

# Lecture 7:

# Training Neural Networks,

## Part 2

# Administrative

- Assignment 1 is being graded, stay tuned
- Project proposals due today by 11:59pm
- Assignment 2 is out, due Thursday May 4 at 11:59pm

# Administrative: Google Cloud

- STOP YOUR INSTANCES when not in use!

Google Cloud Platform Assignment1

Compute Engine

VM instances

CREATE INSTANCE IMPORT VM

Filter by label or name Columns Labels

Name	Zone	Recommendation	Internal IP	External IP	Connect
<input checked="" type="checkbox"/> gpu-instance	us-west1-b		10.138.0.3	35.185.201.171	SSH
<input type="checkbox"/> instance-2	us-west1-b		10.138.0.2	104.196.245.79	SSH

- Start
- Stop
- Reset
- Delete
- New instance group
- View logs

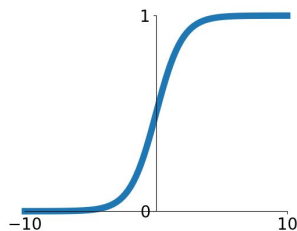
# Administrative: Google Cloud

- STOP YOUR INSTANCES when not in use!
- Keep track of your spending!
- GPU instances are much more expensive than CPU instances - only use GPU instance when you need it (e.g. for A2 only on TensorFlow / PyTorch notebooks)

# Last time: Activation Functions

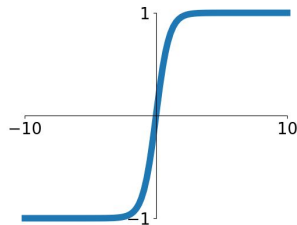
## Sigmoid

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



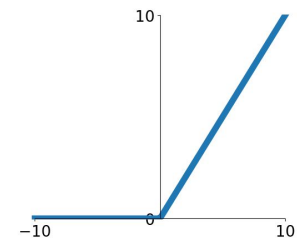
## tanh

$$\tanh(x)$$



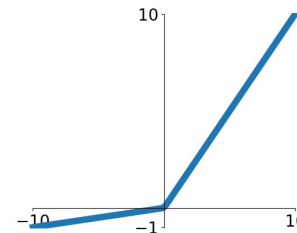
## ReLU

$$\max(0, x)$$



## Leaky ReLU

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

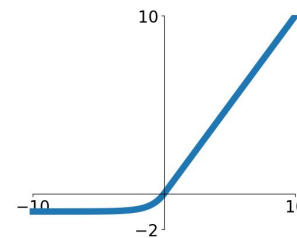


## Maxout

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

## ELU

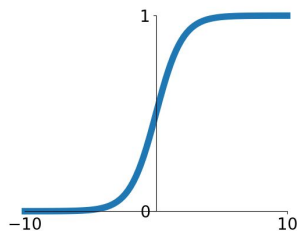
$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



# Last time: Activation Functions

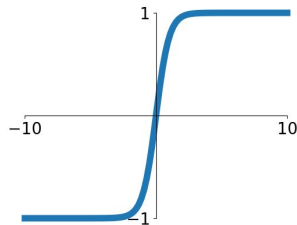
## Sigmoid

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



## tanh

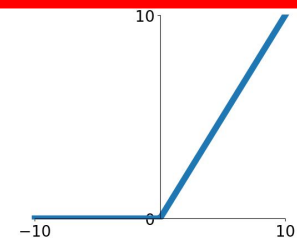
$$\tanh(x)$$



## ReLU

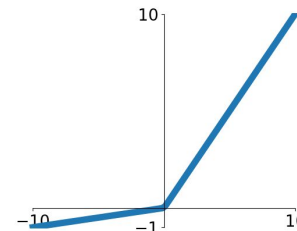
$$\max(0, x)$$

Good default choice



## Leaky ReLU

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

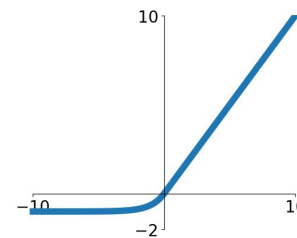


## Maxout

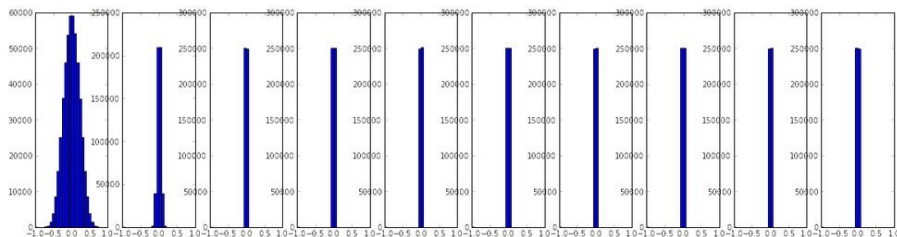
$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

## ELU

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

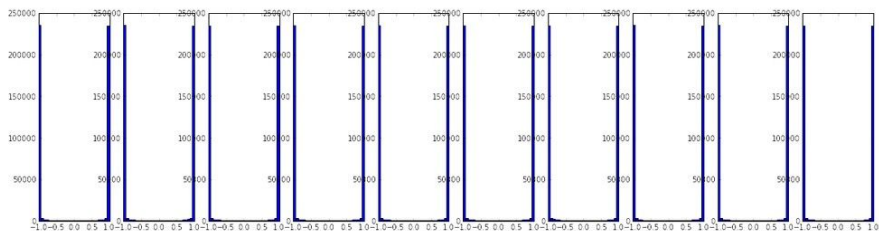


# Last time: Weight Initialization



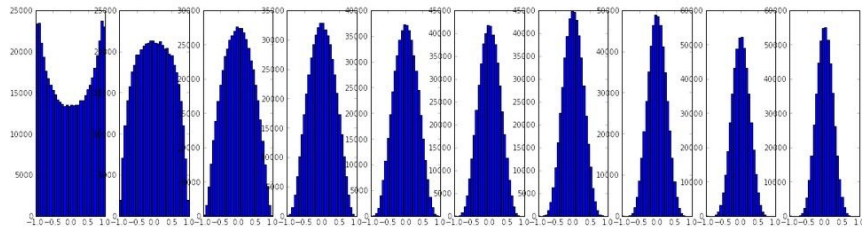
## Initialization too small:

Activations go to zero, gradients also zero,  
No learning



## Initialization too big:

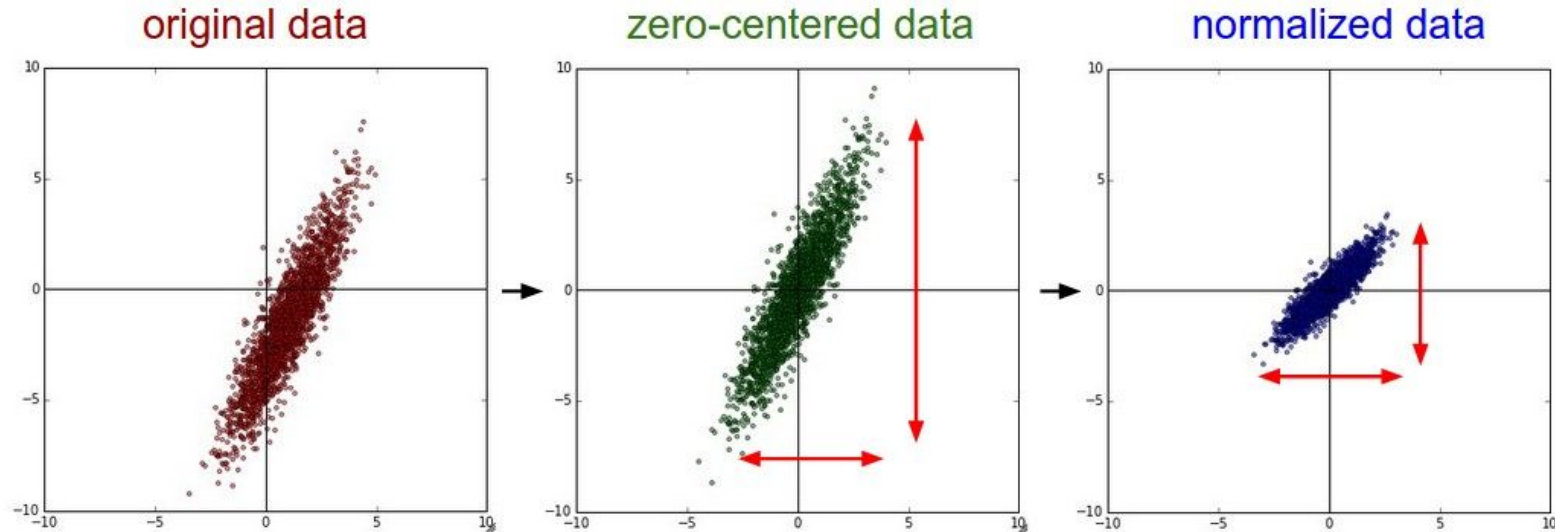
Activations saturate (for tanh),  
Gradients zero, no learning



## Initialization just right:

Nice distribution of activations at all layers,  
Learning proceeds nicely

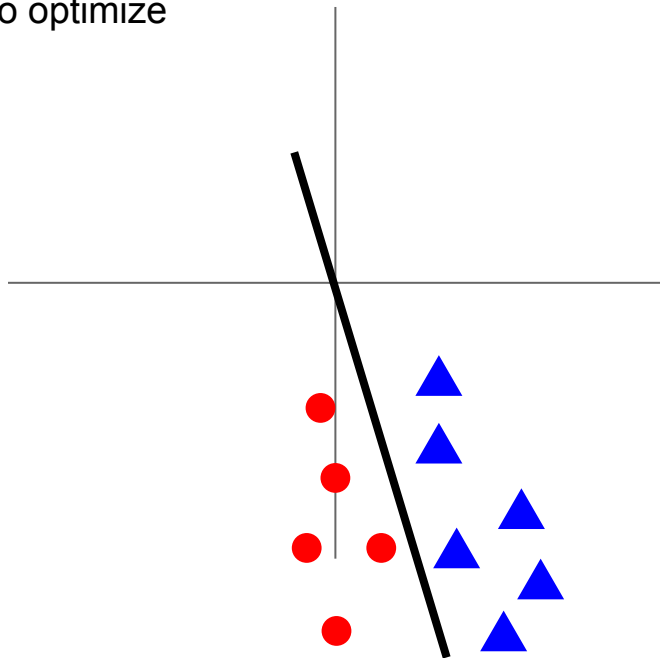
# Last time: Data Preprocessing



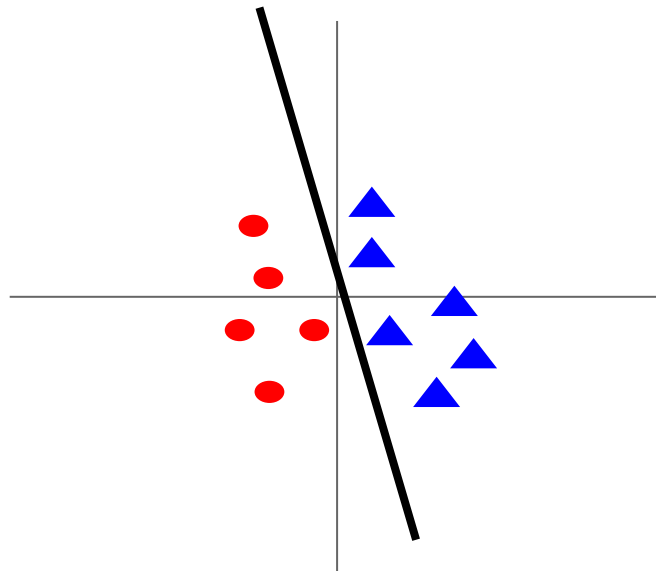


# Last time: Data Preprocessing

**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



# Last time: Batch Normalization

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

**Learnable params:**

$$\gamma, \beta : D$$

**Intermediates:**  $\mu, \sigma : D$   
 $\hat{x} : N \times D$

**Output:**  $y : N \times D$

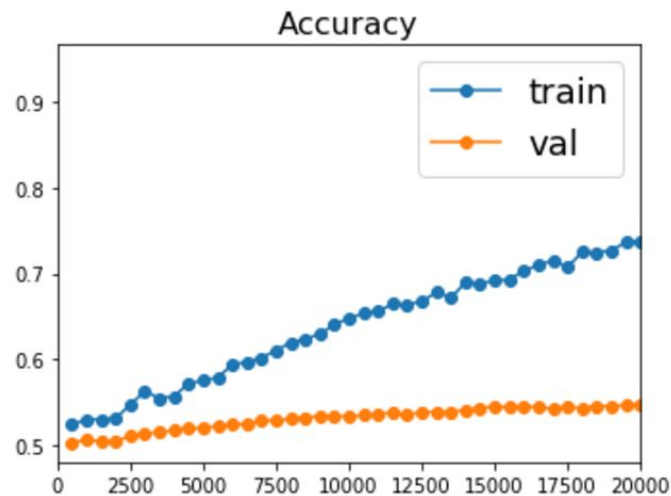
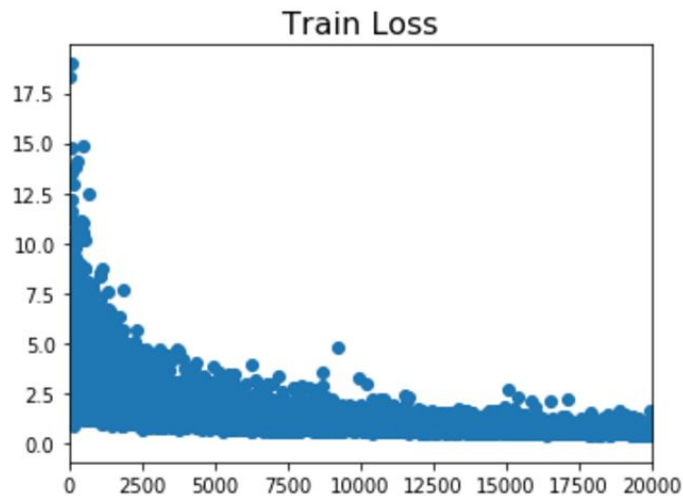
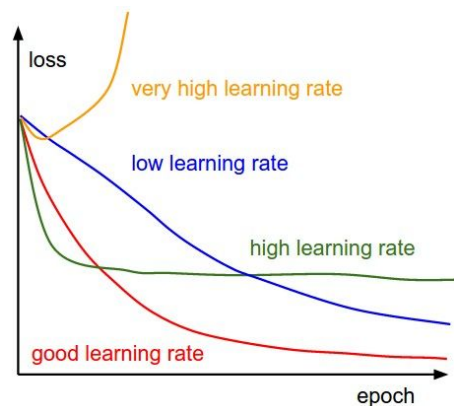
$$\mu_j = \frac{1}{N} \sum_{i=1}^N x_{i,j}$$

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

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

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

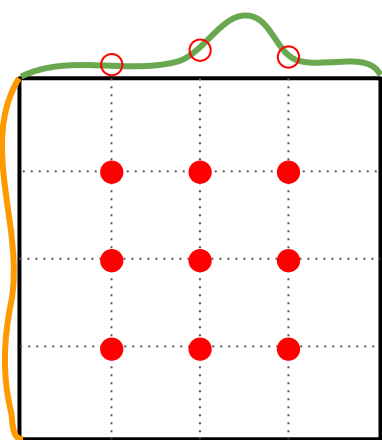
# Last time: Babysitting Learning



# Last time: Hyperparameter Search

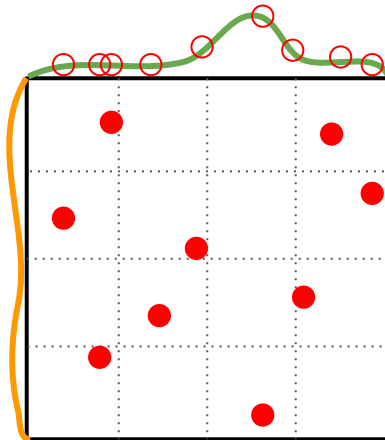
## Coarse to fine search

Grid Layout



Important  
Parameter

Random Layout



Important  
Parameter

```
val_acc: 0.412000, lr: 1.405206e-04, reg: 4.793564e-01, (1 / 100)
val_acc: 0.214000, lr: 7.231888e-06, reg: 2.321281e-04, (2 / 100)
val_acc: 0.208000, lr: 2.119571e-06, reg: 8.011857e+01, (3 / 100)
val_acc: 0.196000, lr: 1.551131e-05, reg: 4.374936e-05, (4 / 100)
val_acc: 0.079000, lr: 1.753300e-05, reg: 1.200424e+03, (5 / 100)
val_acc: 0.223000, lr: 4.215128e-05, reg: 4.196174e+01, (6 / 100)
val_acc: 0.441000, lr: 1.750259e-04, reg: 2.110807e-04, (7 / 100)
val_acc: 0.241000, lr: 6.749231e-05, reg: 4.226413e+01, (8 / 100)
val_acc: 0.482000, lr: 4.296863e-04, reg: 6.642555e-01, (9 / 100)
val_acc: 0.079000, lr: 5.401602e-06, reg: 1.599828e+04, (10 / 100)
val_acc: 0.154000, lr: 1.618508e-06, reg: 4.925252e-01, (11 / 100)
```

```
val_acc: 0.527000, lr: 5.340517e-04, reg: 4.097824e-01, (0 / 100)
val_acc: 0.492000, lr: 2.279484e-04, reg: 9.991345e-04, (1 / 100)
val_acc: 0.512000, lr: 8.680827e-04, reg: 1.349727e-02, (2 / 100)
val_acc: 0.461000, lr: 1.028377e-04, reg: 1.220193e-02, (3 / 100)
val_acc: 0.460000, lr: 1.113730e-04, reg: 5.244309e-02, (4 / 100)
val_acc: 0.498000, lr: 9.477776e-04, reg: 2.001293e-03, (5 / 100)
val_acc: 0.469000, lr: 1.484369e-04, reg: 4.328313e-01, (6 / 100)
val_acc: 0.522000, lr: 5.586261e-04, reg: 2.312685e-04, (7 / 100)
val_acc: 0.530000, lr: 5.808183e-04, reg: 8.259964e-02, (8 / 100)
val_acc: 0.489000, lr: 1.979168e-04, reg: 1.010889e-04, (9 / 100)
val_acc: 0.490000, lr: 2.036031e-04, reg: 2.406271e-03, (10 / 100)
val_acc: 0.475000, lr: 2.021162e-04, reg: 2.287807e-01, (11 / 100)
val_acc: 0.460000, lr: 1.135527e-04, reg: 3.905040e-02, (12 / 100)
val_acc: 0.515000, lr: 6.947668e-04, reg: 1.562808e-02, (13 / 100)
val_acc: 0.531000, lr: 9.471549e-04, reg: 1.433895e-03, (14 / 100)
val_acc: 0.509000, lr: 3.140888e-04, reg: 2.857518e-01, (15 / 100)
val_acc: 0.514000, lr: 6.438349e-04, reg: 3.033781e-01, (16 / 100)
val_acc: 0.502000, lr: 3.921784e-04, reg: 2.707126e-04, (17 / 100)
val_acc: 0.509000, lr: 9.752279e-04, reg: 2.850865e-03, (18 / 100)
val_acc: 0.500000, lr: 2.412048e-04, reg: 4.997821e-04, (19 / 100)
val_acc: 0.466000, lr: 1.319314e-04, reg: 1.189915e-02, (20 / 100)
val_acc: 0.516000, lr: 8.039527e-04, reg: 1.528291e-02, (21 / 100)
```

# Today

- Fancier optimization
- Regularization
- Transfer Learning

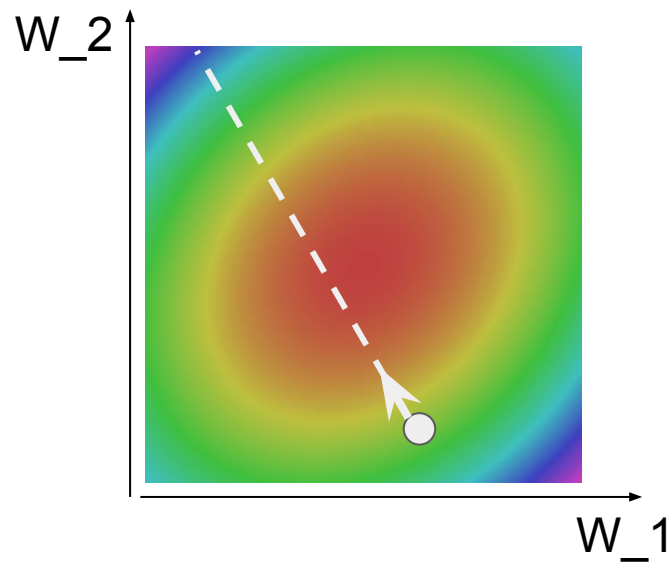
# Optimization

```
# Vanilla Gradient Descent
```

```
while True:
```

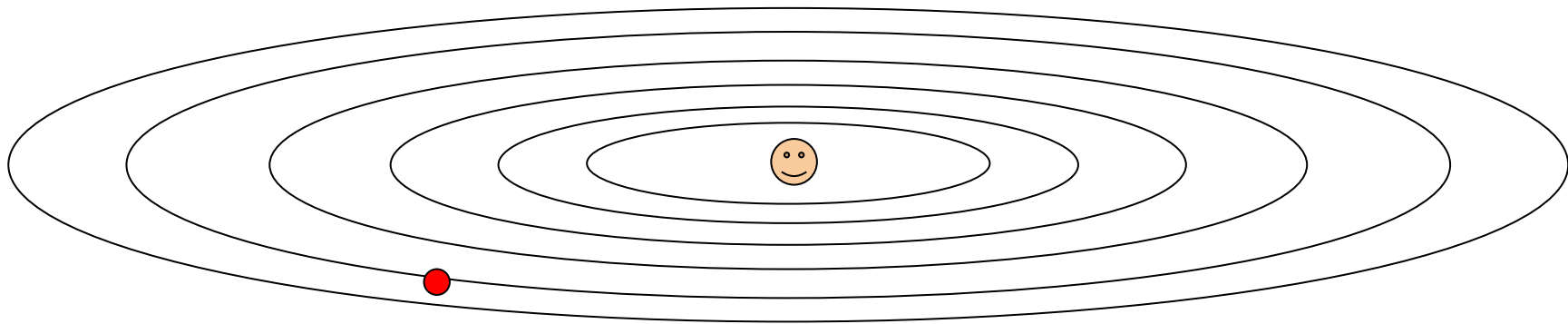
```
    weights_grad = evaluate_gradient(loss_fun, data, weights)
```

```
    weights += - step_size * weights_grad # perform parameter update
```



# Optimization: Problems with SGD

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



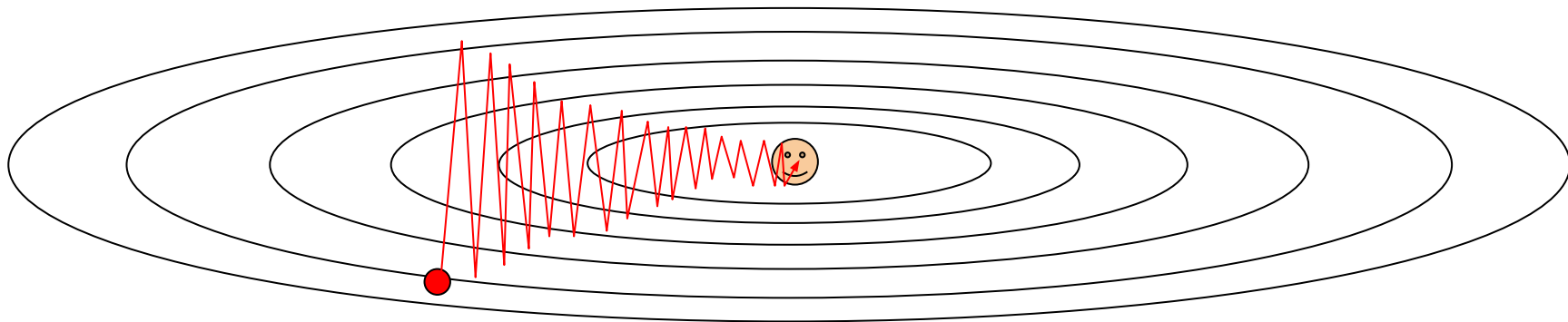
Loss function has high **condition number**: ratio of largest to smallest singular value of the Hessian matrix is large

# Optimization: Problems 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

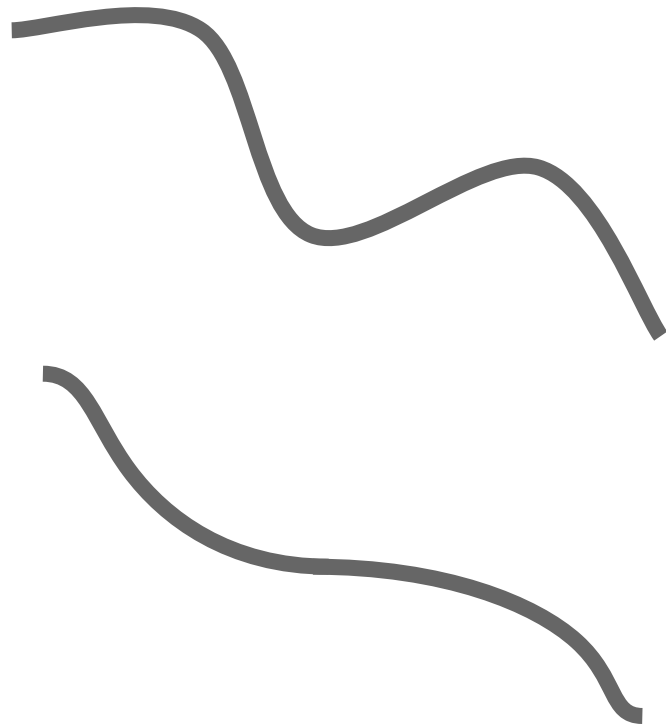


Loss function has high **condition number**: ratio of largest to smallest singular value of the Hessian matrix is large



# Optimization: Problems with SGD

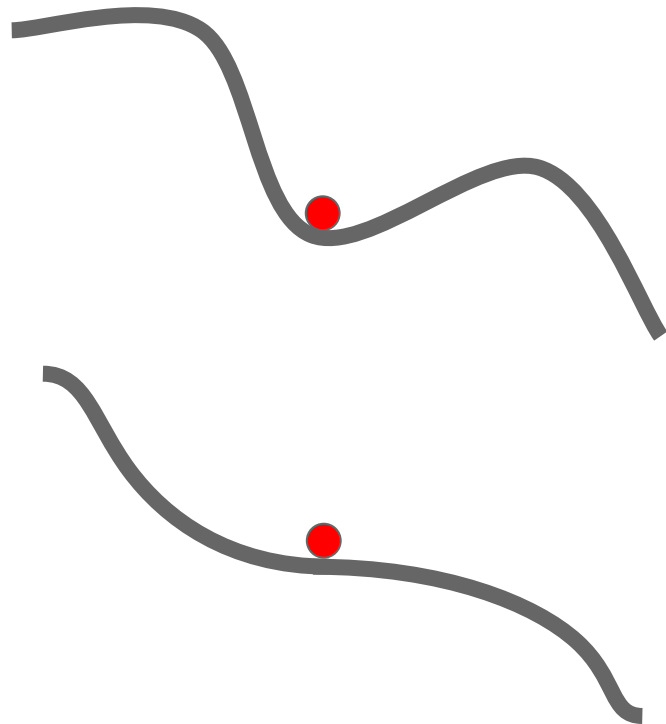
What if the loss function has a **local minima** or **saddle point**?



# Optimization: Problems with SGD

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

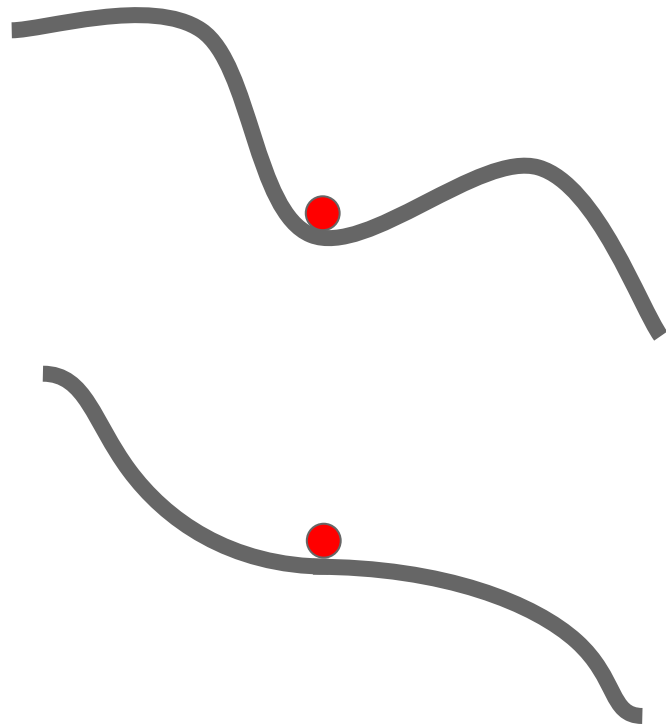
Zero gradient,  
gradient descent  
gets stuck



# Optimization: Problems with SGD

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

Saddle points much more common in high dimension



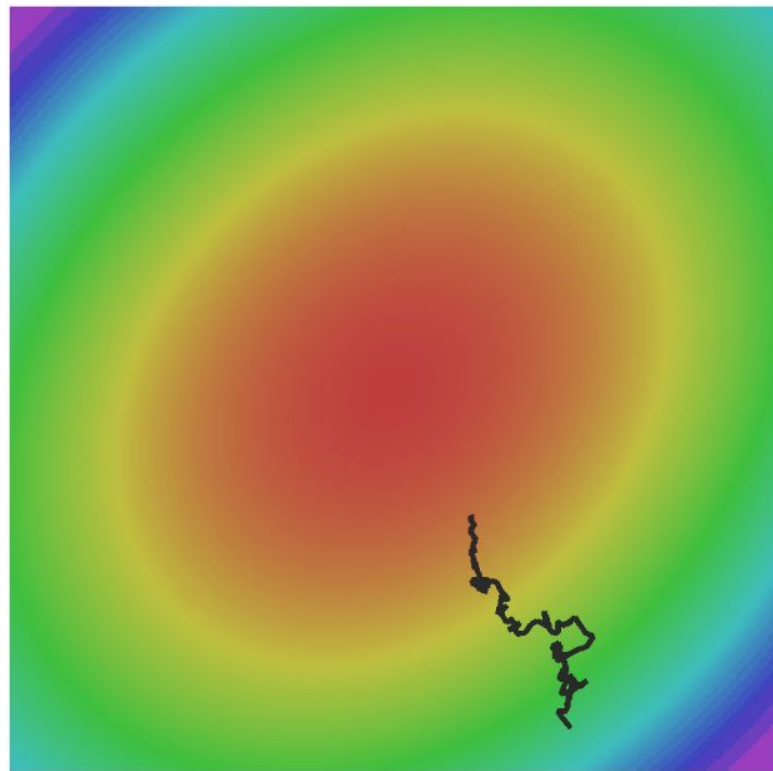
Dauphin et al, "Identifying and attacking the saddle point problem in high-dimensional non-convex optimization", NIPS 2014

# Optimization: Problems 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

## SGD

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

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

## SGD+Momentum

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

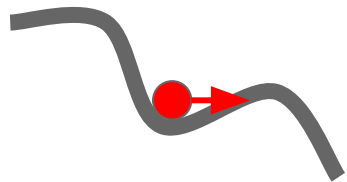
$$x_{t+1} = x_t - \alpha v_{t+1}$$

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

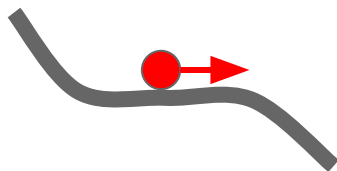
- Build up “velocity” as a running mean of gradients
- Rho gives “friction”; typically rho=0.9 or 0.99

# SGD + Momentum

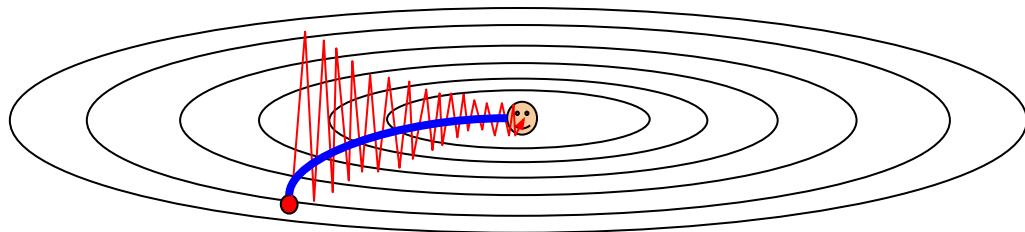
Local Minima



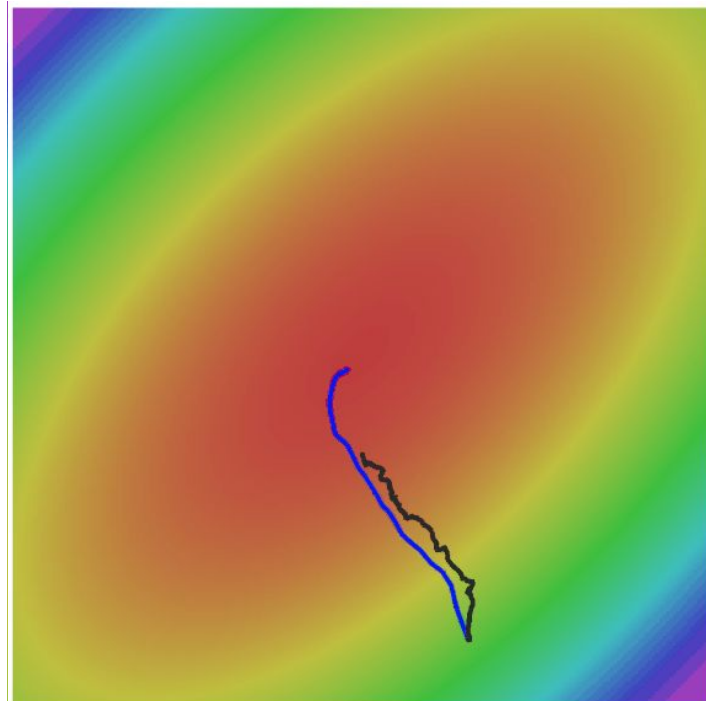
Saddle points



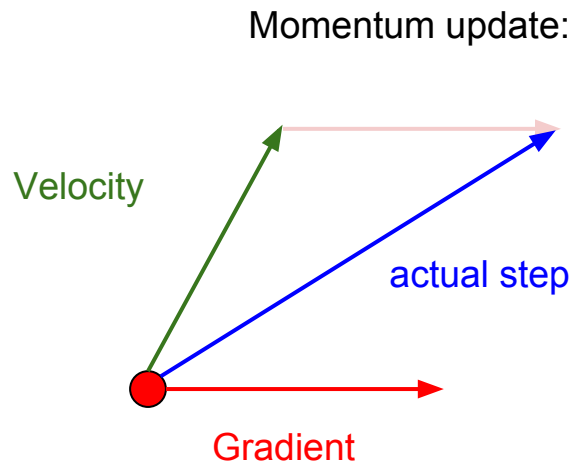
Poor Conditioning



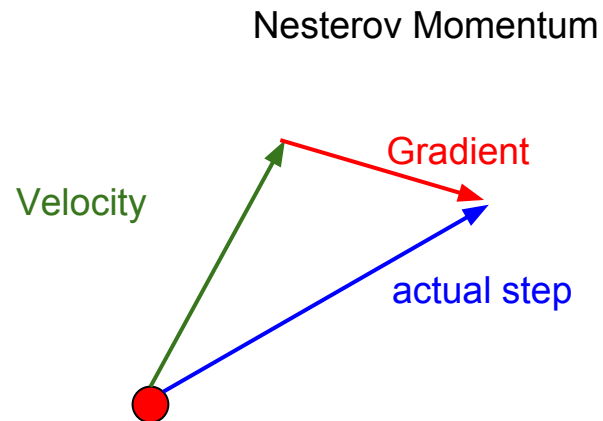
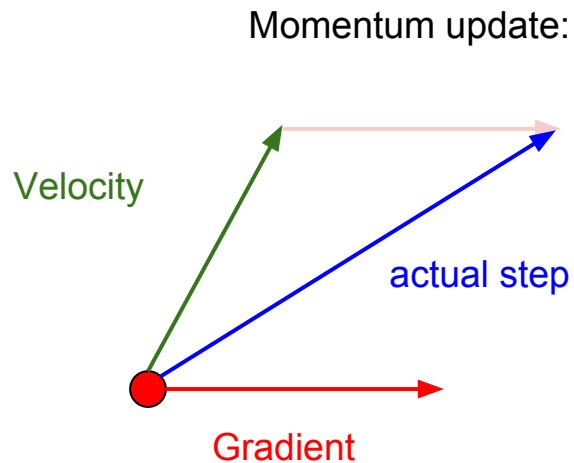
Gradient Noise



# SGD + Momentum



# Nesterov Momentum



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

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

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

# Nesterov Momentum

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

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

# Nesterov Momentum

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

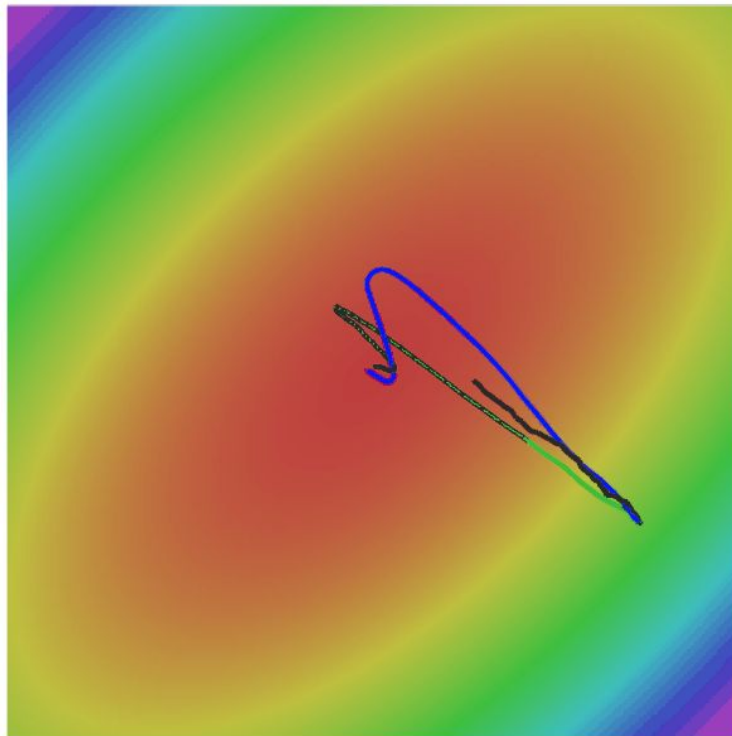
Annoying, usually we want update in terms of  $x_t, \nabla f(x_t)$

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

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

```
dx = compute_gradient(x)
old_v = v
v = rho * v - learning_rate * dx
x += -rho * old_v + (1 + rho) * v
```

# Nesterov Momentum



- SGD
- SGD+Momentum
- Nesterov

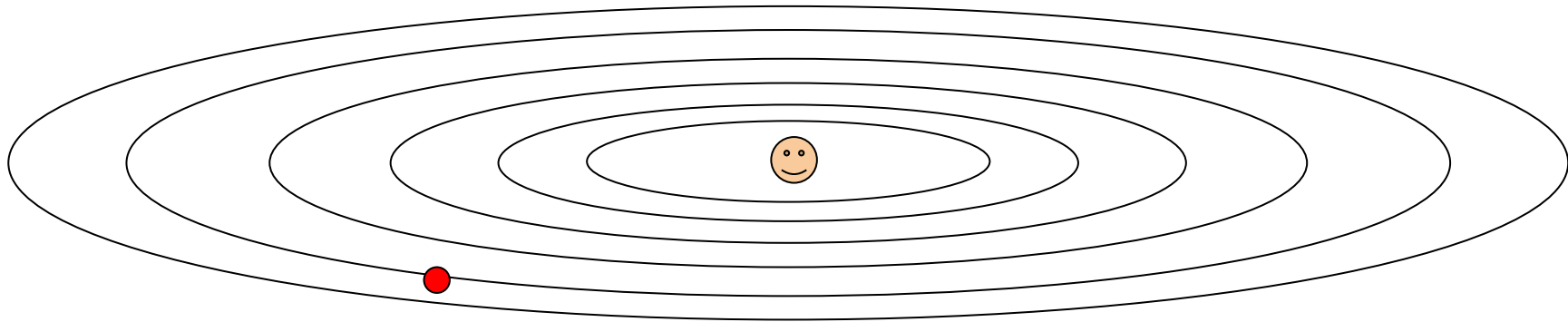
# AdaGrad

```
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

# AdaGrad

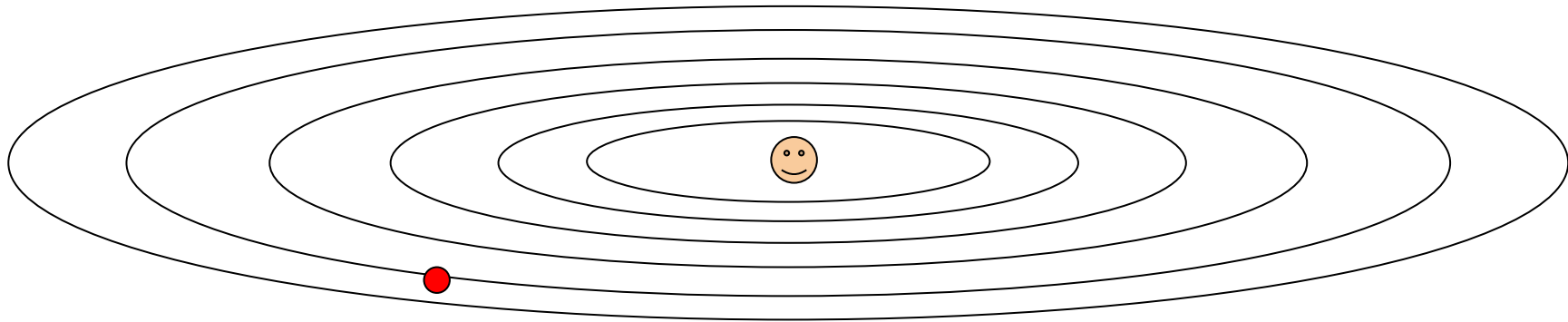
```
grad_squared = 0
while True:
    dx = compute_gradient(x)
    grad_squared += dx * dx
    x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```



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)
```



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

# RMSProp

AdaGrad

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

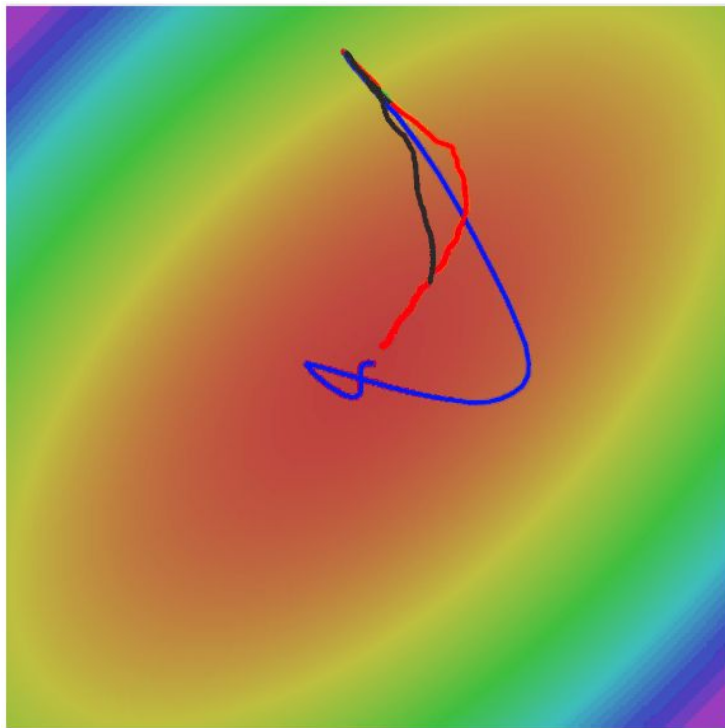


RMSProp

```
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

# 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))
```

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

# 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

Sort of like RMSProp with momentum

Q: What happens at first timestep?

# Adam (full form)

```
first_moment = 0
second_moment = 0
for t in range(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

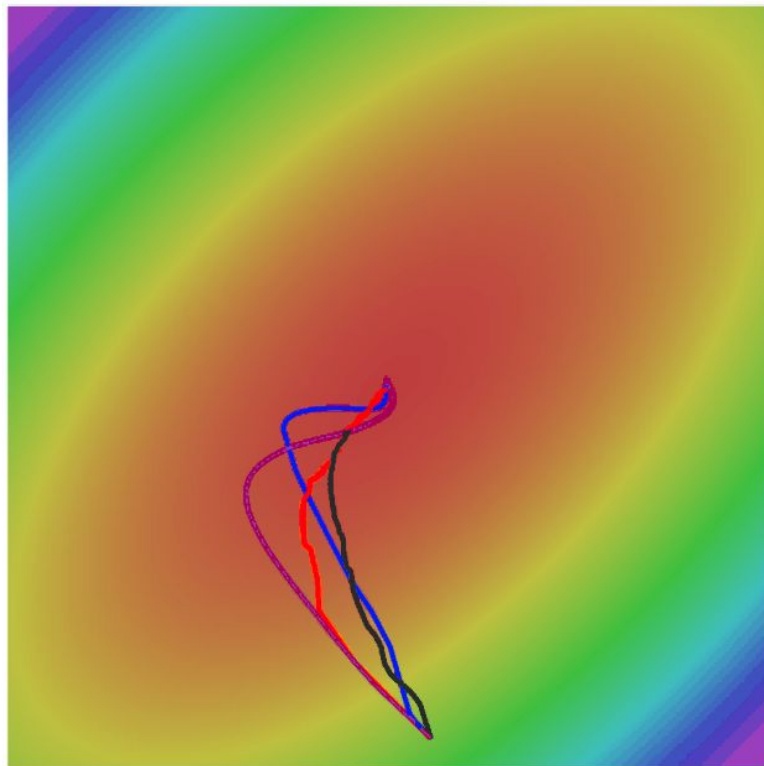
AdaGrad / RMSProp

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

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

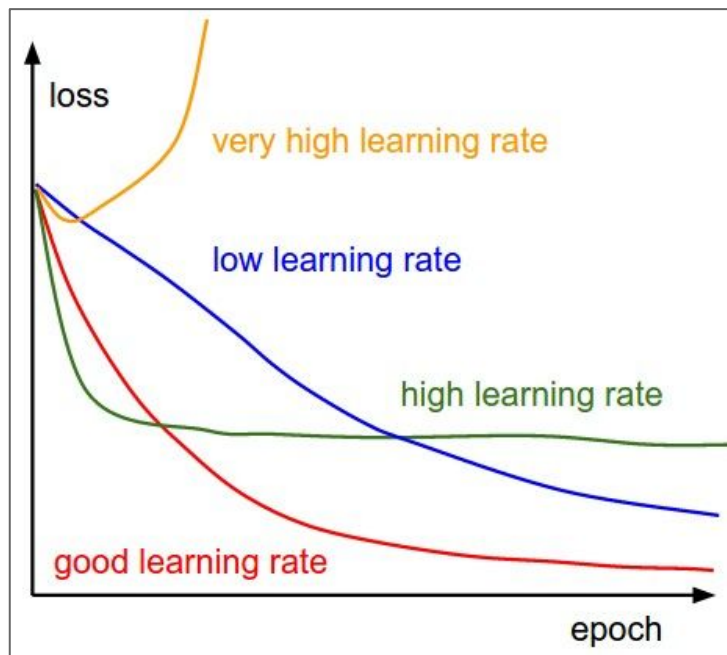
Kingma and Ba, "Adam: A method for stochastic optimization", ICLR 2015

# Adam



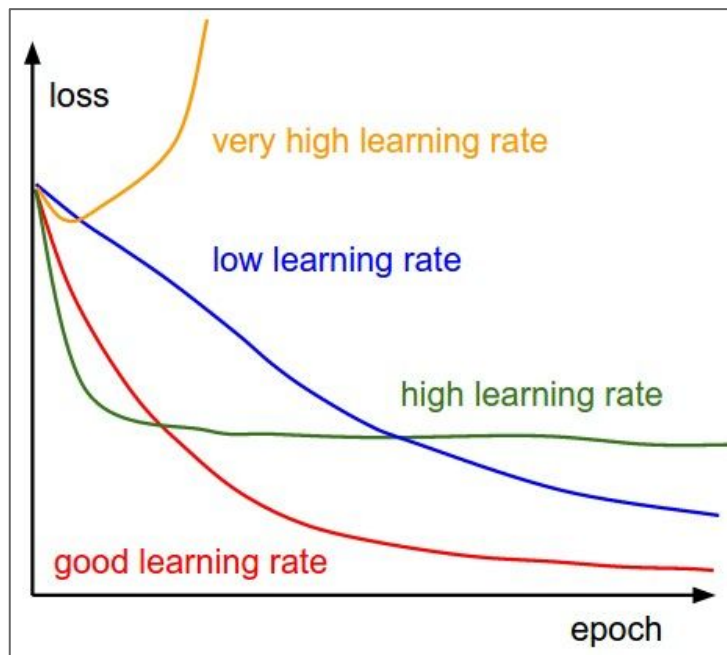
- SGD
- SGD+Momentum
- RMSProp
- Adam

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.



**=> Learning rate decay over time!**

**step decay:**

e.g. decay learning rate by half every few epochs.

**exponential decay:**

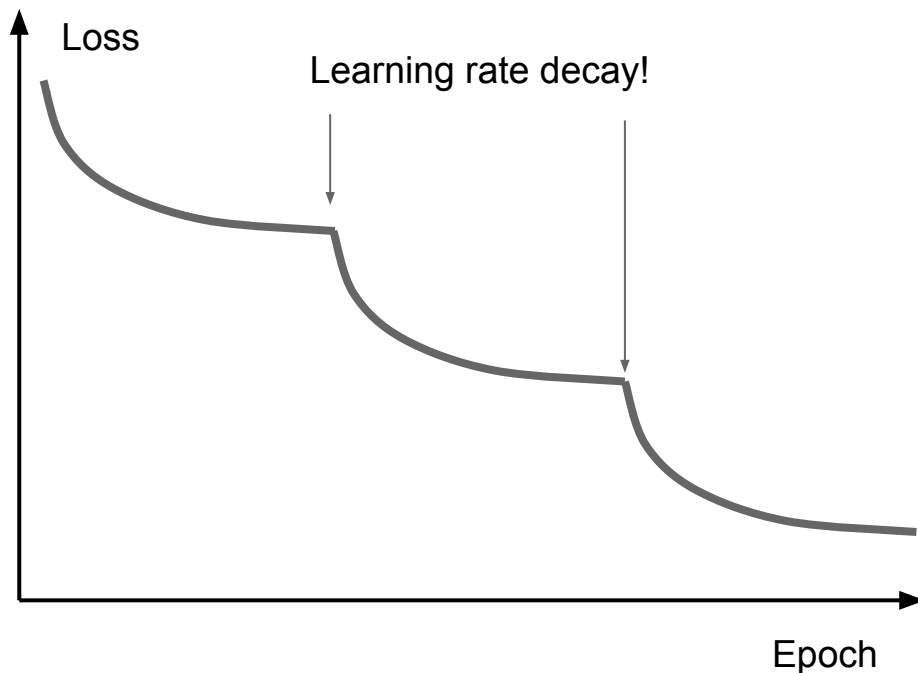
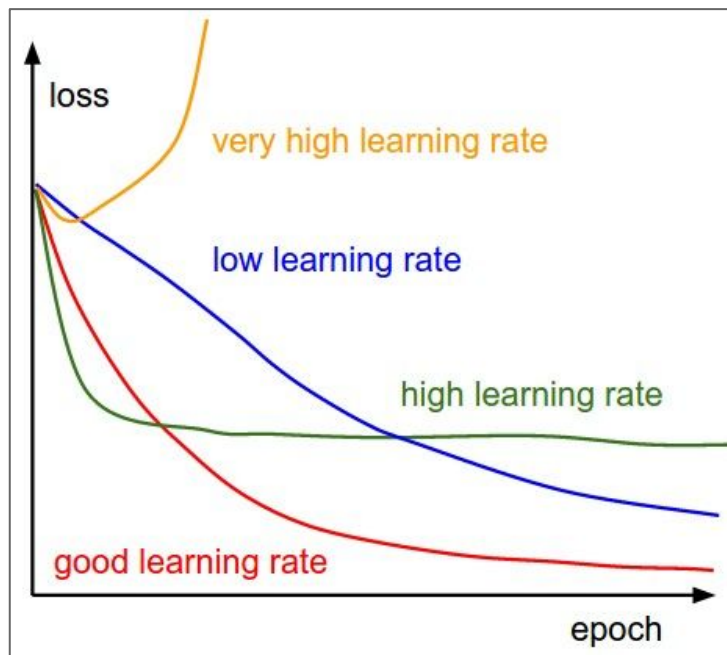
$$\alpha = \alpha_0 e^{-kt}$$

**1/t decay:**

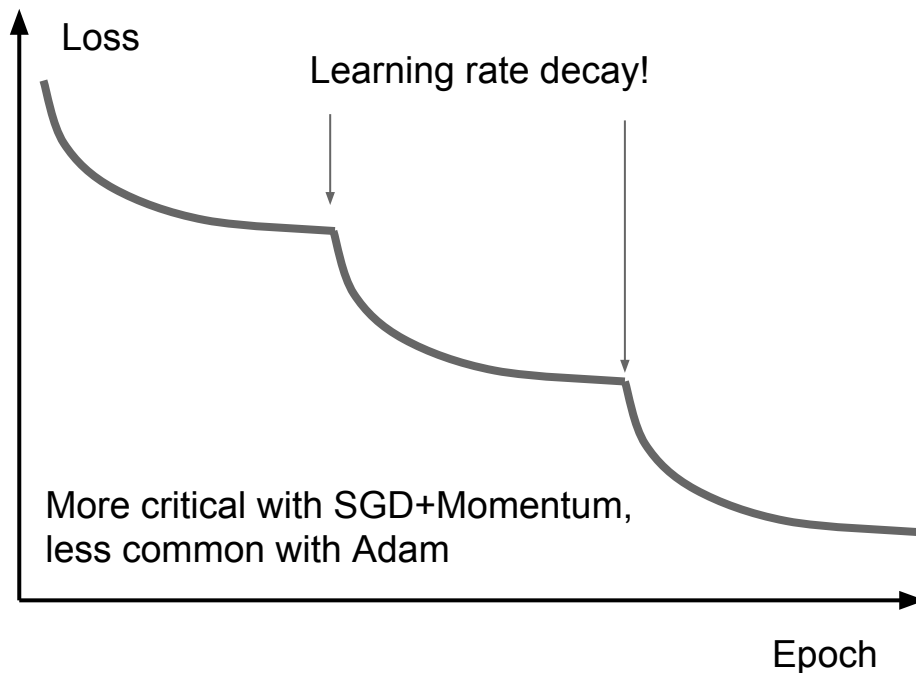
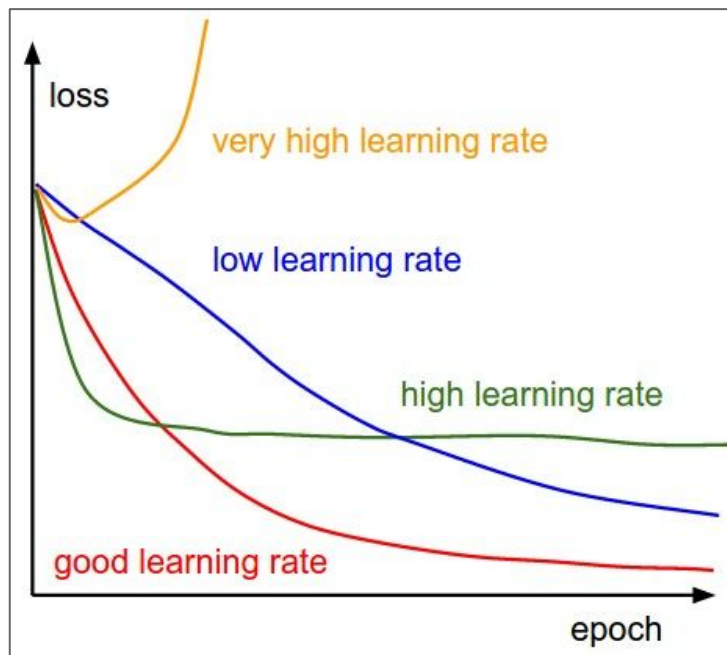
$$\alpha = \alpha_0 / (1 + kt)$$



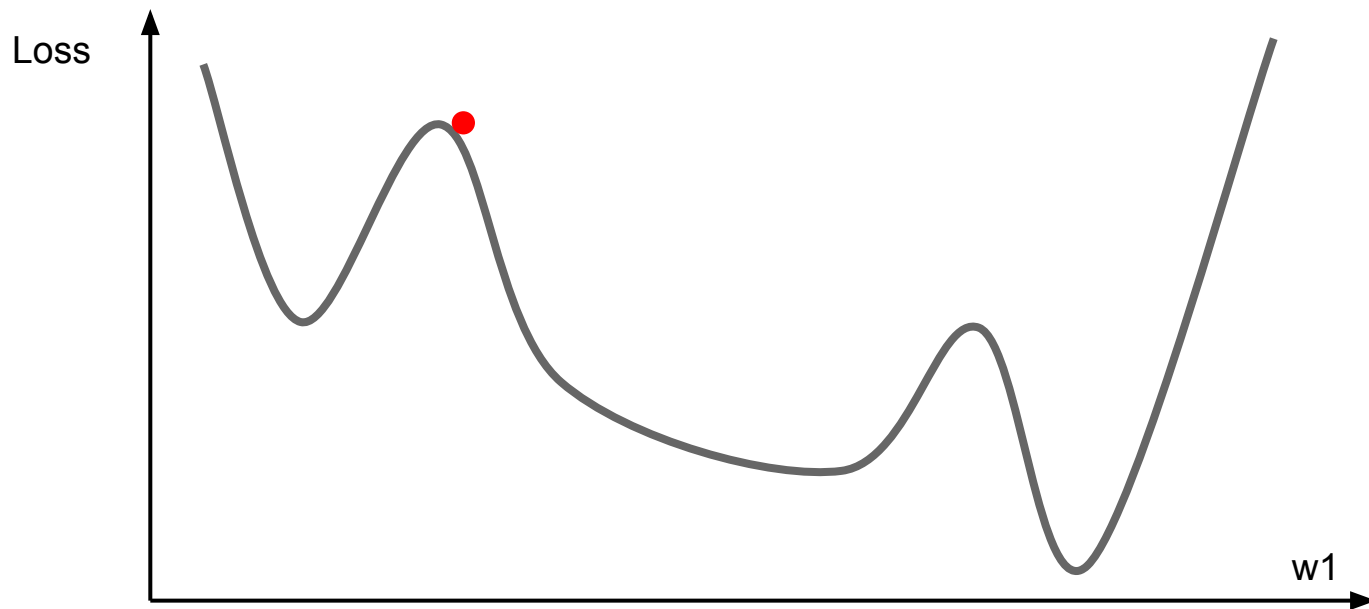
SGD, SGD+Momentum, Adagrad, RMSProp, Adam all have **learning rate** as a hyperparameter.



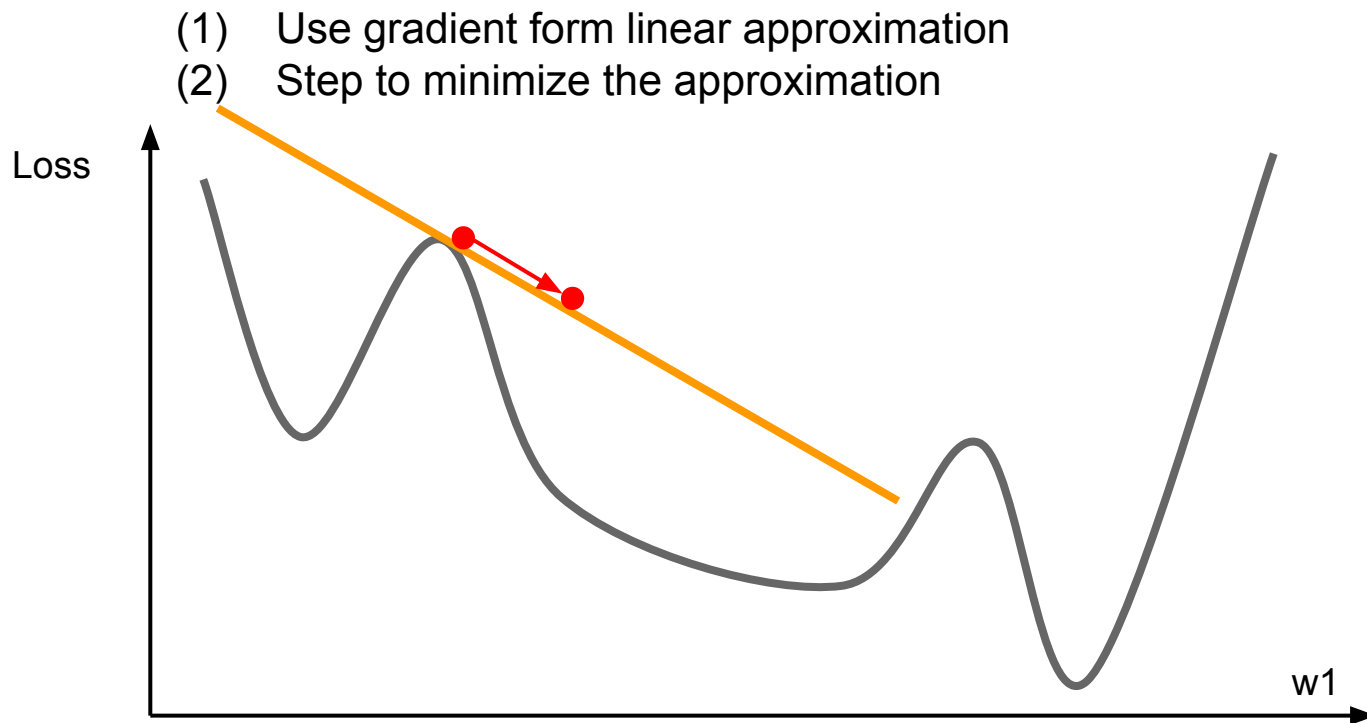
SGD, SGD+Momentum, Adagrad, RMSProp, Adam all have **learning rate** as a hyperparameter.



# First-Order Optimization

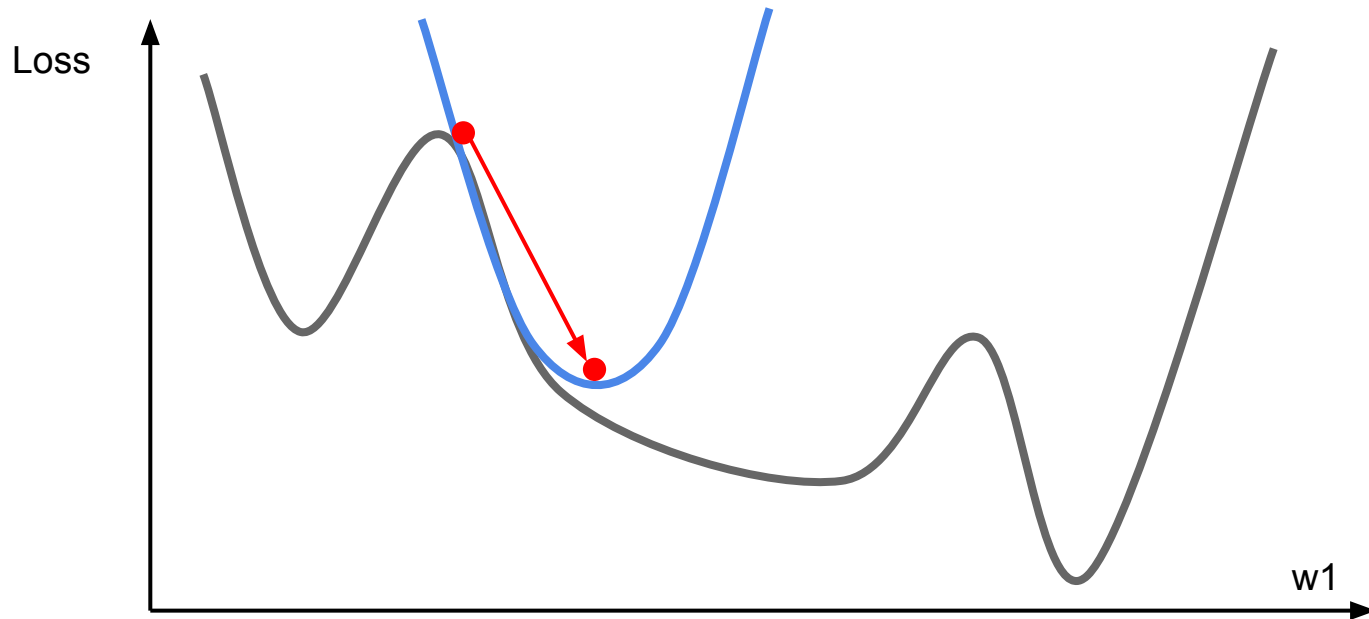


# First-Order Optimization



# 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:

$$J(\boldsymbol{\theta}) \approx J(\boldsymbol{\theta}_0) + (\boldsymbol{\theta} - \boldsymbol{\theta}_0)^\top \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0) + \frac{1}{2} (\boldsymbol{\theta} - \boldsymbol{\theta}_0)^\top \mathbf{H} (\boldsymbol{\theta} - \boldsymbol{\theta}_0)$$

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

$$\boldsymbol{\theta}^* = \boldsymbol{\theta}_0 - \mathbf{H}^{-1} \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0)$$

Q: What is nice about this update?

# Second-Order Optimization

second-order Taylor expansion:

$$J(\boldsymbol{\theta}) \approx J(\boldsymbol{\theta}_0) + (\boldsymbol{\theta} - \boldsymbol{\theta}_0)^\top \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0) + \frac{1}{2} (\boldsymbol{\theta} - \boldsymbol{\theta}_0)^\top \mathbf{H} (\boldsymbol{\theta} - \boldsymbol{\theta}_0)$$

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

$$\boldsymbol{\theta}^* = \boldsymbol{\theta}_0 - \mathbf{H}^{-1} \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0)$$

No hyperparameters!  
No learning rate!

Q: What is nice about this update?

# Second-Order Optimization

second-order Taylor expansion:

$$J(\boldsymbol{\theta}) \approx J(\boldsymbol{\theta}_0) + (\boldsymbol{\theta} - \boldsymbol{\theta}_0)^\top \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0) + \frac{1}{2} (\boldsymbol{\theta} - \boldsymbol{\theta}_0)^\top \mathbf{H} (\boldsymbol{\theta} - \boldsymbol{\theta}_0)$$

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

$$\boldsymbol{\theta}^* = \boldsymbol{\theta}_0 - \mathbf{H}^{-1} \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0)$$

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

Q2: Why is this bad for deep learning?



# Second-Order Optimization

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

- 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.*

# Second-Order Optimization

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

- 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.*

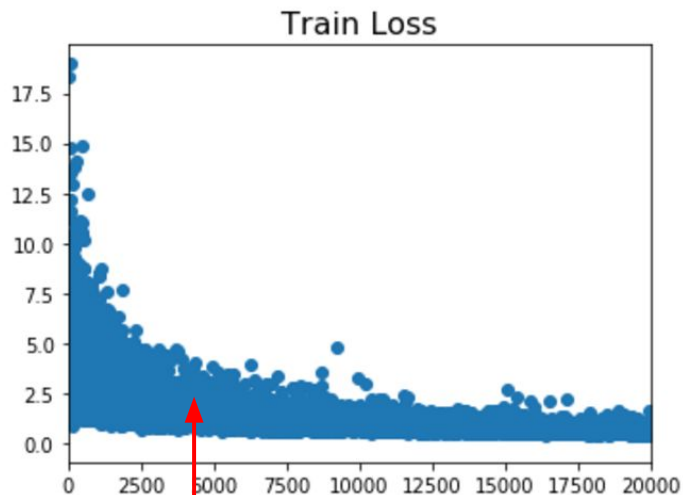
# L-BFGS

- **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 L-BFGS to large-scale, stochastic setting is an active area of research.

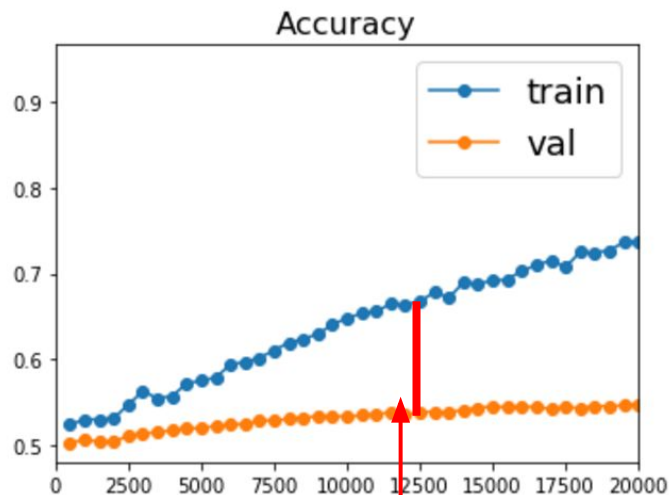
# In practice:

- **Adam** is a good default choice in most cases
- If you can afford to do full batch updates then try out **L-BFGS** (and don't forget to disable all sources of noise)

# Beyond Training Error



Better optimization algorithms  
help reduce training loss



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

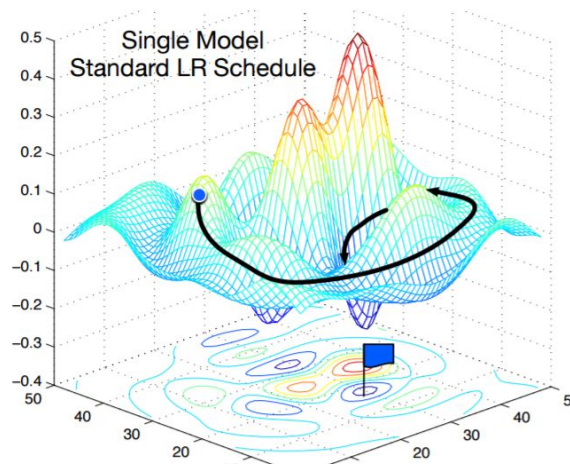
# Model Ensembles

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

Enjoy 2% extra performance

# Model Ensembles: Tips and Tricks

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



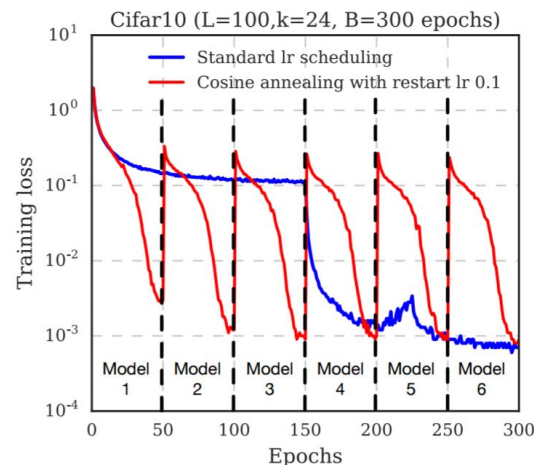
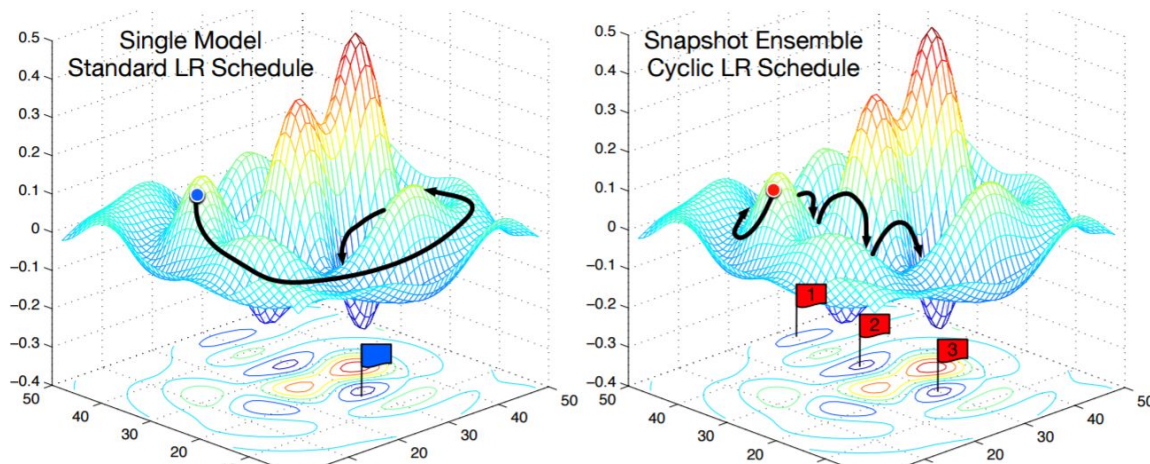
Loshchilov and Hutter, "SGDR: Stochastic gradient descent with restarts", arXiv 2016

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!



Cyclic learning rate schedules can make this work even better!

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.



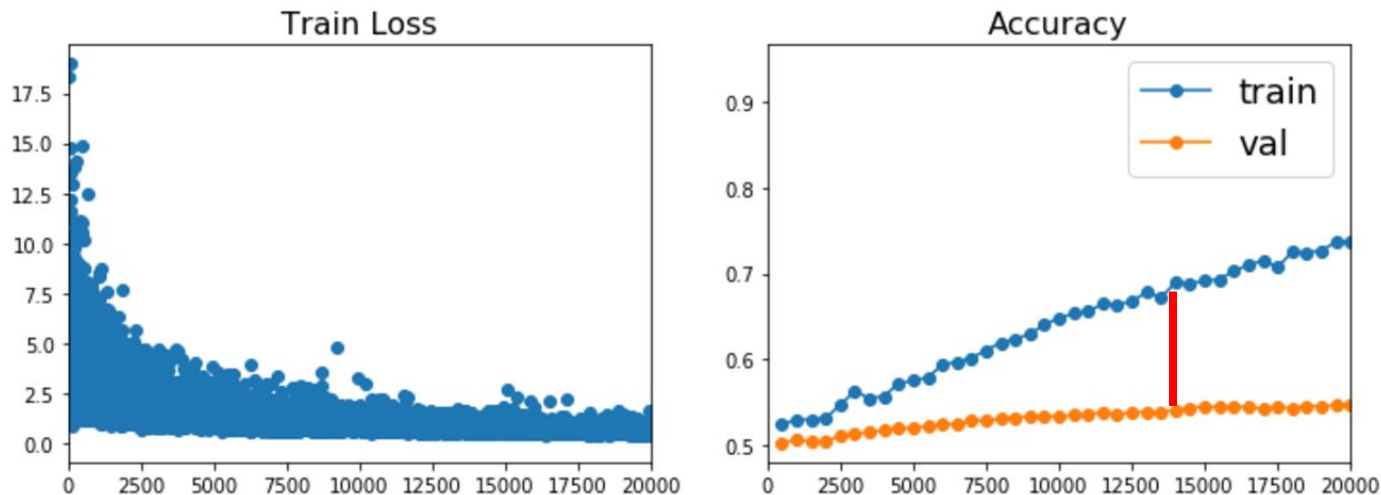
# Model Ensembles: Tips and Tricks

Instead of 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.

# 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**

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

L1 regularization

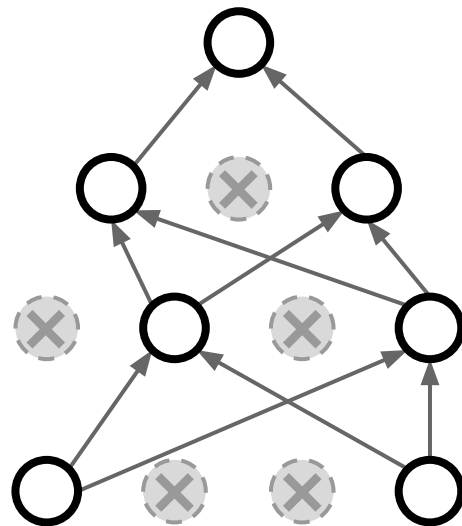
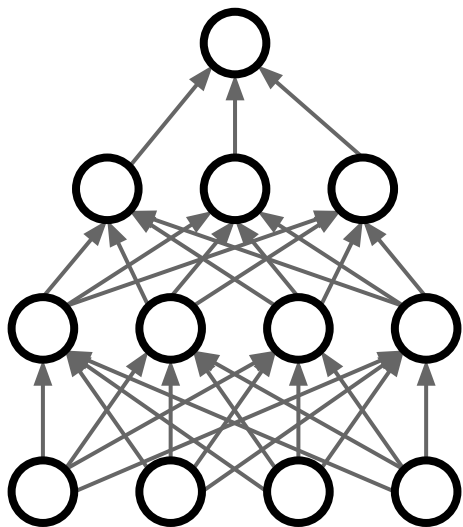
$$R(W) = \sum_k \sum_l |W_{k,l}|$$

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
```

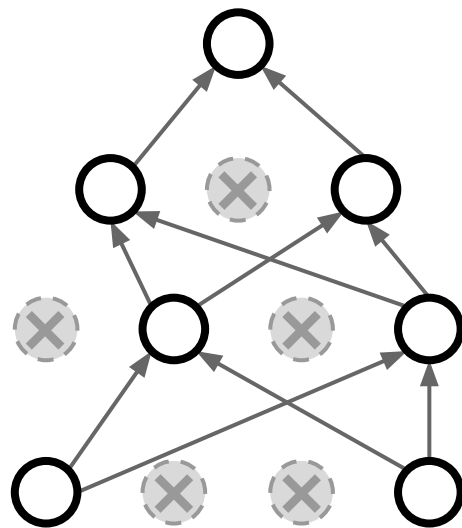
```
    H2 *= U2 # drop!
```

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

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

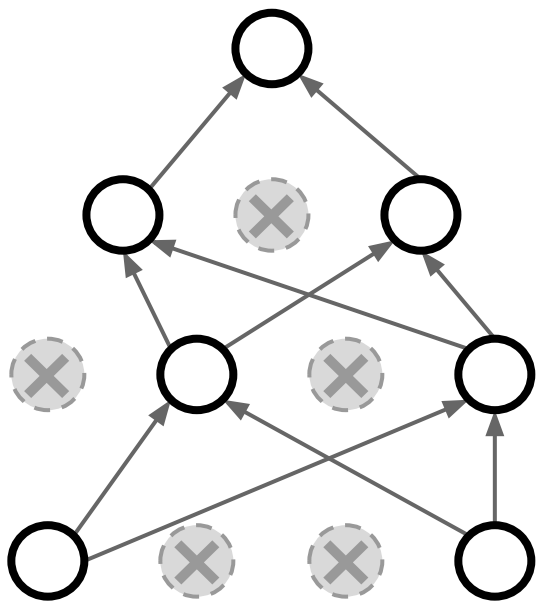
```
    # perform parameter update... (not shown)
```

Example forward  
pass with a  
3-layer network  
using dropout



# Regularization: Dropout

How can this possibly be a good idea?

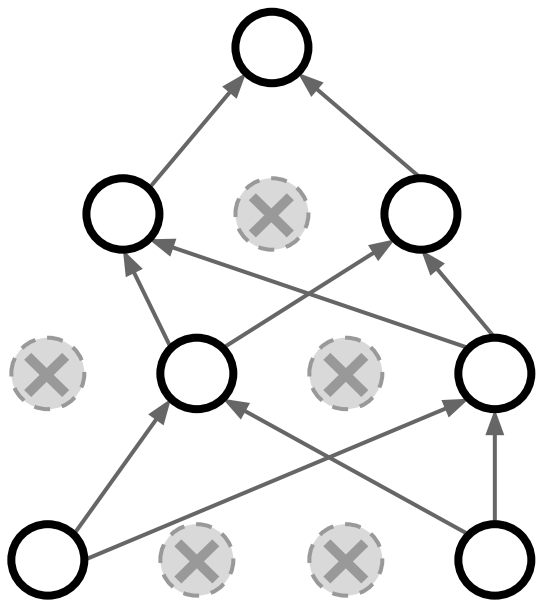


Forces the network to have a redundant representation;  
Prevents co-adaptation of features



# 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!

$$\text{Output (label)} \quad y = f_W(\text{Input (image)} \quad x, z) \quad \text{Random mask}$$

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

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

But this integral seems hard ...

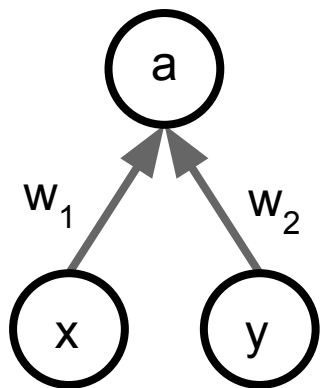


# Dropout: Test time

Want to approximate  
the integral

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

Consider a single neuron.



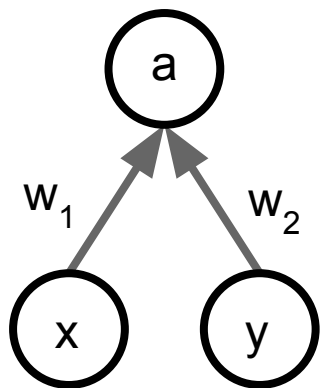
# Dropout: Test time

Want to approximate  
the integral

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

Consider a single neuron.

At test time we have:  $E[a] = w_1x + w_2y$

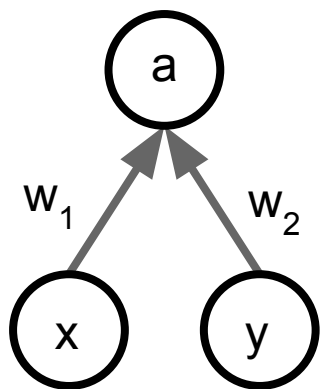


# Dropout: Test time

Want to approximate  
the integral

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

Consider a single neuron.



At test time we have:  $E[a] = w_1x + w_2y$

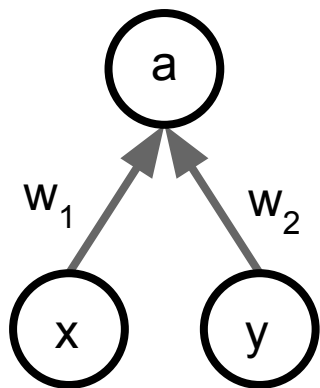
During training we have: 
$$\begin{aligned} E[a] &= \frac{1}{4}(w_1x + w_2y) + \frac{1}{4}(w_1x + 0y) \\ &\quad + \frac{1}{4}(0x + 0y) + \frac{1}{4}(0x + w_2y) \\ &= \frac{1}{2}(w_1x + w_2y) \end{aligned}$$

# Dropout: Test time

Want to approximate the integral

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

Consider a single neuron.



At test time we have:  $E[a] = w_1x + w_2y$

During training 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 forward pass

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!



# Regularization: A common pattern

**Training:** Add some kind of randomness

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

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

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



# Regularization: A common pattern

**Training:** Add some kind of randomness

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

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

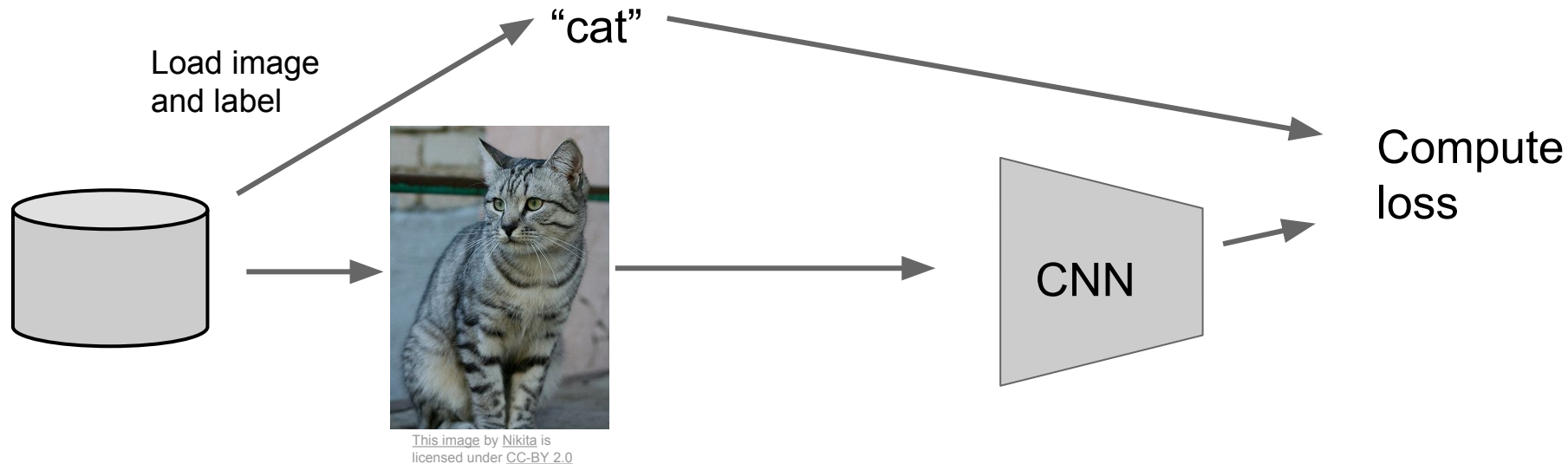
$$y = f(x) = E_z[f(x, z)] = \int p(z)f(x, z)dz$$

**Example:** Batch Normalization

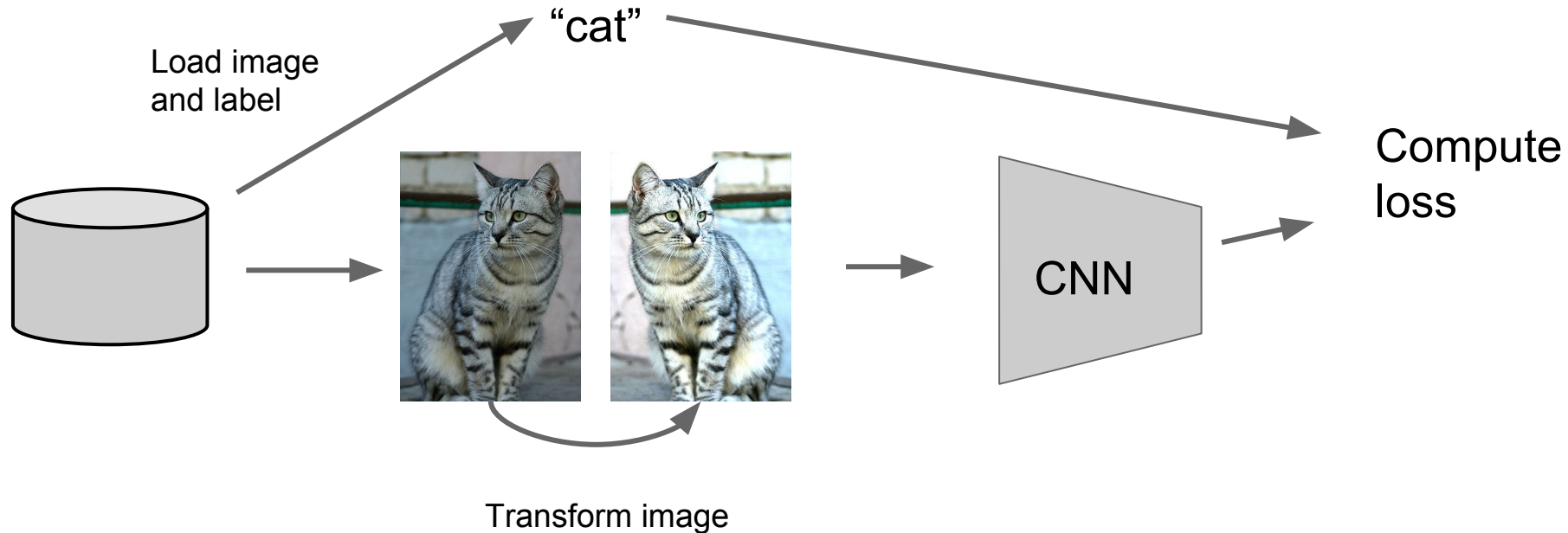
**Training:**  
Normalize using stats from random minibatches

**Testing:** Use fixed stats to normalize

# Regularization: Data Augmentation



# Regularization: Data Augmentation



# Data Augmentation

## Horizontal Flips



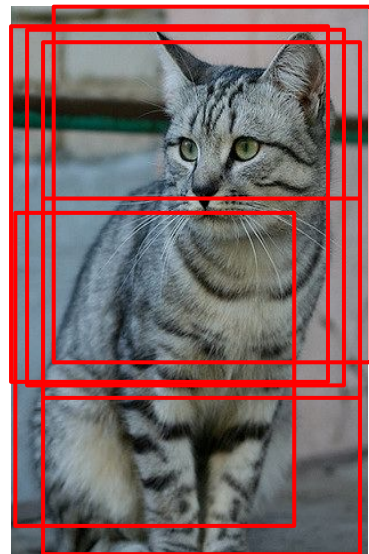
# Data Augmentation

## 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  $224 \times 224$  patch



# Data Augmentation

## Random crops and scales

**Training:** sample random crops / scales

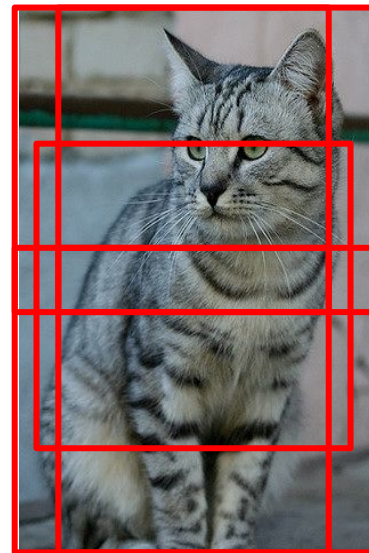
ResNet:

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

Random mix/combinations of :

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

# Regularization: A common pattern

**Training:** Add random noise

**Testing:** Marginalize over the noise

## **Examples:**

Dropout

Batch Normalization

Data Augmentation

# Regularization: A common pattern

**Training:** Add random noise

**Testing:** Marginalize over the noise

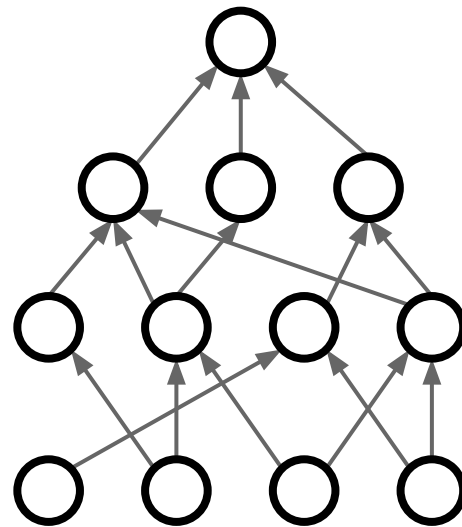
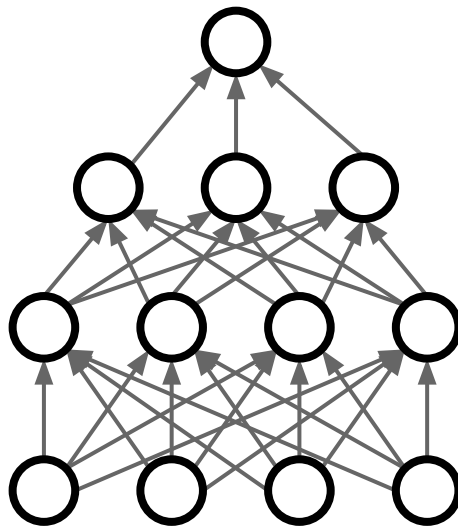
## Examples:

Dropout

Batch Normalization

Data Augmentation

DropConnect



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

# Regularization: A common pattern

**Training:** Add random noise

**Testing:** Marginalize over the noise

## Examples:

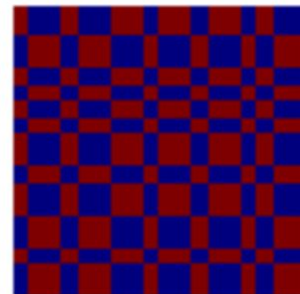
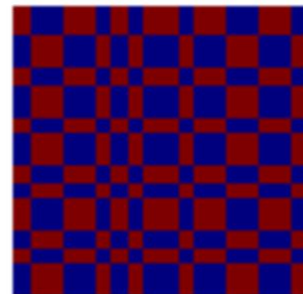
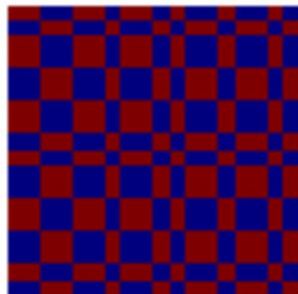
Dropout

Batch Normalization

Data Augmentation

DropConnect

Fractional Max Pooling



Graham, "Fractional Max Pooling", arXiv 2014

# Regularization: A common pattern

**Training:** Add random noise

**Testing:** Marginalize over the noise

## Examples:

Dropout

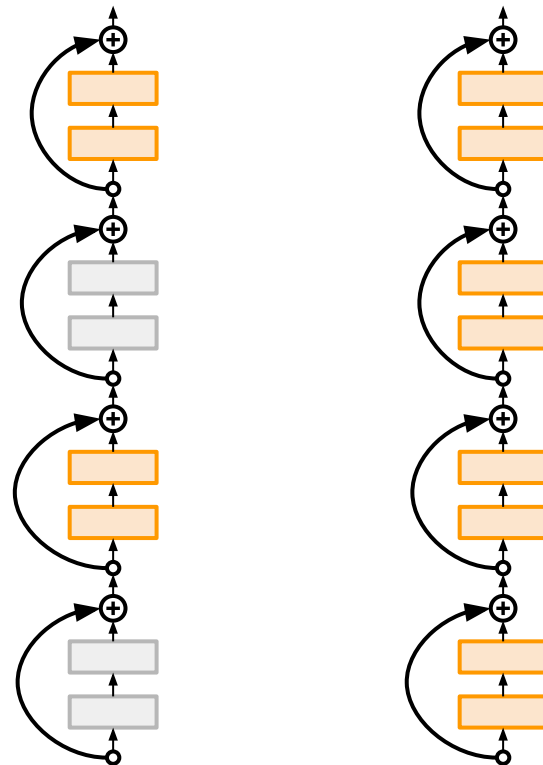
Batch Normalization

Data Augmentation

DropConnect

Fractional Max Pooling

Stochastic Depth



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

# Transfer Learning

“You need a lot of a data if you want to train/use CNNs”

# Transfer Learning

“You need a lot of data if you want to train/use CNNs”

**BUSTED**

# Transfer Learning with CNNs

Donahue et al, "DeCAF: A Deep Convolutional Activation Feature for Generic Visual Recognition", ICML 2014  
Razavian et al, "CNN Features Off-the-Shelf: An Astounding Baseline for Recognition", CVPR Workshops 2014

## 1. Train on Imagenet





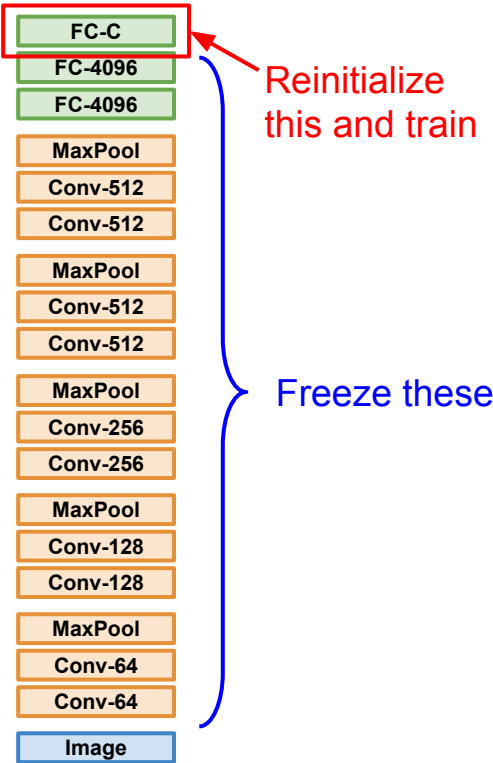
# Transfer Learning with CNNs

Donahue et al, "DeCAF: A Deep Convolutional Activation Feature for Generic Visual Recognition", ICML 2014  
Razavian et al, "CNN Features Off-the-Shelf: An Astounding Baseline for Recognition", CVPR Workshops 2014

## 1. Train on Imagenet



## 2. Small Dataset (C classes)



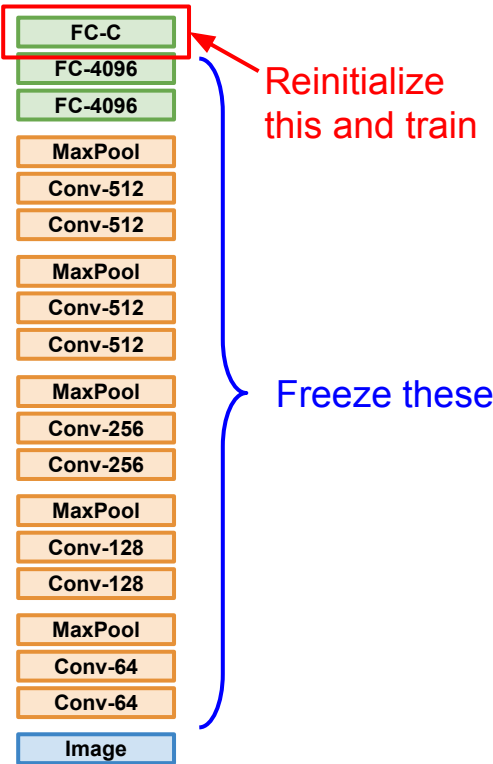
# Transfer Learning with CNNs

Donahue et al, "DeCAF: A Deep Convolutional Activation Feature for Generic Visual Recognition", ICML 2014  
Razavian et al, "CNN Features Off-the-Shelf: An Astounding Baseline for Recognition", CVPR Workshops 2014

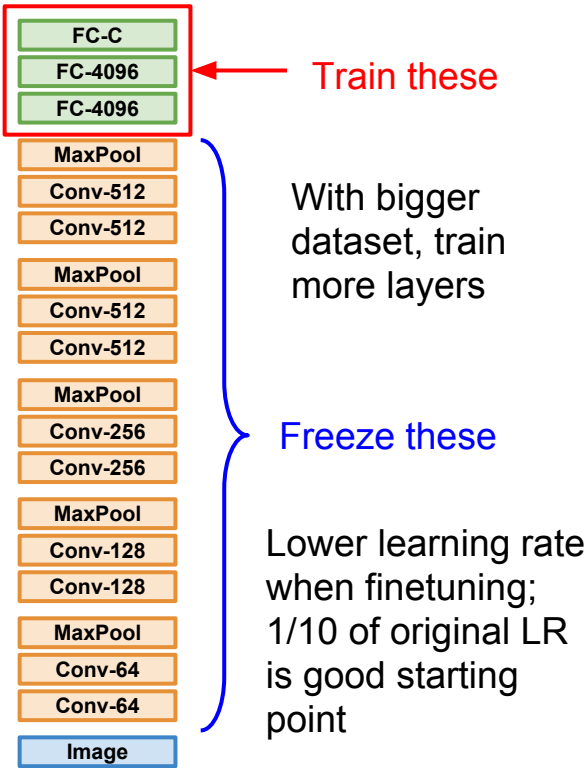
## 1. Train on Imagenet



## 2. Small Dataset (C classes)



## 3. Bigger dataset





More specific

More generic

	<b>very similar dataset</b>	<b>very different dataset</b>
<b>very little data</b>	?	?
<b>quite a lot of data</b>	?	?



More specific

More generic

	<b>very similar dataset</b>	<b>very different dataset</b>
<b>very little data</b>	Use Linear Classifier on top layer	?
<b>quite a lot of data</b>	Finetune a few layers	?



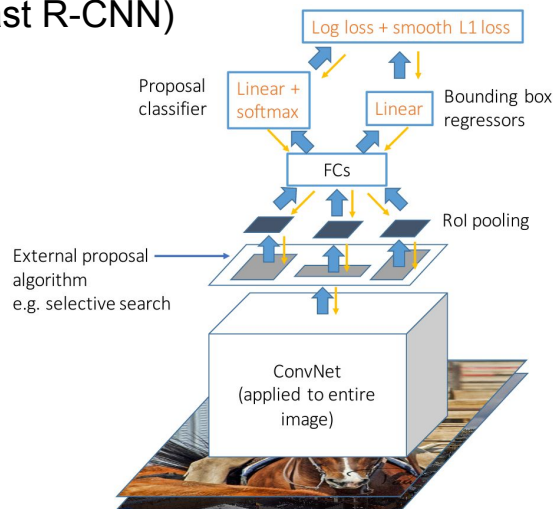
More specific

More generic

	<b>very similar dataset</b>	<b>very different dataset</b>
<b>very little data</b>	Use Linear Classifier on top layer	You're in trouble... Try linear classifier from different stages
<b>quite a lot of data</b>	Finetune a few layers	Finetune a larger number of layers

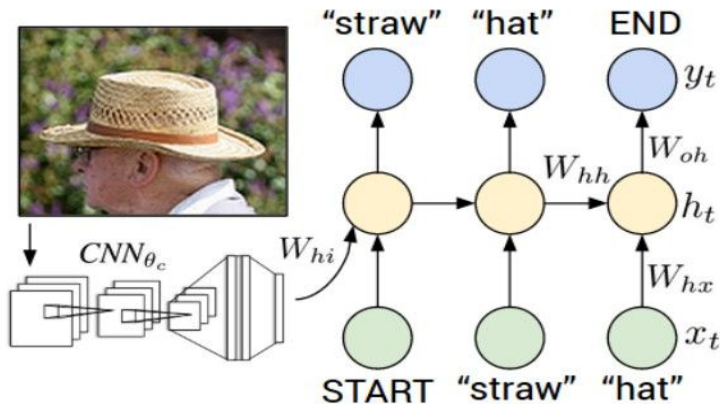
# Transfer learning with CNNs is pervasive... (it's the norm, not an exception)

## Object Detection (Fast R-CNN)



Girshick, "Fast R-CNN", ICCV 2015  
Figure copyright Ross Girshick, 2015. Reproduced with permission.

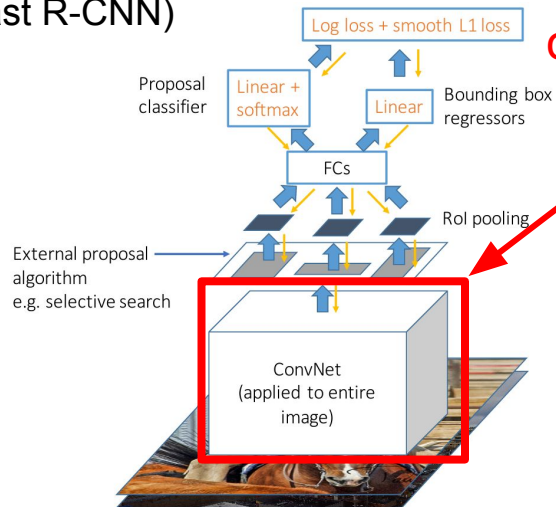
## Image Captioning: CNN + RNN



Karpathy and Fei-Fei, "Deep Visual-Semantic Alignments for  
Generating Image Descriptions", CVPR 2015  
Figure copyright IEEE, 2015. Reproduced for educational purposes.

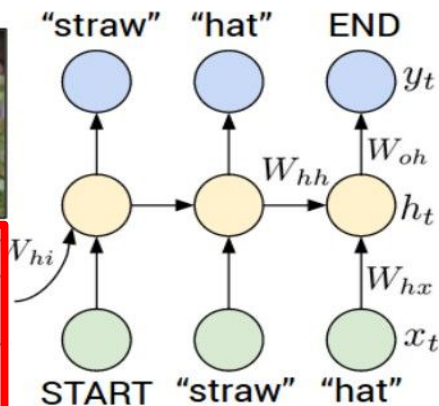
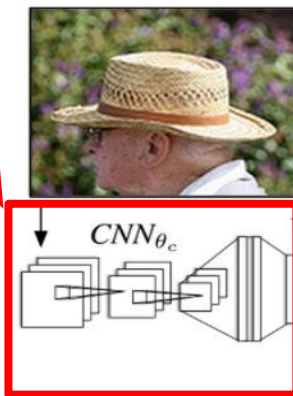
# Transfer learning with CNNs is pervasive... (it's the norm, not an exception)

## Object Detection (Fast R-CNN)



**CNN pretrained  
on ImageNet**

## Image Captioning: CNN + RNN

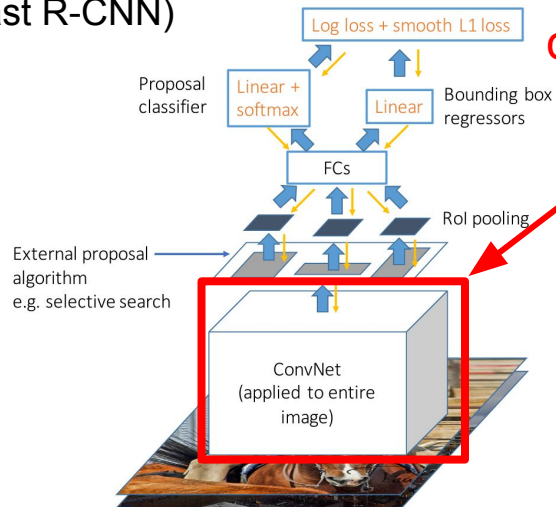


Girshick, "Fast R-CNN", ICCV 2015  
Figure copyright Ross Girshick, 2015. Reproduced with permission.

Karpathy and Fei-Fei, "Deep Visual-Semantic Alignments for  
Generating Image Descriptions", CVPR 2015  
Figure copyright IEEE, 2015. Reproduced for educational purposes.

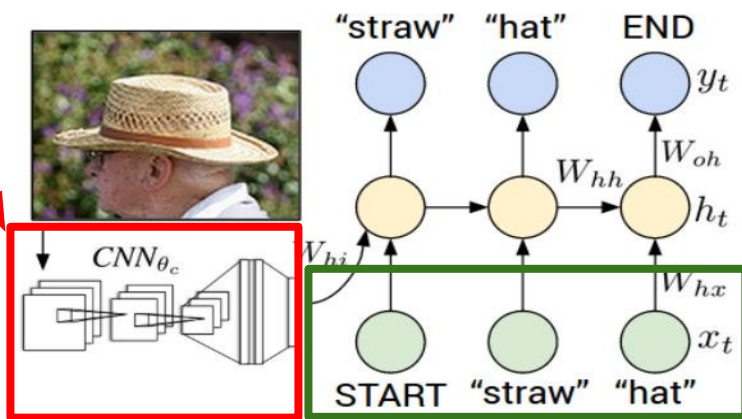
# Transfer learning with CNNs is pervasive... (it's the norm, not an exception)

## Object Detection (Fast R-CNN)



**CNN pretrained  
on ImageNet**

## Image Captioning: CNN + RNN



**Word vectors pretrained  
with word2vec**

Girshick, "Fast R-CNN", ICCV 2015  
Figure copyright Ross Girshick, 2015. Reproduced with permission.

Karpathy and Fei-Fei, "Deep Visual-Semantic Alignments for  
Generating Image Descriptions", CVPR 2015  
Figure copyright IEEE, 2015. Reproduced for educational purposes.



# Takeaway for your projects and beyond:

Have some dataset of interest but it has  $< \sim 1\text{M}$  images?

1. Find a very large dataset that has similar data, train a big ConvNet there
2. Transfer learn to your dataset

Deep learning frameworks provide a “Model Zoo” of pretrained models so you don’t need to train your own

Caffe: <https://github.com/BVLC/caffe/wiki/Model-Zoo>

TensorFlow: <https://github.com/tensorflow/models>

PyTorch: <https://github.com/pytorch/vision>

# Summary

- Optimization
  - Momentum, RMSProp, Adam, etc
- Regularization
  - Dropout, etc
- Transfer learning
  - Use this for your projects!

Next time: Deep Learning Software!