HA NOI UNIVERSITY OF SCIENCE AND TECHNOLOGY
SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY

Lesson 5: Training neural networks

(Part 2)

Viet-Trung Tran

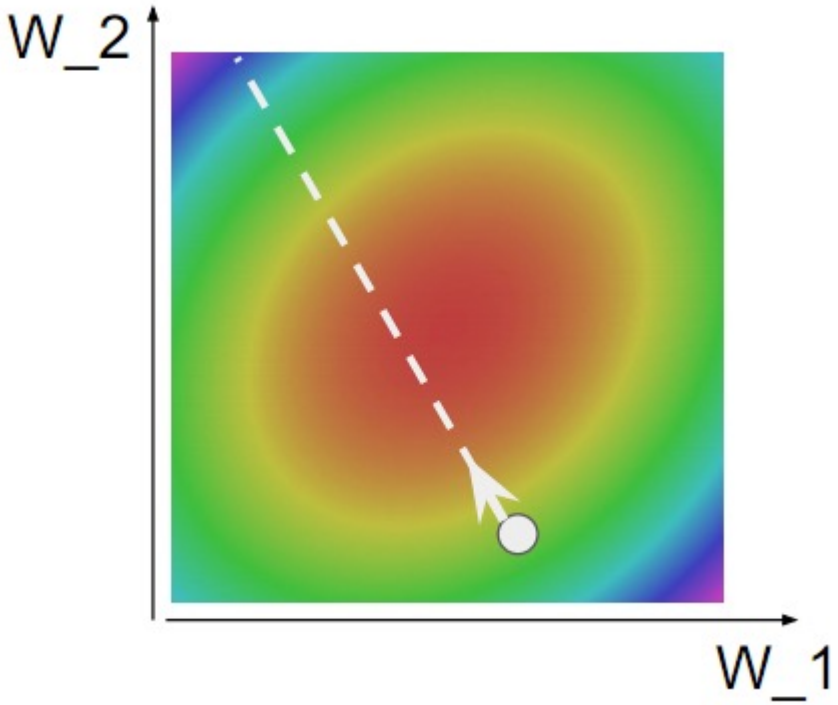
Outline

- Optimization algorighms for neural networks
- Learning rate schedules
- Anti-overfitting techniques
- Data enrichment (data augmentation)
- Choosing Hyperparameters
- Techniques for combining multiple models (ensemble methods)
- Transfer learning

Optimization algorighms for neural networks

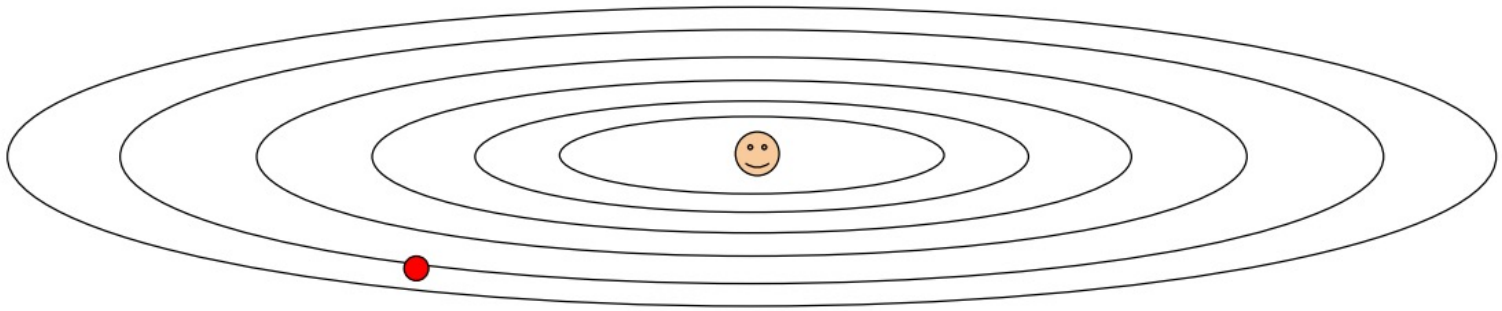
Stochastic gradient descent (SGD)

```
# Vanilla Gradient Descent

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



Problem #1 with SGD

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

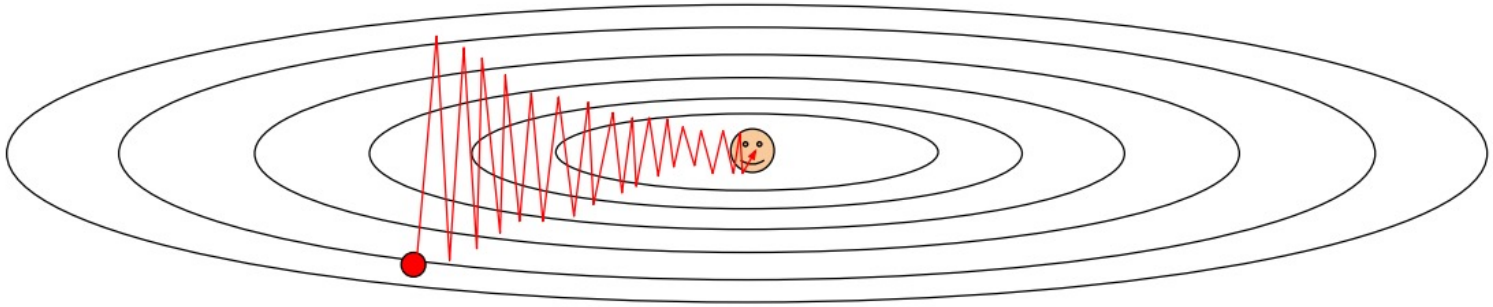


Loss function has high condition number: ratio of largest to smallest singular value of the Hessian matrix is large

(Hessian is a square matrix of second-order partial derivatives of a scalar-valued function, or scalar field. It describes the local curvature of a function of many variables.)

Problem #1 with SGD (2)

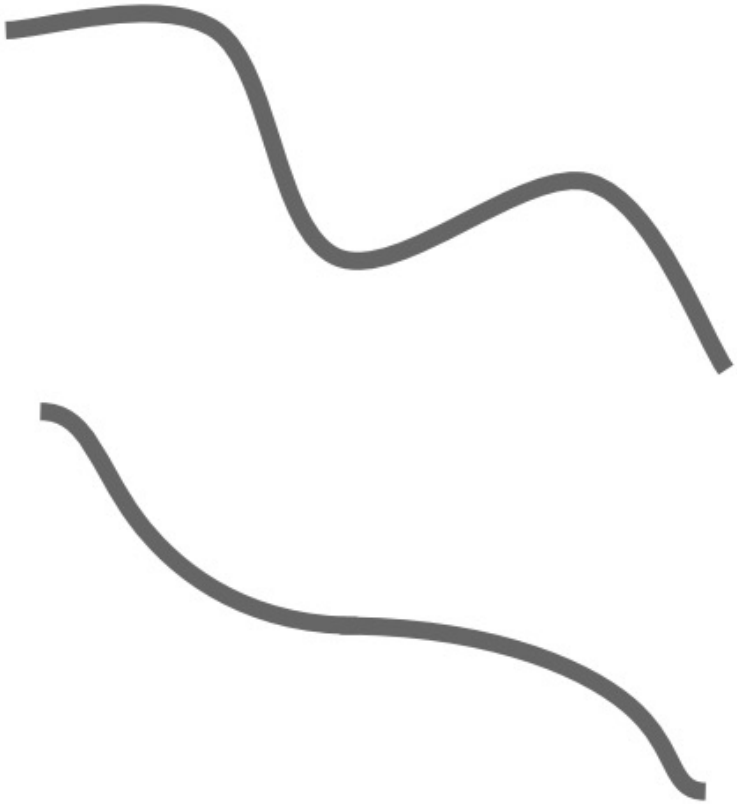
- What if loss changes quickly in one direction and slowly in another? What does gradient descent do?
- Very slow progress along shallow dimension, jitter along steep direction



Loss function has high condition number: ratio of largest to smallest singular value of the Hessian matrix is large (Hessian is a square matrix of second-order partial derivatives of a scalar-valued function, or scalar field. It describes the local curvature of a function of many variables.)

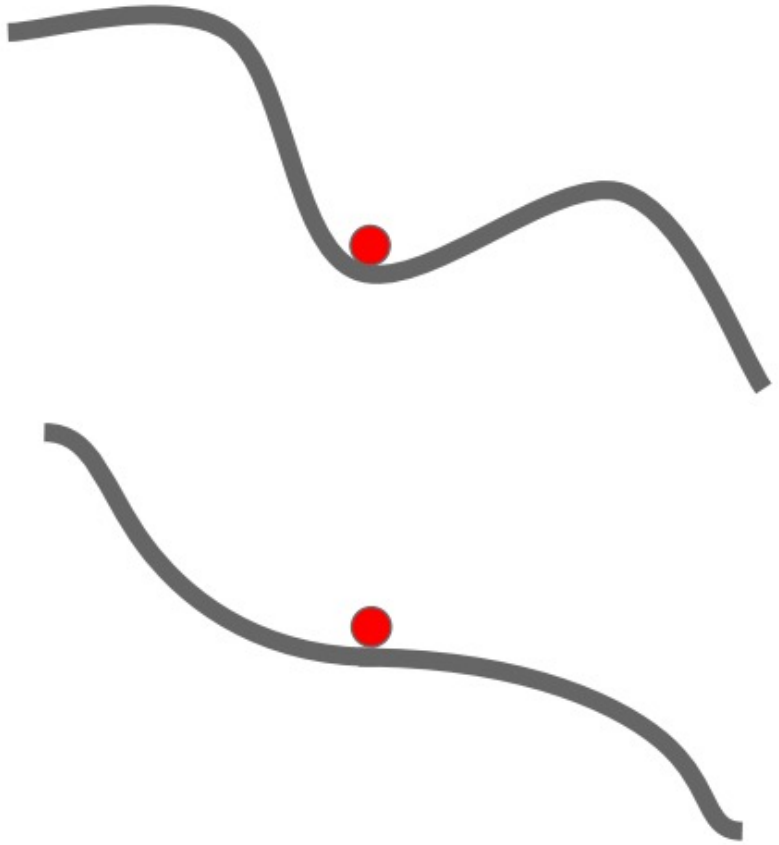
Problem #2 with SGD

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



Problem #2 with SGD

- What if the loss function has a **local minima or saddle point**?
- Zero gradient, gradient descent gets stuck
- Saddle points often appear with multivariable objective functions

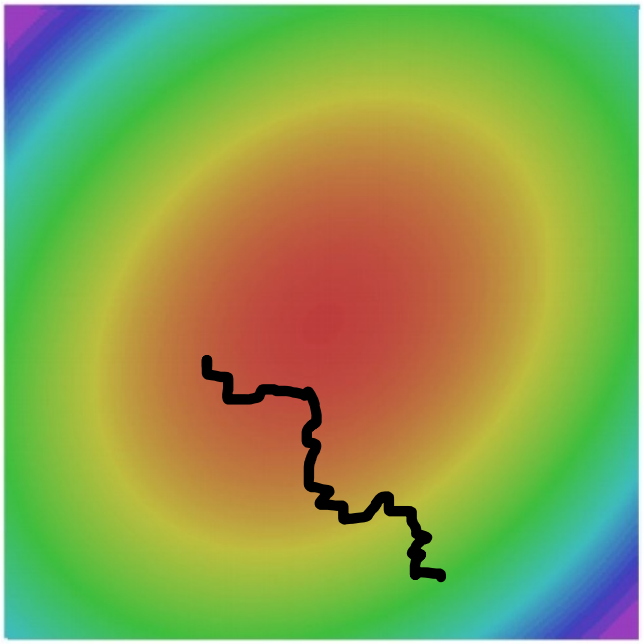


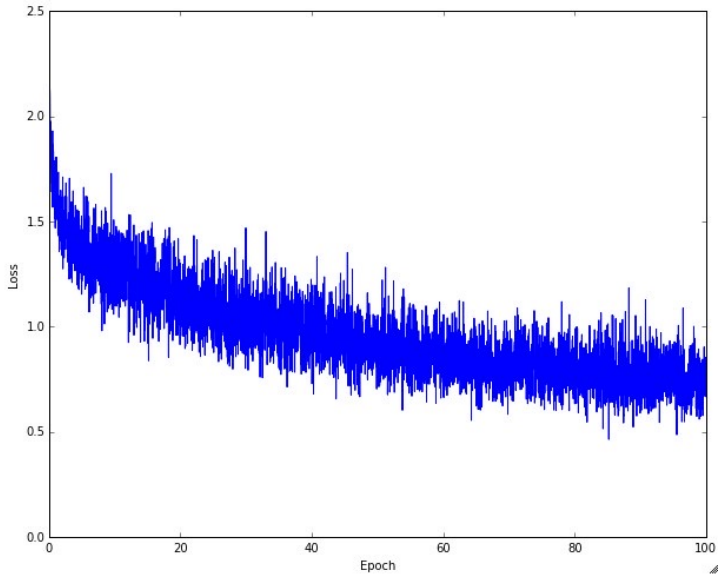
Problem #3 with SGD

- 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

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
```

- Continue moving in the general direction as the previous iterations
- Build up “velocity” as a running mean of gradients
- Rho gives “friction”; typically, rho=0.9 or 0.99
- At the beginning, rho the score may be lower due to unclear redirection, e.g., rho = 0.5

SGD + momentum (2)

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
```

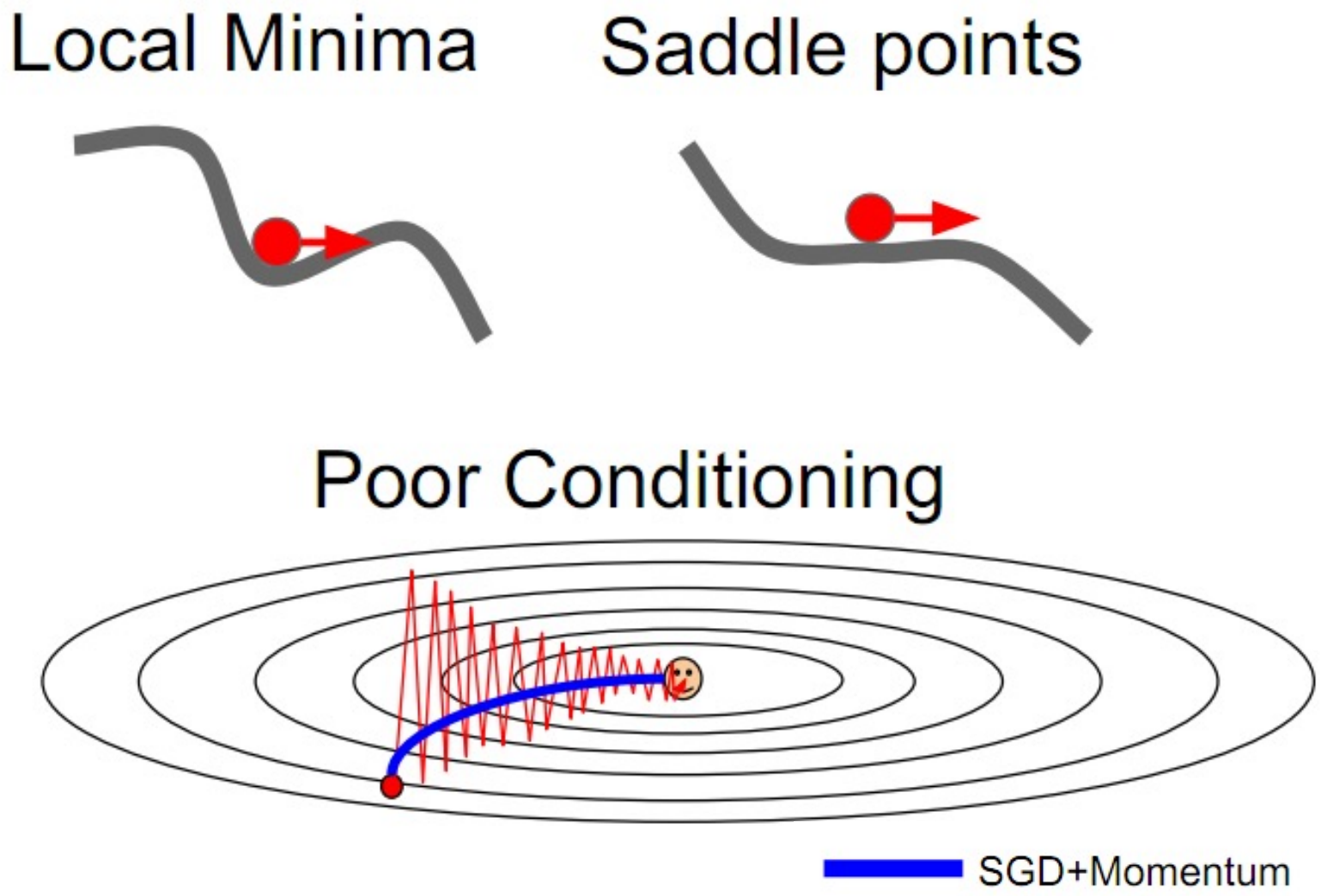
SGD+Momentum

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

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

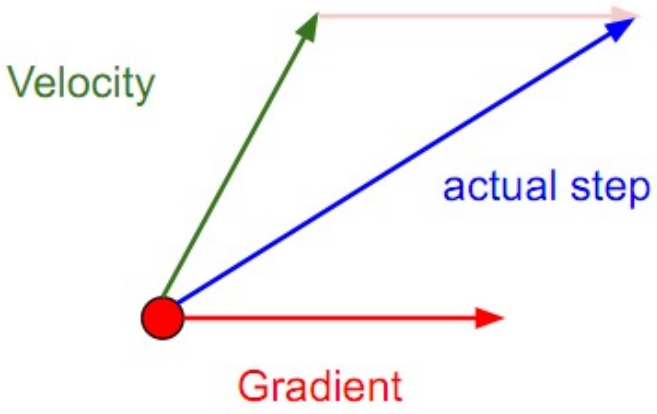
- Alternative equivalent formulation
- You may see SGD+Momentum formulated different ways,
- But they are equivalent - give same sequence of x

SGD + momentum (3)



Nesterov Momentum

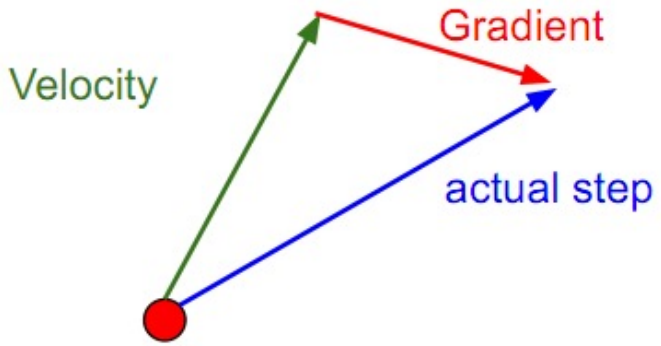
Momentum update:



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

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

Nesterov Momentum

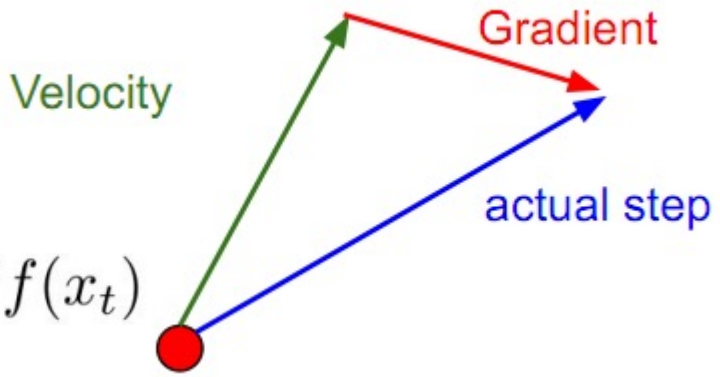


$$v_{t+1} = \rho v_t - \alpha \nabla f(x_t + \rho v_t)$$
$$x_{t+1} = x_t + v_{t+1}$$

"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 (2)

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

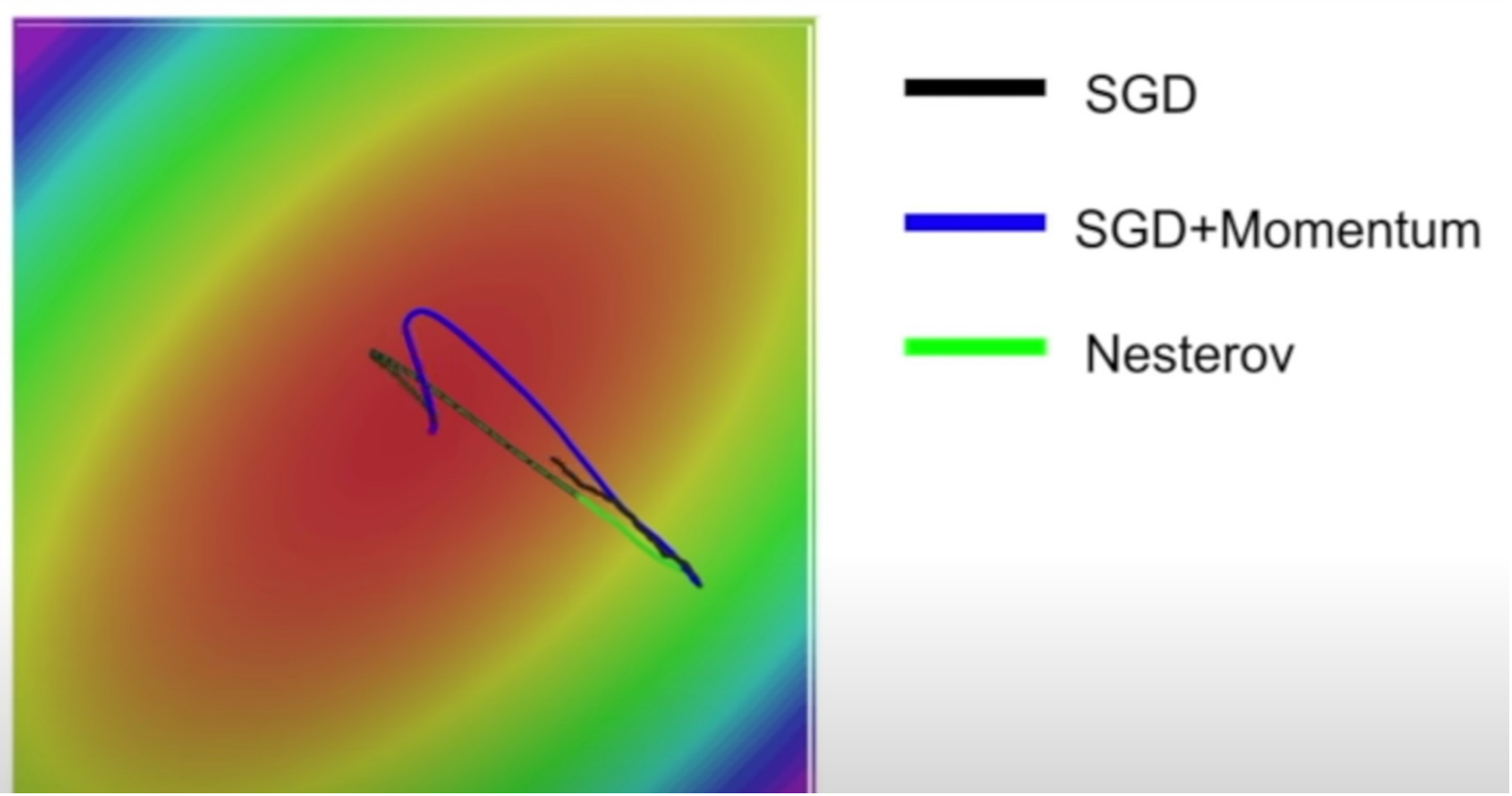


- Annoying, usually we want update in terms of $x_t, \nabla f(x_t)$
- Let $\tilde{x}_t = x_t + \rho v_t$ and rearrange

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

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

Nesterov Momentum (3)



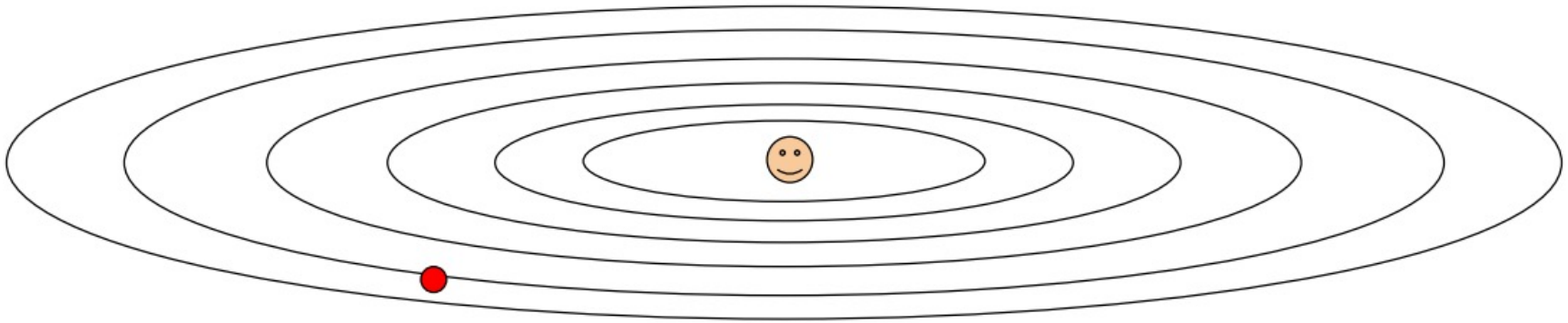
AdaGrad

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

- Added element-wise scaling of the gradient based on the historical sum of squares in each dimension
- “Per-parameter learning rates” or “adaptive learning rates”

AdaGrad

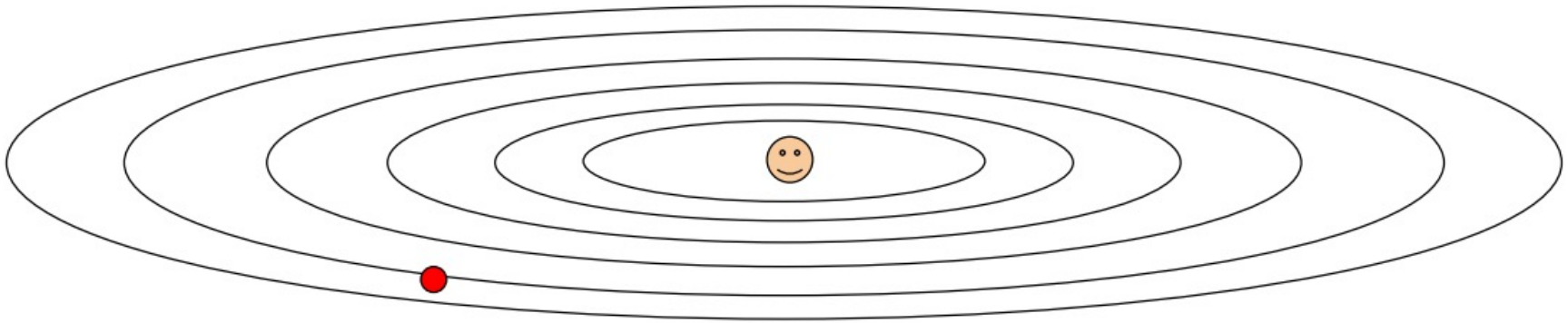
```
grad_squared = 0
while True:
  dx = compute_gradient(x)
  grad_squared += dx * dx
  x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```



- Q1: What happens with AdaGrad?

AdaGrad

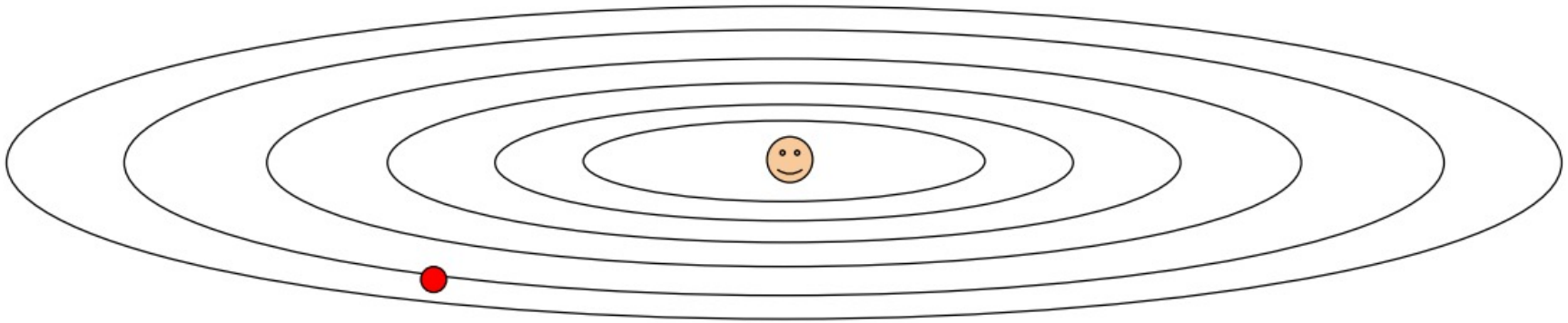
```
grad_squared = 0
while True:
  dx = compute_gradient(x)
  grad_squared += dx * dx
  x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```



- Q1: What happens with AdaGrad?
- Progress along “steep” directions is damped
- Progress along “flat” directions is accelerated

AdaGrad

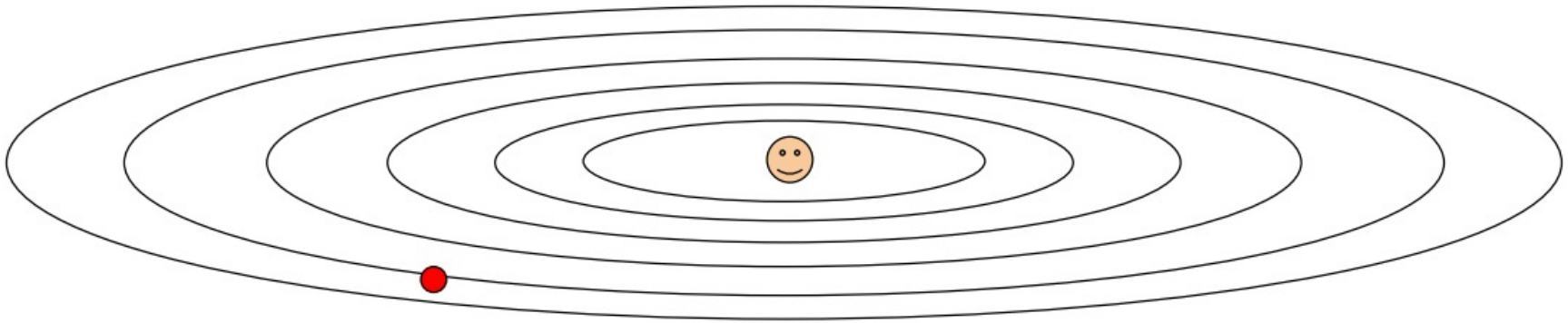
```
grad_squared = 0
while True:
  dx = compute_gradient(x)
  grad_squared += dx * dx
  x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```



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

AdaGrad

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



- Q2: What happens to the step size over long time?
- Decays to zero
- The learning rate is monotonically smaller

RMSProp: “Leaky AdaGrad”

AdaGrad

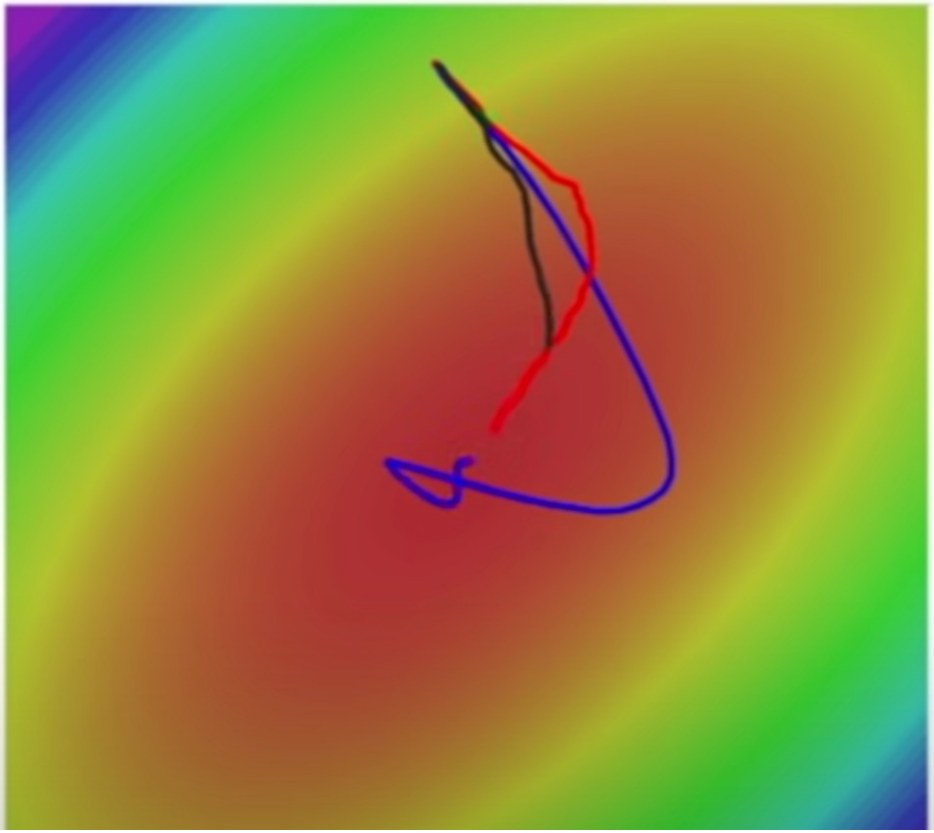
```
grad_squared = 0
while True:
  dx = compute_gradient(x)
  grad_squared += dx * dx
  x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```

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)
```

RSMProp use a moving average of squared gradients
Recommendation: decay_rate = [0.9, 0.99, 0.999]
Unlike Adagrad the updates do not get monotonically smaller

RSMProp



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))
```

Momentum

AdaGrad / RMSProp

Sort of like RMSProp with momentum

Q: What happens at first timestep?

beta1 = 0.9, beta2 = 0.999

First and second moment start at zero

Adam (full form)

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

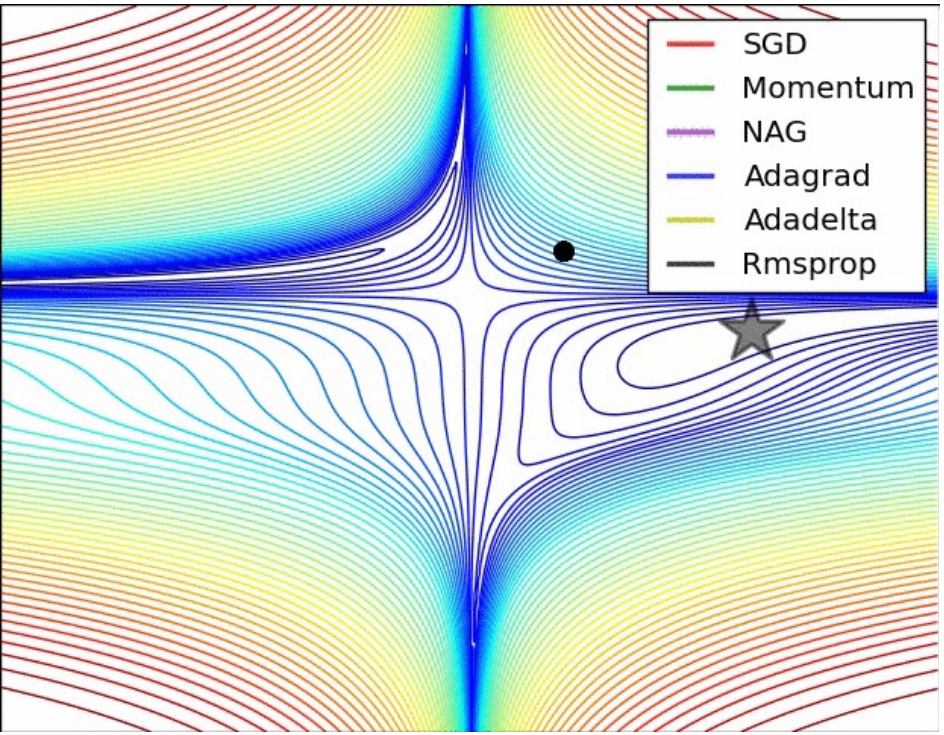
Momentum

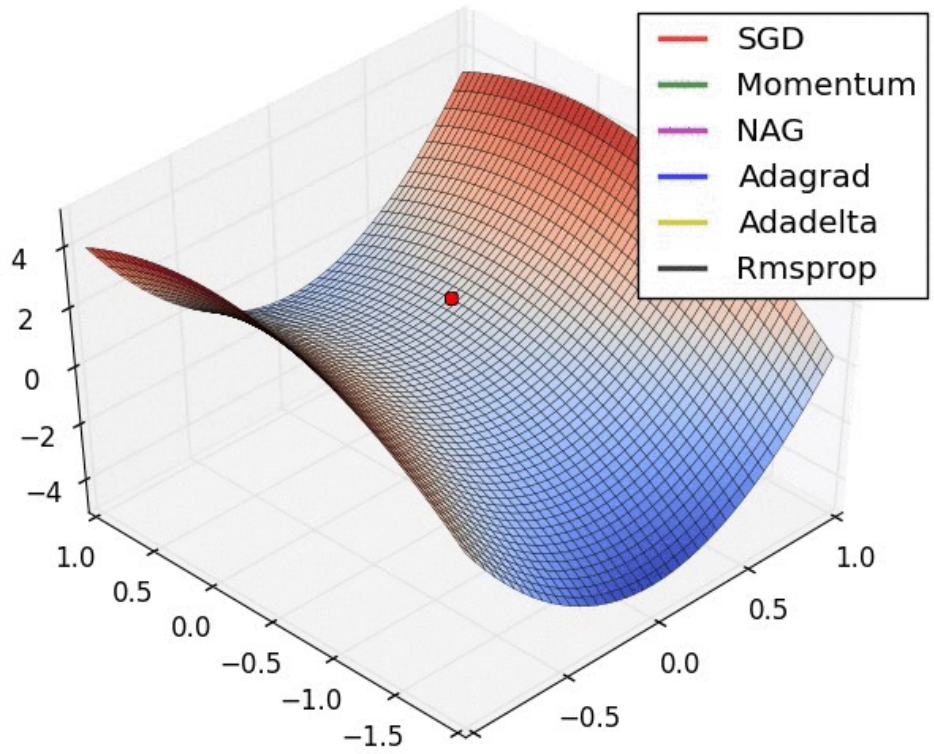
Bias correction

AdaGrad / RMSProp

- Bias correction for the fact that first and second moment estimates start at zero. It makes the algorithm more stable during warm up at the first few steps.
- Adam with beta1 = 0.9, beta2 = 0.999, and learning_rate = 1e-3 or 5e-4 are good default parameters for many models!

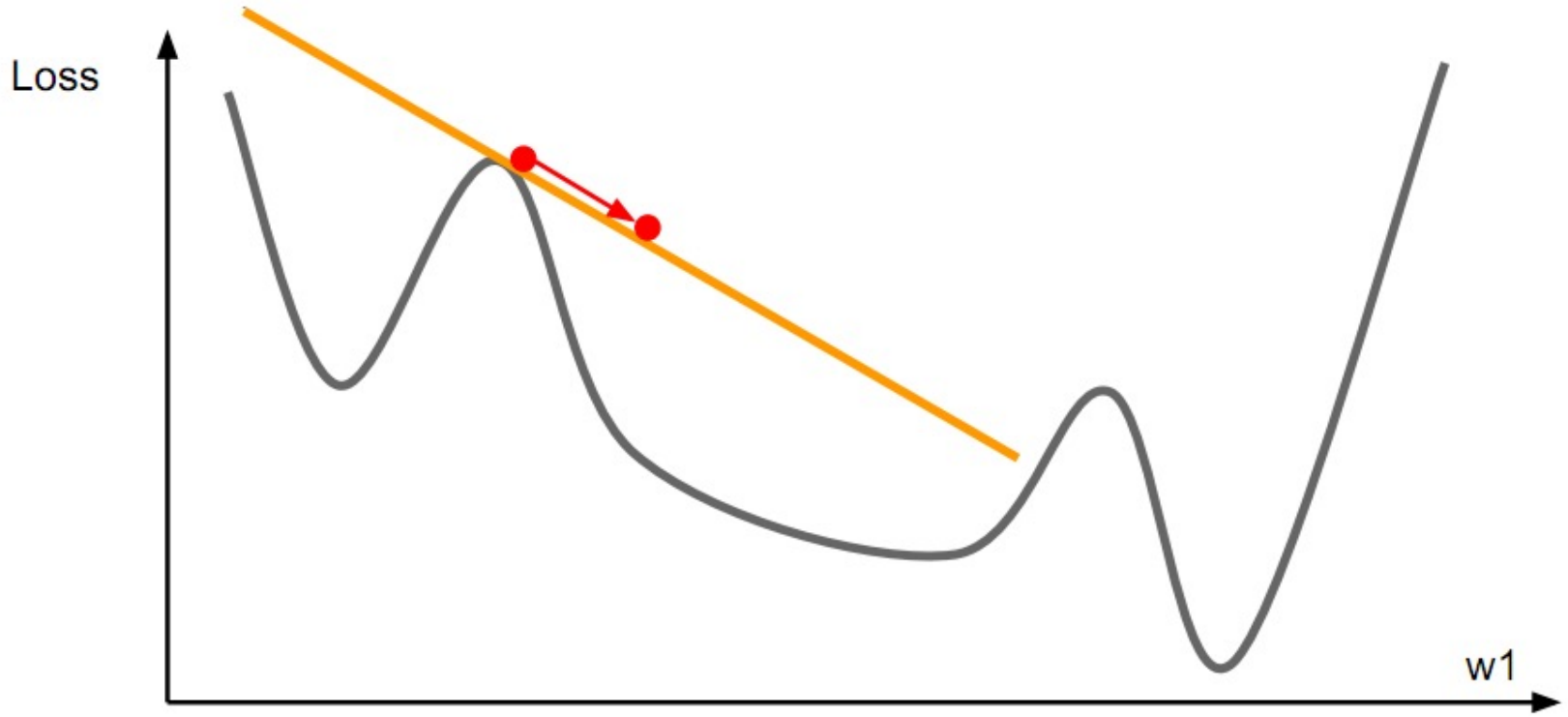
Visual examples of the learning process





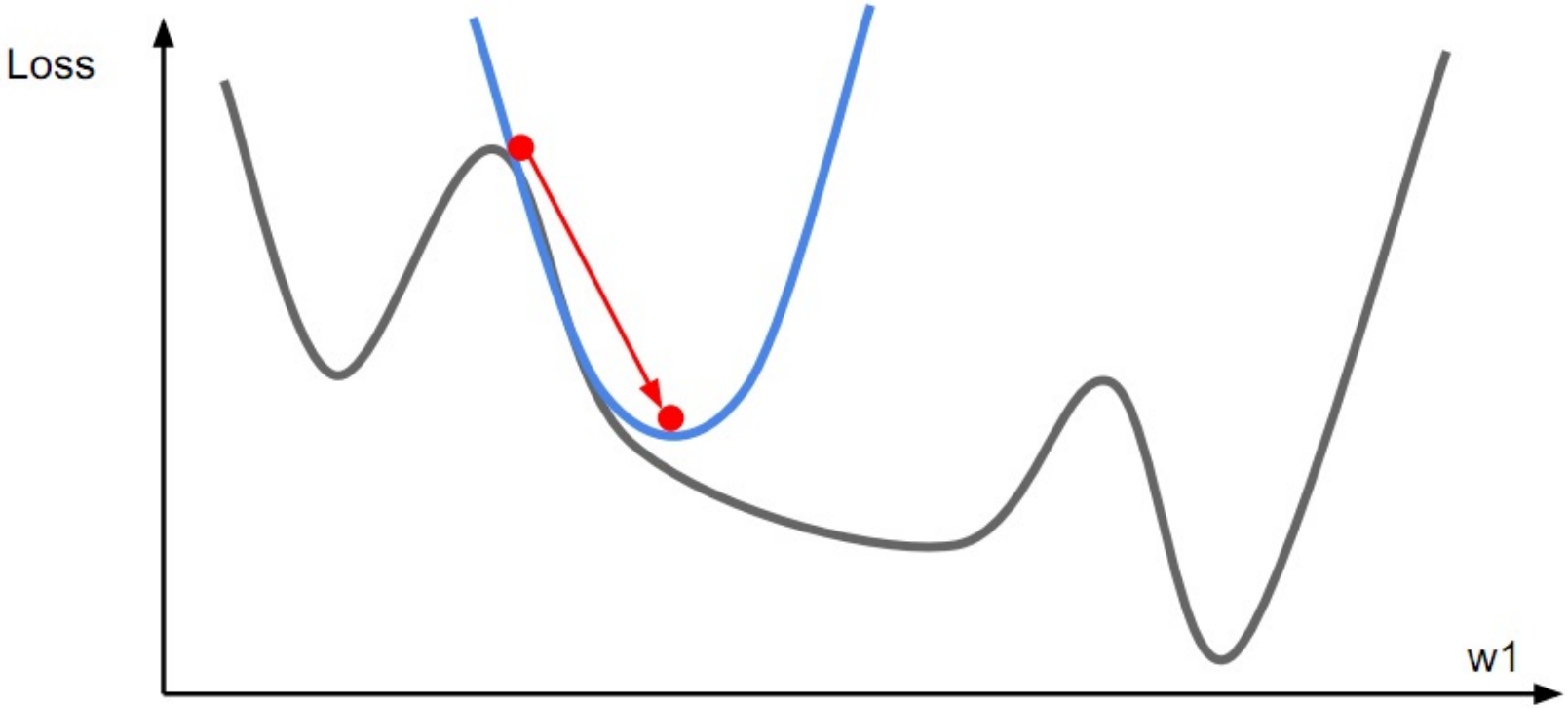
(c) Alec Radford.

First-order optimization



Second-order optimization

- Using the Hessian, which is a square matrix of second-order partial derivatives of the function.



computing (and inverting) the Hessian in its explicit form is a very costly process in both space and time

Second-order optimization

- Taylor expansion

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

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

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

- Not good for DL (due to matrix inverse complexity of O(n^3))
  - Hessian has O(N^2) elements
  - Inverting takes O(N^3)
  - N = (Tens or Hundreds of) Millions
- Quasi-Newton (BGFS)
  - 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
- **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.

SOTA optimizers

- NAdam = Adam + Nesterov's Accelerated Gradient (NAG)
- RAdam (Rectified Adam)
- LookAhead
- Ranger = RAdam + LookAhead

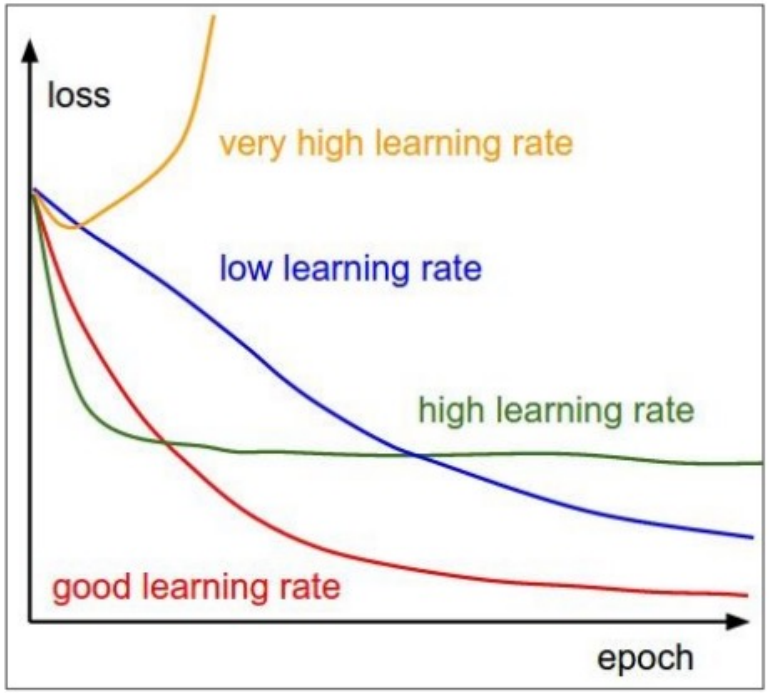
In practice

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

Learning rate schedules

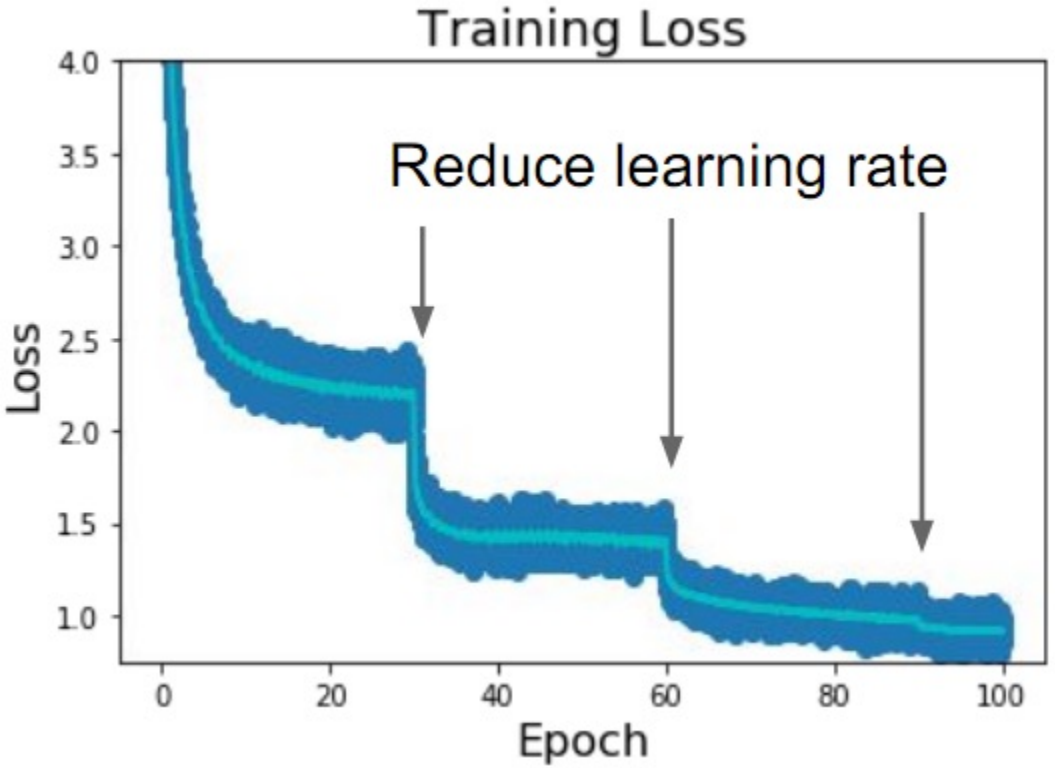
Learning rate

- SGD, SGD+Momentum, Adagrad, RMSProp, Adam all have learning rate as a hyperparameter.
- Q: Which one of these learning rates is best to use?
  - Usually starts with a large value and decreases over time



Learning rate decays over time

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



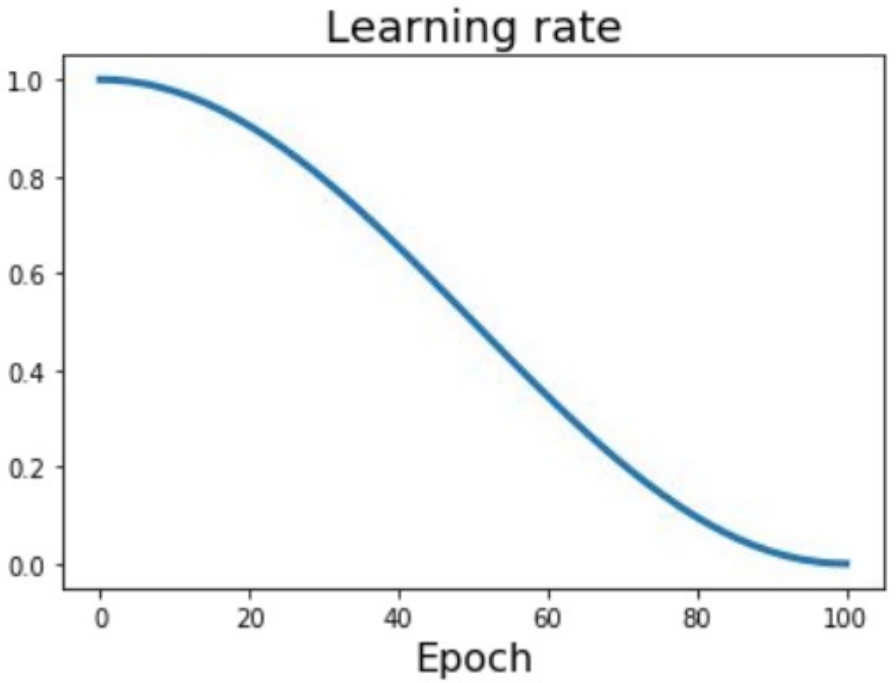
Cosine rate decay

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

α_0 : Initial learning rate

α_t : Learning rate at epoch t

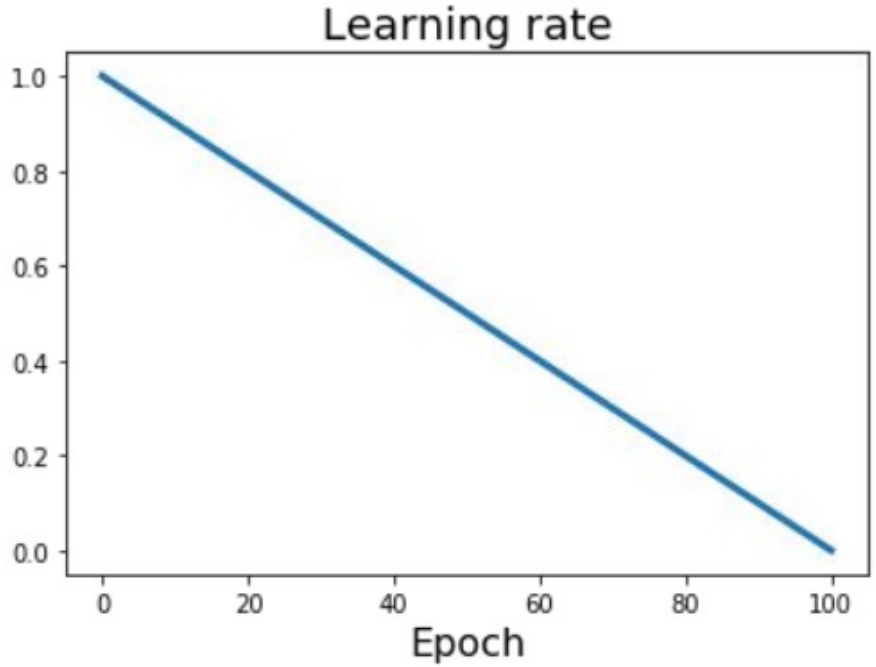
T : Total number of epochs



Linear rate decay

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

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



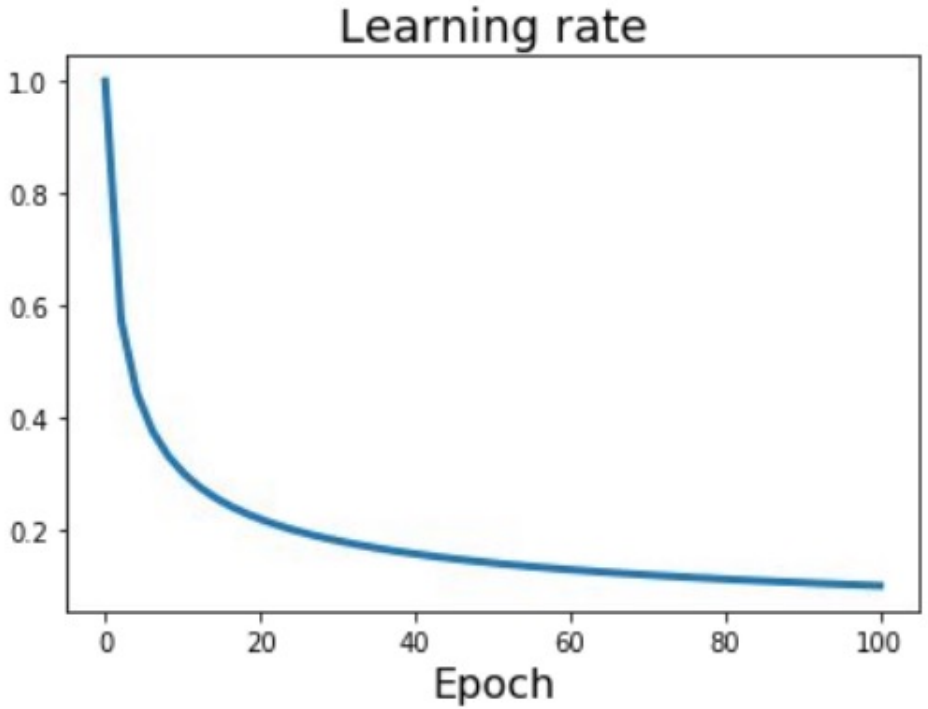
Inverse sqrt of total number of epochs

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

α_0 : Initial learning rate

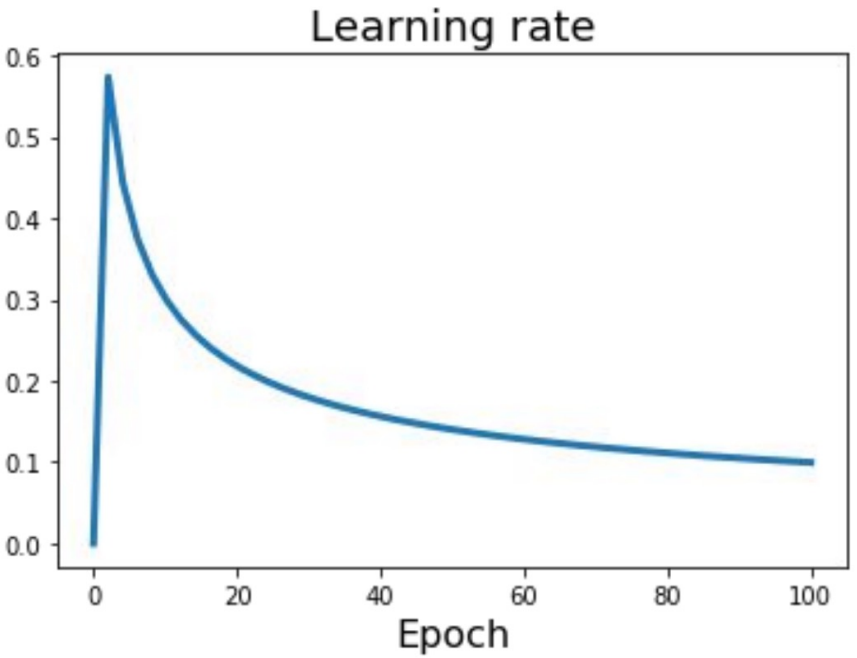
α_t : Learning rate at epoch t

T : Total number of epochs



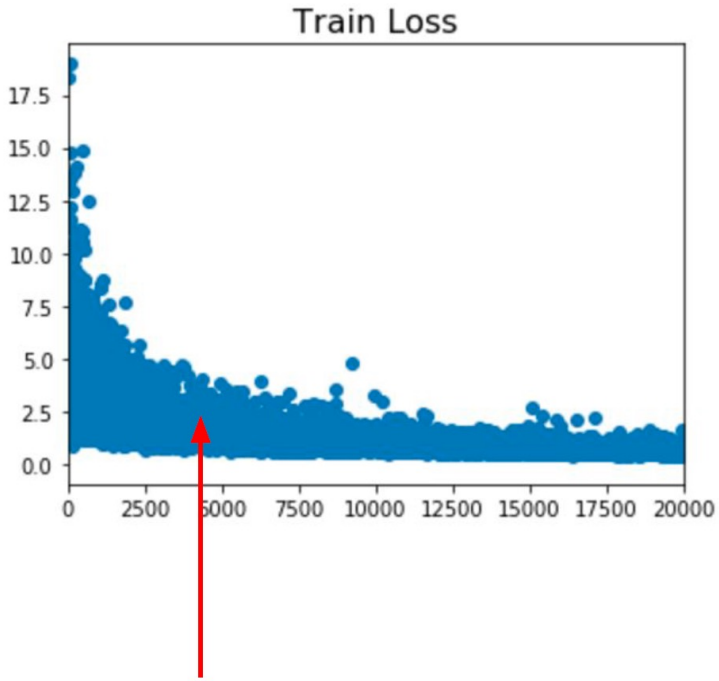
Linear Warmup

- 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

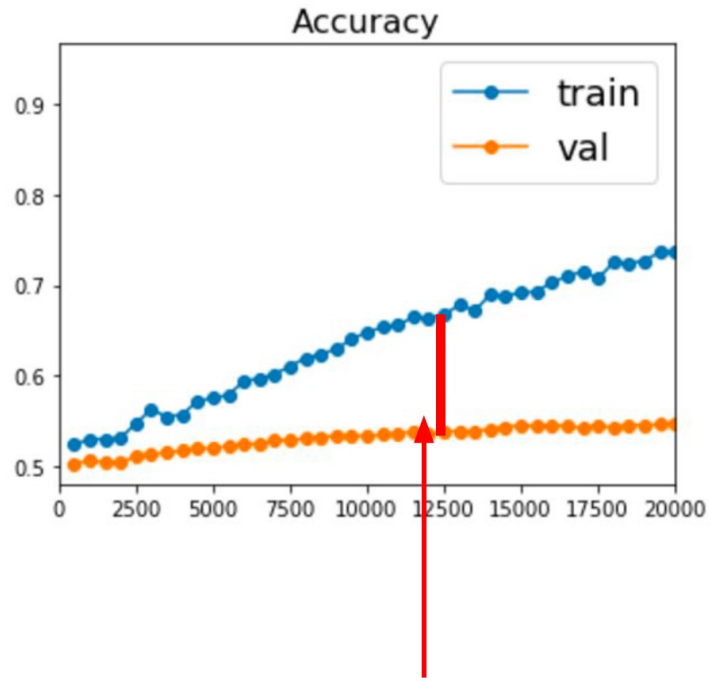


Anti-overfitting techniques

Beyond Training Error



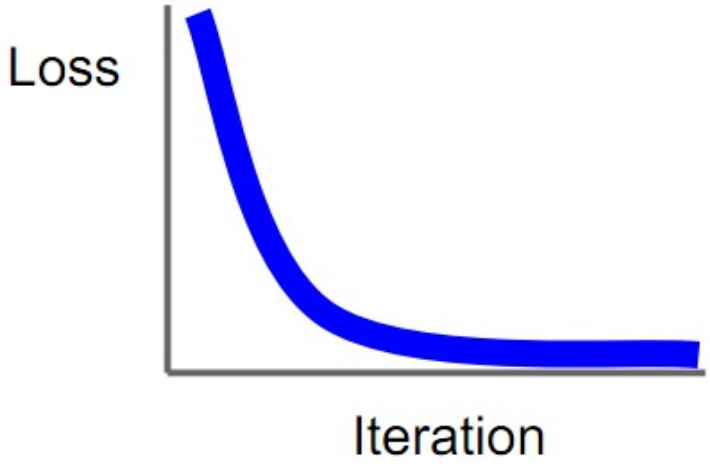
Better optimization algorithms help reduce training loss

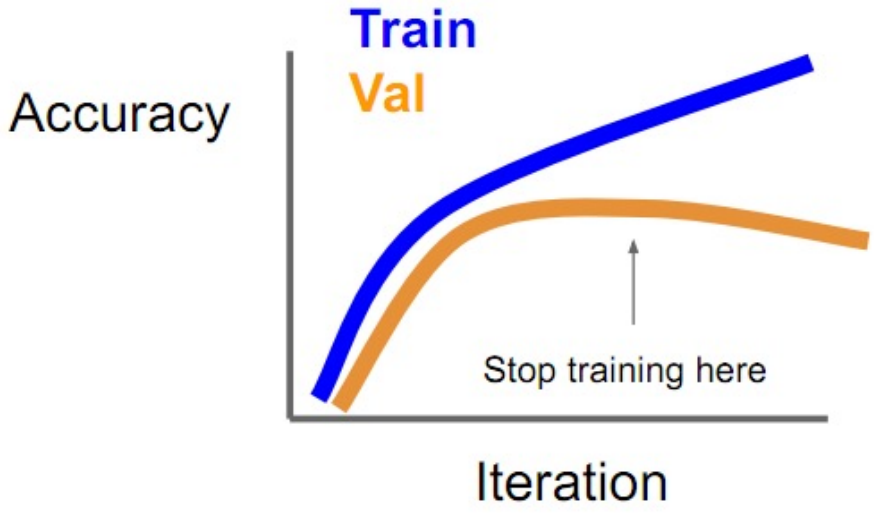


But we really care about error on new data - how to reduce the gap?

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





Regularization: Add term to loss

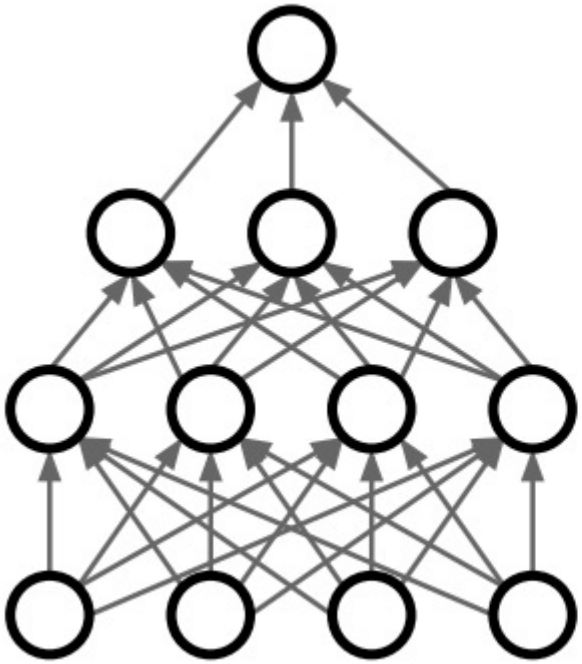
$$L = \frac{1}{N}\sum_{i=1}^{N}\sum_{j\neq y_i}\max(0, f(x_i;W)_j - f(x_i;W)_{y_i} + 1) + \boxed{\lambda R(W)}$$

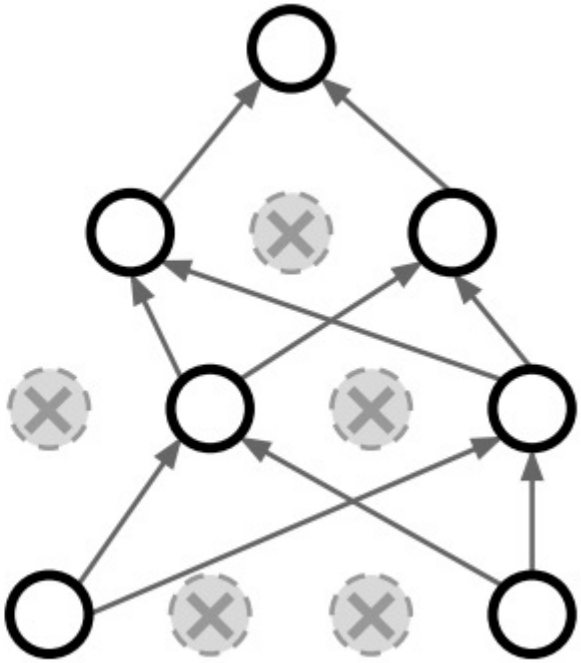
Common regularization

L2 regularization	$R(W) = \sum_k \sum_l W_{k,l}^2$	(Weight decay)
L1 regularization	$R(W) = \sum_k \sum_l \lvert W_{k,l}\rvert$	
Elastic net (L1 + L2)	$R(W) = \sum_k \sum_l \beta W_{k,l}^2 + \lvert W_{k,l}\rvert$	

Regularization: Dropout

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





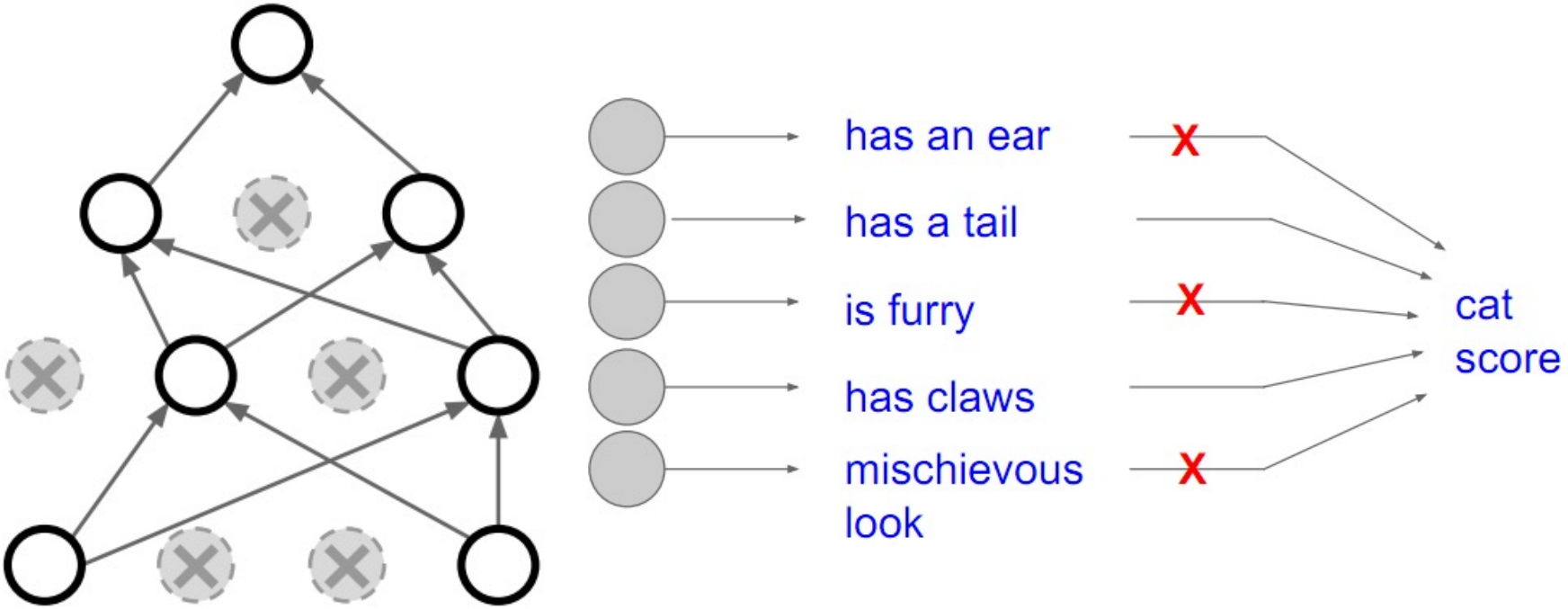
Regularization: Dropout

- Example forward pass with a 3-layer network using 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)
```

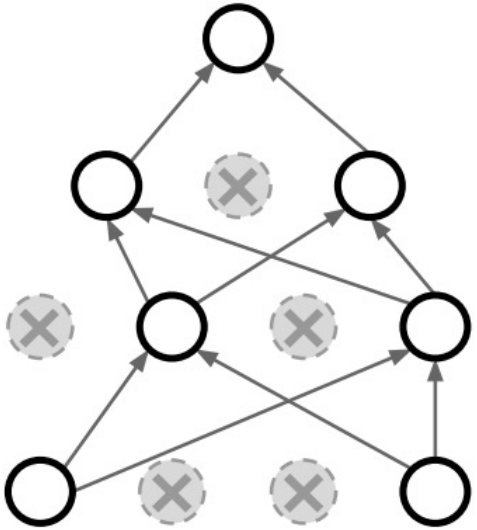
Dropout effects

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



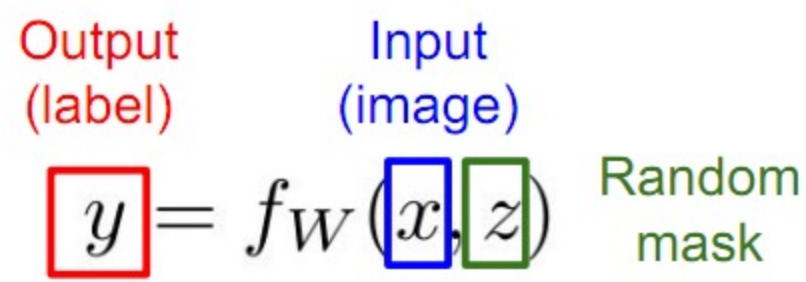
Dropout effects (2)

- 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 10^{82} atoms in the universe!



Dropout: Test time

- Dropout makes our output random!



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

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

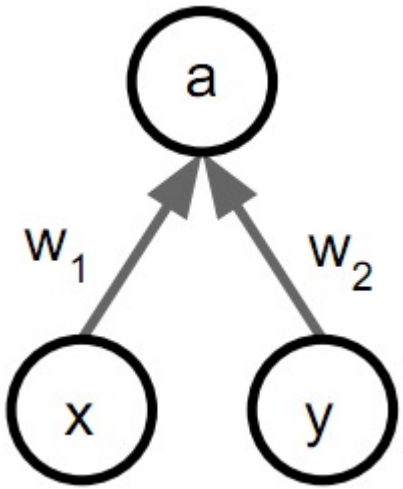
- But this integral seems hard …

Dropout: Test time (2)

- 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:

$$\begin{aligned} E\big[a\big] =& \frac{1}{4}(w_1 x + w_2 y) + \frac{1}{4}(w_1 x + 0y) \\ &+ \frac{1}{4}(0x + 0y) + \frac{1}{4}(0x + w_2 y) \\ =& \frac{1}{2}(w_1 x + w_2 y) \end{aligned}$$

Dropout: Test time (3)

- 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
- At test time, multiply by dropout probability

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

Dropout Summary

```
""" 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

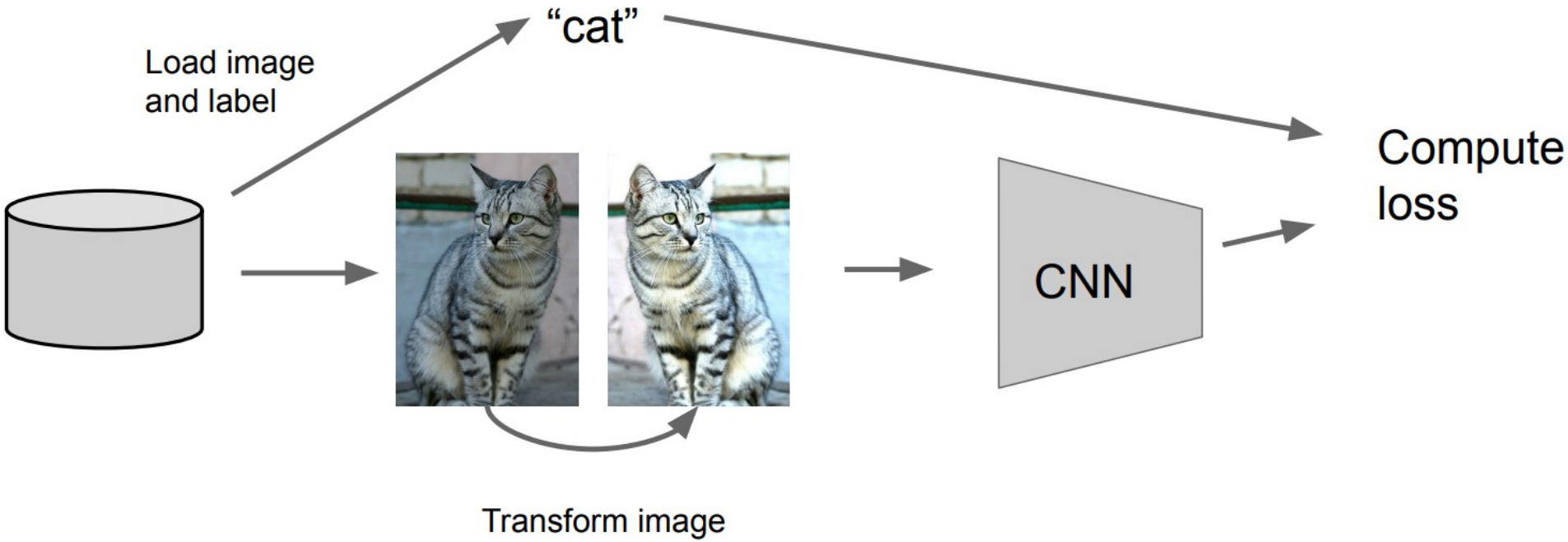
More common: “Inverted dropout”

```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!

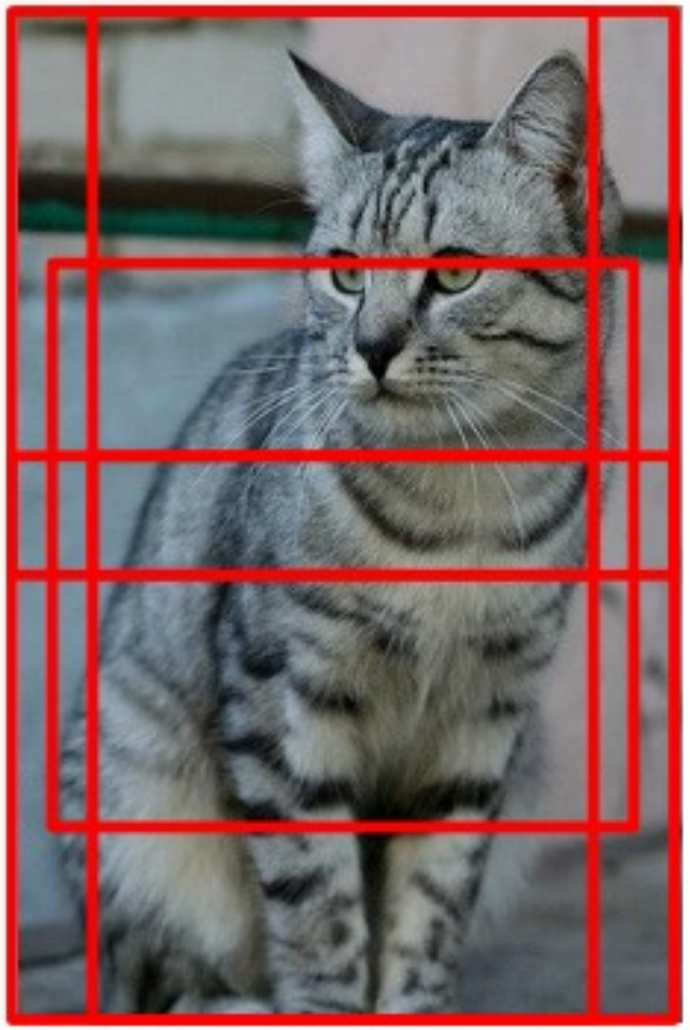
Data Augmentation

Horizontal Flip



Random crops and scales

- Training: sample random crops / scales ResNet:
  - Pick random L in range [256, 480]
  - Resize training image, short side = L
  - Sample random 224 x 224 patch
- Testing: average a fixed set of crops ResNet:
  - Resize image at 5 scales: {224, 256, 384, 480, 640}
  - For each size, use 10 224 x 224 crops: 4 corners + center, + flips

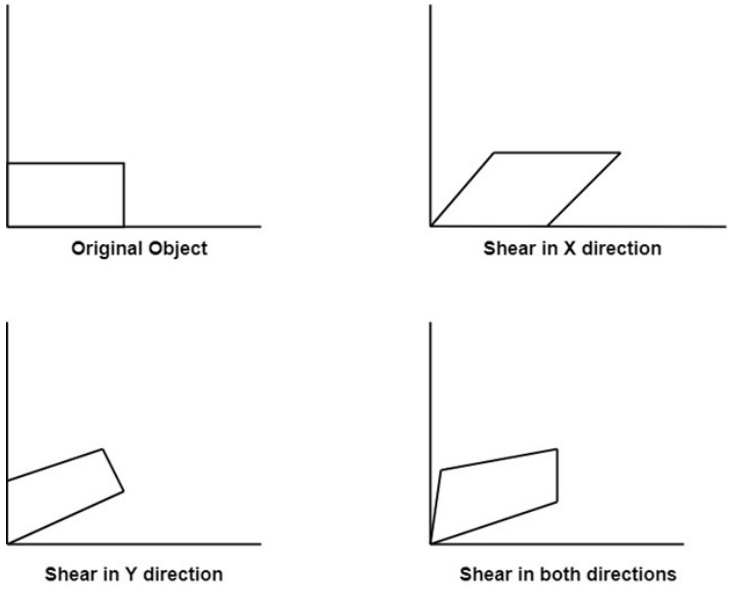


Color Jitter

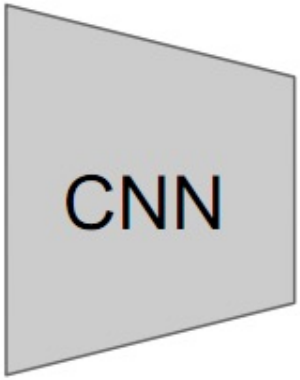
Simple:
Randomize
contrast and
brightness

Other transformations

- Translation
- Rotation
- Stretching
- Shearing
- lens distortions
- … (go crazy)



Mixup



Target label:
cat: 0.4
dog: 0.6

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

Some libraries

1. Albumentations

https://github.com/albumentations-team/albumentations

2. Imgaug

https://github.com/aleju/imgaug

3. Augmentor

https://github.com/mdbloice/Augmentor

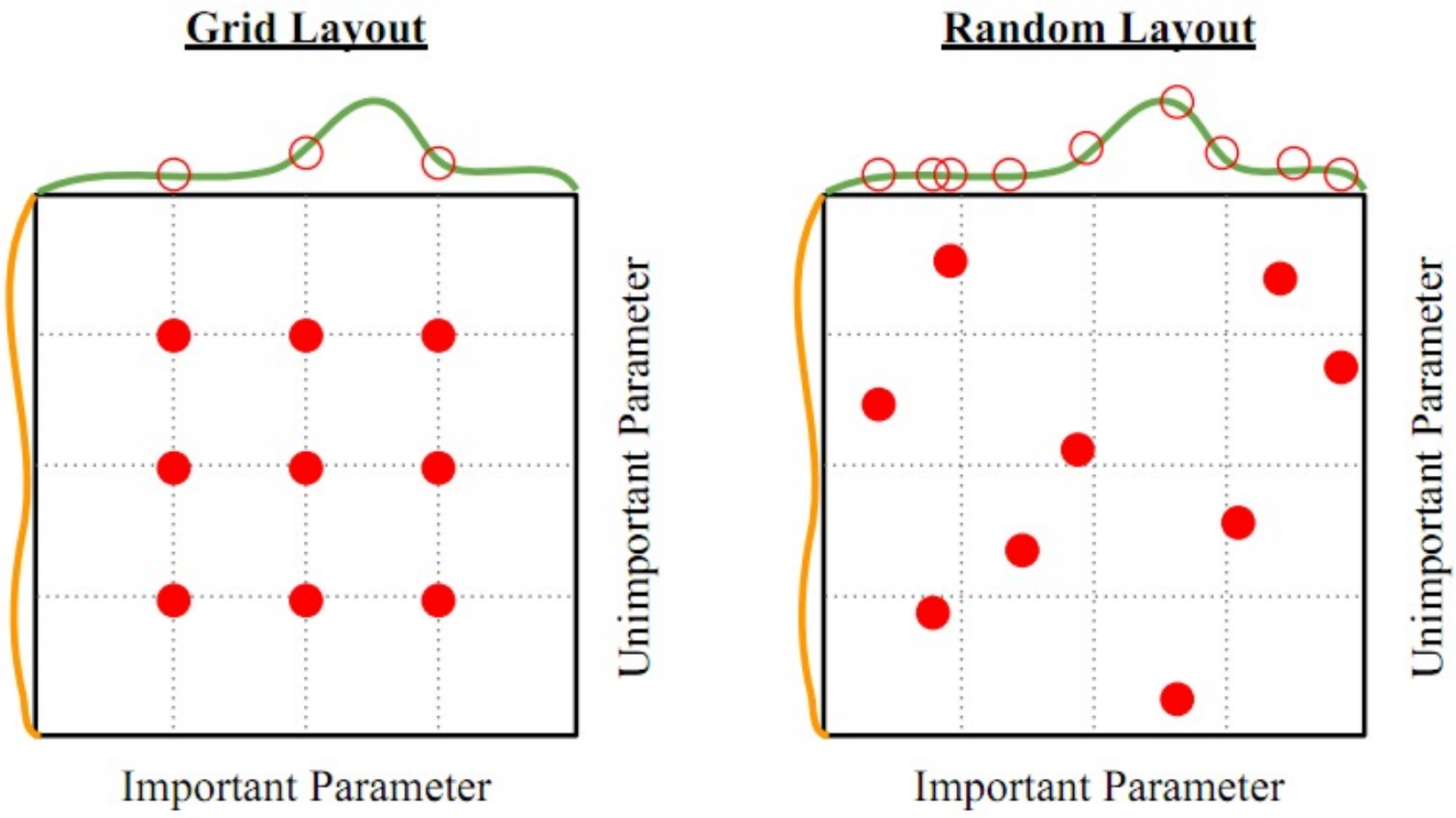
Choosing hyperparameters

Hyperparameters

- Network Architecture
- Learning rate, parameters in learning rate change strategy, optimization algorithm
- Control coefficients (L2 weight decay, drop rate)

neural networks practitioner
music = loss function

Random Search vs Grid Search



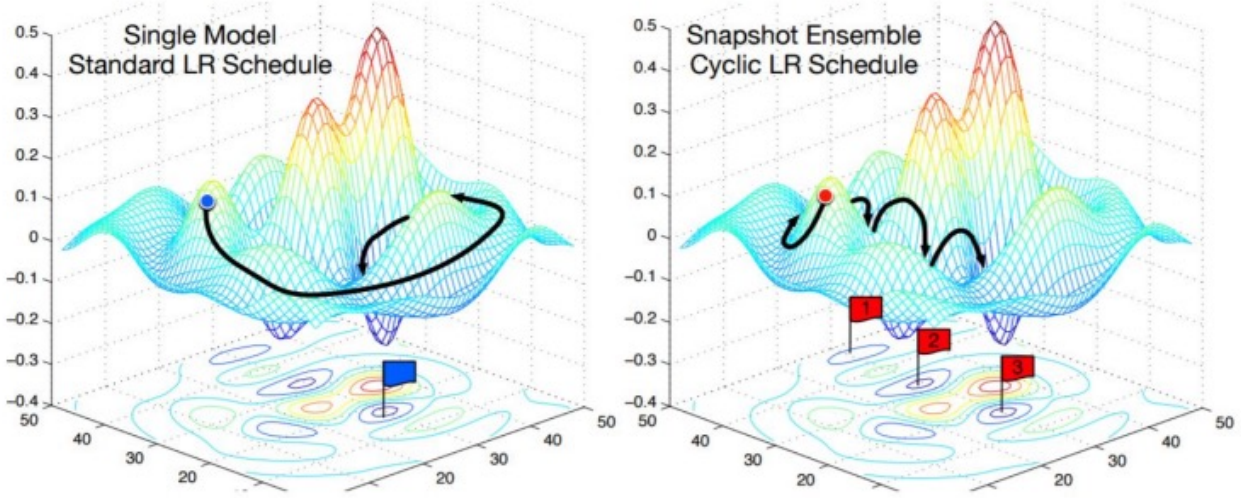
Techniques for combining multiple models (ensemble methods)

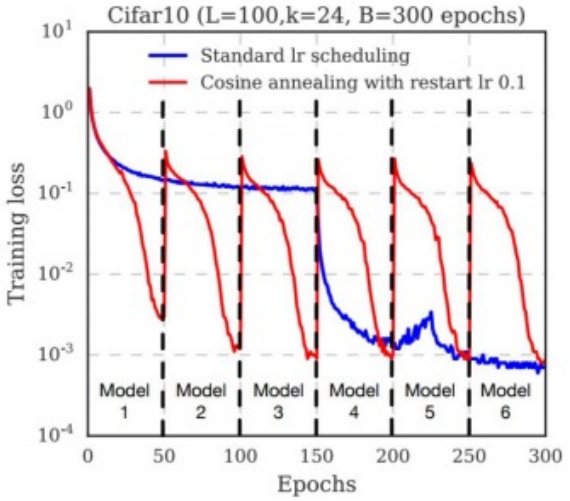
Model Ensembles

- Train multiple independent models
- At test time average their results
  - Take average of predicted probability distributions, then choose argmax
- Enjoy 2% extra performance

Model Ensembles

- Instead of training multiple models independently, multiple snapshots of the same model can be used during training

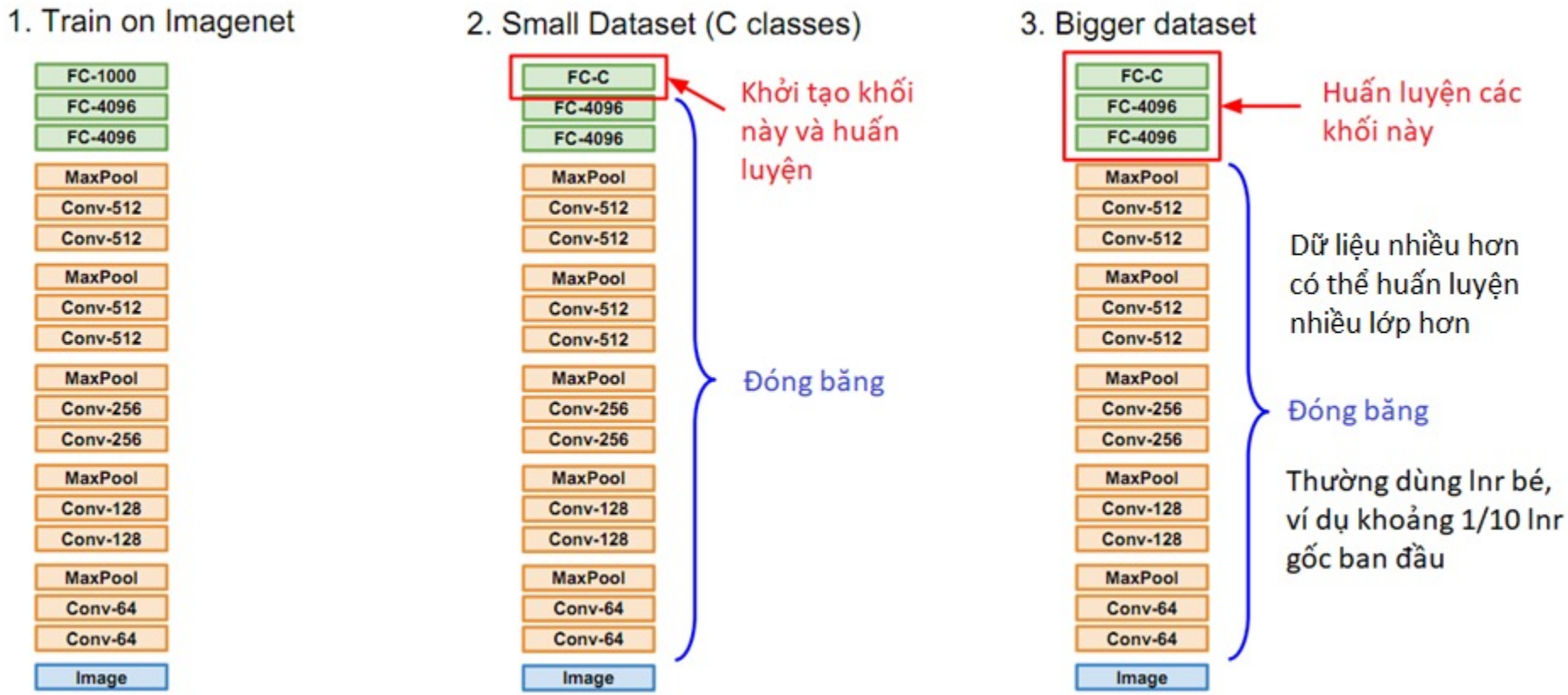




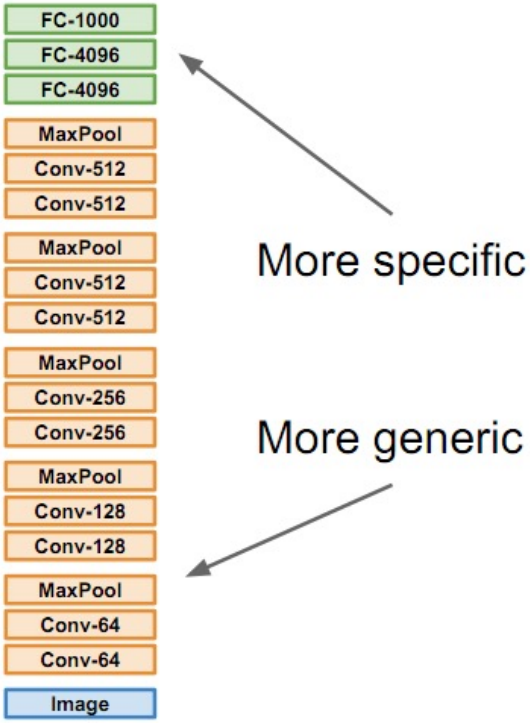
Transfer learning

Transfer learning

Train the network on a large available data set, then train with your data set



Transfer learning



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

More tips and tricks

- Machine Learning Yearning by Andrew Ng

https://d2wvfoqc9gyqzf.cloudfront.net/content/uploads/2018/09/Ng-MLY01-13.pdf

References

1. http://cs231n.stanford.edu
2. Adam:

https://towardsdatascience.com/adam-latest-trends-in-deep-learning-optimization-6be9a291375c

3. Stanford lecture note:

http://cs231n.github.io/neural-networks-3/

ĐẠI HỌC BÁCH KHOA

25 YEARS ANNIVERSARY SOICT

VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY

Thank you for your attention!!!