# Lecture 7:
# Training Neural Networks, Part II

# Administrative: Assignment 2

Has been released

Due 10/22 11:59pm

- Multi-layer Neural Networks,

- Image Features,

- Optimizers
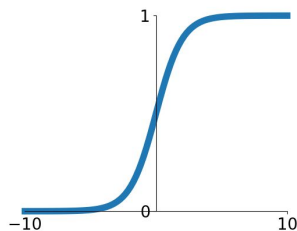
# Administrative: Course Project

Groups assigned

Proposal due 2/29

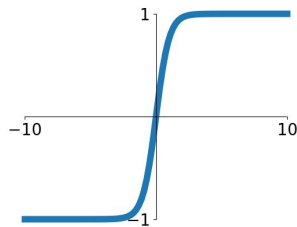Come to OH to talk about projects

# Last time: Activation Functions

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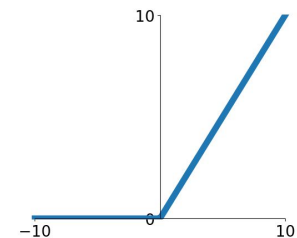
$\sigma(x) = \frac{1}{1+e^{-x}}$
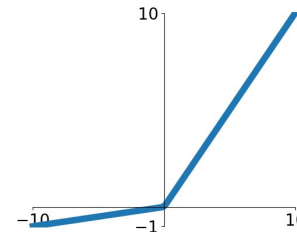
**tanh**

$\tanh(x)$

**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**

$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$
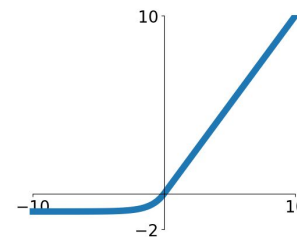
# Last time: Activation Functions

**Sigmoid**

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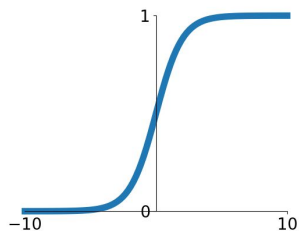
**tanh**

$\tanh(x)$

**ReLU**

$\max(0, x)$

**Good default choice**

**Leaky ReLU**

$\max(0.1x, x)$

**Maxout**

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

**ELU**
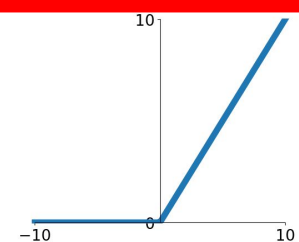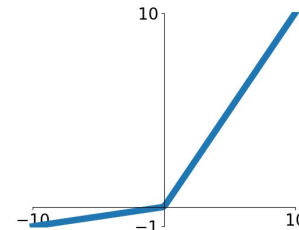
$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$
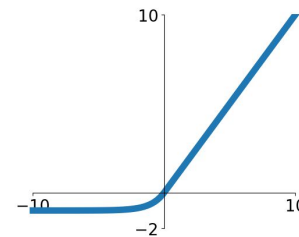
# Last time: Weight Initialization
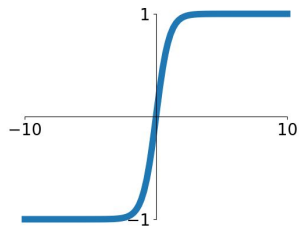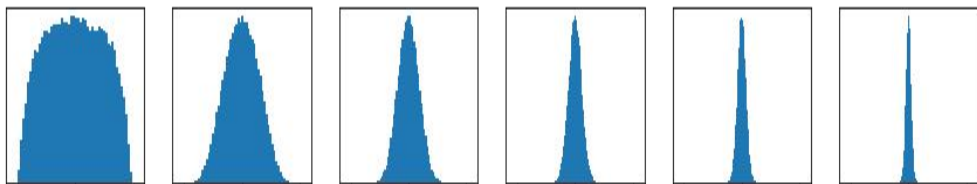


**Initialization too small**:
Activations go to zero, gradients also zero,
No learning =(
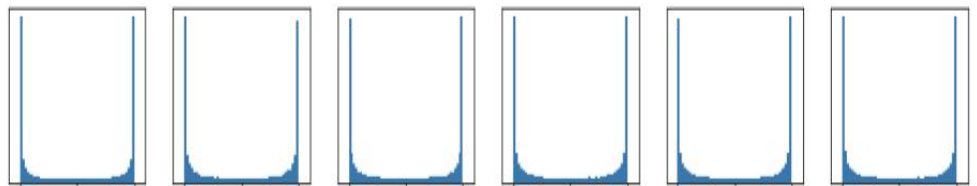
**Initialization too big**:
Activations saturate (for tanh),
Gradients zero, no learning =(
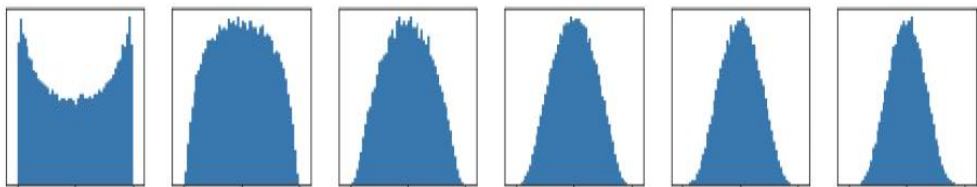
**Initialization just right**:
Nice distribution of activations at all layers,
Learning proceeds nicely =)

# Last time: Data Preprocessing



original data          zero-centered data          normalized data
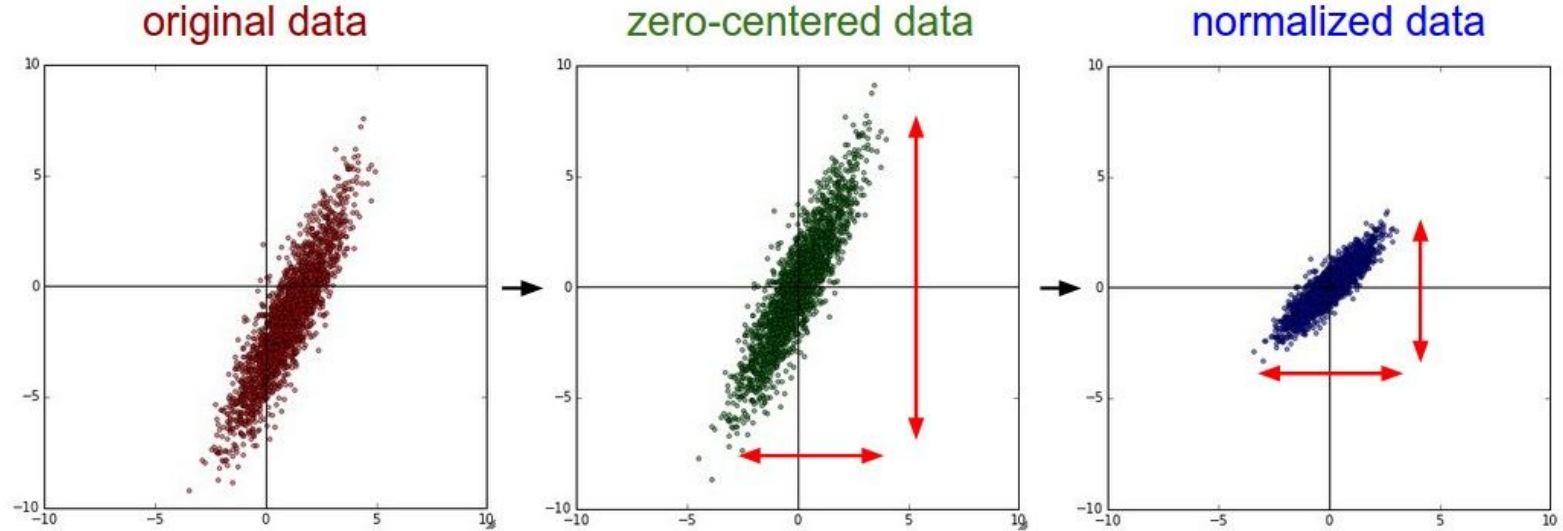
# Last Time: Batch Normalization

**Input**: $x : N \times D$

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

Per-channel mean, shape is D

**Learnable scale and shift parameters:**

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

Per-channel var, shape is D

$$\gamma, \beta : D$$

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

Normalized x, Shape is N x D

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

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

Output, Shape is N x D

# Batch Normalization: Test-Time

**Input**: $x : N \times D$

$\mu_j =$ <span style="color:red">(Running) average of values seen during training</span>    Per-channel mean, shape is D

**Learnable scale and shift parameters:**

$\sigma_j^2 =$ <span style="color:red">(Running) average of values seen during training</span>    Per-channel var, shape is D

$\gamma, \beta : D$

$\hat{x}_{i,j} = \dfrac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \varepsilon}}$    Normalized x, Shape is N x D

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

$y_{i,j} = \gamma_j \hat{x}_{i,j} + \beta_j$    Output, Shape is N x D

# Batch Normalization

FC

BN

tanh

FC

BN

tanh

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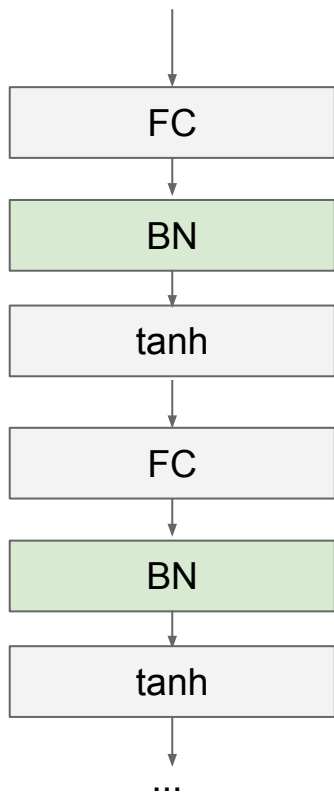
Usually inserted after Fully Connected or Convolutional layers, and before nonlinearity.

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

# Batch Normalization

FC

BN

tanh

FC

BN

tanh

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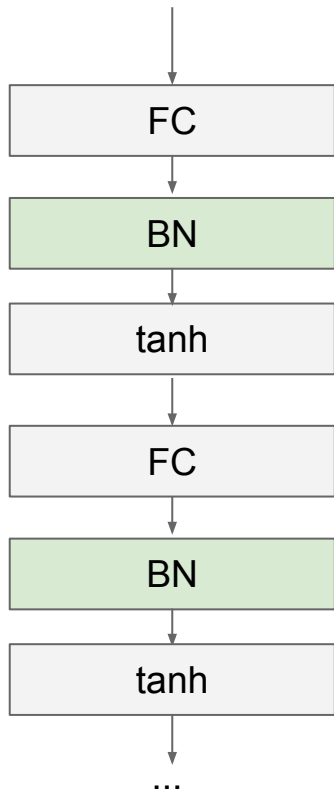
- Makes deep networks **much** easier to train!
- Improves gradient flow
- Allows higher learning rates, faster convergence
- Networks become more robust to initialization
- Acts as regularization during training
- Zero overhead at test-time: can be fused with conv!
- Behaves differently during training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

# Batch Normalization for ConvNets

Batch Normalization for
**fully-connected** networks

Batch Normalization for
**convolutional** networks
(Spatial Batchnorm, BatchNorm2D)

$$\mathbf{x}:\ N\ \times\ D$$

Normalize

$$\boldsymbol{\mu},\boldsymbol{\sigma}:\ 1\ \times\ D$$
$$\gamma,\beta:\ 1\ \times\ D$$
$$y\ =\ \gamma(\mathbf{x}-\boldsymbol{\mu})/\sigma+\beta$$

$$\mathbf{x}:\ N{\times}C{\times}H{\times}W$$

Normalize

$$\boldsymbol{\mu},\boldsymbol{\sigma}:\ 1{\times}C{\times}1{\times}1$$
$$\gamma,\beta:\ 1{\times}C{\times}1{\times}1$$
$$y\ =\ \gamma(\mathbf{x}-\boldsymbol{\mu})/\sigma+\beta$$

# Layer Normalization

**Batch Normalization** for
fully-connected networks

**Layer Normalization** for
fully-connected networks
Same behavior at train and test!
Can be used in recurrent networks

$$\texttt{x: N × D}$$

Normalize

$$\boldsymbol{\mu},\boldsymbol{\sigma}: \texttt{ 1 × D}$$

$$\texttt{γ,β: 1 × D}$$

$$\texttt{y = γ(x−}\boldsymbol{\mu}\texttt{)/}\boldsymbol{\sigma}\texttt{+β}$$

$$\texttt{x: N × D}$$

Normalize

$$\boldsymbol{\mu},\boldsymbol{\sigma}: \texttt{ N × 1}$$

$$\texttt{γ,β: 1 × D}$$

$$\texttt{y = γ(x−}\boldsymbol{\mu}\texttt{)/}\boldsymbol{\sigma}\texttt{+β}$$

Ba, Kiros, and Hinton, "Layer Normalization", arXiv 2016

# Instance Normalization

**Batch Normalization** for convolutional networks

$$x: \texttt{N×C×H×W}$$

Normalize

$$\boldsymbol{\mu},\boldsymbol{\sigma}: \texttt{1×C×1×1}$$

$$\gamma,\beta: \texttt{1×C×1×1}$$

$$y = \gamma(x-\boldsymbol{\mu})/\boldsymbol{\sigma}+\beta$$

**Instance Normalization** for convolutional networks
Same behavior at train / test!

$$x: \texttt{N×C×H×W}$$

Normalize

$$\boldsymbol{\mu},\boldsymbol{\sigma}: \texttt{N×C×1×1}$$

$$\gamma,\beta: \texttt{1×C×1×1}$$

$$y = \gamma(x-\boldsymbol{\mu})/\boldsymbol{\sigma}+\beta$$

Ulyanov et al, Improved Texture Networks: Maximizing Quality and Diversity in Feed-forward Stylization and Texture Synthesis, CVPR 2017

# Comparison of Normalization Layers



Batch Norm       Layer Norm       Instance Norm
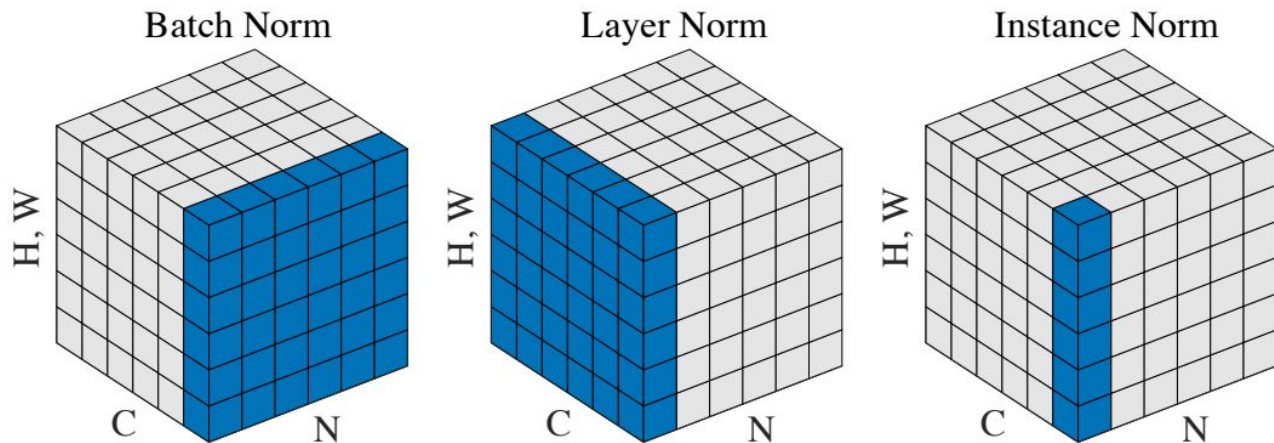
Wu and He, "Group Normalization", ECCV 2018

# Group Normalization



Batch Norm | Layer Norm | Instance Norm | **Group Norm**
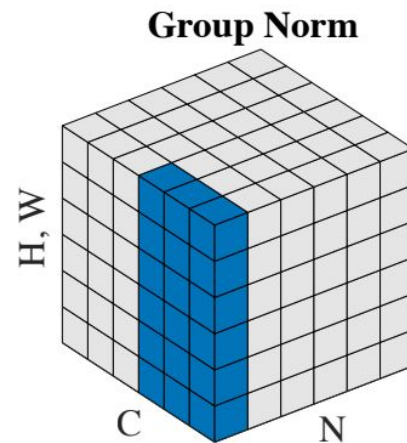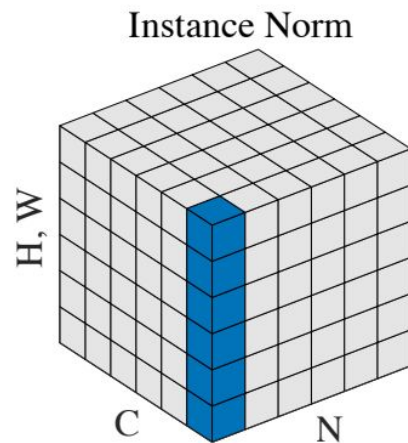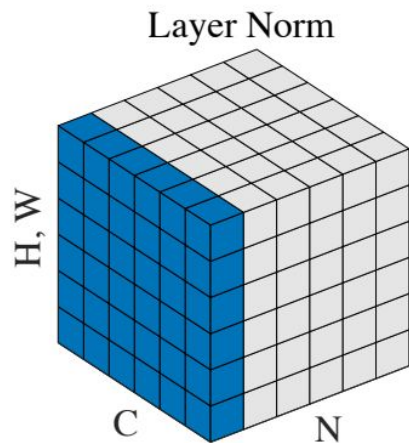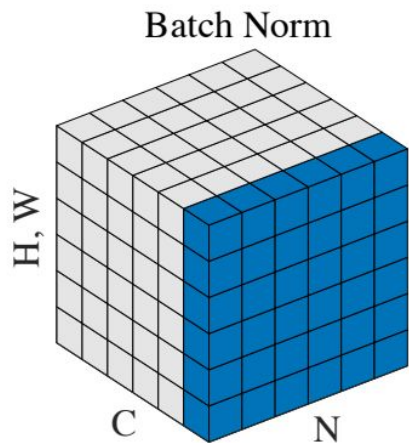
Wu and He, "Group Normalization", ECCV 2018

# Today

- Improve your training error:
    - (Fancier) Optimizers
    - Learning rate schedules
- Improve your test error:

    - Regularization
    - Choosing Hyperparameters

(Fancier) Optimizers

# Optimization



```
# Vanilla Gradient Descent

while True:
    weights_grad = evaluate_gradient(loss_fun, data, weights)
    weights += - step_size * weights_grad # perform parameter update
```
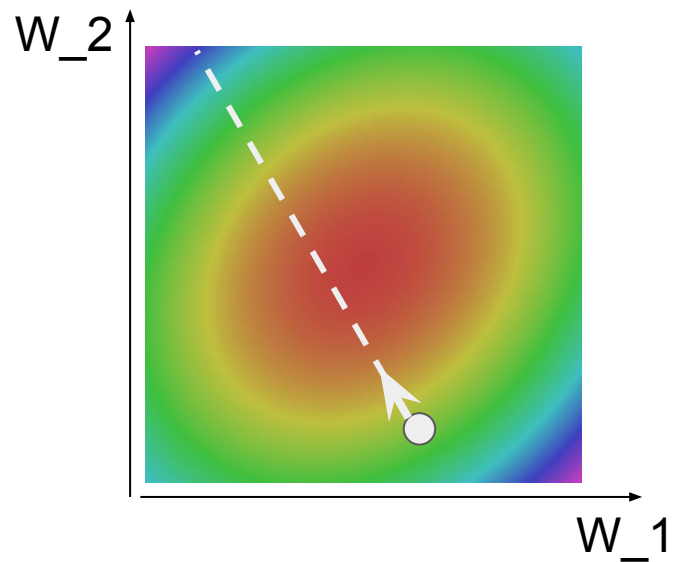
# Optimization: Problem #1 with SGD

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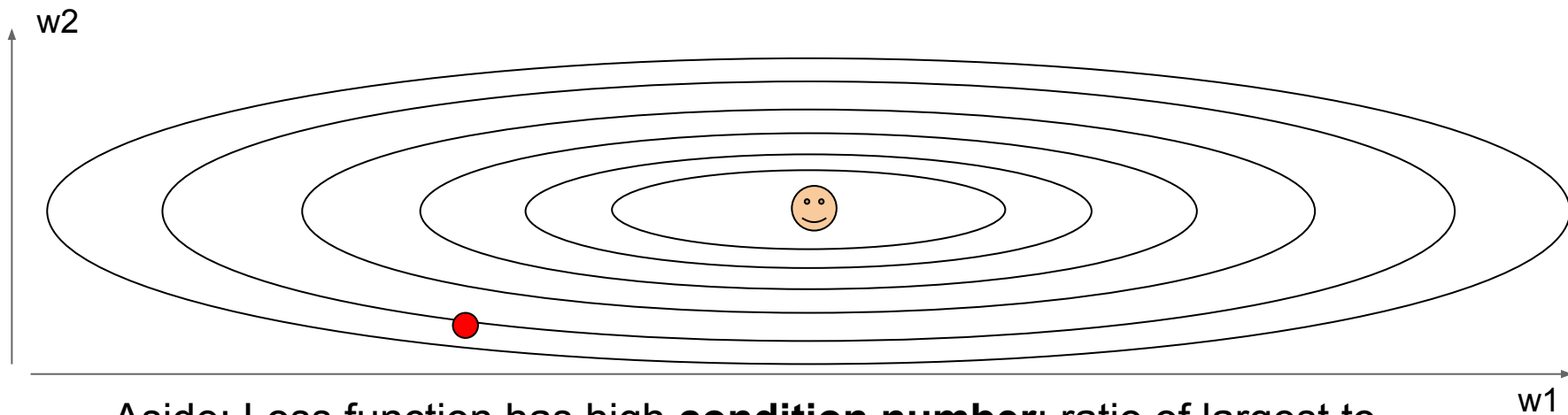


Aside: Loss function has high **condition number**: ratio of largest to smallest singular value of the Hessian matrix is large

# Optimization: Problem #1 with SGD

What if loss changes quickly in one direction and slowly in another?
What does gradient descent do?
<span style="color:red">Very slow progress along shallow dimension, jitter along steep direction</span>
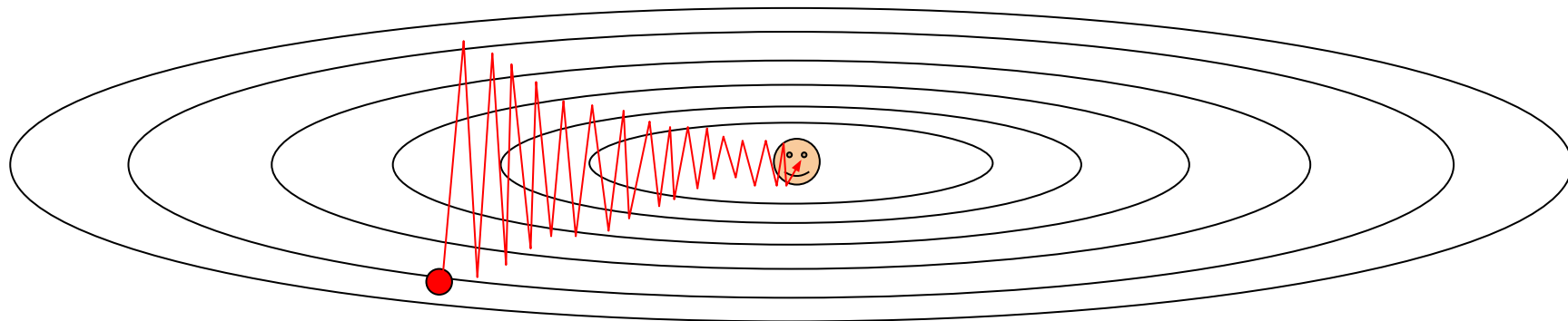


Loss function has high **condition number**: ratio of largest to smallest singular value of the Hessian matrix is large

# Optimization: Problem #2 with SGD

What if the loss function has a **local minima** or **saddle point**?
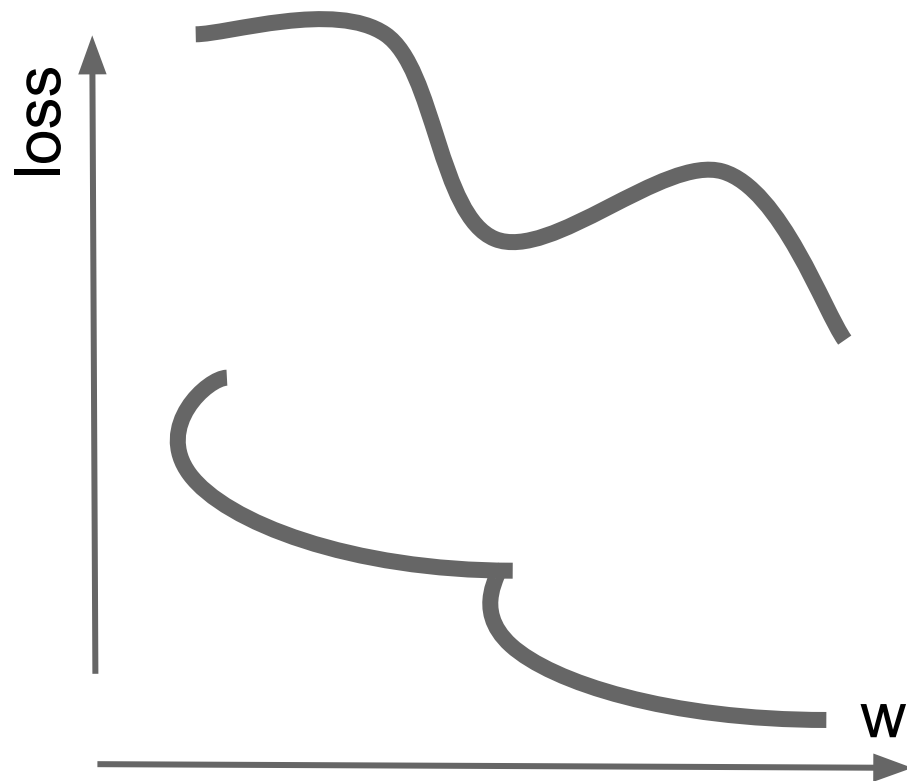
# Optimization: Problem #2 with SGD

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

Zero gradient, gradient descent gets stuck
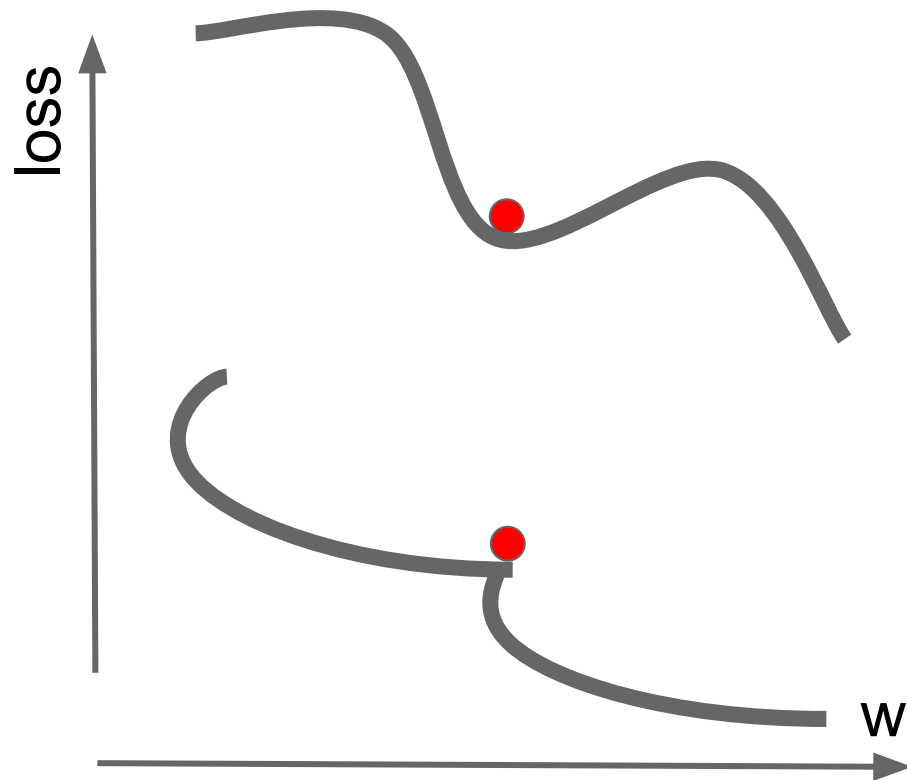
# Optimization: Problem #2 with SGD

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

Saddle points much more common in high dimension
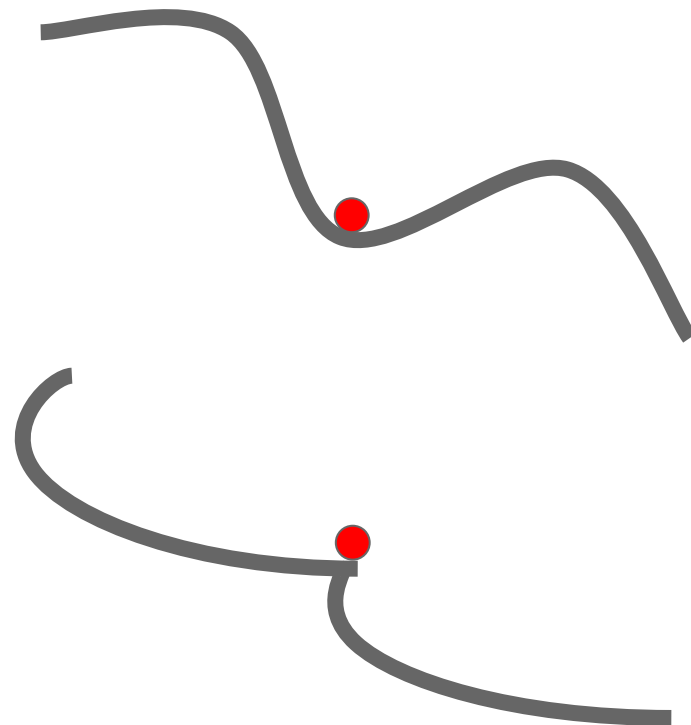
Dauphin et al, "Identifying and attacking the saddle point problem in high-dimensional non-convex optimization", NIPS 2014

# Optimization: Problem #3 with SGD

Our gradients come from minibatches so they can be noisy!

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

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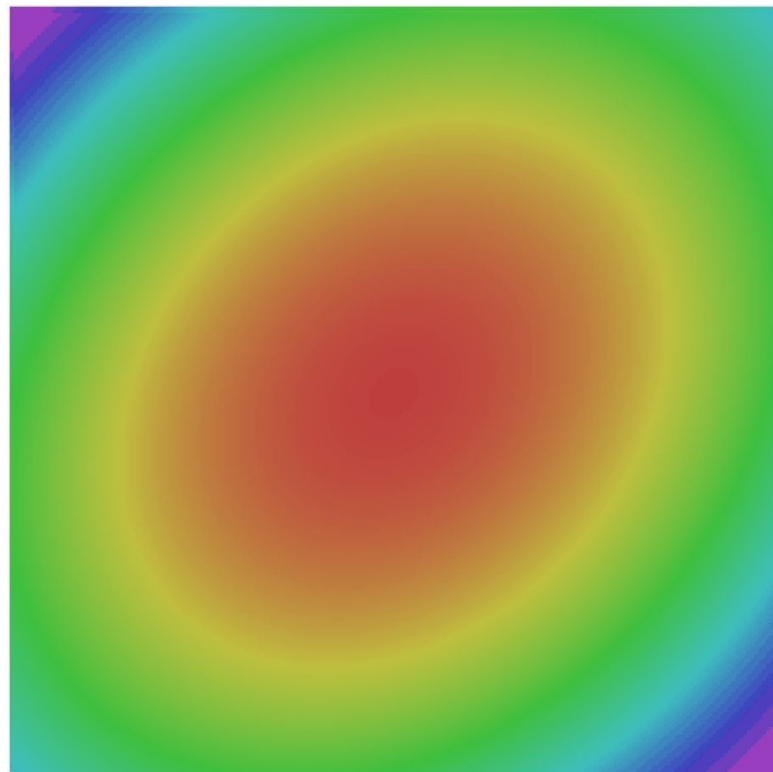
# SGD + Momentum

## Gradient Noise

### Local Minima
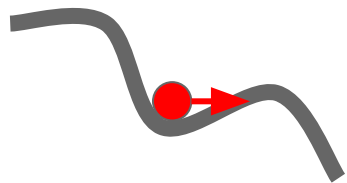
### Saddle points
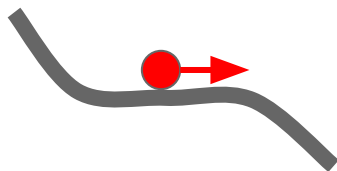
### Poor Conditioning
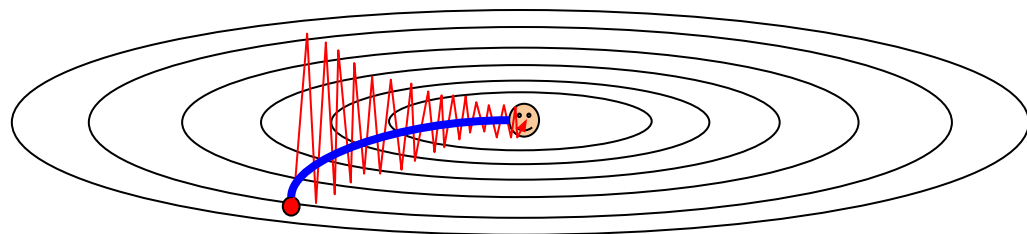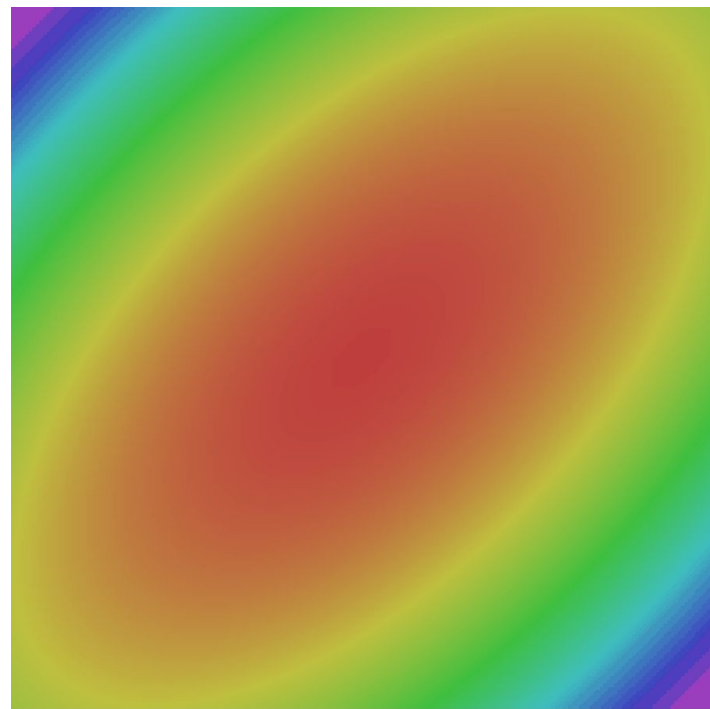
SGD          SGD+Momentum

# SGD: the simple two line update code

SGD

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

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

# SGD + Momentum:

continue moving in the general direction as the previous iterations

| SGD | SGD+Momentum |
|---|---|

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

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

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

- 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

# 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

# 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

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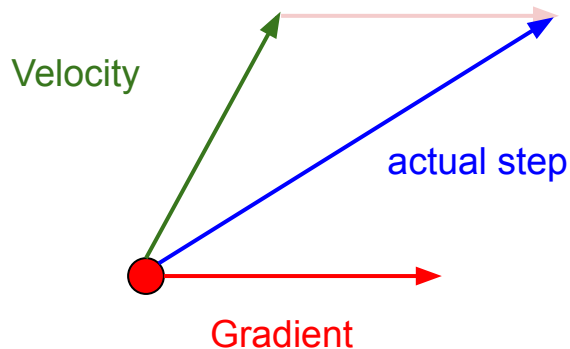
# SGD+Momentum

## Momentum update:



Velocity

actual step

Gradient

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:
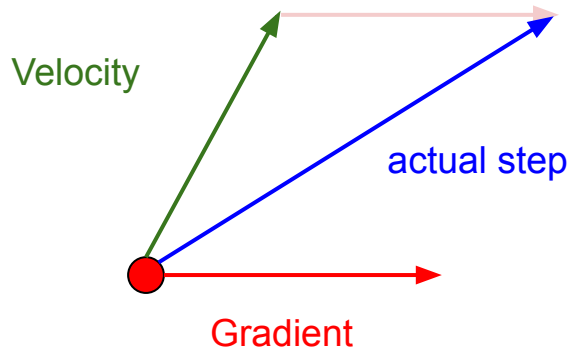


Combine gradient at current point with velocity to get step used to update weights

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

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

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

Velocity

Gradient

actual step

"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

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

Annoying, usually we want update in terms of $x_t, \nabla f(x_t)$



"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

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

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

Change of variables $\tilde{x}_t = \boxed{x_t} + \rho v_t$ and rearrange:

Annoying, usually we want update in terms of $x_t, \nabla f(x_t)$



"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

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

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

Annoying, usually we want update in terms of $x_t, \nabla f(x_t)$

Change of variables $\tilde{x}_t = x_t + \rho v_t$ and rearrange:

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

$$\tilde{x}_{t+1} = \tilde{x}_t - \rho v_t + (1 + \rho)v_{t+1}$$

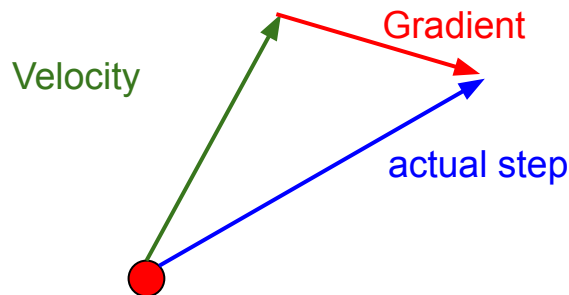$$= \tilde{x}_t + v_{t+1} + \rho(v_{t+1} - v_t)$$
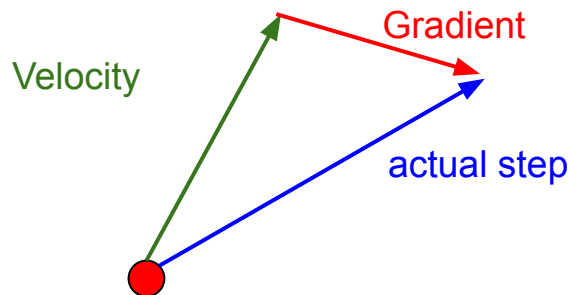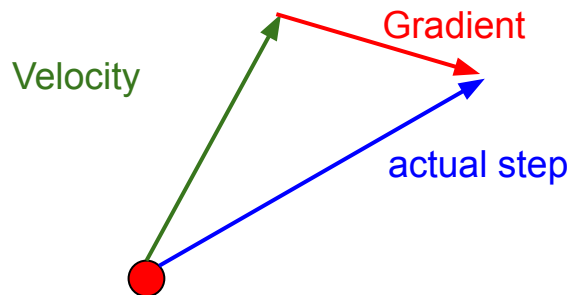
Gradient

Velocity

actual step

"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
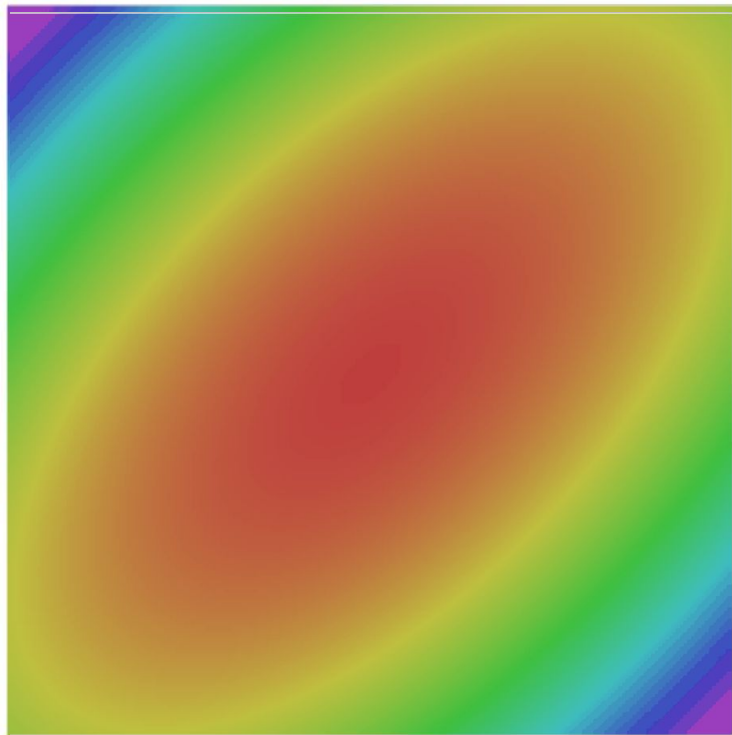
# Nesterov Momentum



━━━   SGD

━━━   SGD+Momentum

━━━   Nesterov

# 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"

Duchi et al, "Adaptive subgradient methods for online learning and stochastic optimization", JMLR 2011

# AdaGrad

```python
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

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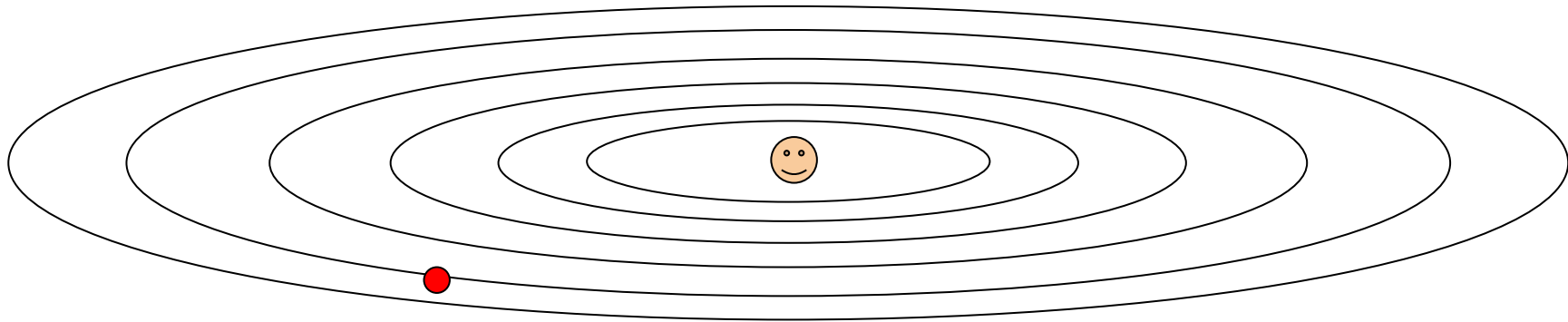


Q2: What happens to the step size over long time?

# AdaGrad

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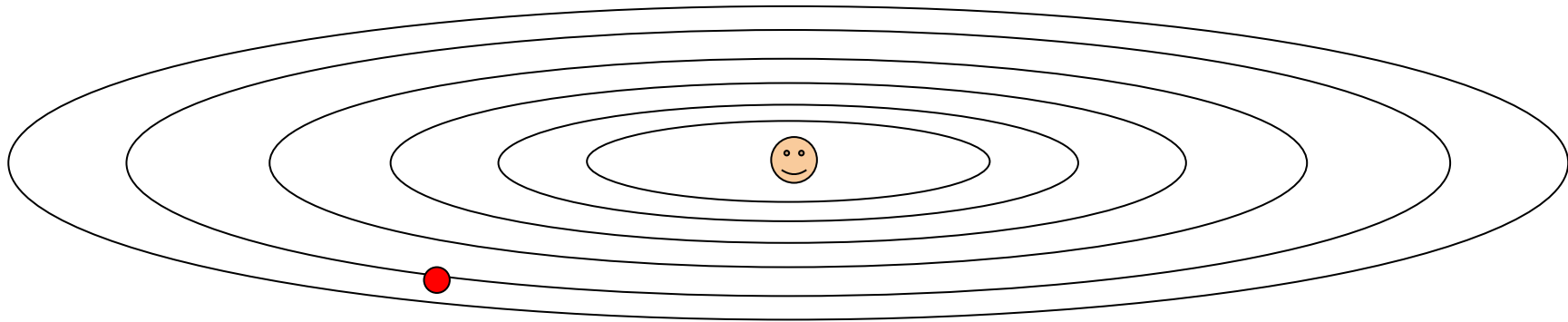


Q2: What happens to the step size over long time?  Decays to zero
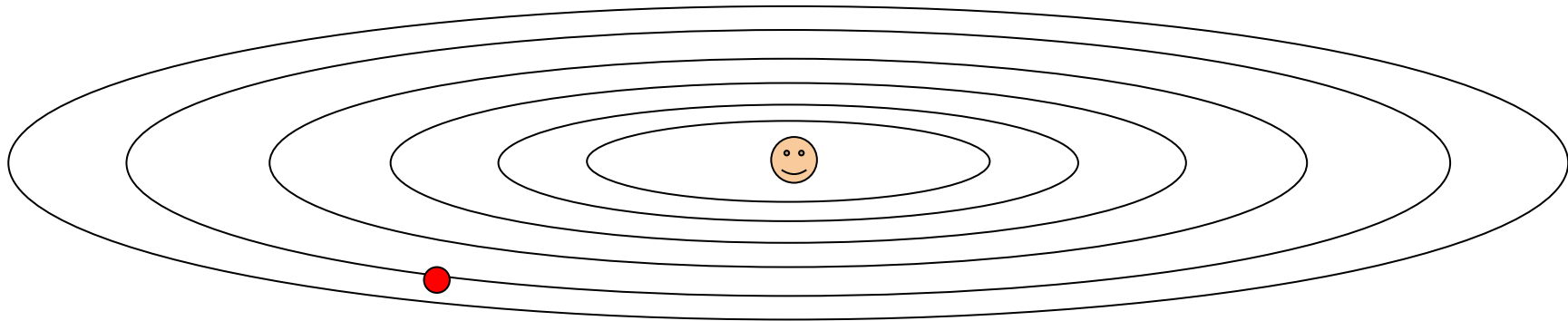
# RMSProp: "Leaky AdaGrad"

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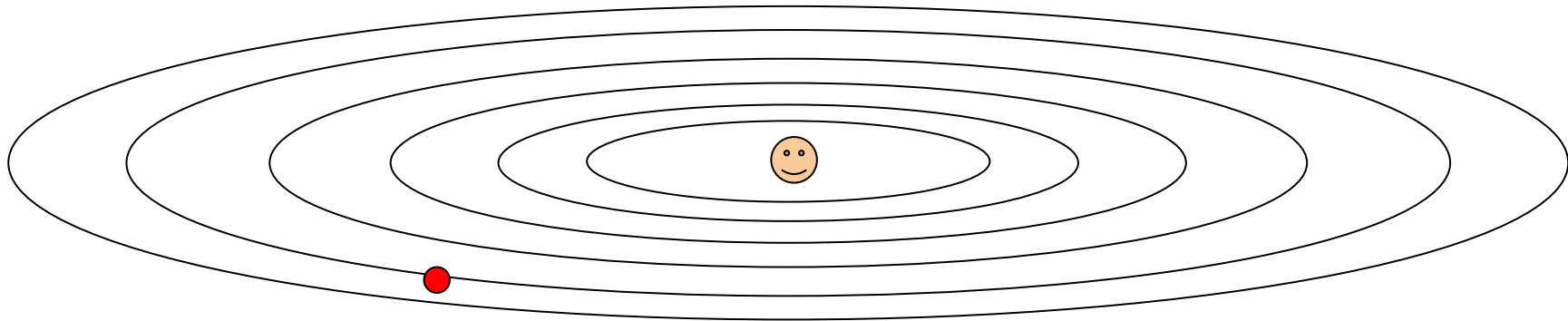
```
grad_squared = 0
while True:
  dx = compute_gradient(x)
  grad_squared += dx * dx
  x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```

**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)
```

Tieleman and Hinton, 2012
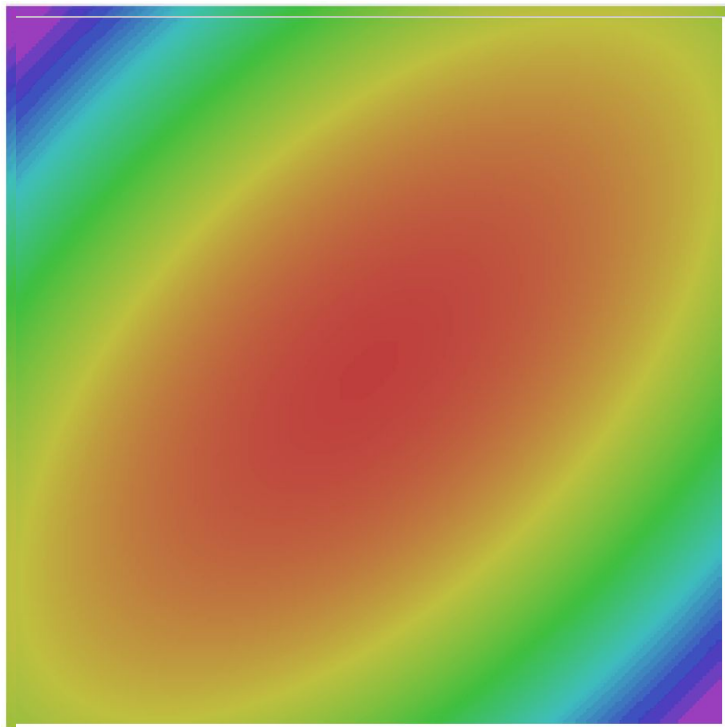
# RMSProp



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

# Adam (almost)

```python
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))
```

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

# Adam (almost)

```python
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

Sort of like RMSProp with momentum

Q: What happens at first timestep?

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

# Adam (full form)

```python
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

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

# 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

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

# Adam



SGD
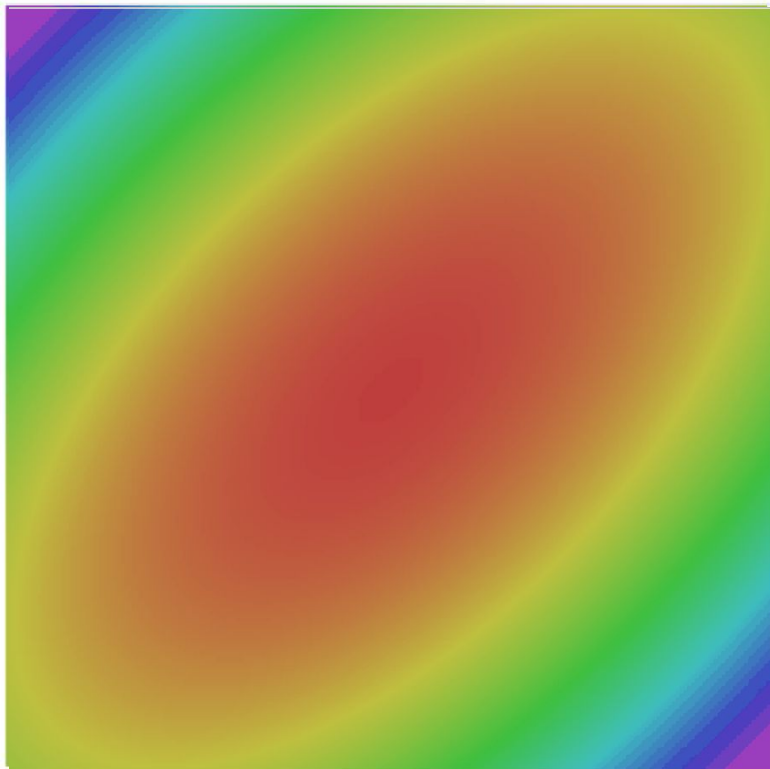
SGD+Momentum

RMSProp

Adam

# L2 Regularization vs Weight Decay

**Optimization Algorithm**

$L(w) = L_{data}(w) + L_{reg}(w)$

$g_t = \nabla L(w_t)$

$s_t = optimizer(g_t)$

$w_{t+1} = w_t - \alpha s_t$

**L2 Regularization**

$L(w) = L_{data}(w) + \lambda |w|^2$

$g_t = \nabla L(w_t) = \nabla L_{data}(w_t) + 2\lambda w_t$

$s_t = optimizer(g_t)$

$w_{t+1} = w_t - \alpha s_t$

# L2 Regularization vs Weight Decay

**Optimization Algorithm**

$L(w) = L_{data}(w) + L_{reg}(w)$

$g_t = \nabla L(w_t)$

$s_t = optimizer(g_t)$

$w_{t+1} = w_t - \alpha s_t$

L2 Regularization and Weight Decay are equivalent for SGD, SGD+Momentum so people often use the terms interchangeably!

**L2 Regularization**

$L(w) = L_{data}(w) + \lambda |w|^2$

$g_t = \nabla L(w_t) = \nabla L_{data}(w_t) + 2\lambda w_t$

$s_t = optimizer(g_t)$

$w_{t+1} = w_t - \alpha s_t$

**Weight Decay**

$L(w) = L_{data}(w)$

$g_t = \nabla L_{data}(w_t)$

$s_t = optimizer(g_t) + 2\lambda w_t$

$w_{t+1} = w_t - \alpha s_t$

# L2 Regularization vs Weight Decay

**Optimization Algorithm**

$$L(w) = L_{data}(w) + L_{reg}(w)$$
$$g_t = \nabla L(w_t)$$
$$s_t = optimizer(g_t)$$
$$w_{t+1} = w_t - \alpha s_t$$

L2 Regularization and Weight Decay are equivalent for SGD, SGD+Momentum so people often use the terms interchangeably!

But they are not the same for adaptive methods (AdaGrad, RMSProp, Adam, etc)

**L2 Regularization**

$$L(w) = L_{data}(w) + \lambda|w|^2$$
$$g_t = \nabla L(w_t) = \nabla L_{data}(w_t) + 2\lambda w_t$$
$$s_t = optimizer(g_t)$$
$$w_{t+1} = w_t - \alpha s_t$$

**Weight Decay**

$$L(w) = L_{data}(w)$$
$$g_t = \nabla L_{data}(w_t)$$
$$s_t = optimizer(g_t) + 2\lambda w_t$$
$$w_{t+1} = w_t - \alpha s_t$$

# AdamW: Decoupled Weight Decay

**Algorithm 2** Adam with $L_2$ regularization and Adam with decoupled weight decay (AdamW)

1: **given** $\alpha = 0.001, \beta_1 = 0.9, \beta_2 = 0.999, \epsilon = 10^{-8}, \lambda \in \mathbb{R}$
2: **initialize** time step $t \leftarrow 0$, parameter vector $\boldsymbol{\theta}_{t=0} \in \mathbb{R}^n$, first moment vector $\boldsymbol{m}_{t=0} \leftarrow \boldsymbol{0}$, second moment vector $\boldsymbol{v}_{t=0} \leftarrow \boldsymbol{0}$, schedule multiplier $\eta_{t=0} \in \mathbb{R}$
3: **repeat**
4: $\quad t \leftarrow t + 1$
5: $\quad \nabla f_t(\boldsymbol{\theta}_{t-1}) \leftarrow \text{SelectBatch}(\boldsymbol{\theta}_{t-1})$ $\qquad \triangleright$ select batch and return the corresponding gradient
6: $\quad \boldsymbol{g}_t \leftarrow \nabla f_t(\boldsymbol{\theta}_{t-1}) \boxed{+\lambda\boldsymbol{\theta}_{t-1}}$
7: $\quad \boldsymbol{m}_t \leftarrow \beta_1 \boldsymbol{m}_{t-1} + (1 - \beta_1)\boldsymbol{g}_t$ $\qquad \triangleright$ here and below all operations are element-wise
8: $\quad \boldsymbol{v}_t \leftarrow \beta_2 \boldsymbol{v}_{t-1} + (1 - \beta_2)\boldsymbol{g}_t^2$
9: $\quad \hat{\boldsymbol{m}}_t \leftarrow \boldsymbol{m}_t/(1 - \beta_1^t)$ $\qquad \triangleright$ $\beta_1$ is taken to the power of $t$
10: $\quad \hat{\boldsymbol{v}}_t \leftarrow \boldsymbol{v}_t/(1 - \beta_2^t)$ $\qquad \triangleright$ $\beta_2$ is taken to the power of $t$
11: $\quad \eta_t \leftarrow \text{SetScheduleMultiplier}(t)$ $\qquad \triangleright$ can be fixed, decay, or also be used for warm restarts
12: $\quad \boldsymbol{\theta}_t \leftarrow \boldsymbol{\theta}_{t-1} - \eta_t \left( \alpha \hat{\boldsymbol{m}}_t/(\sqrt{\hat{\boldsymbol{v}}_t} + \epsilon) \boxed{+\lambda\boldsymbol{\theta}_{t-1}} \right)$
13: **until** *stopping criterion is met*
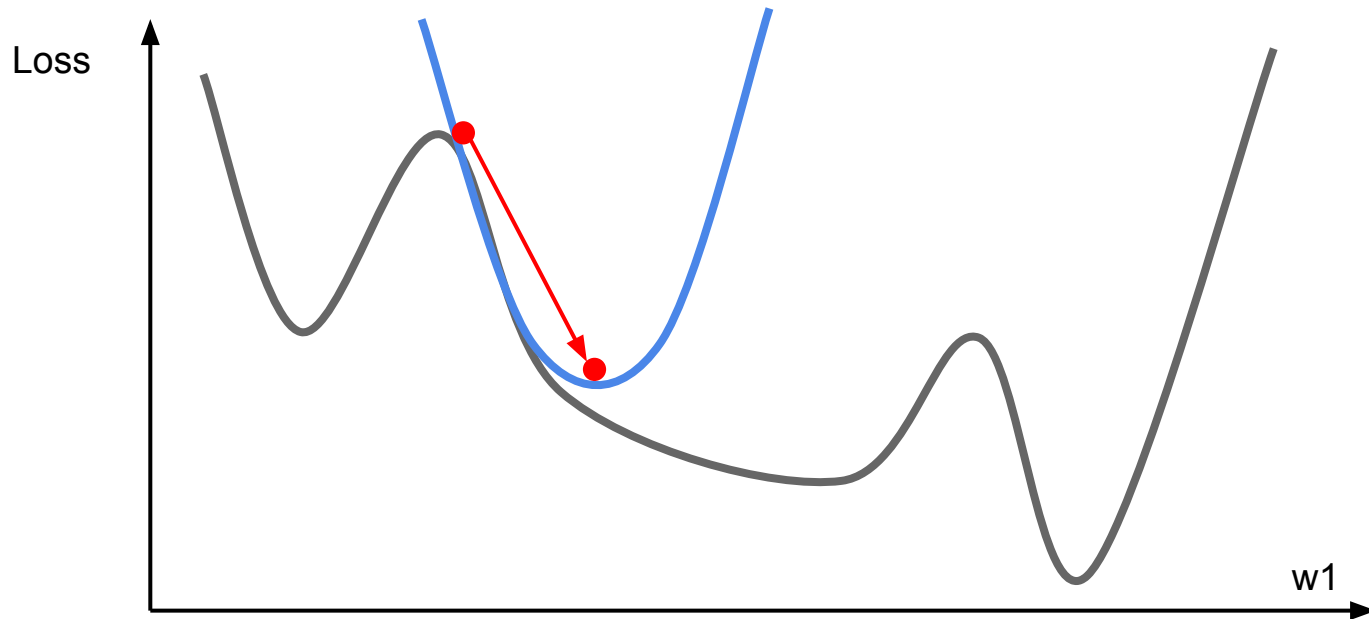14: **return** optimized parameters $\boldsymbol{\theta}_t$

# First-Order Optimization
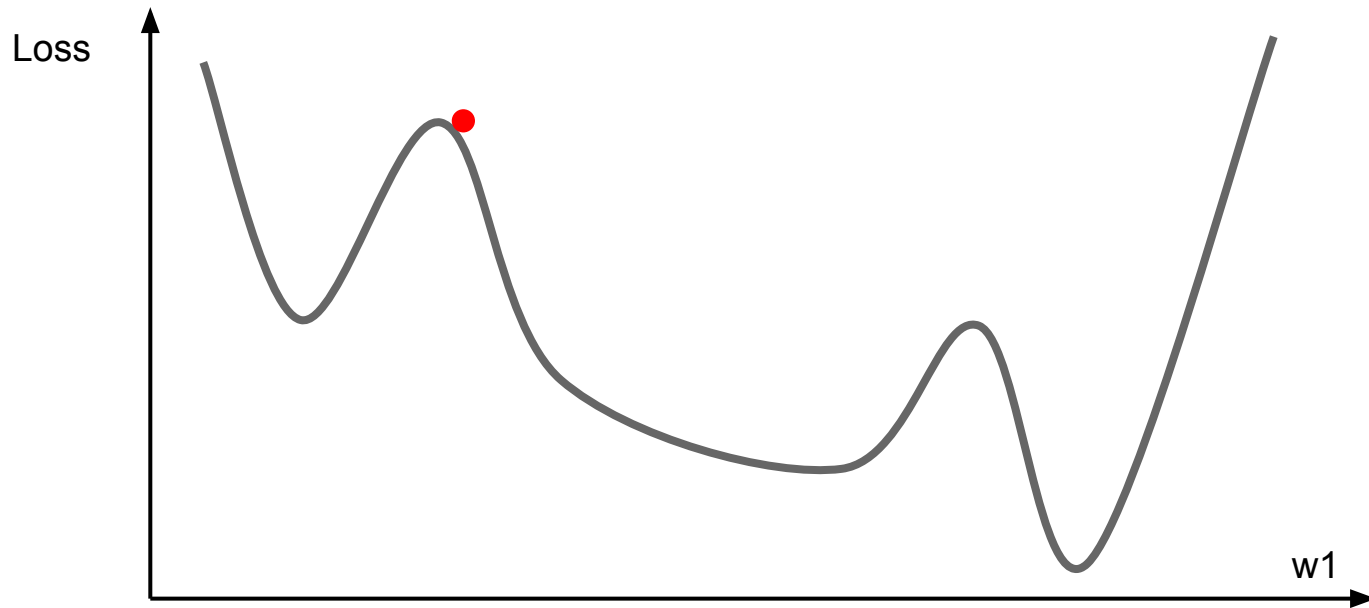
# First-Order Optimization

(1)   Use gradient form linear approximation
(2)   Step to minimize the approximation

Loss

w1
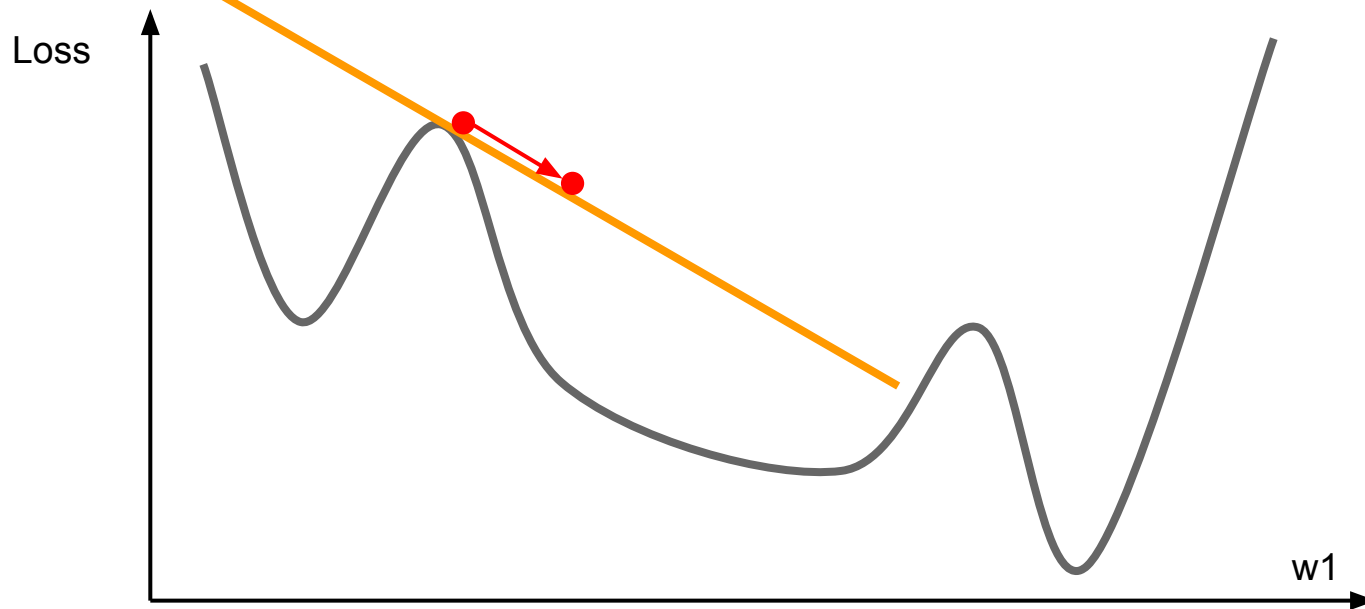
# Second-Order Optimization

(1) Use gradient **and Hessian** to form **quadratic** approximation
(2) Step to the **minima** of the approximation

# Second-Order Optimization

second-order Taylor expansion:

$$L(\theta) \approx L(\theta_0) + (\theta - \theta_0)^T \nabla_\theta L(\theta_0) + \frac{1}{2}(\theta - \theta_0)^T H (\theta - \theta_0)$$

Solving for the critical point we obtain the Newton parameter update:

$$\theta^* = \theta_0 - H^{-1} \nabla_\theta L(\theta_0)$$

Q: Why is this bad for deep learning?

# Second-Order Optimization

second-order Taylor expansion:

$$L(\theta) \approx L(\theta_0) + (\theta - \theta_0)^T \nabla_\theta L(\theta_0) + \frac{1}{2}(\theta - \theta_0)^T H (\theta - \theta_0)$$

Solving for the critical point we obtain the Newton parameter update:

$$\theta^* = \theta_0 - H^{-1} \nabla_\theta L(\theta_0)$$

Hessian has O(N^2) elements
Inverting takes O(N^3)
N = (Tens or Hundreds of) Millions

Q: Why is this bad for deep learning?

# Second-Order Optimization

$$\theta^* = \theta_0 - H^{-1}\nabla_\theta L(\theta_0)$$

- Quasi-Newton methods (**BGFS** most popular):
  *instead of inverting the Hessian (O(n^3)), approximate inverse Hessian with rank 1 updates over time (O(n^2) each).*

- **L-BFGS** (Limited memory BFGS):
  *Does not form/store the full inverse Hessian.*

# L-BFGS

- **Usually works very well in full batch, deterministic mode** i.e. if you have a single, deterministic f(x) then L-BFGS will probably work very nicely

- **Does not transfer very well to mini-batch setting**. Gives bad results. Adapting second-order methods to large-scale, stochastic setting is an active area of research.

Le et al, "On optimization methods for deep learning, ICML 2011"
Ba et al, "Distributed second-order optimization using Kronecker-factored approximations", ICLR 2017

# In practice:

- **AdamW** should probably be your "default" optimizer for new problems
- **Adam** is a good second choice in many cases; it often works ok even with constant learning rate
- **SGD+Momentum** can outperform Adam but may require more tuning of LR and schedule
    - Try cosine schedule, very few hyperparameters!

# Learning rate schedules

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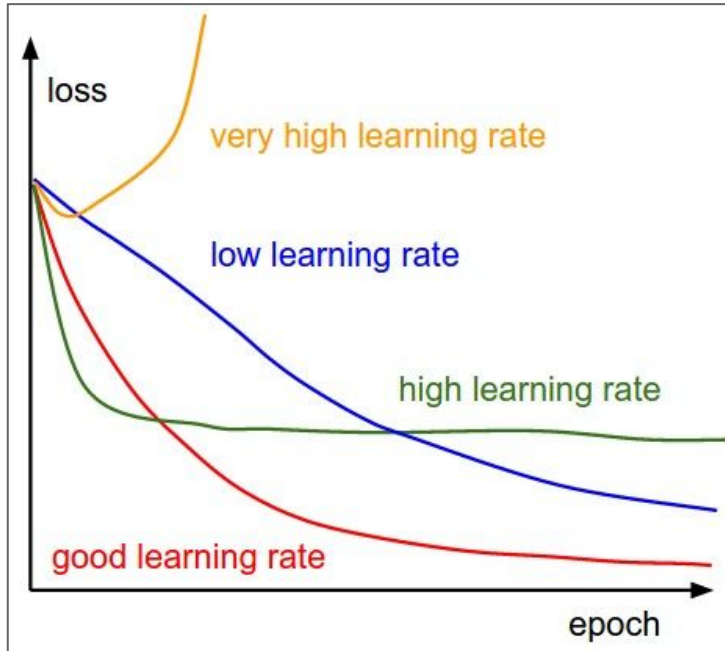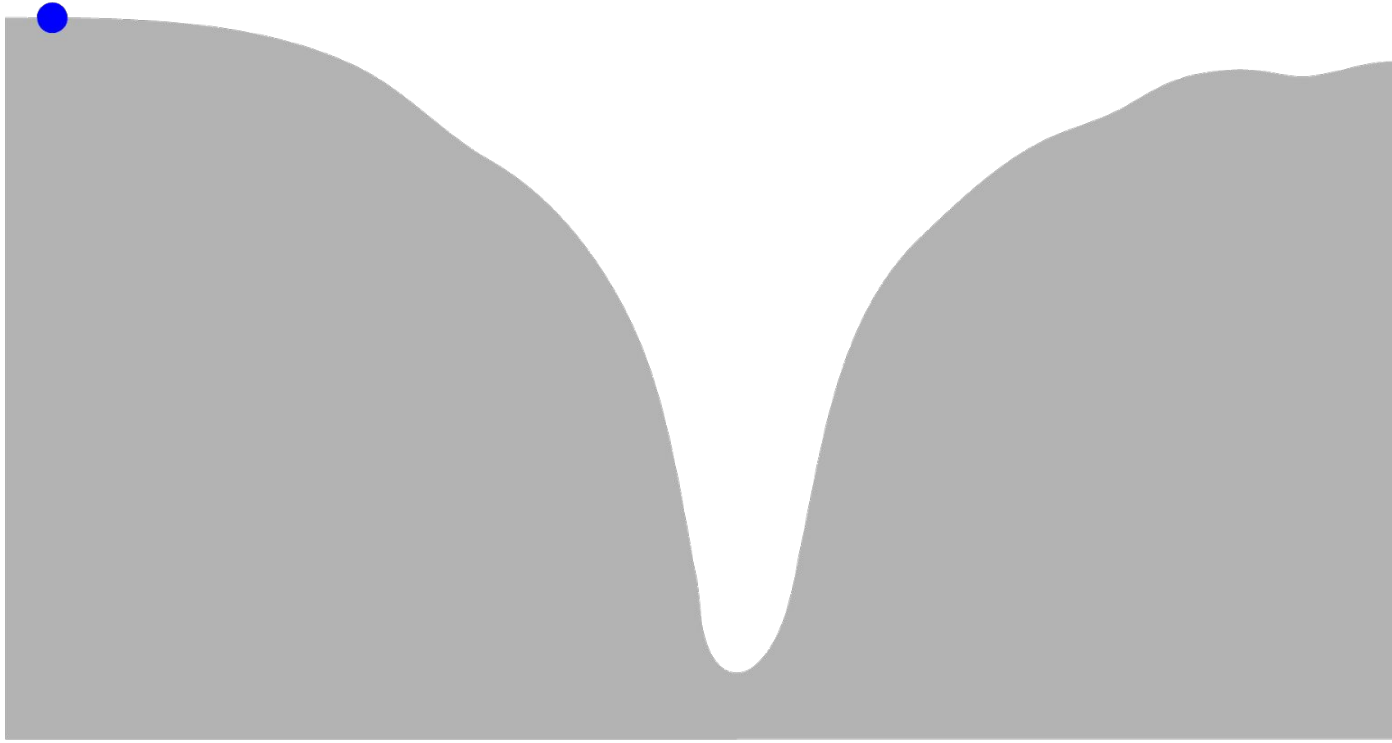


Q: Which one of these learning rates is best to use?

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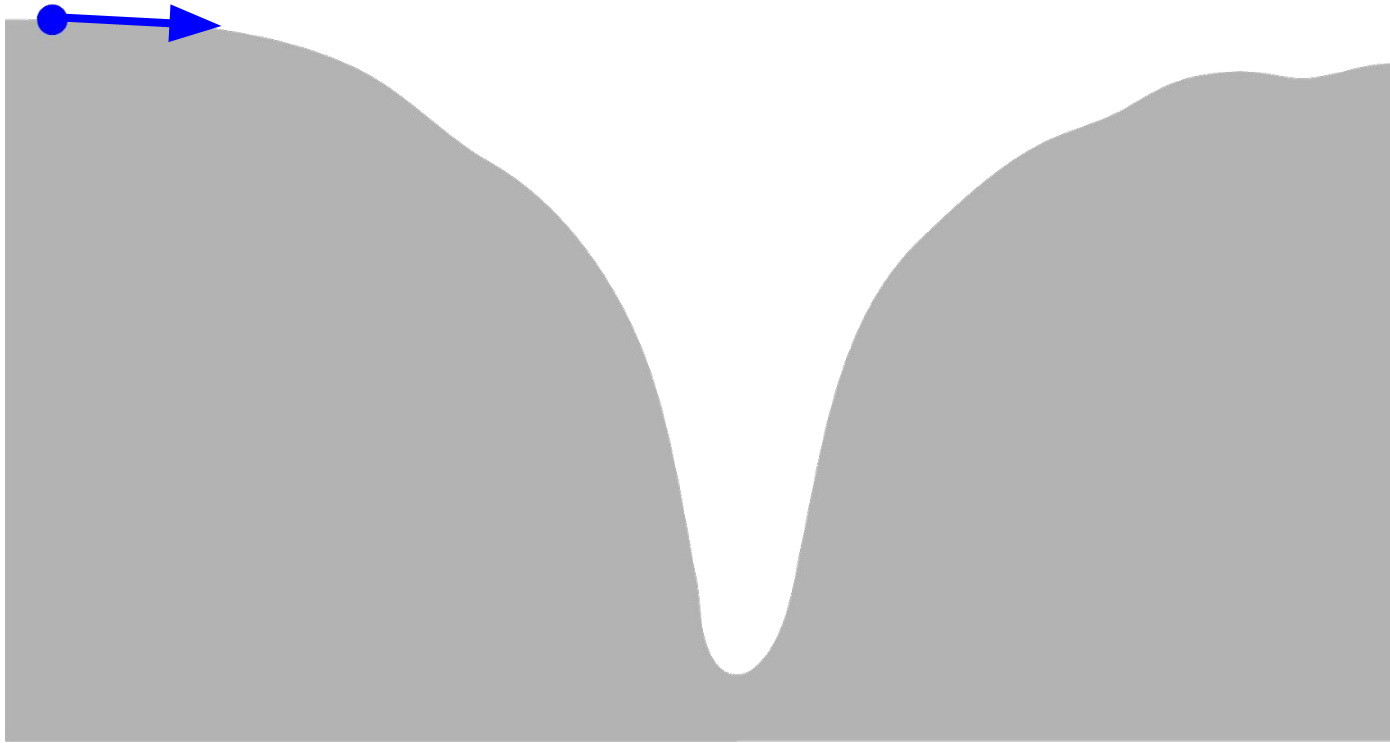


loss

very high learning rate

low learning rate

high learning rate

good learning rate

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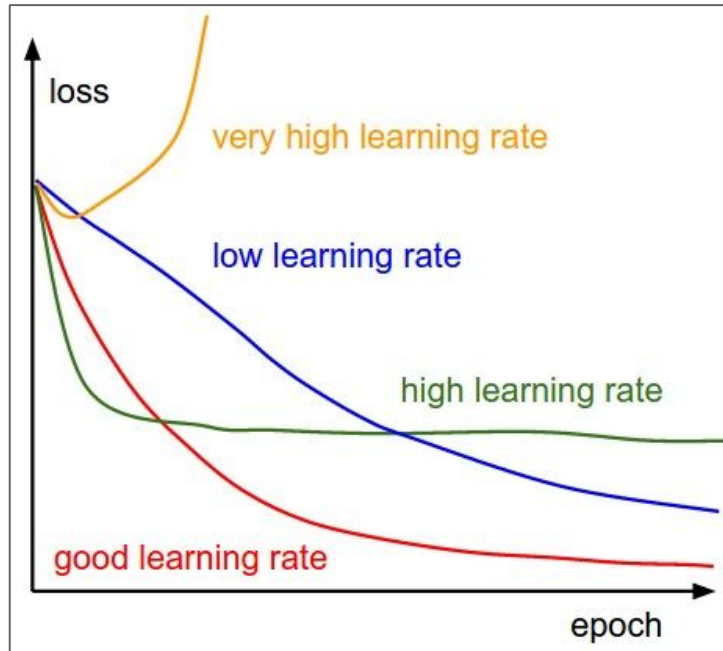
Q: Which one of these learning rates is best to use?

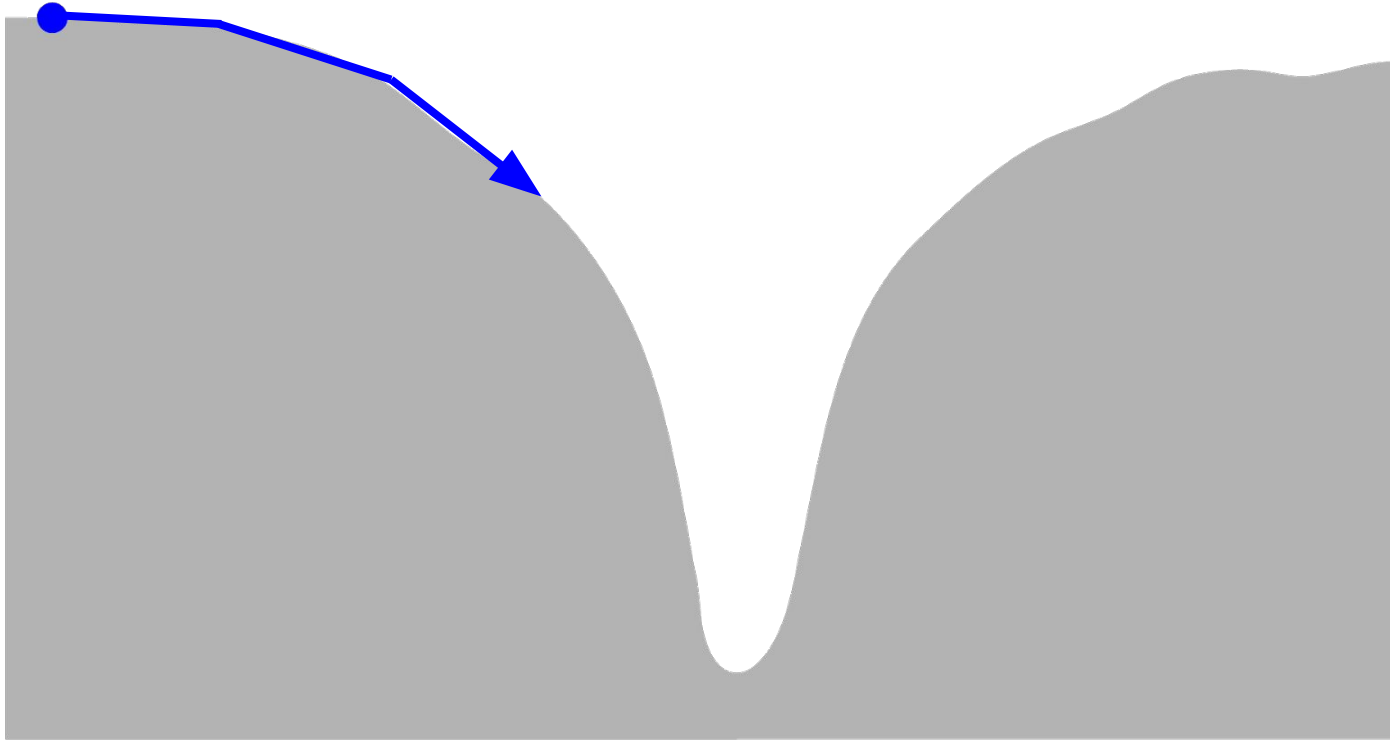A: In reality, all of these are good learning rates.
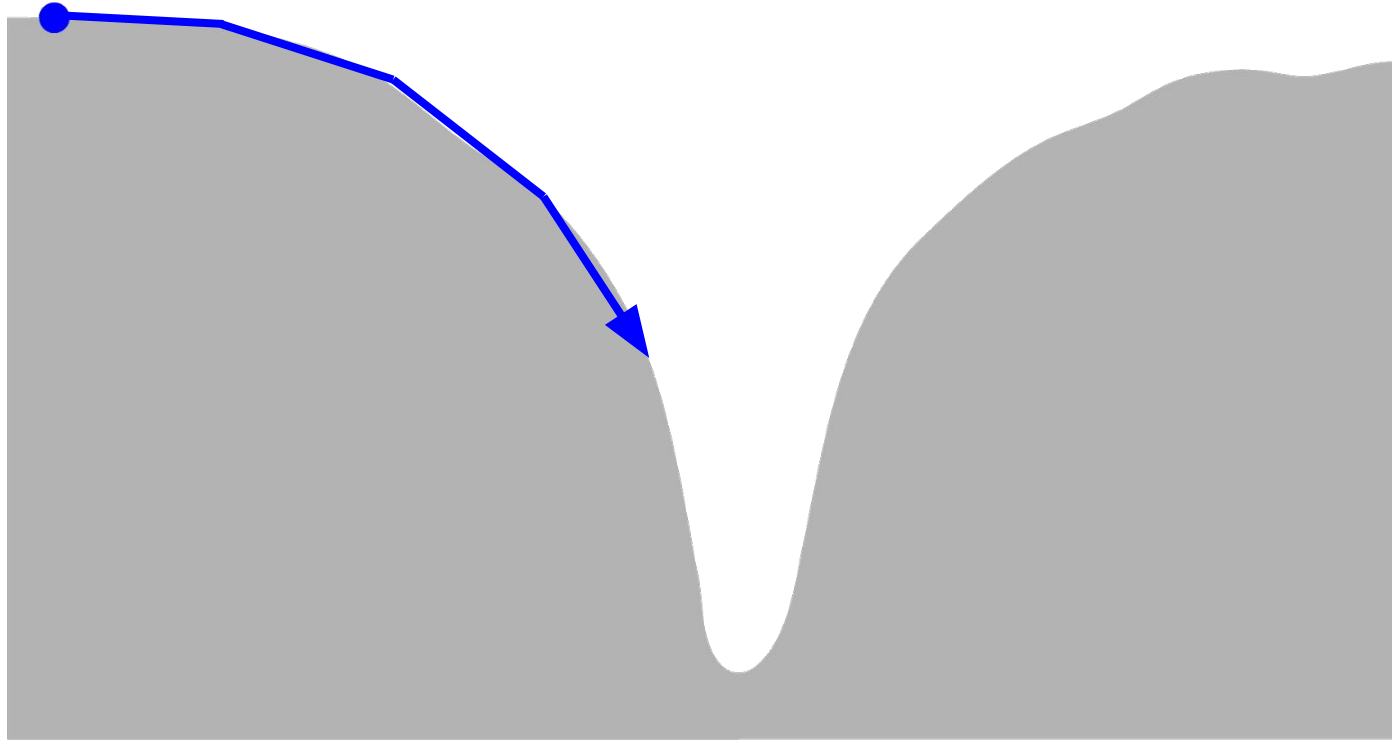
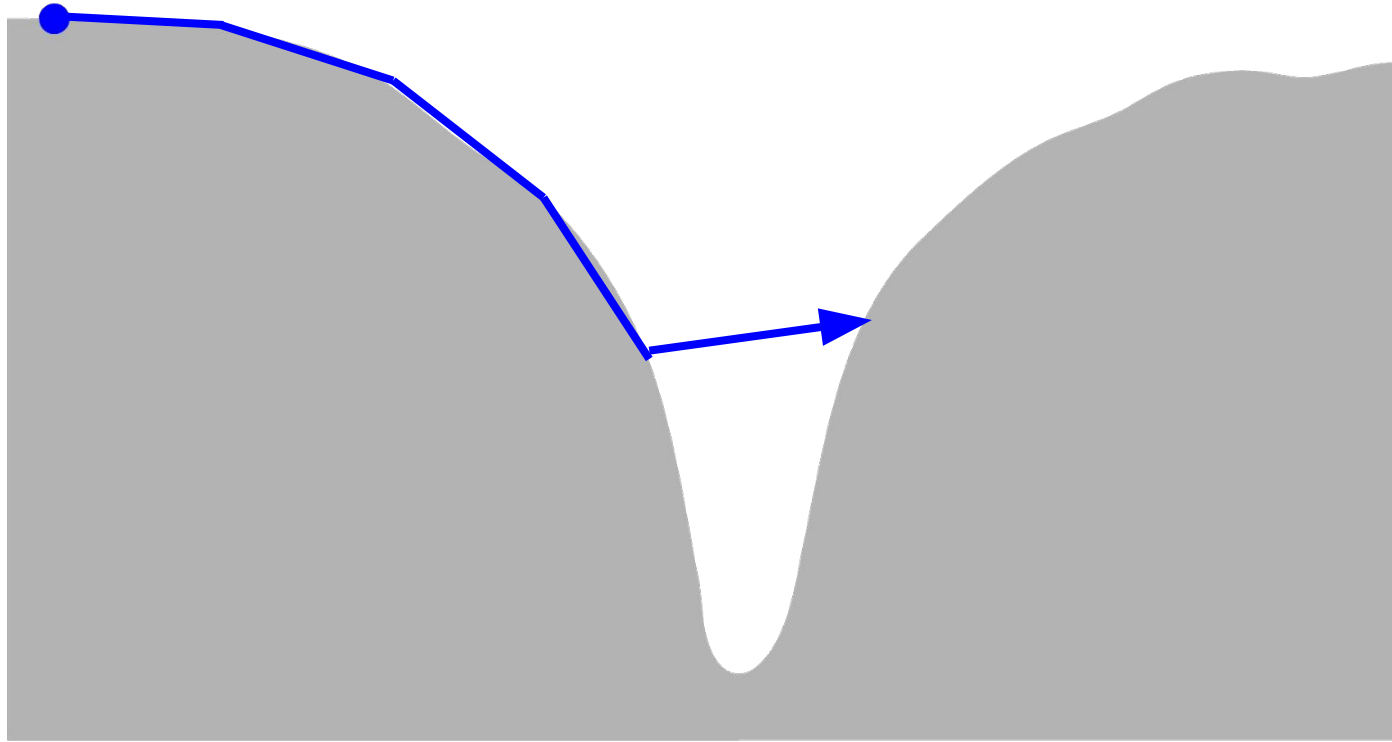# Phases of learning...
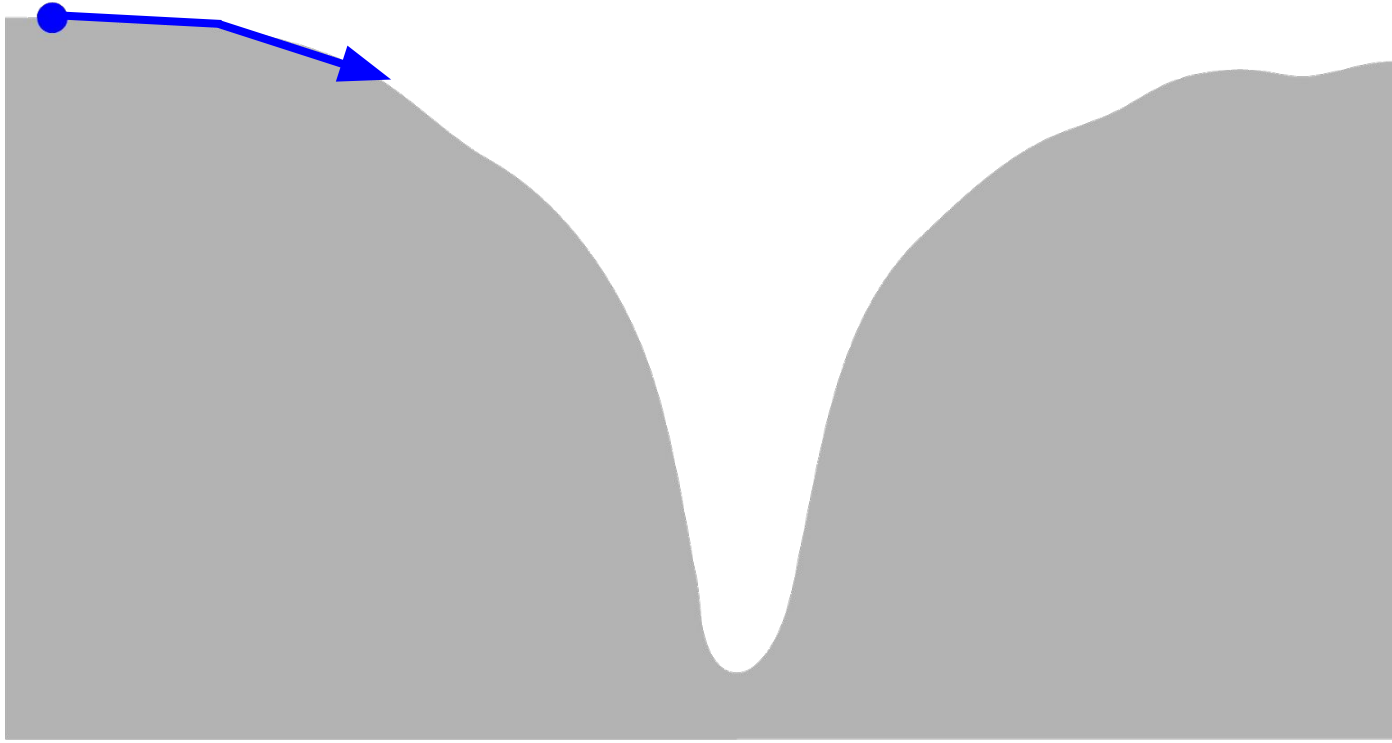
# Phases of learning...

# Phases of learning...

# Phases of learning...
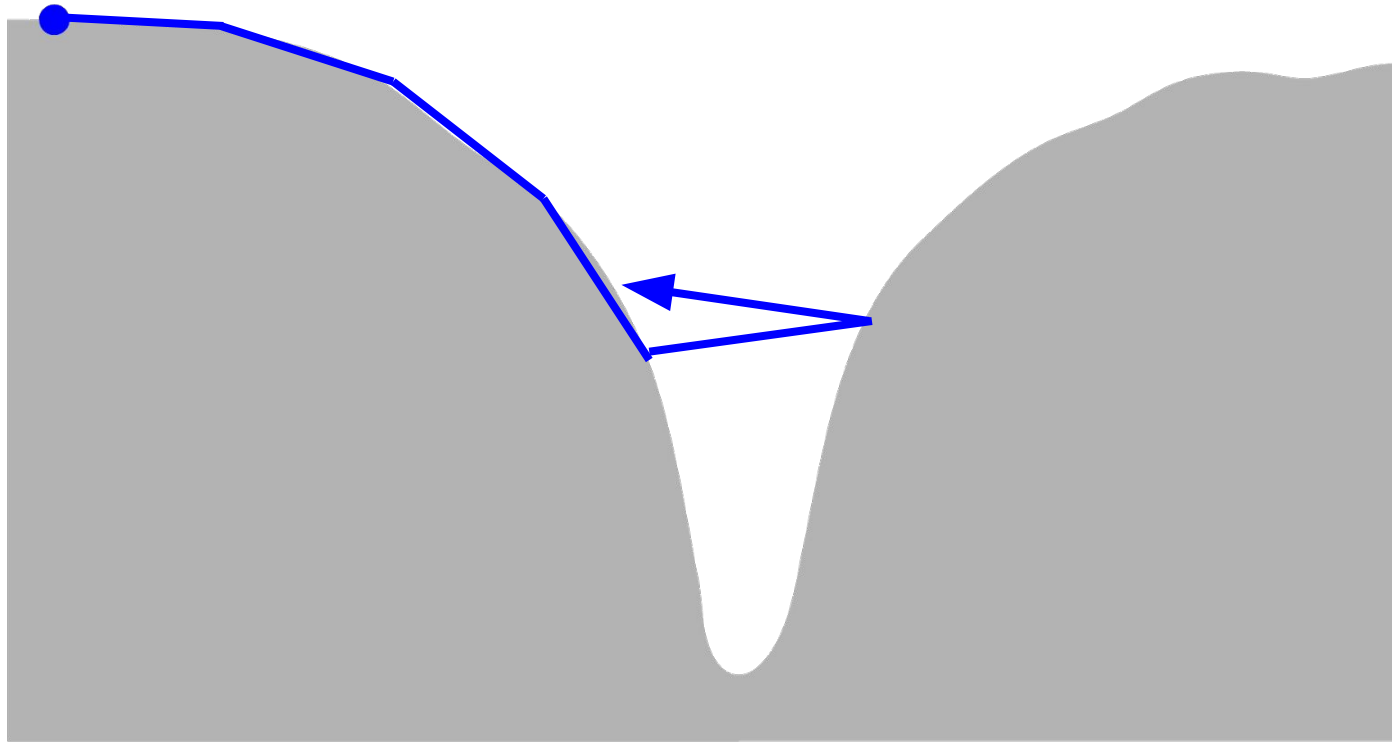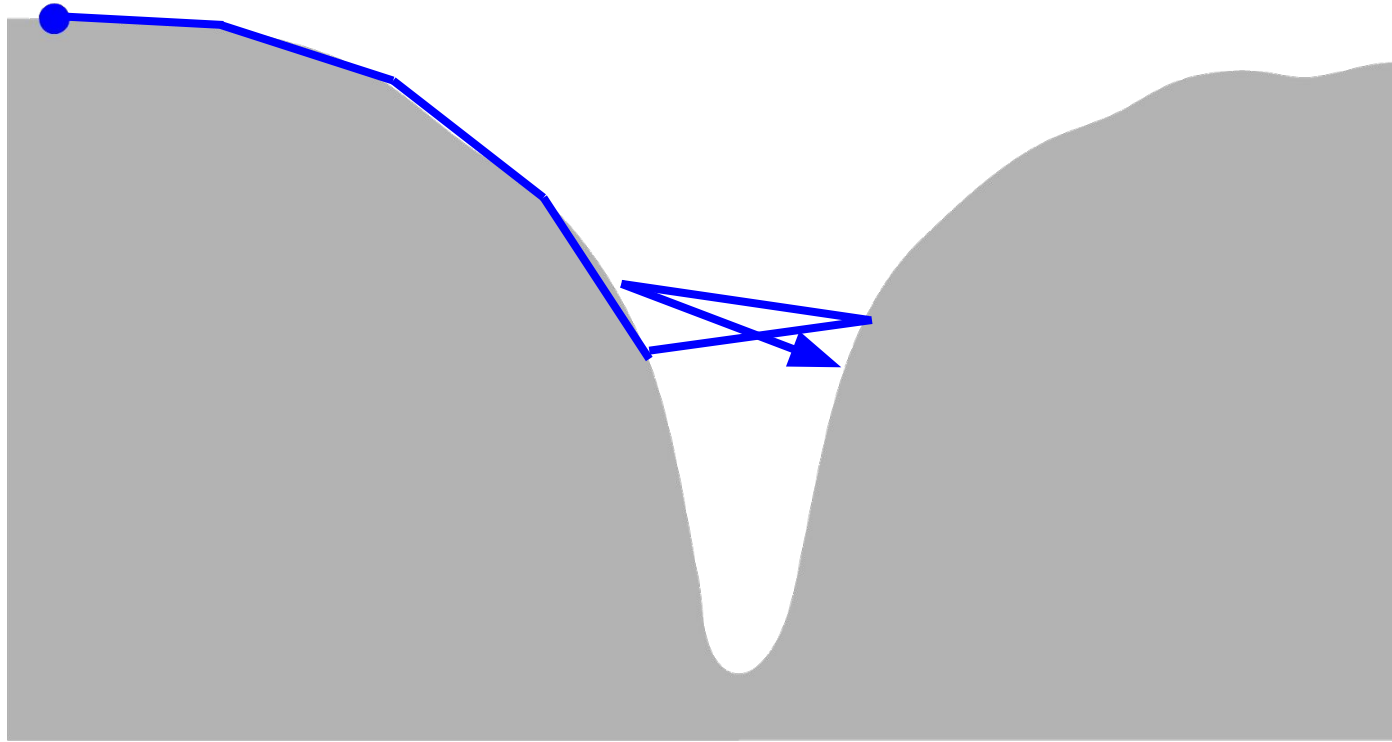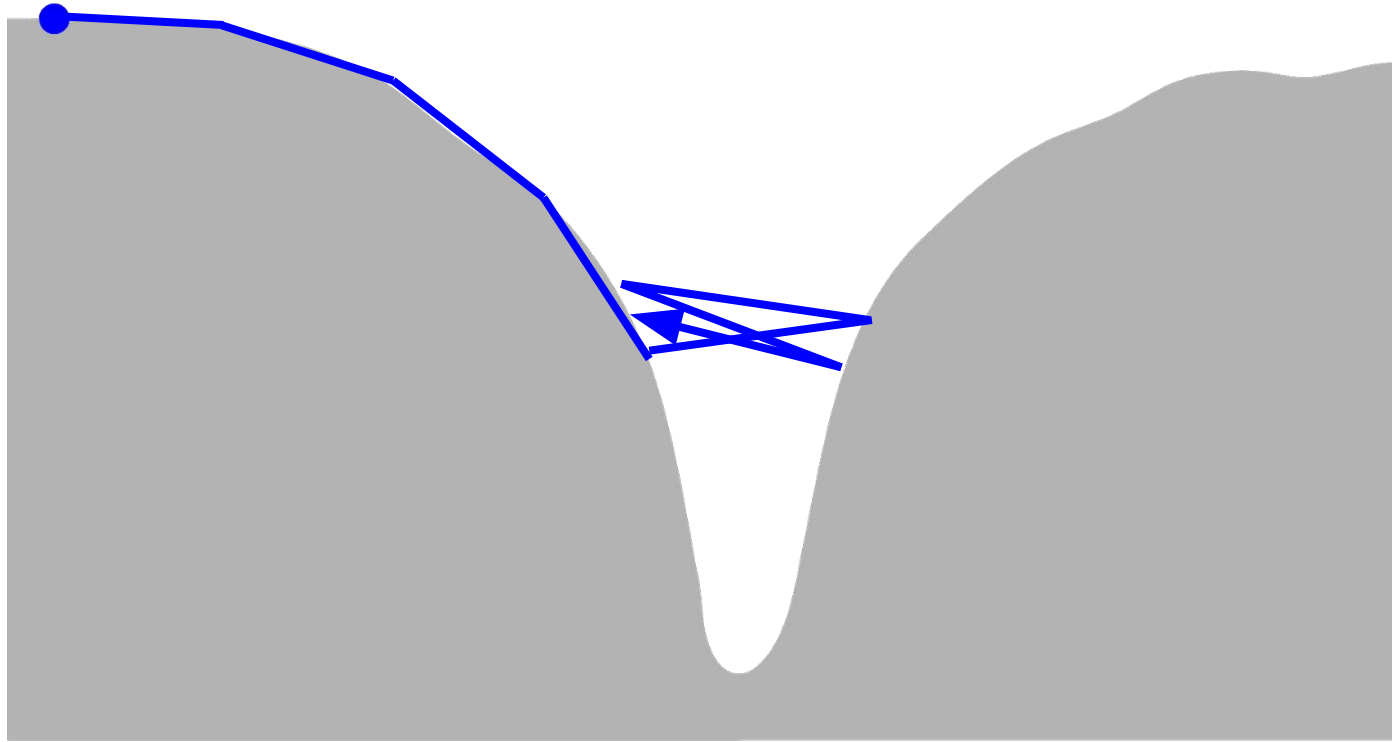
# Phases of learning...

# Phases of learning...

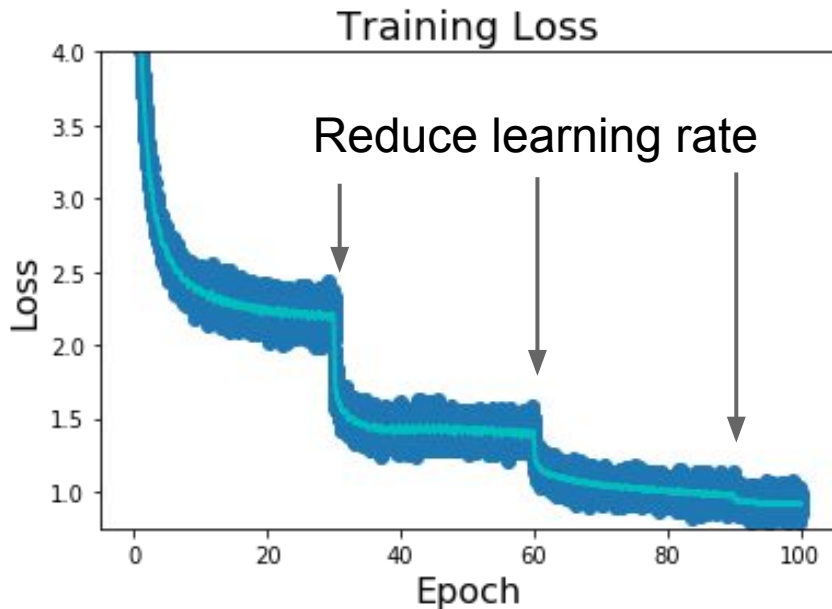# Phases of learning...

# Phases of learning...

# Phases of learning...

# Learning rate decays over time


Training Loss

Reduce learning rate

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

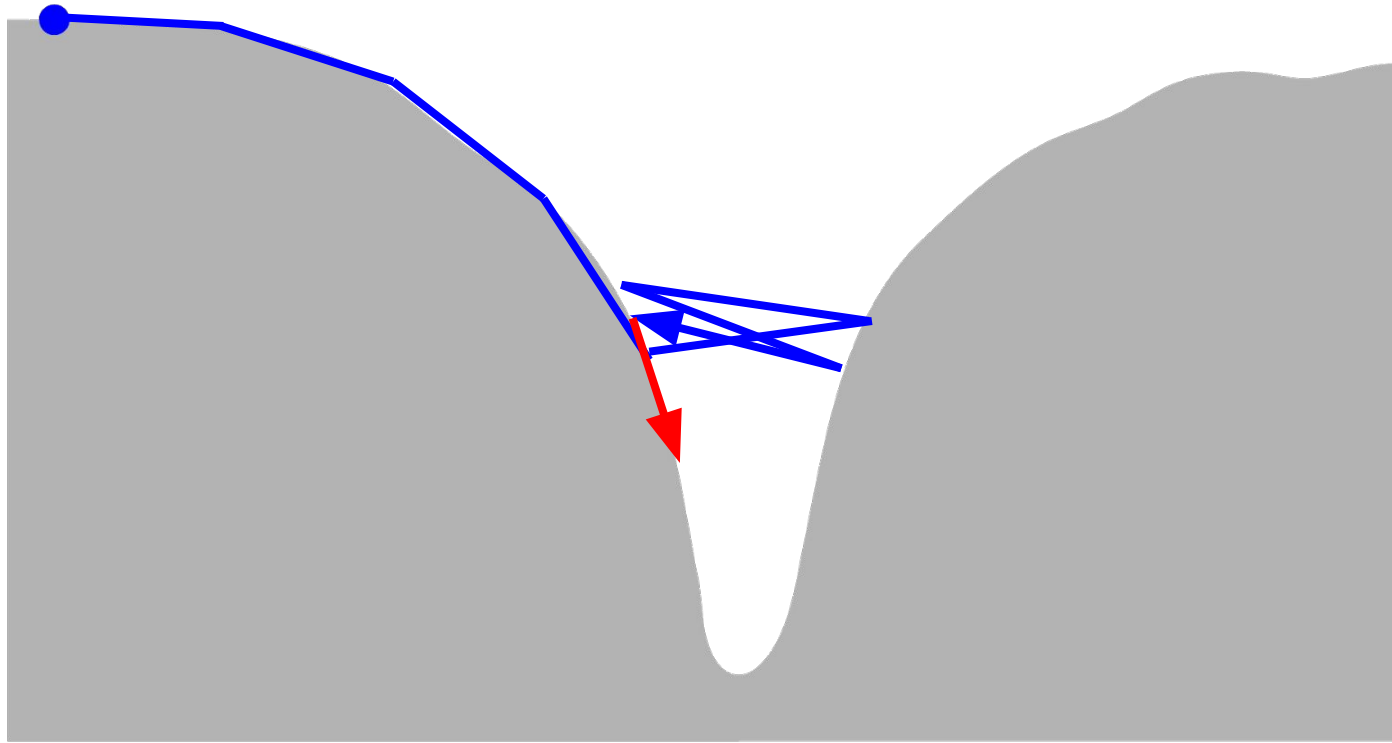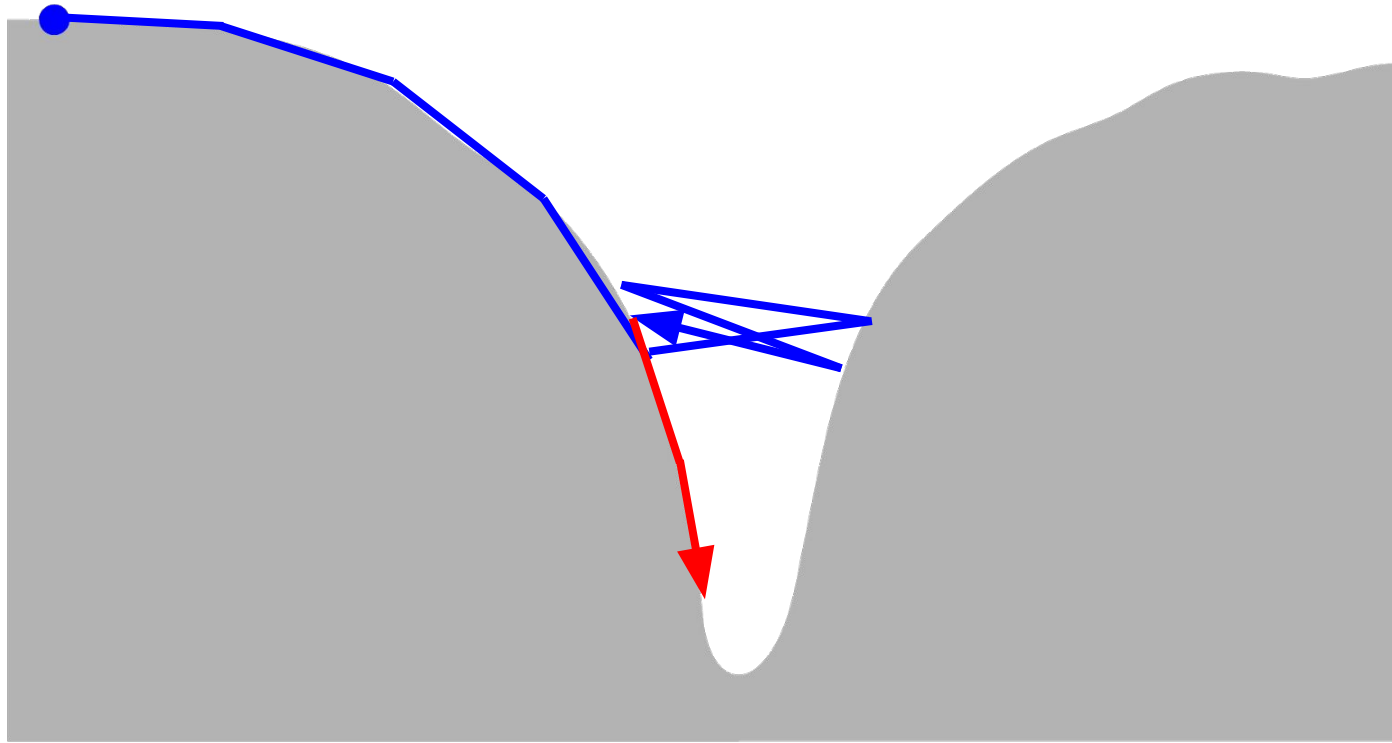# Phases of learning...

# Phases of learning...

# Learning Rate Decay

Learning rate



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

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 at al, "Generating Long Sequences with Sparse Transformers", arXiv 2019

$\alpha_0$ : Initial learning rate
$\alpha_t$ : Learning rate at epoch t
$T$ : Total number of epochs

# Learning Rate Decay



Training Loss

**Step:** Reduce learning rate at a few fixed points. E.g. for ResNets, multiply LR by 0.1 after epochs 30, 60, and 90.
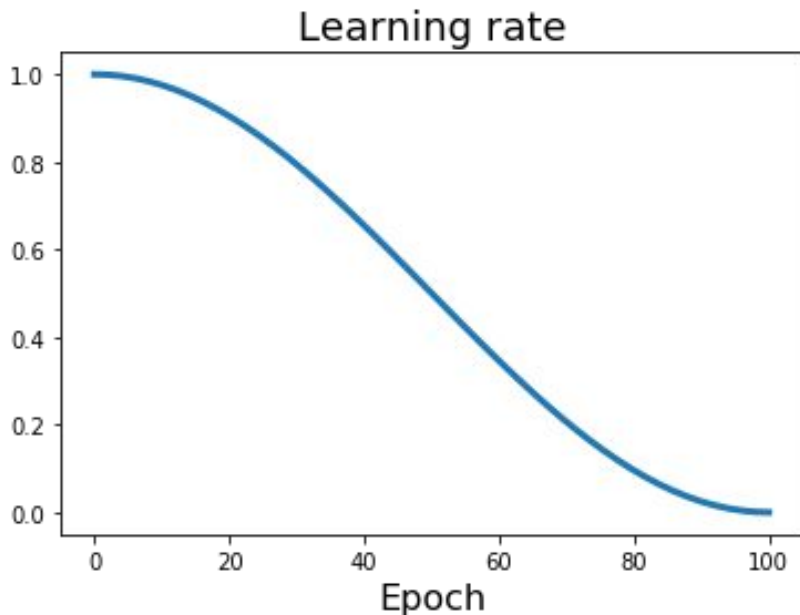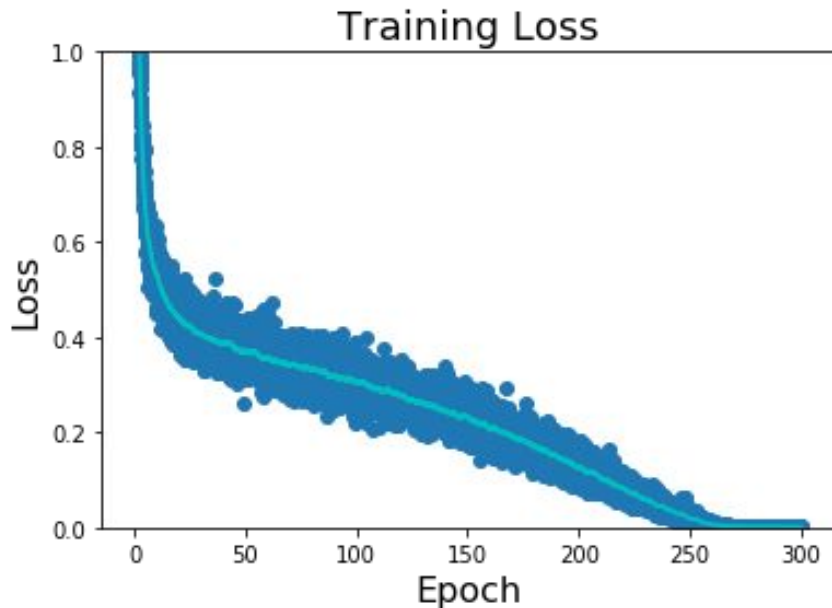
**Cosine**: $\alpha_t = \frac{1}{2}\alpha_0\left(1 + \cos(t\pi/T)\right)$

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 at al, "Generating Long Sequences with Sparse Transformers", arXiv 2019

$\alpha_0$ : Initial learning rate
$\alpha_t$ : Learning rate at epoch t
$T$ : Total number of epochs

# Learning Rate Decay

Learning rate



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

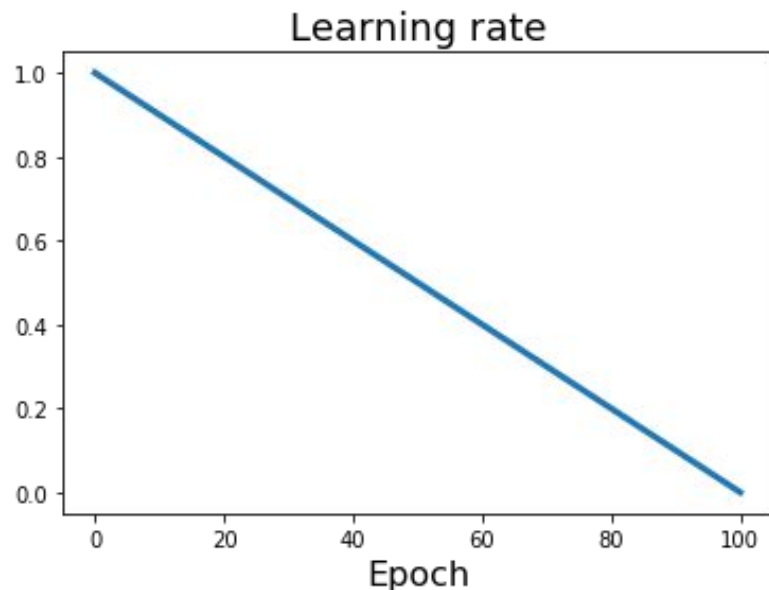**Cosine**: $\alpha_t = \frac{1}{2}\alpha_0\left(1 + \cos(t\pi/T)\right)$

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

$\alpha_0$ : Initial learning rate
$\alpha_t$ : Learning rate at epoch t
$T$ : Total number of epochs

Devlin et al, "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding", 2018

# Learning Rate Decay



Learning rate

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

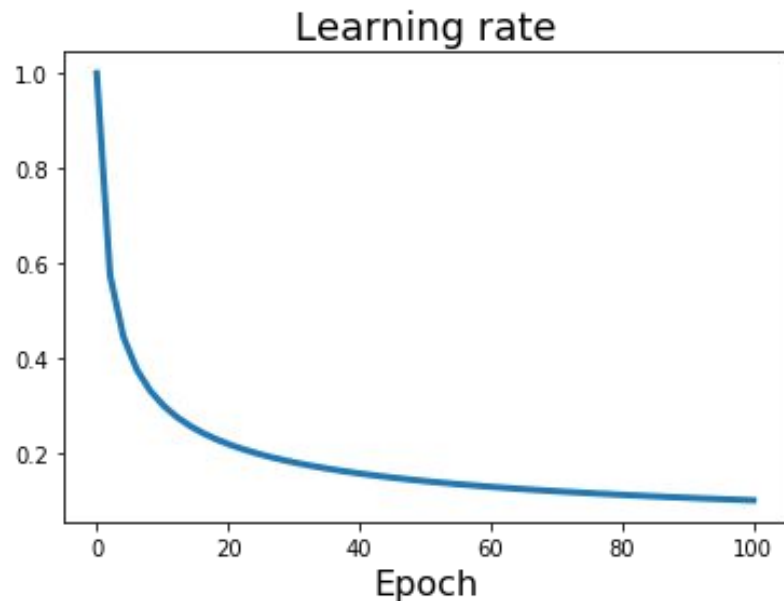**Cosine**: $\alpha_t = \frac{1}{2}\alpha_0 \left(1 + \cos(t\pi/T)\right)$

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

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

$\alpha_0$ : Initial learning rate
$\alpha_t$ : Learning rate at epoch t
$T$ : Total number of epochs

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

# Learning Rate Decay



Learning rate

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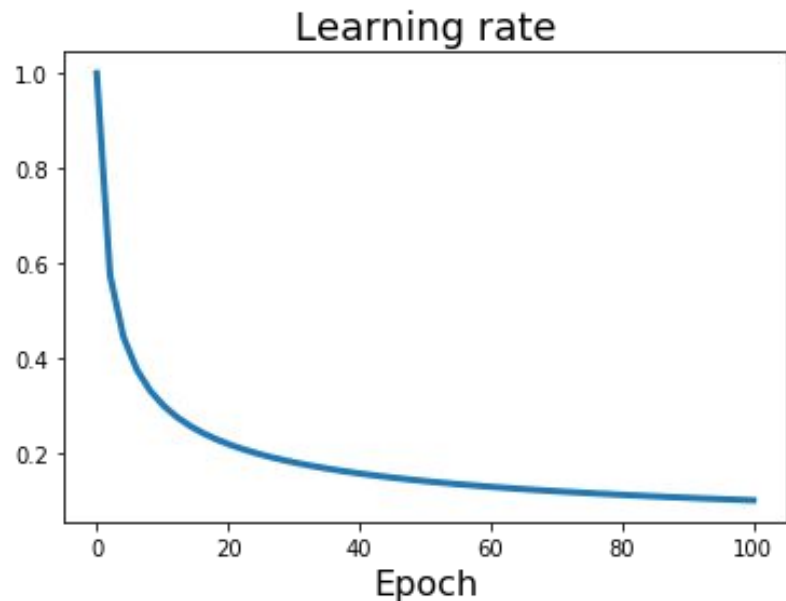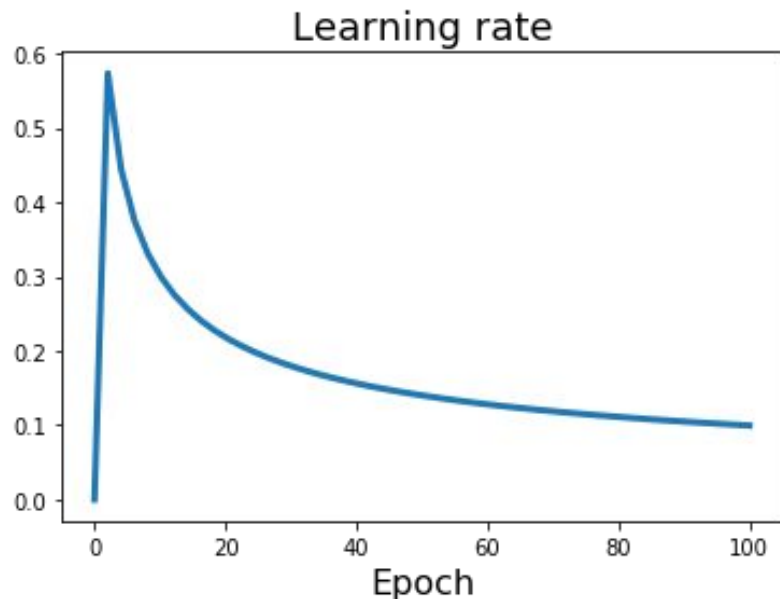
**Linear:** $\alpha_t = \alpha_0(1 - t/T)$

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

**Constant:** $\alpha_t = \alpha_0$

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

# Learning Rate Decay: Linear Warmup

## Learning rate



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

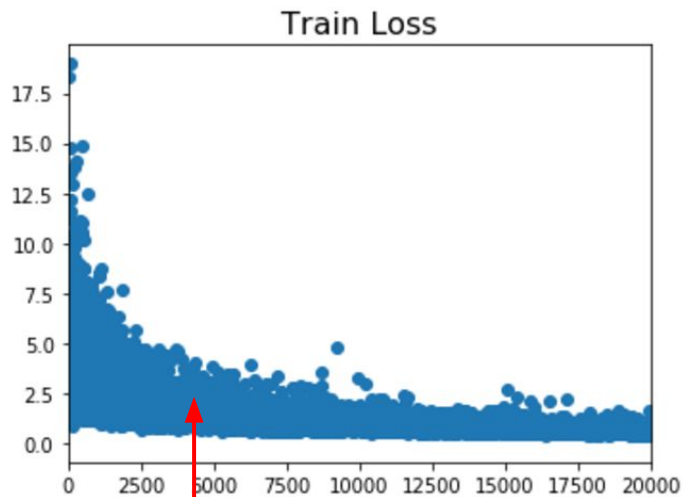Empirical rule of thumb: If you increase the batch size by N, also scale the initial learning rate by N

Goyal et al, "Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour", arXiv 2017
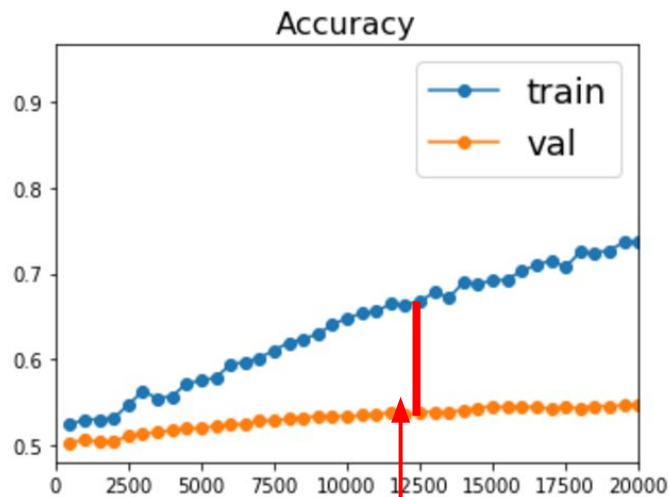
# with cosine and warmup



train_loss

info_info_normalize_proj_head_prior_elanding_60_apr_13 ●
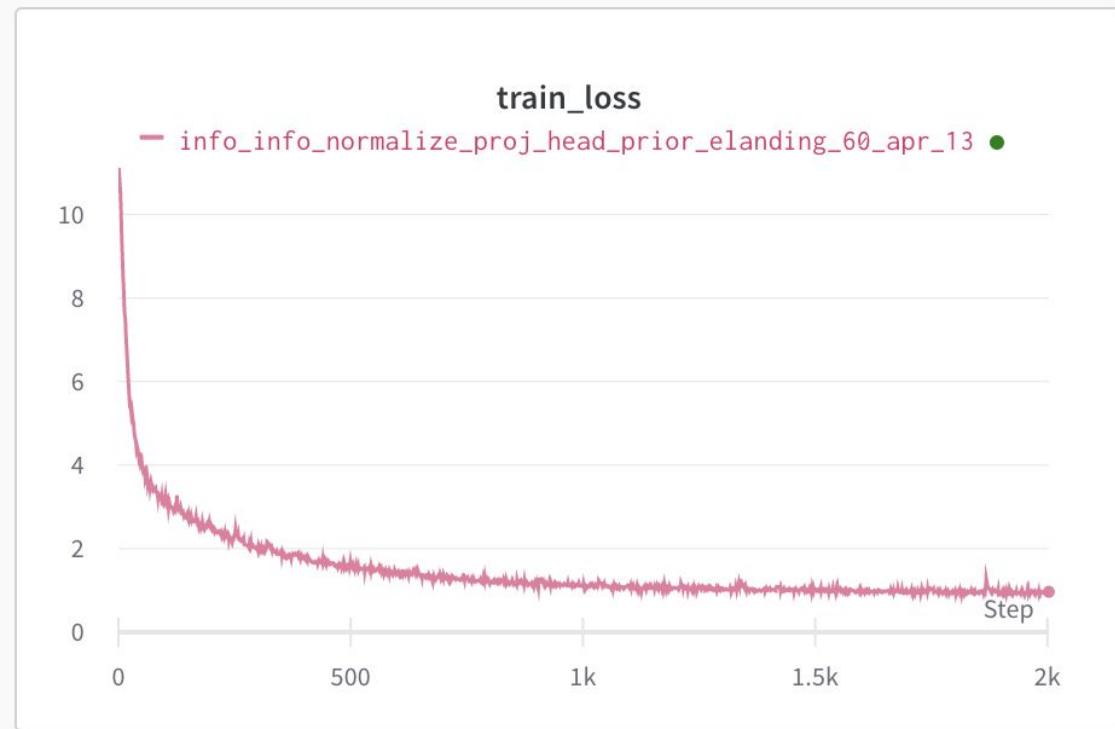
# Improve test error
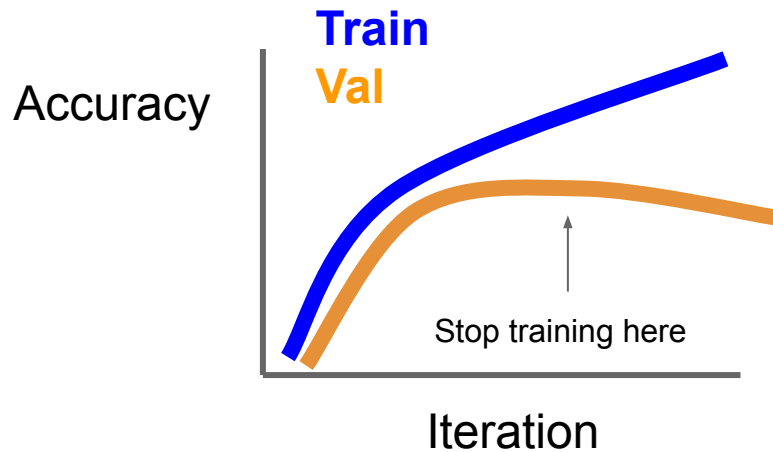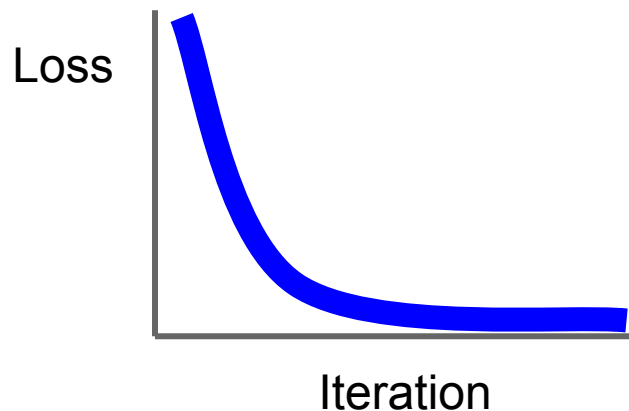
# Beyond Training Error



Better optimization algorithms help reduce training loss

But we really 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
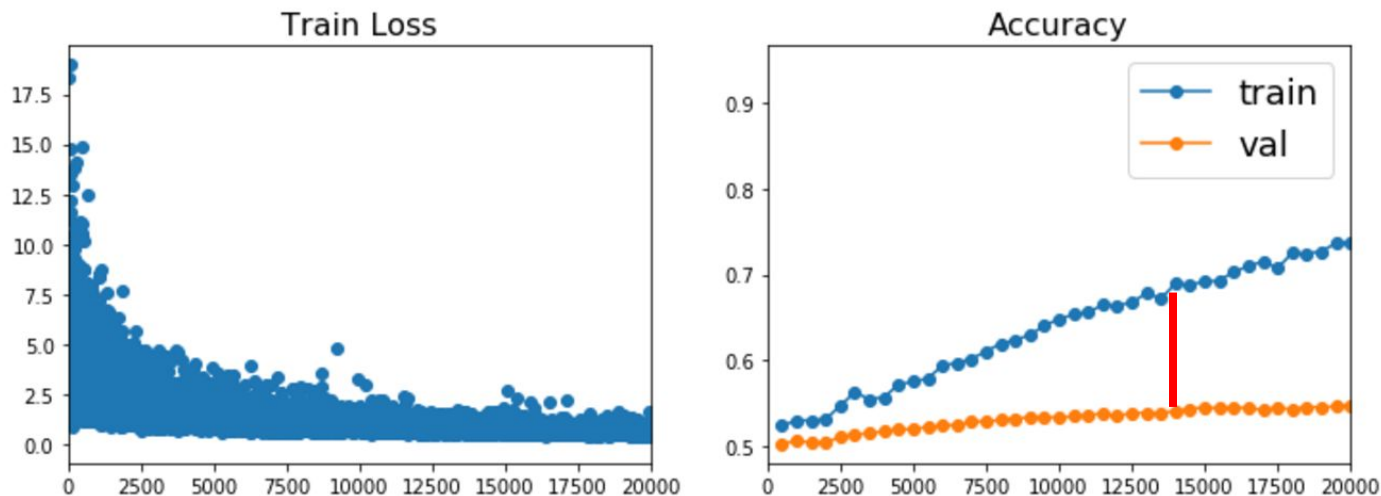Or train for a long time, but always keep track of the model snapshot
that worked best on val

# Model Ensembles

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

    (Take average of predicted probability distributions, then choose argmax)

Enjoy 2% extra performance

# How to improve single-model performance?



## 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     $R(W) = \sum_k \sum_l W_{k,l}^2$   (Weight decay)
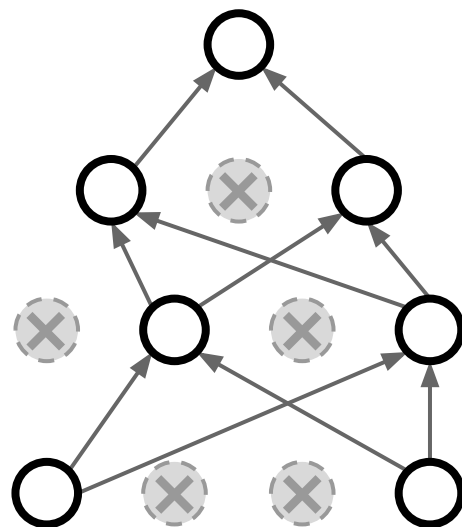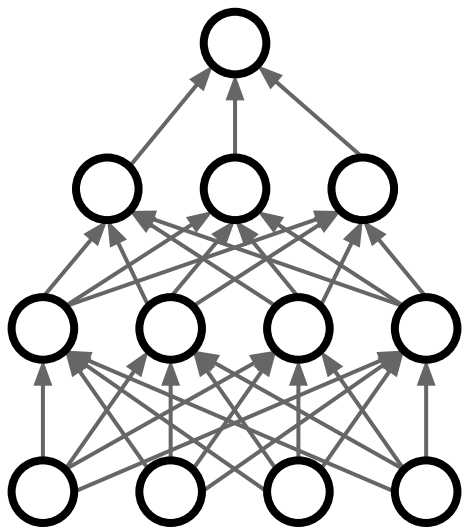
L1 regularization     $R(W) = \sum_k \sum_l |W_{k,l}|$

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

Example forward pass with a 3-layer network using 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)
```
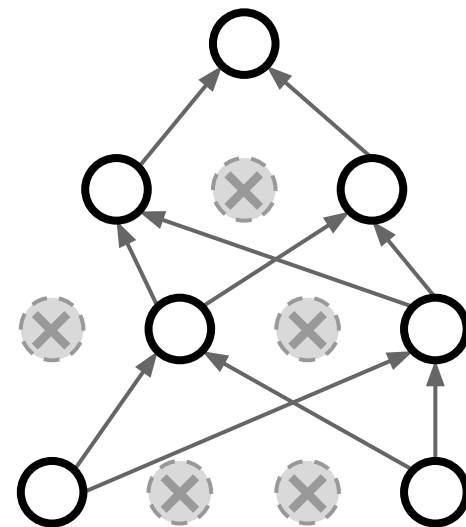
# Regularization: Dropout

How can this possibly be a good idea?
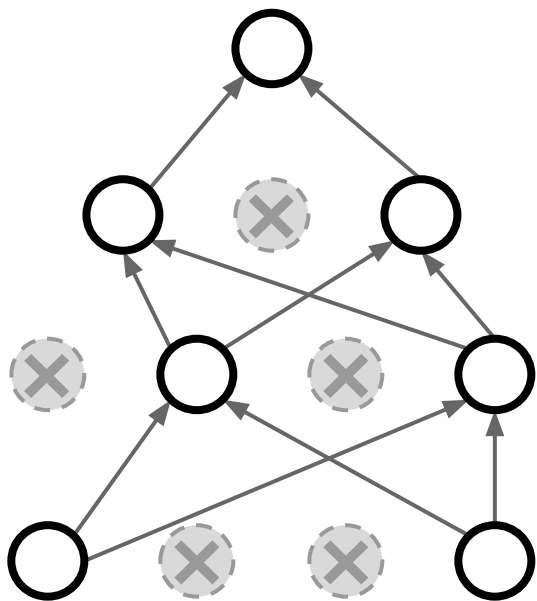


Forces the network to have a redundant representation;
Prevents co-adaptation of features

has an ear ✗

has a tail

is furry ✗

has claws

mischievous look ✗

cat score

# Regularization: Dropout

How can this possibly be a good idea?
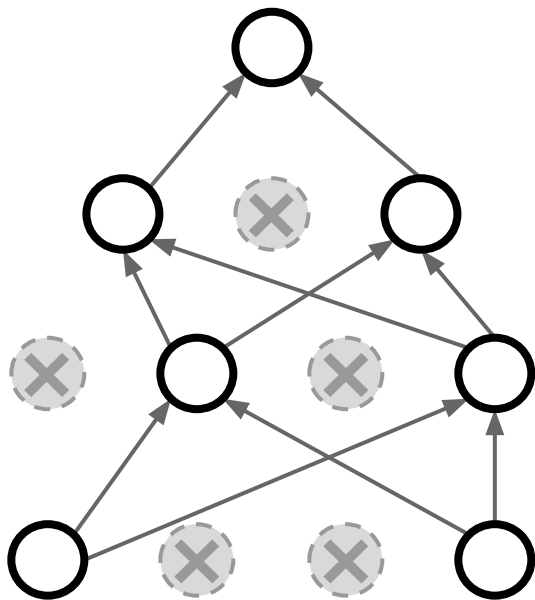


Another interpretation:

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!

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

Want to "average out" the randomness at test-time

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

But this integral seems hard …

# Dropout: Test time

Want to approximate
the integral

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

Consider a single neuron.
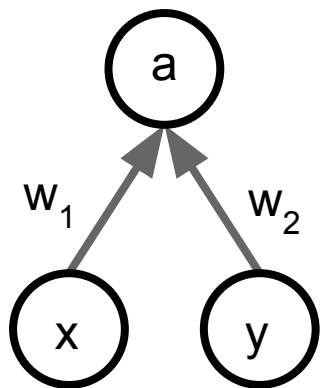
# Dropout: Test time

Want to approximate the integral

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

Consider a single neuron.

At test time we have: $E\big[a\big] = w_1 x + w_2 y$

# Dropout: Test time

Want to approximate
the integral

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

Consider a single neuron.



At test time we have: $E\big[a\big] = w_1 x + w_2 y$

During training we have:

$$E[a] = \tfrac{1}{4}(w_1 x + w_2 y) + \tfrac{1}{4}(w_1 x + w_2 0)$$

$$+ \tfrac{1}{4}(w_1 0 + w_2 0) + \tfrac{1}{4}(w_1 0 + w_2 y)$$

$$= \tfrac{1}{2}(w_1 x + w_2 y)$$
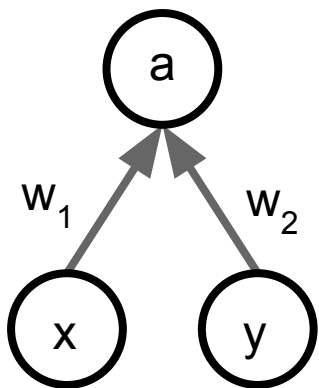
# Dropout: Test time

Want to approximate
the integral

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

Consider a single neuron.



At test time we have: $E[a] = w_1 x + w_2 y$
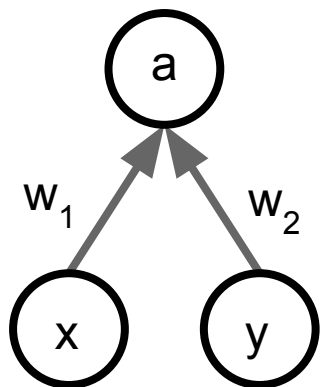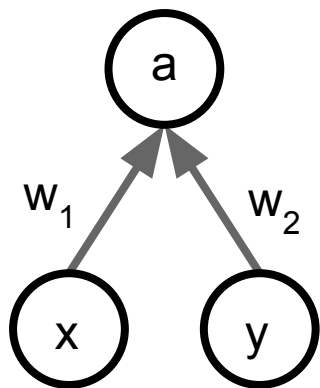
During training we have: $E[a] = \frac{1}{4}(w_1 x + w_2 y) + \frac{1}{4}(w_1 x + w_2 0)$

$$+ \frac{1}{4}(w_1 0 + w_2 0) + \frac{1}{4}(w_1 0 + w_2 y)$$

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

$$= \frac{1}{2}(w_1 x + w_2 y)$$

# 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:
<u>output at test time</u> = <u>expected output at training time</u>

# Dropout Summary

```python
""" 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"

```python
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!

# Regularization: A common pattern

**Training**: Add some kind
of randomness

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

**Testing:** Average out randomness
(sometimes approximate)

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

# Regularization: A common pattern

**Training**: Add some kind of randomness

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

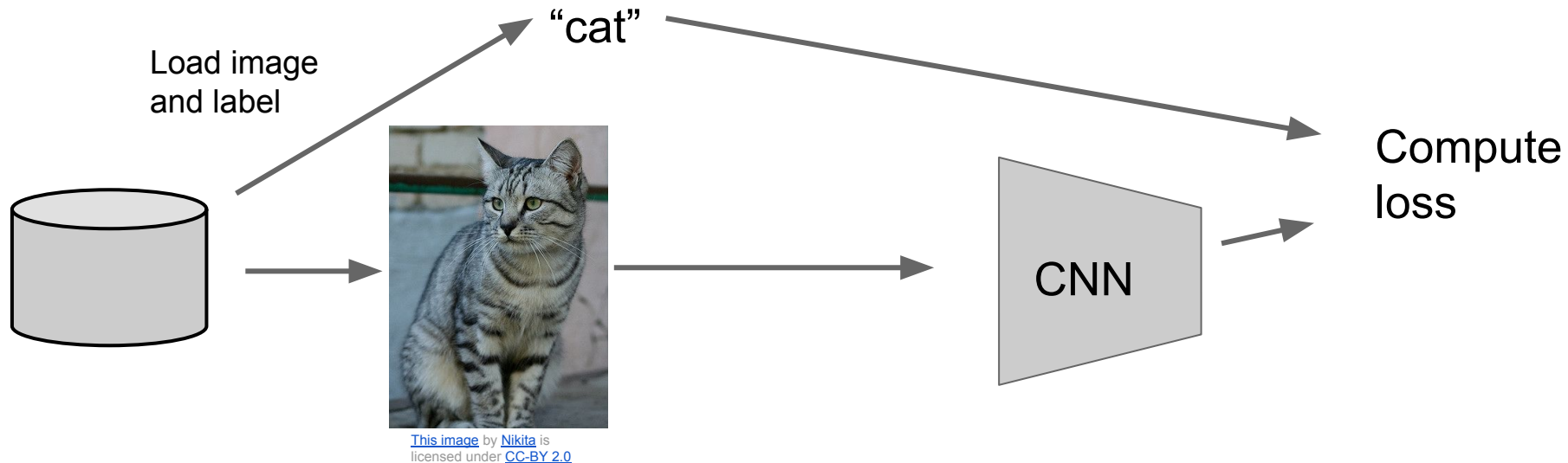**Testing:** Average out randomness (sometimes approximate)

$$y = f(x) = E_z\left[f(x, z)\right] = \int p(z) f(x, z) dz$$
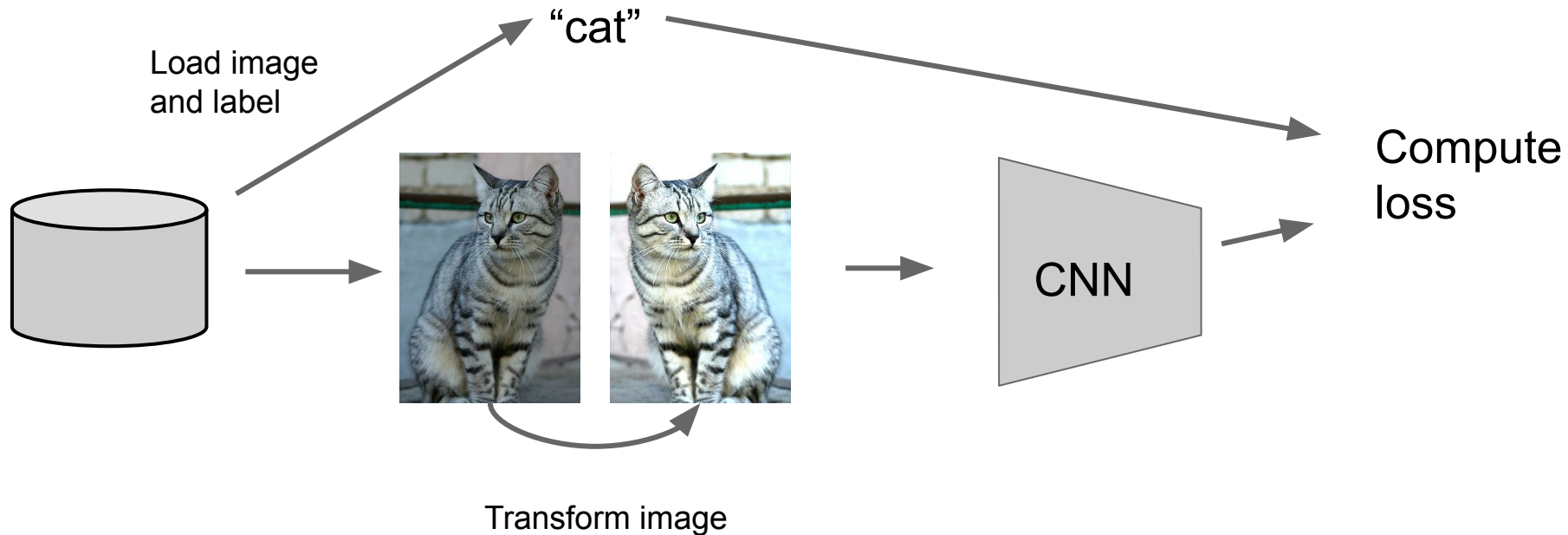
**Example**: Batch Normalization

**Training**: Normalize using stats from random minibatches

**Testing**: Use fixed stats to normalize

# Regularization: Data Augmentation



Load image and label

"cat"

Compute loss

CNN

This image by Nikita is licensed under CC-BY 2.0

# 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
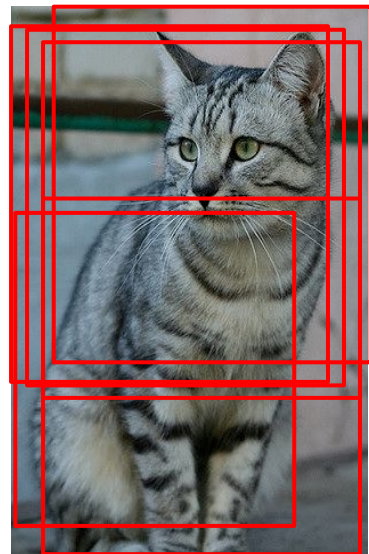3. Sample random 224 x 224 patch

# Data Augmentation
## Random crops and scales

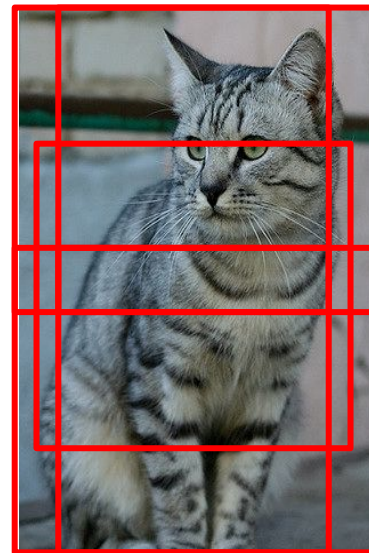**Training**: sample random crops / scales
ResNet:
1. Pick random L in range [256, 480]
2. Resize training image, short side = L
3. Sample random 224 x 224 patch

**Testing**: average a fixed set of crops
ResNet:
1. Resize image at 5 scales:  {224, 256, 384, 480, 640}
2. For each size, use 10 224 x 224 crops: 4 corners + center, + flips

# 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,
- lens distortions, …  (go crazy)

# Automatic Data Augmentation



Cubuk et al., "AutoAugment: Learning Augmentation Strategies from Data", CVPR 2019

# Regularization: A common pattern

**Training**: Add random noise
**Testing**: Marginalize over the noise

**Examples**:
Dropout
Batch Normalization
Data Augmentation

# Regularization: DropConnect

**Training**: Drop connections between neurons (set weights to 0)
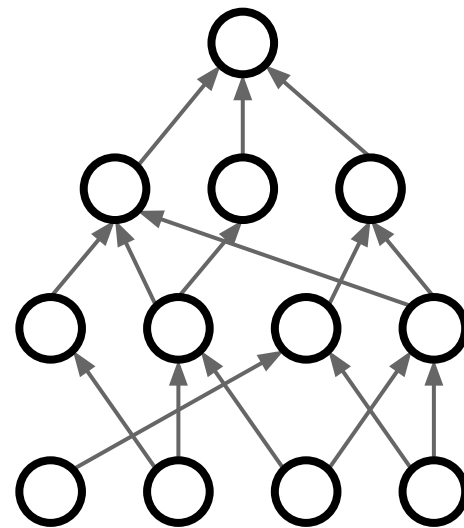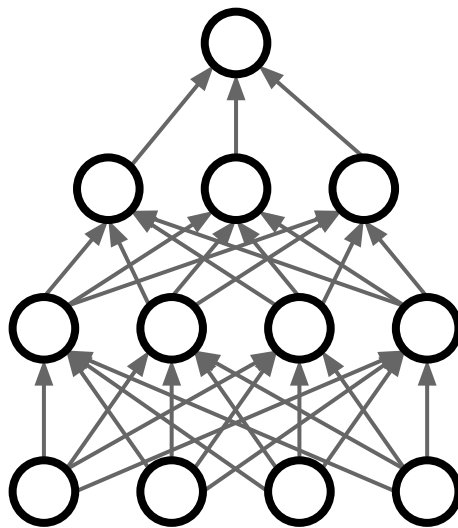**Testing**: Use all the connections

**Examples**:
Dropout
Batch Normalization
Data Augmentation
DropConnect



Wan et al, "Regularization of Neural Networks using DropConnect", ICML 2013

# Regularization: Fractional Pooling

**Training**: Use randomized pooling regions
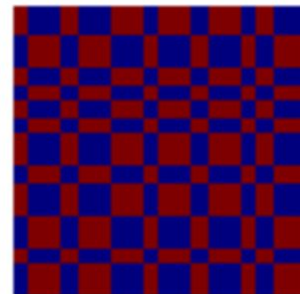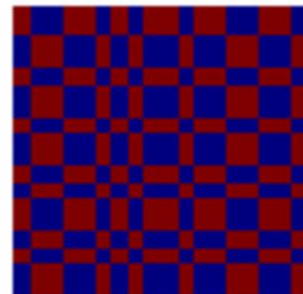**Testing**: Average predictions from several regions

**Examples**:
Dropout
Batch Normalization
Data Augmentation
DropConnect
Fractional Max Pooling



Graham, "Fractional Max Pooling", arXiv 2014
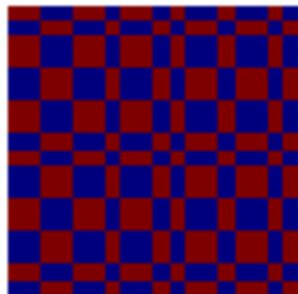
# Regularization: Stochastic Depth

**Training**: Skip some layers in the network

**Testing**: Use all the layer

**Examples**:

Dropout

Batch Normalization

Data Augmentation

DropConnect

Fractional Max Pooling

Stochastic Depth (will become more
clear in next week's lecture)

Huang et al, "Deep Networks with Stochastic Depth", ECCV 2016

# Regularization: Cutout

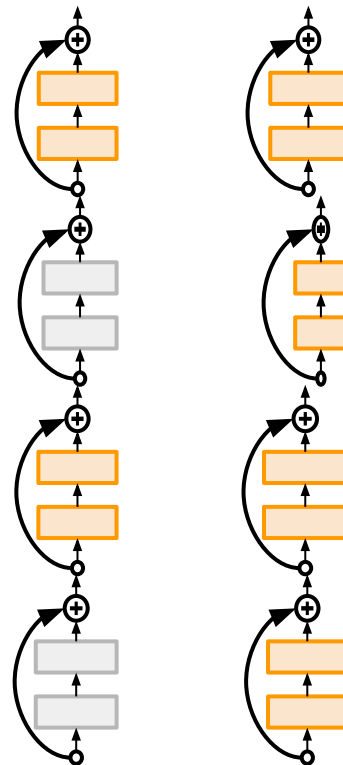**Training**: Set random image regions to zero

**Testing**: Use full image

**Examples**:

Dropout

Batch Normalization

Data Augmentation

DropConnect

Fractional Max Pooling

Stochastic Depth

Cutout / Random Crop

DeVries and Taylor, "Improved Regularization of Convolutional Neural Networks with Cutout", arXiv 2017
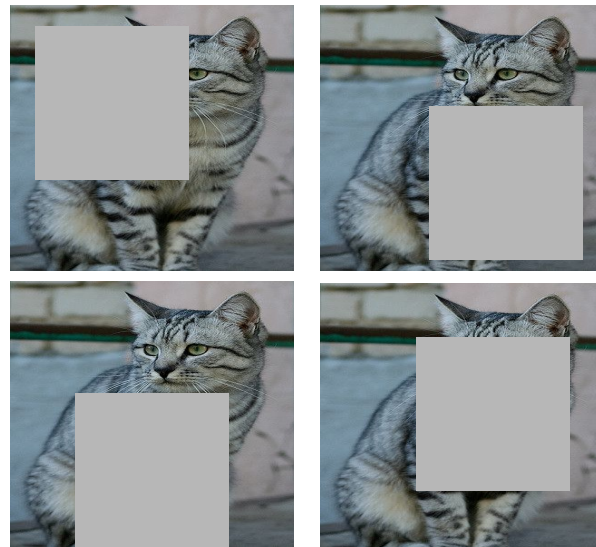


Works very well for small datasets like CIFAR, less common for large datasets like ImageNet

# Regularization: Mixup

**Training**: Train on random blends of images

**Testing**: Use original images

**Examples**:

Dropout
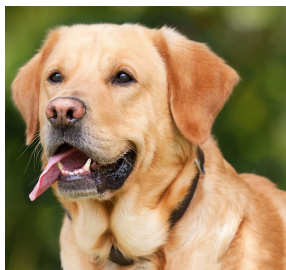Batch Normalization
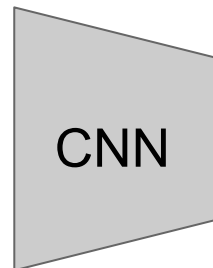Data Augmentation
DropConnect
Fractional Max Pooling
Stochastic Depth
Cutout / Random Crop
Mixup



Randomly blend the pixels of pairs of training images, e.g. 40% cat, 60% dog

CNN

Target label:
cat: 0.4
dog: 0.6

Zhang et al, "*mixup*: Beyond Empirical Risk Minimization", ICLR 2018

# Regularization: CutMix

**Training**: Train on random blends of images
**Testing**: Use original images

**Examples**:
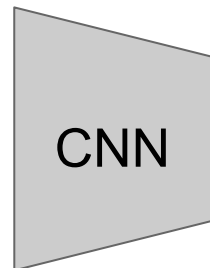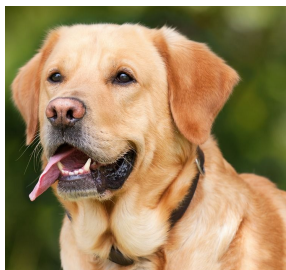Dropout
Batch Normalization
Data Augmentation
DropConnect
Fractional Max Pooling
Stochastic Depth
Cutout / Random Crop
Mixup



CNN

Target label:
cat: 0.4
dog: 0.6

Replace random crops of one image with another: e.g. 60% of pixels from cat, 40% from dog

Yun et al, "CutMix: Regularization Strategies to Train Strong Classifiers with Localizable Features", ICCV 2019

# Regularization: Label Smoothing

**Training**: Change target distribution
**Testing**: Take argmax over predictions

**Examples**:

Dropout
Batch Normalization
Data Augmentation
DropConnect
Fractional Max Pooling
Stochastic Depth
Cutout / Random Crop
Mixup
Label Smoothing



Standard Training:
Cat: 100%
Dog: 0%
Fish: 0%

Label Smoothing:
Cat: 90%
Dog: 5%
Fish: 5%

Set target distribution to be $1 - \frac{K-1}{K}\epsilon$ on the correct category and $\epsilon/K$ on all other categories, with $K$ categories and $\epsilon \in 0,1$. Loss is cross-entropy between predicted and target distribution.

Szegedy et al, "Rethinking the Inception Architecture for Computer Vision", CVPR 2015

# Regularization - In practice

**Training**: Add random noise
**Testing**: Marginalize over the noise

**Examples**:

Dropout
Batch Normalization
Data Augmentation
DropConnect
Fractional Max Pooling
Stochastic Depth
Cutout / Random Crop
Mixup

- Use **dropout** for large fully-connected layers
- Using **batchnorm** is always a good idea
- Try **Cutout, MixUp, CutMix, Stochastic Depth, Label Smoothing** to squeeze out a bit of extra performance

# Choosing Hyperparameters

# Choosing Hyperparameters: Grid Search

Choose several values for each hyperparameter
(Often space choices log-linearly)

**Example**:
Weight decay: [$1\times10^{-4}$, $1\times10^{-3}$, $1\times10^{-2}$, $1\times10^{-1}$]
Learning rate: [$1\times10^{-4}$, $1\times10^{-3}$, $1\times10^{-2}$, $1\times10^{-1}$]

Evaluate all possible choices on this hyperparameter grid

# Choosing Hyperparameters: Random search

Choose several values for each hyperparameter
(Often space choices log-linearly)

**Example:**
Weight decay: **log-uniform** on $[1 \times 10^{-4}, 1 \times 10^{-1}]$
Learning rate: **log-uniform** on $[1 \times 10^{-4}, 1 \times 10^{-1}]$

Run many different trials

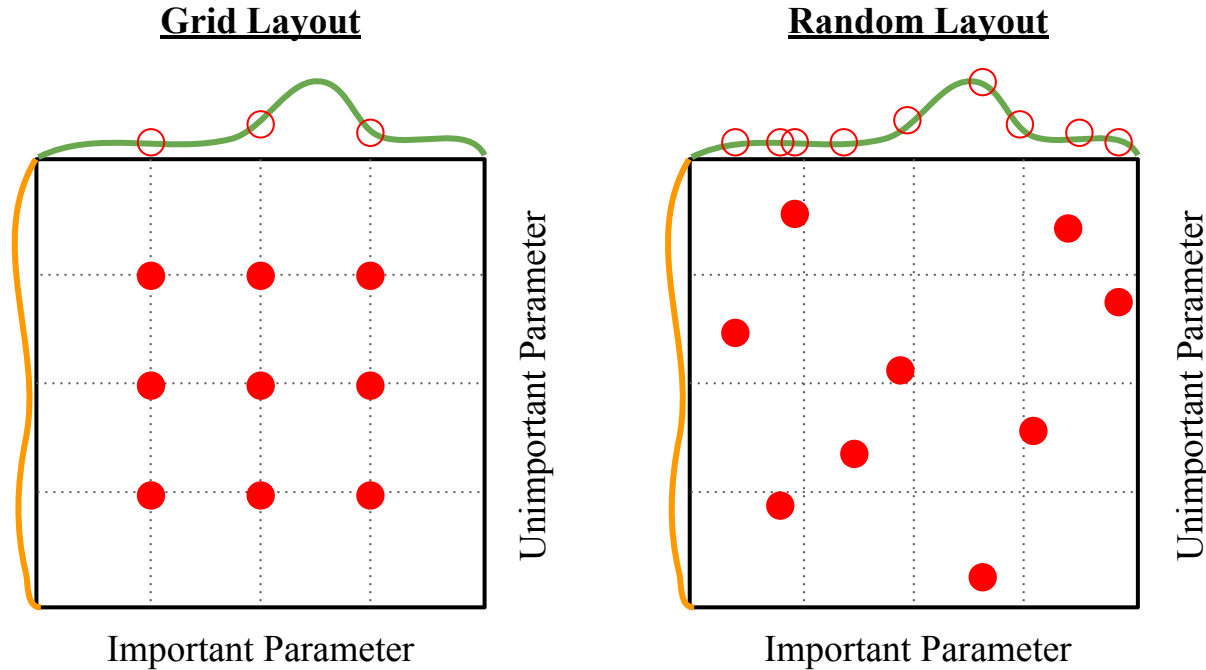# Random Search vs. Grid Search

Illustration of Bergstra et al., 2012 by Shayne Longpre, copyright CS231n 2017

# Choosing Hyperparameters

(without tons of GPUs)

# Choosing Hyperparameters

**Step 1**: <span style="color:red">Check initial loss</span>

Turn off weight decay, sanity check loss at initialization
e.g. log(C) for softmax with C classes

# Choosing Hyperparameters

**Step 1**: Check initial loss
**Step 2**: <span style="color:red">Overfit a small sample</span>

Try to train to 100% training accuracy on a small sample of training data (~5-10 minibatches); fiddle with architecture, learning rate, weight initialization

Loss not going down? LR too low, bad initialization
Loss explodes to Inf or NaN? LR too high, bad initialization

# Choosing Hyperparameters

**Step 1**: Check initial loss
**Step 2**: Overfit a small sample
**Step 3**: <span style="color:red">Find LR that makes loss go down</span>

Use the architecture from the previous step, use all training data, turn on small weight decay, find a learning rate that makes the loss drop significantly within ~100 iterations

Good learning rates to try: 1e-1, 1e-2, 1e-3, 1e-4

# Choosing Hyperparameters

**Step 1**: Check initial loss
**Step 2**: Overfit a small sample
**Step 3**: Find LR that makes loss go down
**Step 4**: <span style="color:red">Coarse grid, train for ~1-5 epochs</span>

Choose a few values of learning rate and weight decay around what worked from Step 3, train a few models for ~1-5 epochs.

Good weight decay to try: 1e-4, 1e-5, 0

# Choosing Hyperparameters

**Step 1**: Check initial loss
**Step 2**: Overfit a small sample
**Step 3**: Find LR that makes loss go down
**Step 4**: Coarse grid, train for ~1-5 epochs
**Step 5**: <span style="color:red">Refine grid, train longer</span>

Pick best models from Step 4, train them for longer (~10-20 epochs) without learning rate decay

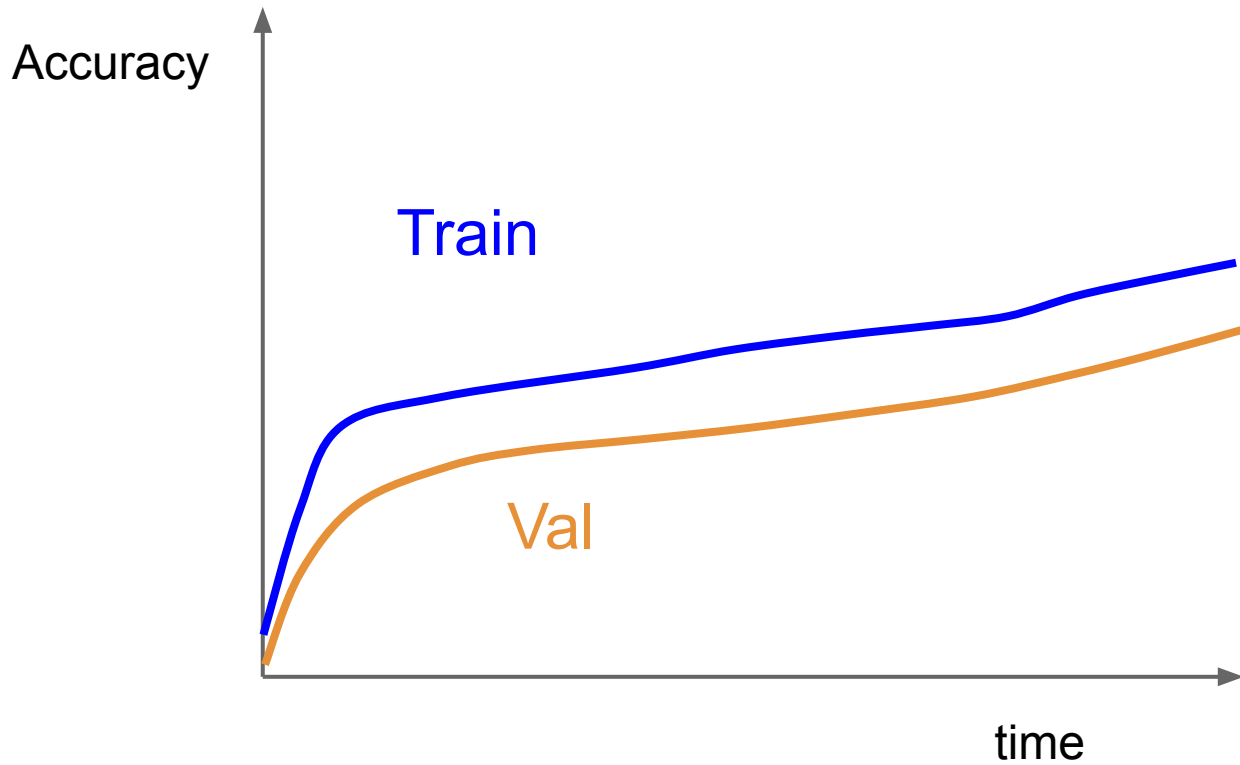# Choosing Hyperparameters

**Step 1**: Check initial loss
**Step 2**: Overfit a small sample
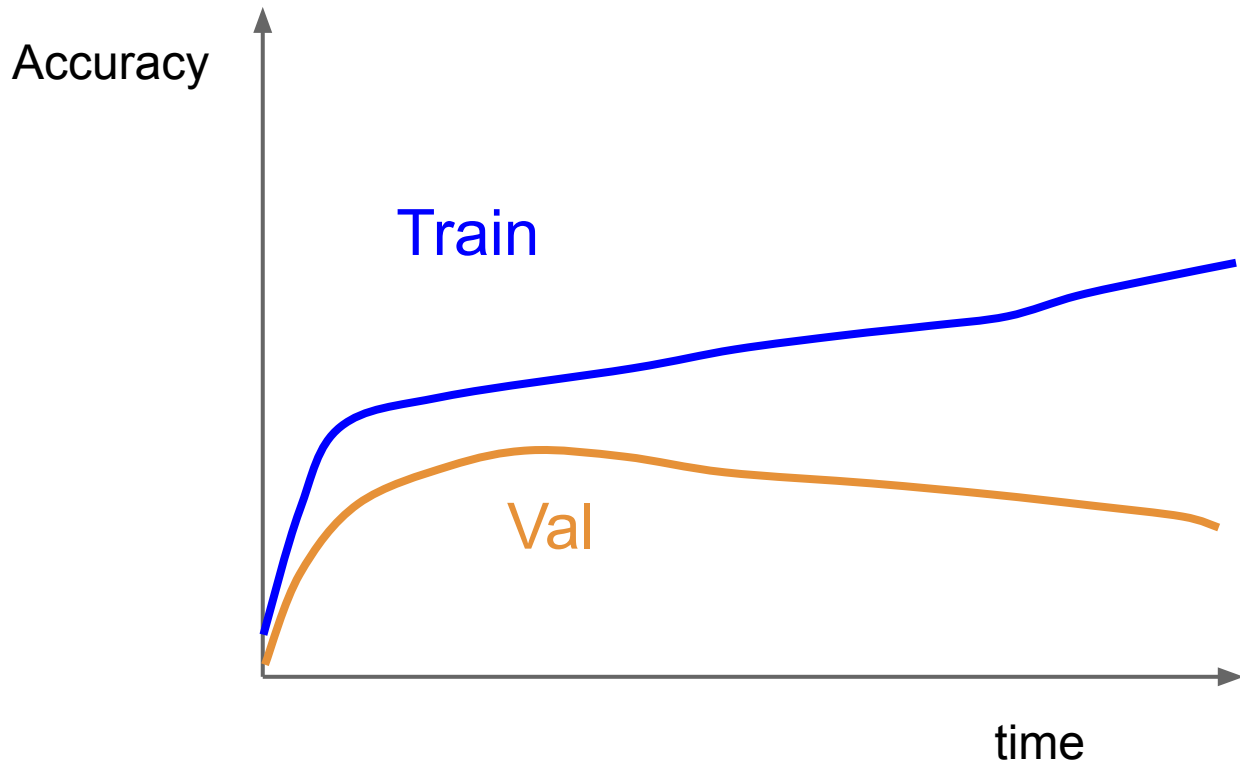**Step 3**: Find LR that makes loss go down
**Step 4**: Coarse grid, train for ~1-5 epochs
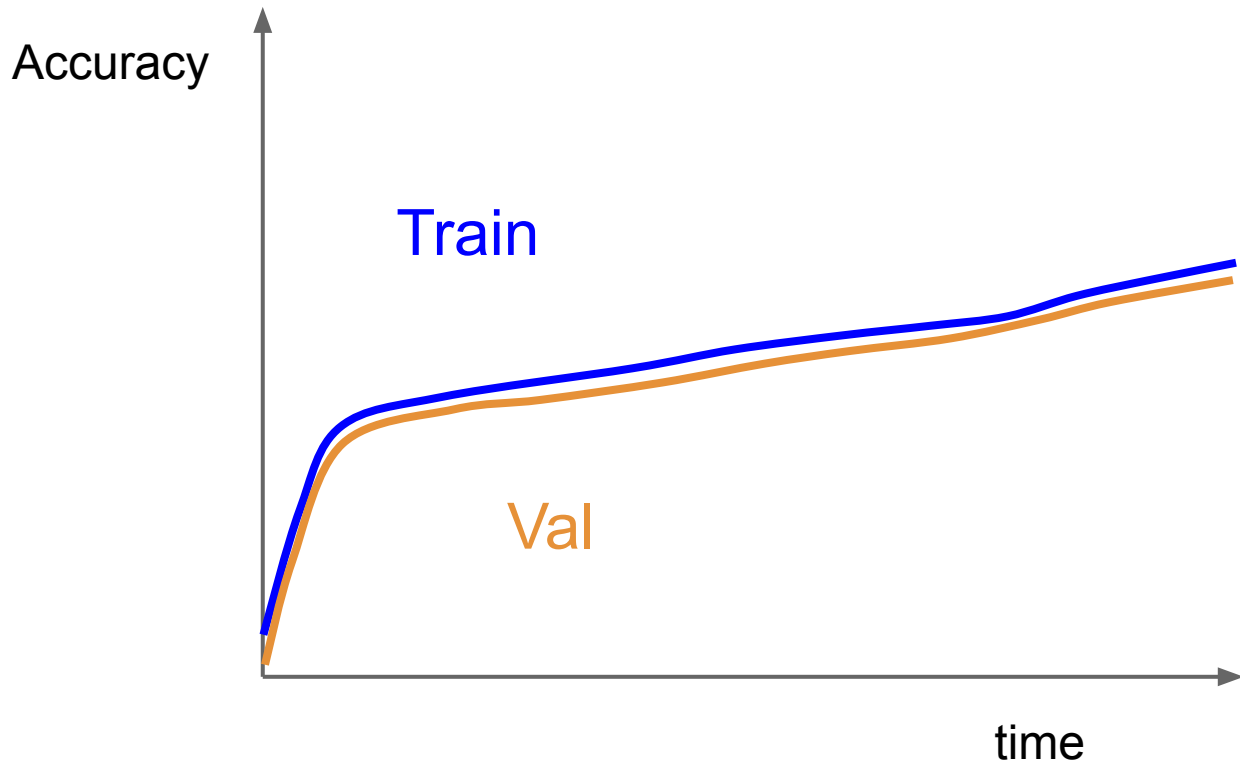**Step 5**: Refine grid, train longer
**Step 6**: Look at loss and accuracy curves

Accuracy

Train

Val

time

Q1. You see this. What should you do?
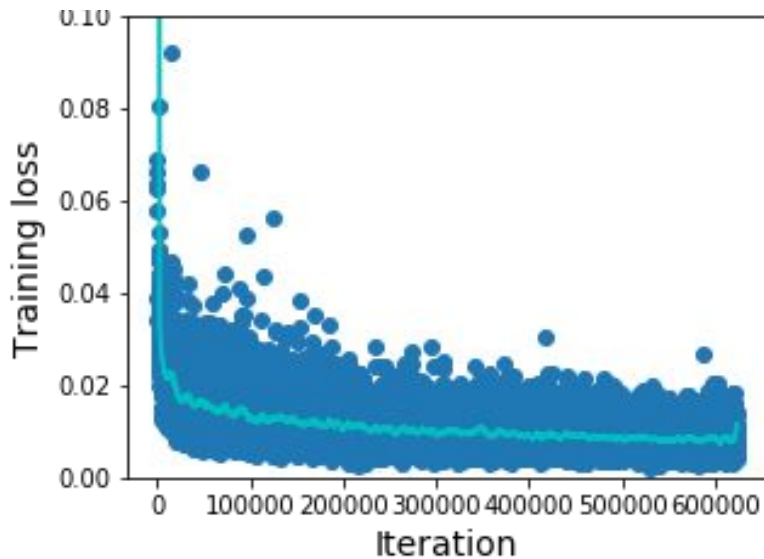
Accuracy

Train
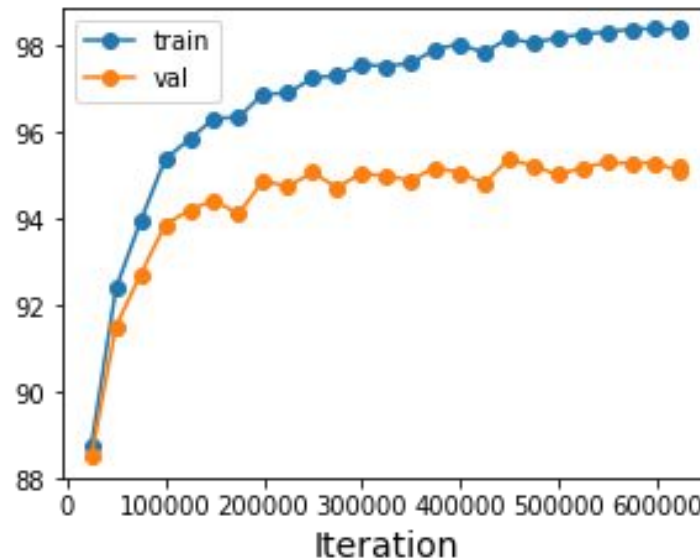
Val

time

Q2. You see this. What should you do?

Q3. You see this. What should you do?
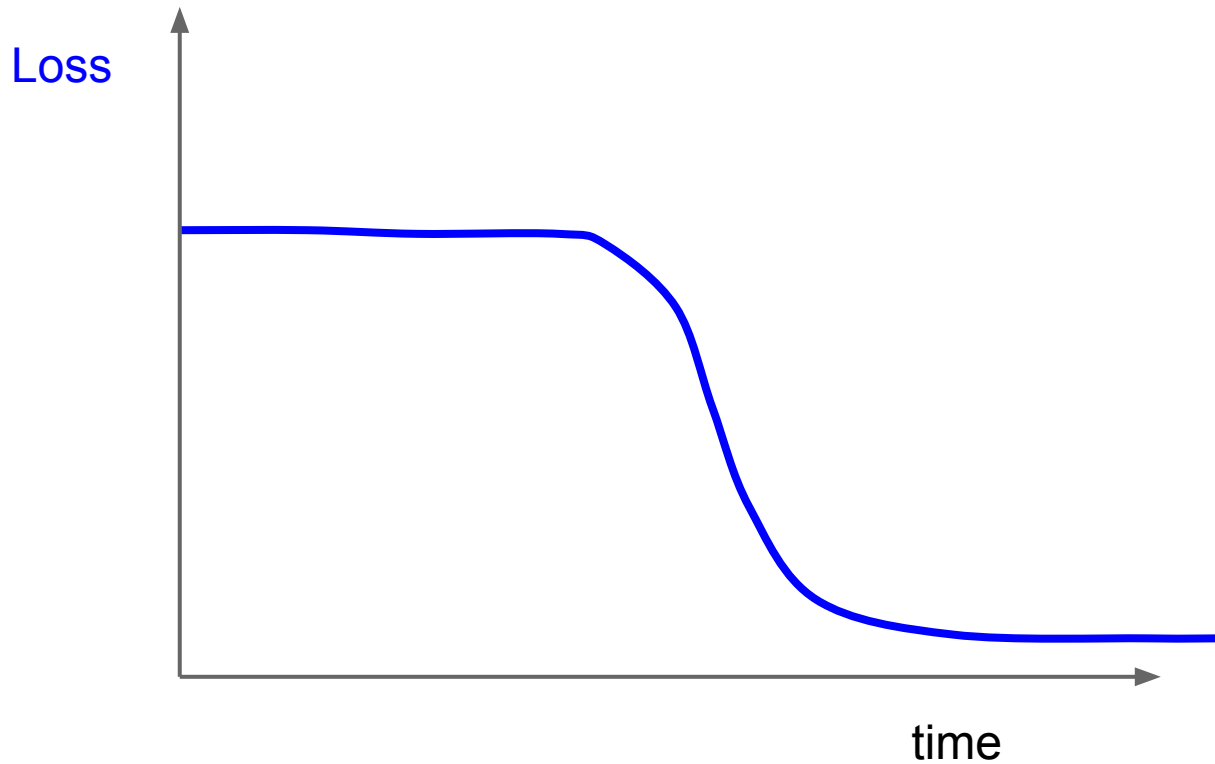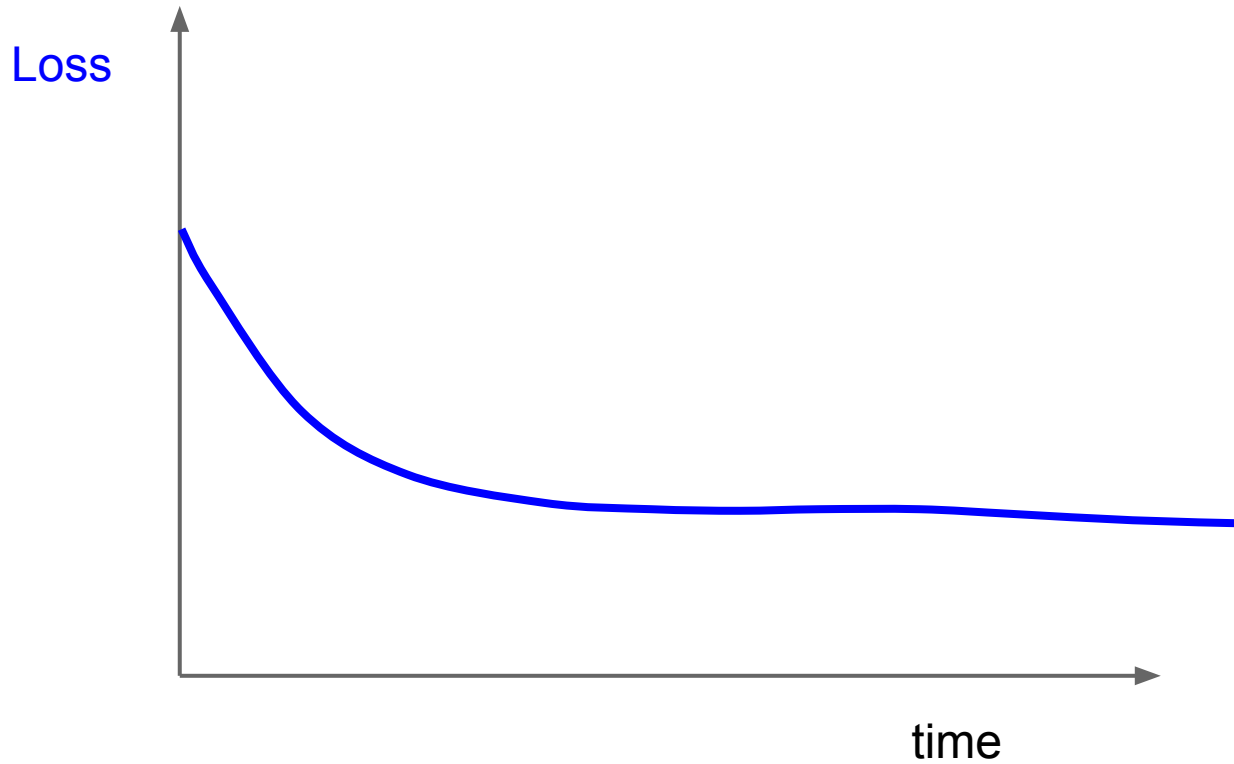
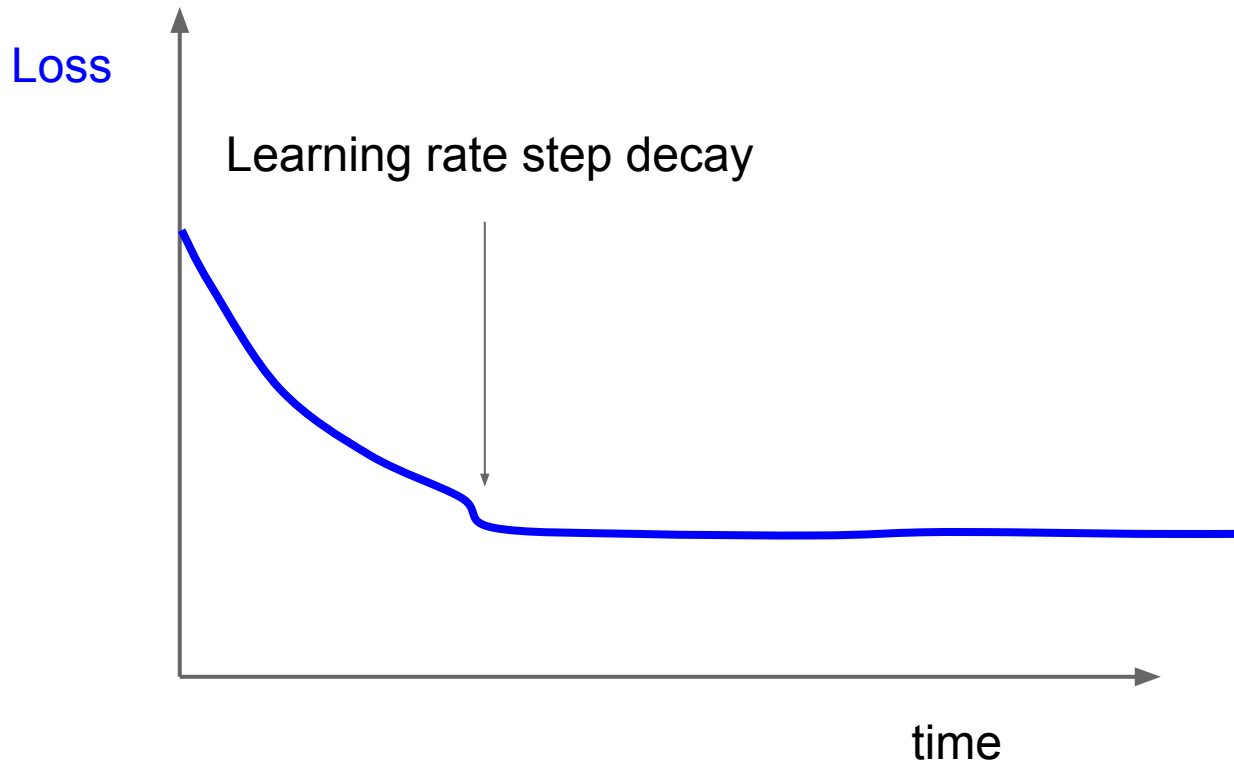# Look at learning curves!



Training Loss

Train / Val Accuracy

Losses may be noisy, use a
scatter plot and also plot moving
average to see trends better

Loss

time

Q4. You see this. What should you do?

Loss

time

Q5. You see this. What should you do?

Loss
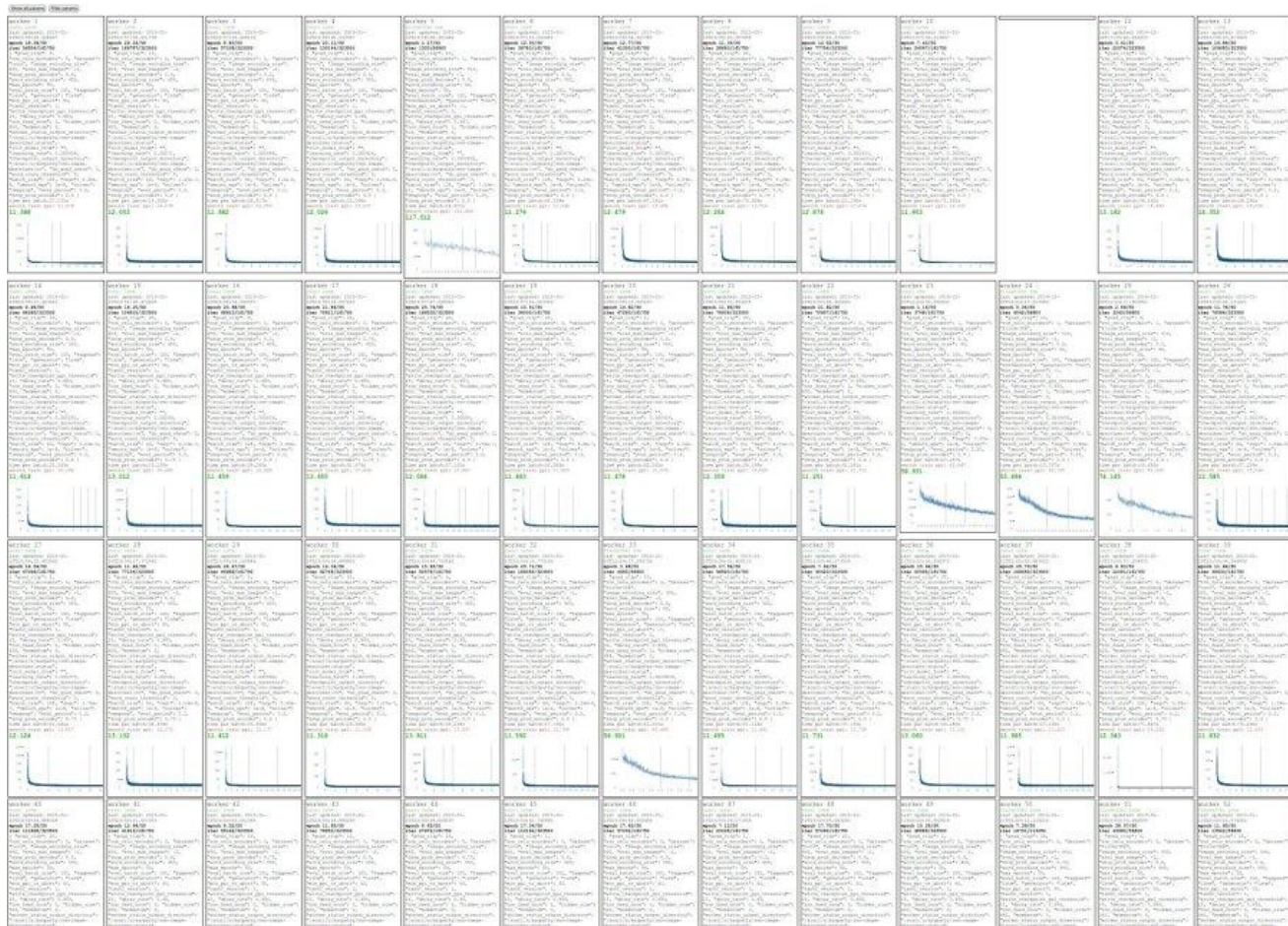
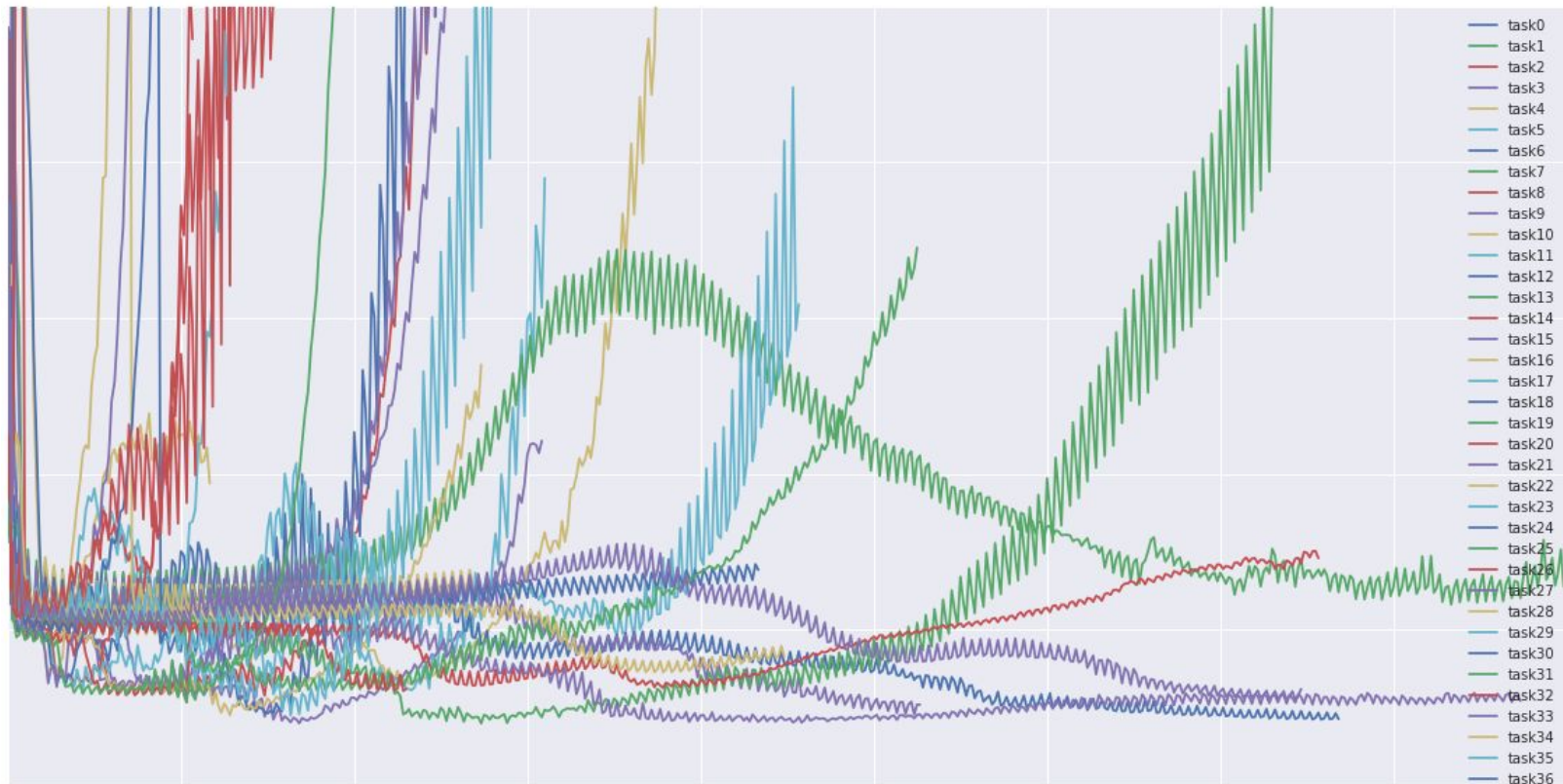Learning rate step decay

time

Q5. You see this. What should you do?

Cross-validation

We develop
"command centers"
to visualize all our
models training with
different
hyperparameters

check out weights
and biases

You can plot all your loss curves for different hyperparameters on a single plot

# Choosing Hyperparameters

**Step 1**: Check initial loss
**Step 2**: Overfit a small sample
**Step 3**: Find LR that makes loss go down
**Step 4**: Coarse grid, train for ~1-5 epochs
**Step 5**: Refine grid, train longer
**Step 6**: Look at loss and accuracy curves
**Step 7**: <span style="color:red">GOTO step 5</span>

# Hyperparameters to play with:

- learning rate,
- Its decay schedule, update type
- regularization (L2/Dropout strength)

# Summary

- Improve your training error:
    - Optimizers
    - Learning rate schedules
- Improve your test error:

    - Regularization
    - Choosing Hyperparameters

Next time: Visualizing and understanding neural networks

# Model Ensembles: Tips and Tricks

Instead of training independent models, use multiple snapshots of a single model during training!
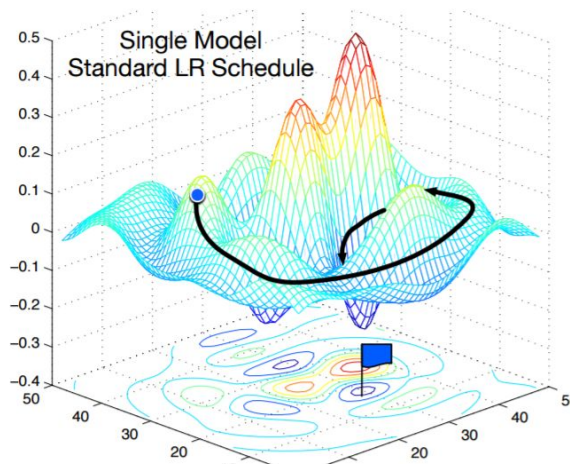


Loshchilov and Hutter, "SGDR: Stochastic gradient descent with restarts", arXiv 2016
Huang et al, "Snapshot ensembles: train 1, get M for free", ICLR 2017
Figures copyright Yixuan Li and Geoff Pleiss, 2017. Reproduced with permission.

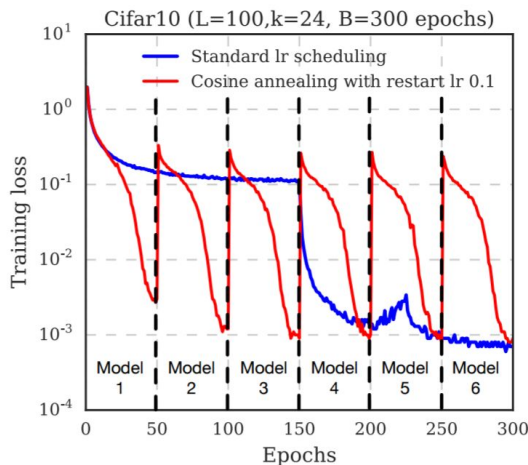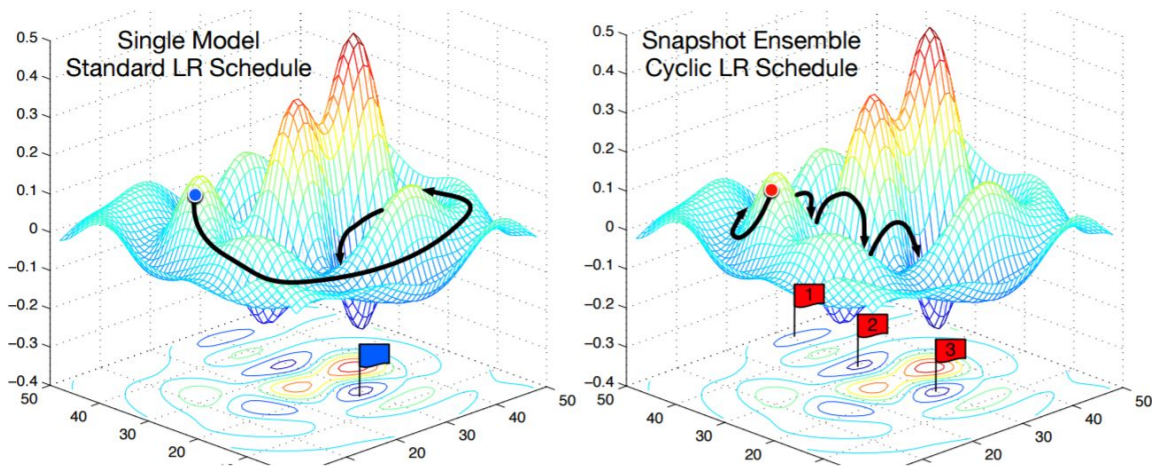# Model Ensembles: Tips and Tricks

Instead of training independent models, use multiple snapshots of a single model during training!



Loshchilov and Hutter, "SGDR: Stochastic gradient descent with restarts", arXiv 2016
Huang et al, "Snapshot ensembles: train 1, get M for free", ICLR 2017
Figures copyright Yixuan Li and Geoff Pleiss, 2017. Reproduced with permission.

Cyclic learning rate schedules can make this work even better!

# Model Ensembles: Tips and Tricks

Instead of using actual parameter vector, keep a moving average of the parameter vector and use that at test time (Polyak averaging)

```
while True:
    data_batch = dataset.sample_data_batch()
    loss = network.forward(data_batch)
    dx = network.backward()
    x += - learning_rate * dx
    x_test = 0.995*x_test + 0.005*x # use for test set
```

Polyak and Juditsky, "Acceleration of stochastic approximation by averaging", SIAM Journal on Control and Optimization, 1992.

# Track the ratio of weight updates / weight magnitudes:

```python
# assume parameter vector W and its gradient vector dW
param_scale = np.linalg.norm(W.ravel())
update = -learning_rate*dW # simple SGD update
update_scale = np.linalg.norm(update.ravel())
W += update # the actual update
print update_scale / param_scale # want ~1e-3
```

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