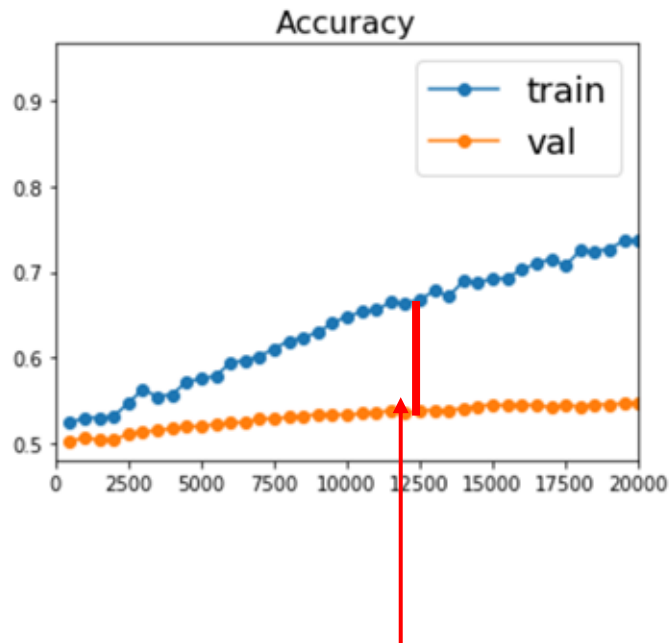
T : Total number of epochs

Regularization

Beyond Training Error

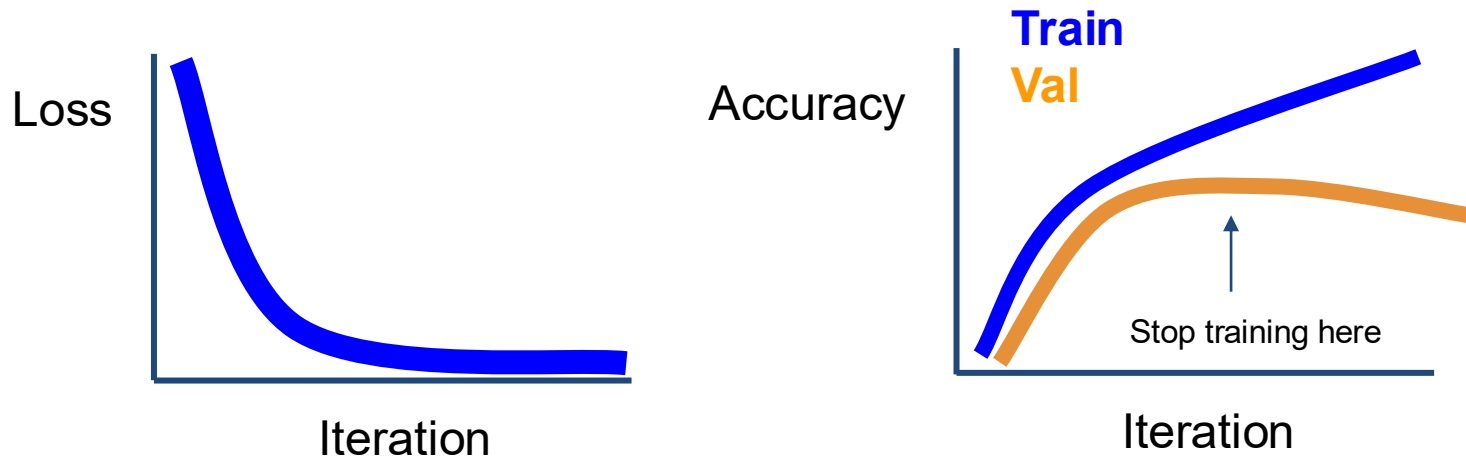


Better optimization algorithms
help reduce training loss



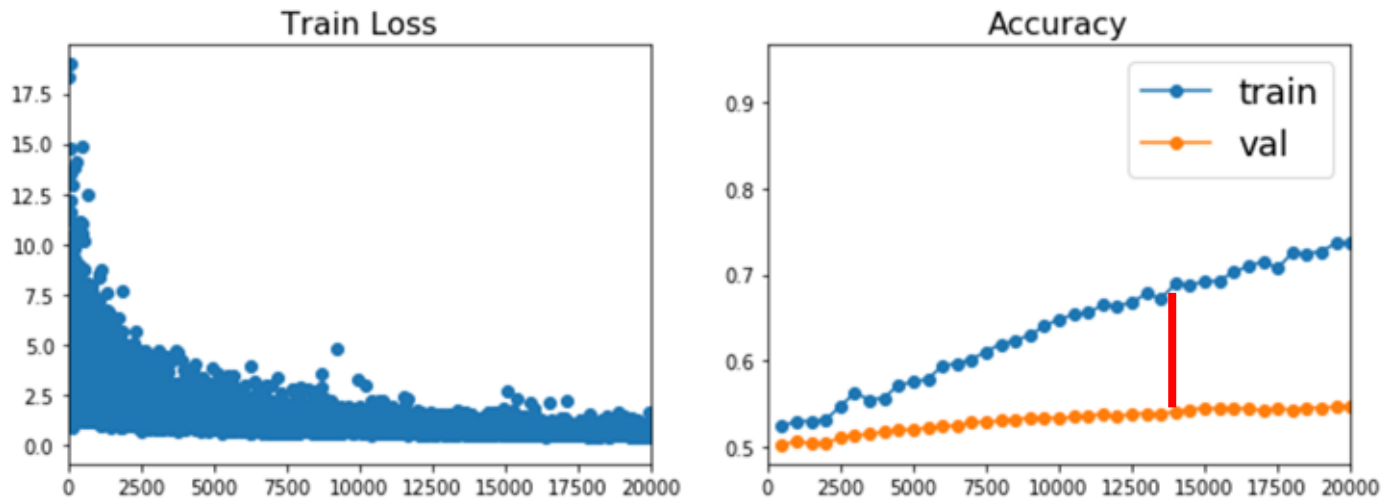
But we care about error on new data
- how to reduce the gap?

Early Stopping: Always do this



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

How to improve generalization?



Regularization

Regularization: Add term to loss

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

In common use:

L2 regularization

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

L1 regularization

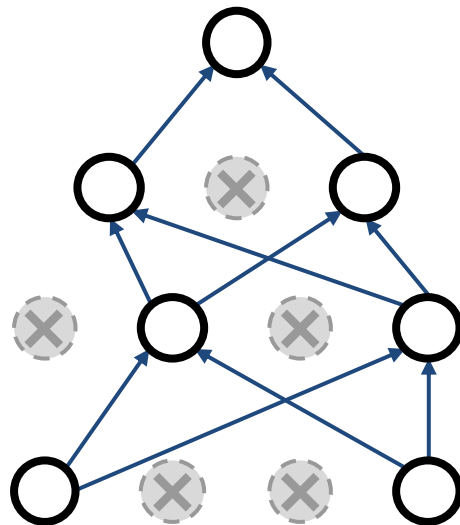
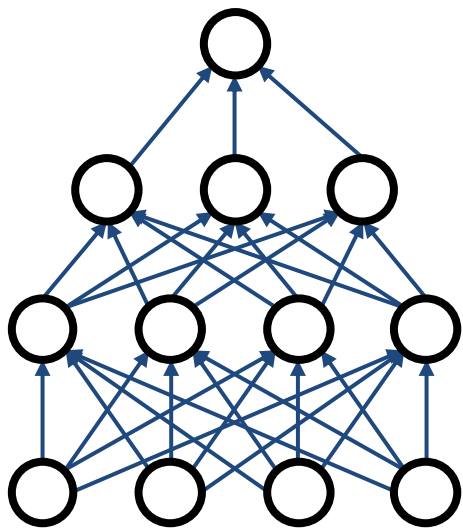
$$R(W) = \sum_k \sum_l |W_{k,l}|$$

Elastic net (L1 + L2)

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

Regularization: Dropout

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



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

Regularization: Dropout

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

```
def train_step(X):
```

```
    """ X contains the data """
```

```
    # forward pass for example 3-layer neural network
```

```
    H1 = np.maximum(0, np.dot(W1, X) + b1)
```

```
    U1 = np.random.rand(*H1.shape) < p # first dropout mask
```

```
    H1 *= U1 # drop!
```

```
    H2 = np.maximum(0, np.dot(W2, H1) + b2)
```

```
    U2 = np.random.rand(*H2.shape) < p # second dropout mask
```

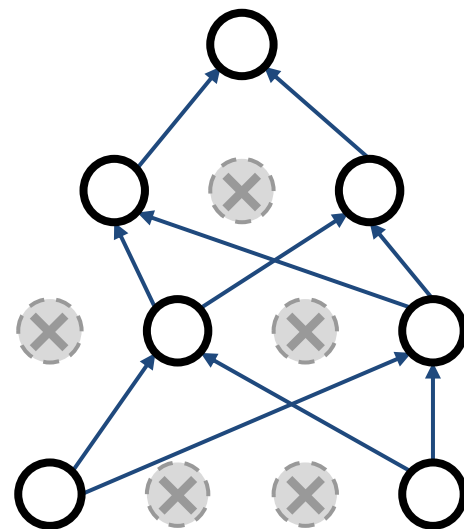
```
    H2 *= U2 # drop!
```

```
    out = np.dot(W3, H2) + b3
```

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

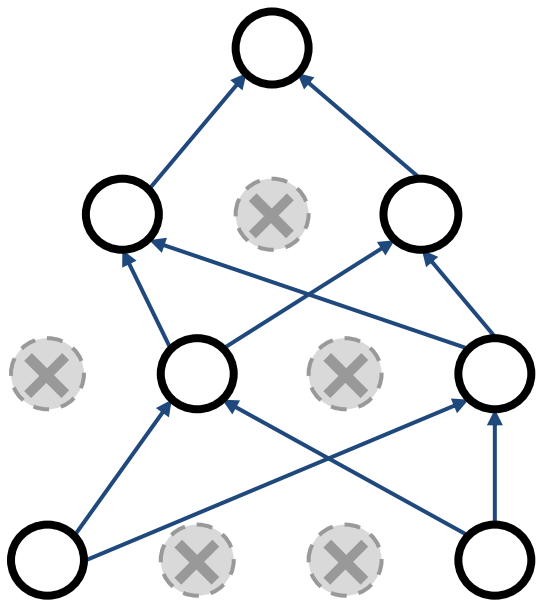
```
    # perform parameter update... (not shown)
```

Example forward pass with a 3-layer network using dropout



Regularization: Dropout

How can this possibly be a good idea?

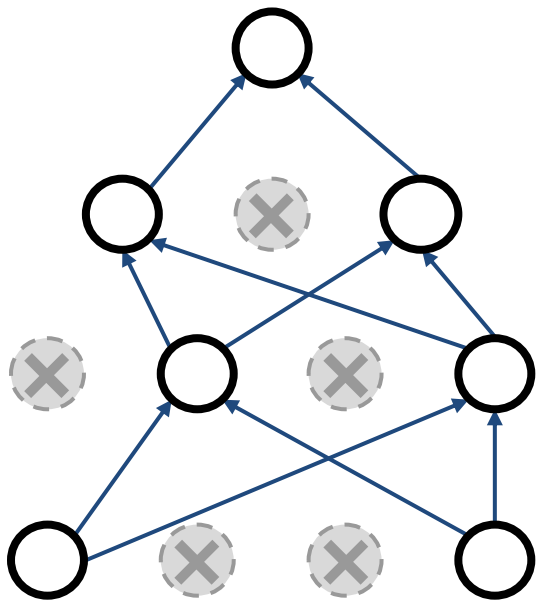


Forces the network to have a redundant representation;
Prevents co-adaptation of features



Regularization: Dropout

How can this possibly be a good idea?



Another interpretation:

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

Each binary mask is one model

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

Dropout: Test time

Dropout makes our output random!

Output (label) y = f_W (Input (image) x , Random mask z)

Test-time behavior should be deterministic

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

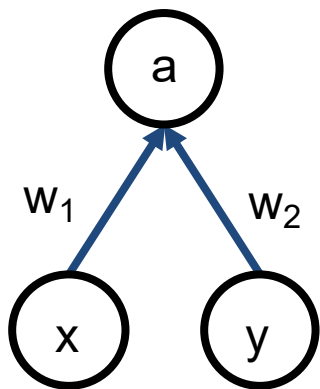
$$y = f(x) = E_z[f(x, z)] = \int p(z)f(x, z)dz$$

Dropout: Test time

Compute the expectation

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

Consider a single neuron.



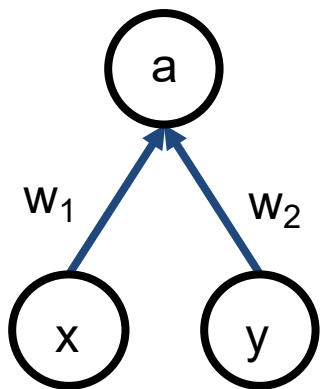
Dropout: Test time

Compute the expectation

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

Consider a single neuron.

Without dropout: $E[a] = w_1x + w_2y$

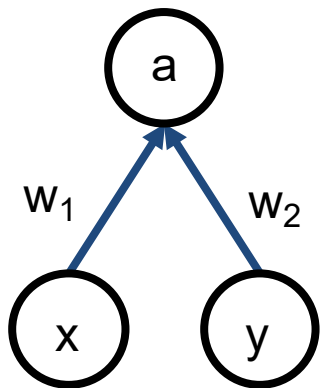


Dropout: Test time

Compute the expectation

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

Consider a single neuron.



Without dropout:

$$E[a] = w_1x + w_2y$$

With dropout we have:

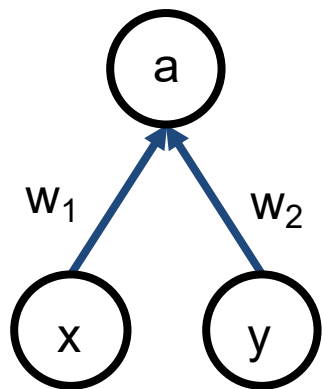
$$\begin{aligned} E[a] &= \frac{1}{4}(w_1x + w_2y) + \frac{1}{4}(w_1x + 0y) \\ &\quad + \frac{1}{4}(0x + 0y) + \frac{1}{4}(0x + w_2y) \\ &= \frac{1}{2}(w_1x + w_2y) \end{aligned}$$

Dropout: Test time

Compute the expectation

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

Consider a single neuron.



Without dropout:

$$E[a] = w_1x + w_2y$$

With dropout we have:

$$\begin{aligned} E[a] &= \frac{1}{4}(w_1x + w_2y) + \frac{1}{4}(w_1x + 0y) \\ &\quad + \frac{1}{4}(0x + 0y) + \frac{1}{4}(0x + w_2y) \\ &= \frac{1}{2}(w_1x + w_2y) \end{aligned}$$

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

Dropout: Test time

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

At test time all neurons are active always

=> We must scale the activations so that for each neuron:

output at test time = expected output at training time

Dropout Summary

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

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

```
def train_step(X):
```

```
    """ X contains the data """
```

```
    # forward pass for example 3-layer neural network
```

```
    H1 = np.maximum(0, np.dot(W1, X) + b1)
```

```
    U1 = np.random.rand(*H1.shape) < p # first dropout mask
```

```
    H1 *= U1 # drop!
```

```
    H2 = np.maximum(0, np.dot(W2, H1) + b2)
```

```
    U2 = np.random.rand(*H2.shape) < p # second dropout mask
```

```
    H2 *= U2 # drop!
```

```
    out = np.dot(W3, H2) + b3
```

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

```
    # perform parameter update... (not shown)
```

```
def predict(X):
```

```
    # ensembled forward pass
```

```
    H1 = np.maximum(0, np.dot(W1, X) + b1) * p # NOTE: scale the activations
```

```
    H2 = np.maximum(0, np.dot(W2, H1) + b2) * p # NOTE: scale the activations
```

```
    out = np.dot(W3, H2) + b3
```

drop in train time

scale at test time

More common: “Inverted dropout”

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

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

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

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

test time is unchanged!



Similar to BatchNorm, different behavior train vs test!

Regularization: A common strategy

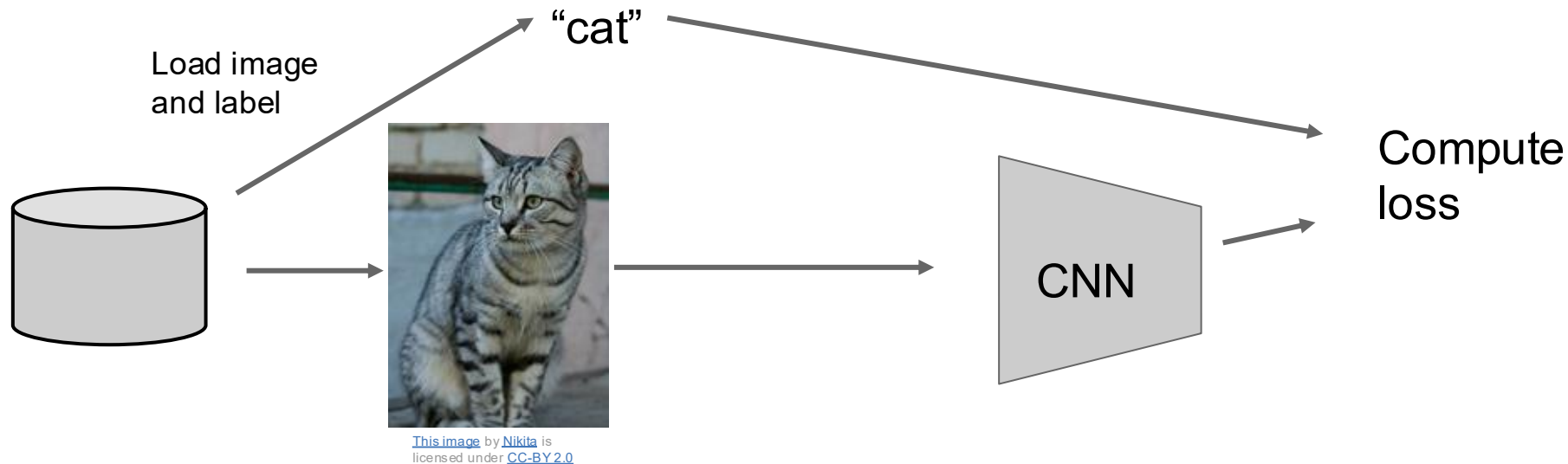
Training: Add some kind of randomness

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

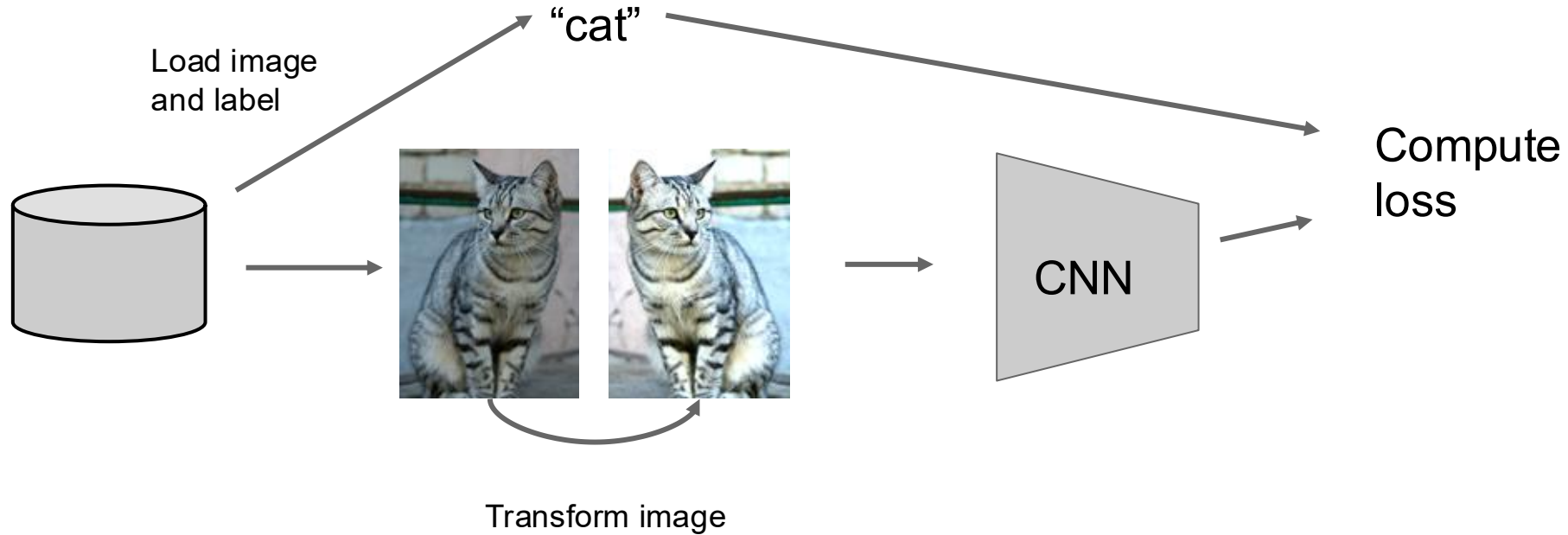
Testing: Average out randomness (sometimes approximate)

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

Regularization: Data Augmentation



Regularization: Data Augmentation



Data Augmentation

Horizontal Flips



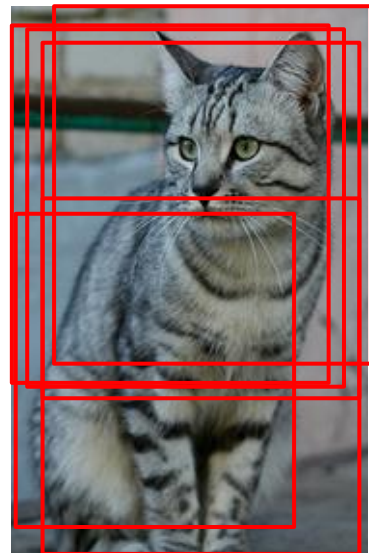
Data Augmentation

Random crops and scales

Training: sample random crops / scales

ResNet:

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



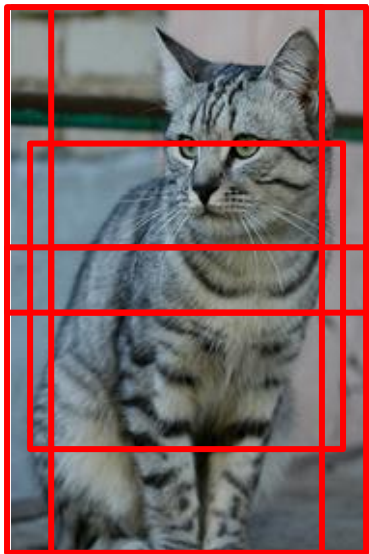
Data Augmentation

Random crops and scales

Training: sample random crops / scales

ResNet:

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



Testing (test-time augmentation):

take votes / average from a fixed set of crops

1. Resize image at 5 scales: $\{224, 256, 384, 480, 640\}$
2. For each size, use 10 224×224 crops: 4 corners + center, + flips
3. Make prediction on all crops, use the majority vote as the final output.

Data Augmentation

Random crops and scales

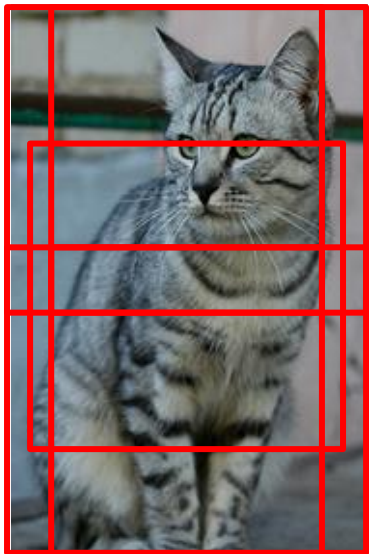
Training: sample random crops / scales

ResNet:

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

Testing (deterministic):

- Take a center crop of 224 by 224 .
- Or crop by longer dimension and resize.



Data Augmentation

Color Jitter

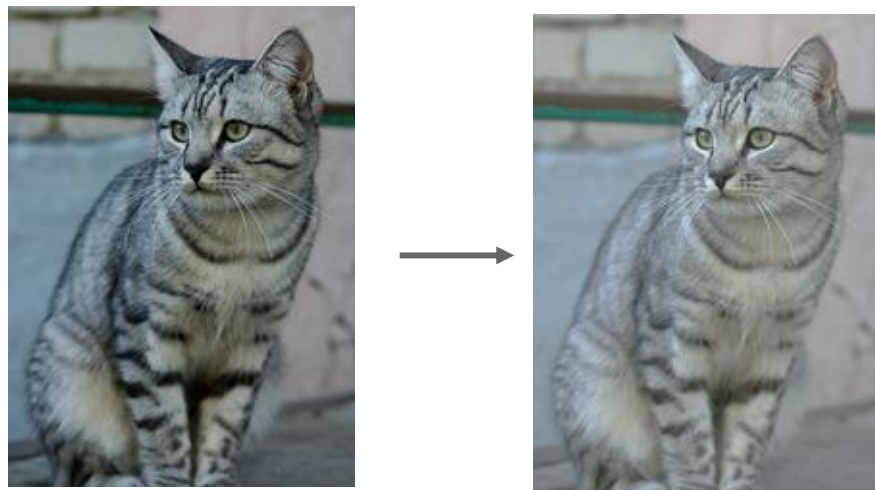
Simple: Randomize
contrast and brightness



Data Augmentation

Color Jitter

Simple: Randomize contrast and brightness



More Complex:

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

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

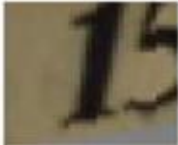
Data Augmentation

Get creative for your problem!

Examples of data augmentations:

- translation
- rotation
- stretching
- shearing,
- chromatic aberration
- lens distortions, ... (go crazy)

Automatic Data Augmentation

	Original	Sub-policy 1	Sub-policy 2	Sub-policy 3	Sub-policy 4	Sub-policy 5
Batch 1						
Batch 2						
Batch 3						
		ShearX, 0.9, 7 Invert, 0.2, 3	ShearY, 0.7, 6 Solarize, 0.4, 8	ShearX, 0.9, 4 AutoContrast, 0.8, 3	Invert, 0.9, 3 Equalize, 0.6, 3	ShearY, 0.8, 5 AutoContrast, 0.7, 3

Gradient clipping: prevent large gradient step

Large gradient step will likely destabilize training (gradients are noisy!)

Large gradient update can be caused by many issues, e.g., large weights, large input, bad loss function / activation function, ...

Should always first try to fix the root cause (normalization, better loss / activation function, etc.)

But if all things fail ... just clip the gradient

$$g_{new} = \min\left(1, \frac{\lambda}{||g||}\right) \times g$$

g : original gradient

λ : clipping threshold

If $||g|| \leq \lambda$, no effect

```
# Zero the gradients.
optimizer.zero_grad()

# Perform forward pass.
outputs = model(inputs)

# Compute the loss.
loss = loss_function(outputs, targets)

# Perform backward pass (compute gradients).
loss.backward()

# Clip the gradients.
torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0)

# Update the model parameters.
optimizer.step()
```

Next Time:

Training Deep Neural Networks

- Details of the non-linear activation functions
- Data normalization
- Weight Initialization
- Batch Normalization
- Advanced Optimization
- Regularization
- Data Augmentation
- **Transfer learning**
- **Hyperparameter Tuning**
- **Model Ensemble**