Машинное обучение

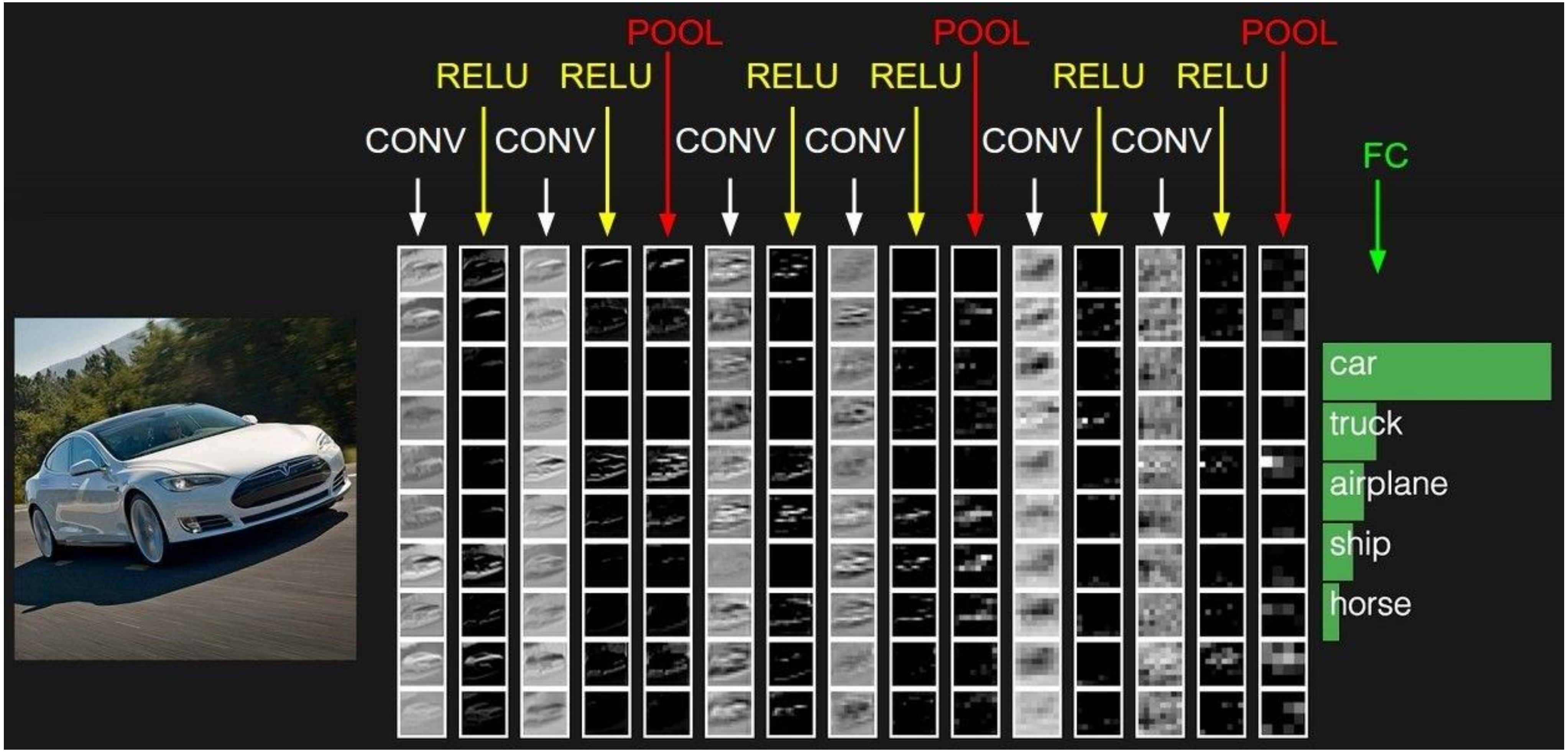
на примере глубокого обучения в компьютерного зрения

Занятие 5
Тренировка нейронных сетей часть 2
Архитектуры CNN

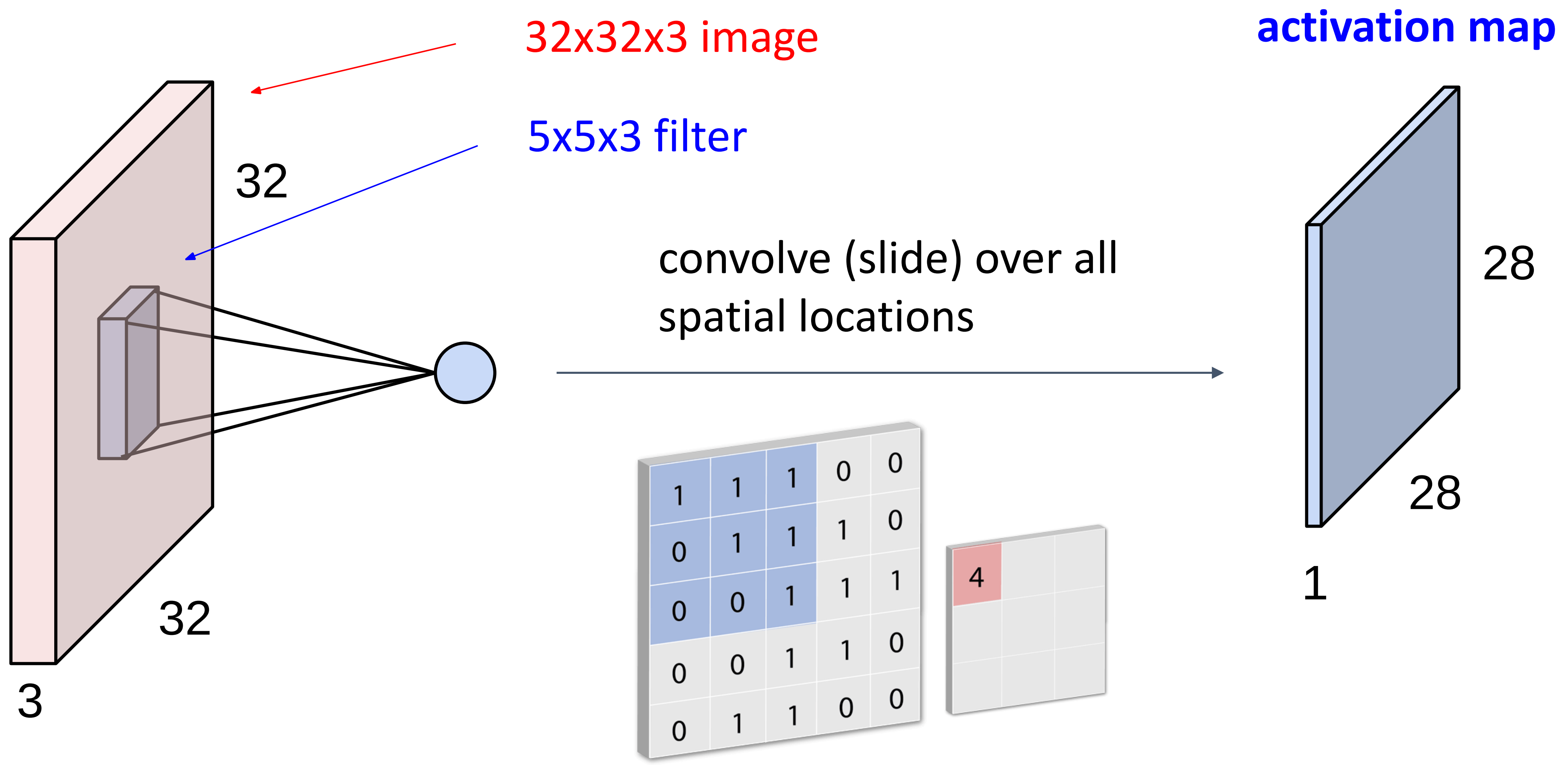
Дмитрий Яшунин, к.ф.-м.н
IntelliVision

e-mail: yashuninda@yandex.ru

На прошлом занятии:
Сверточная нейронная сеть

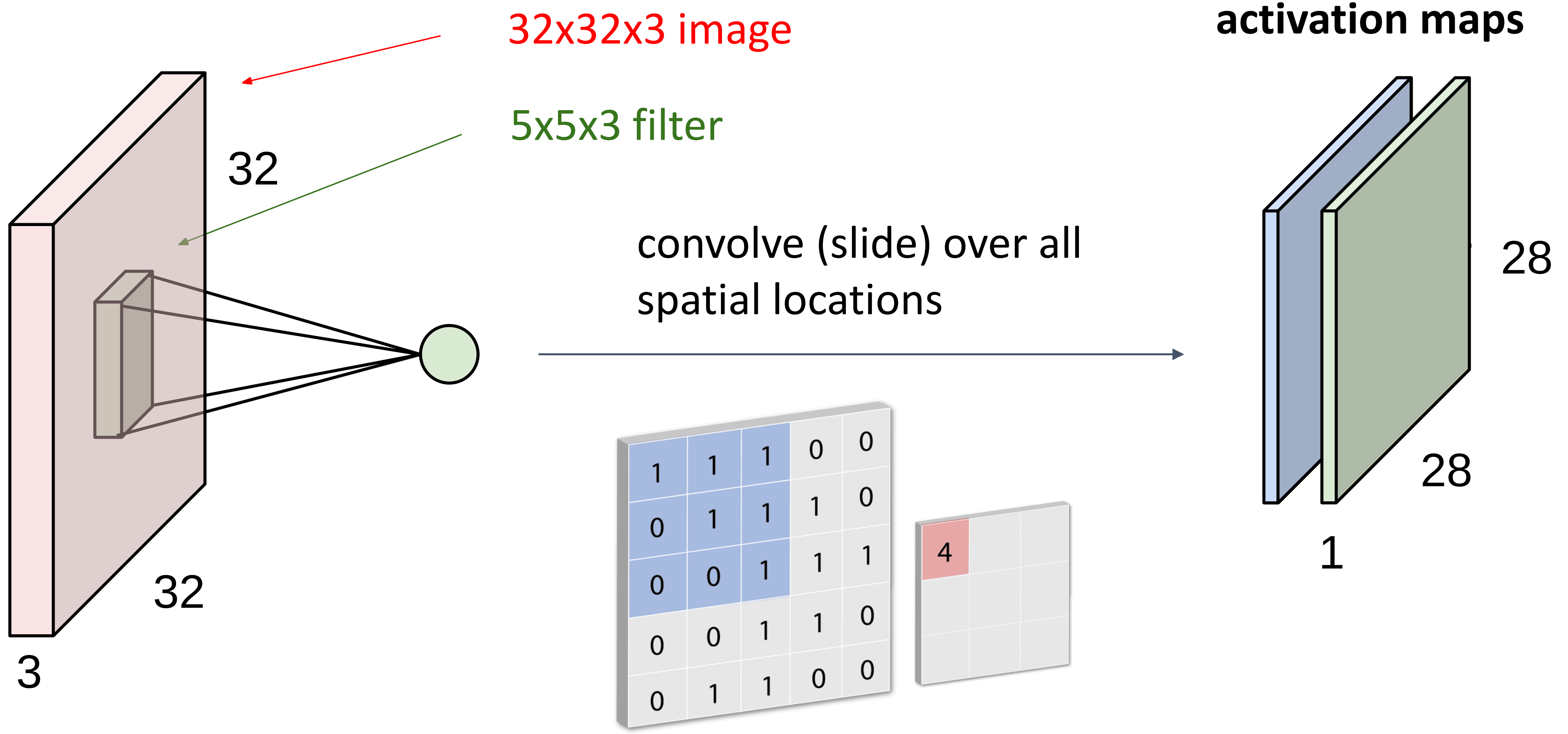


На прошлом занятии: **Convolutional Layer**



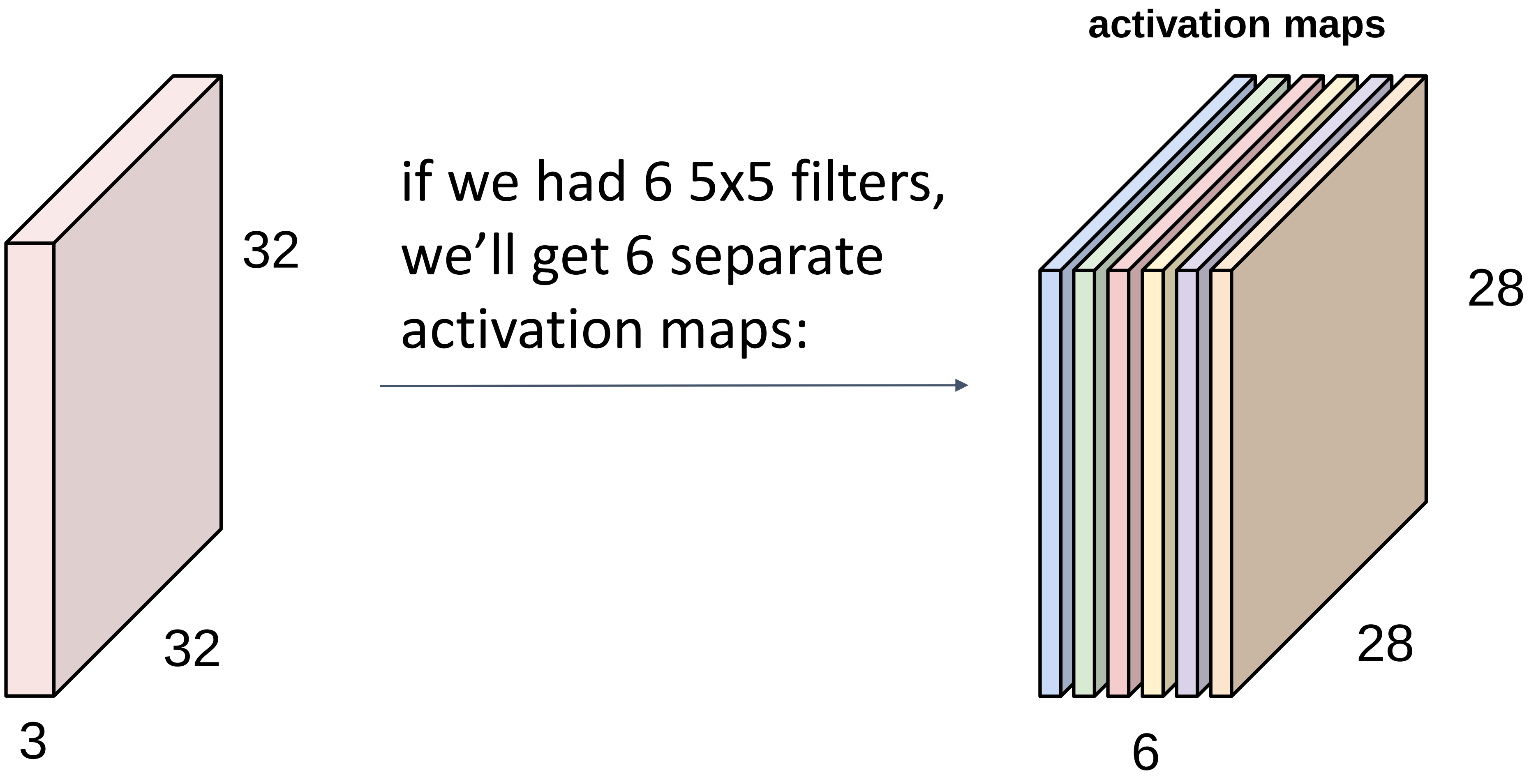
example: image 5x5, filter 3x3

На прошлом занятии: **Convolutional Layer**



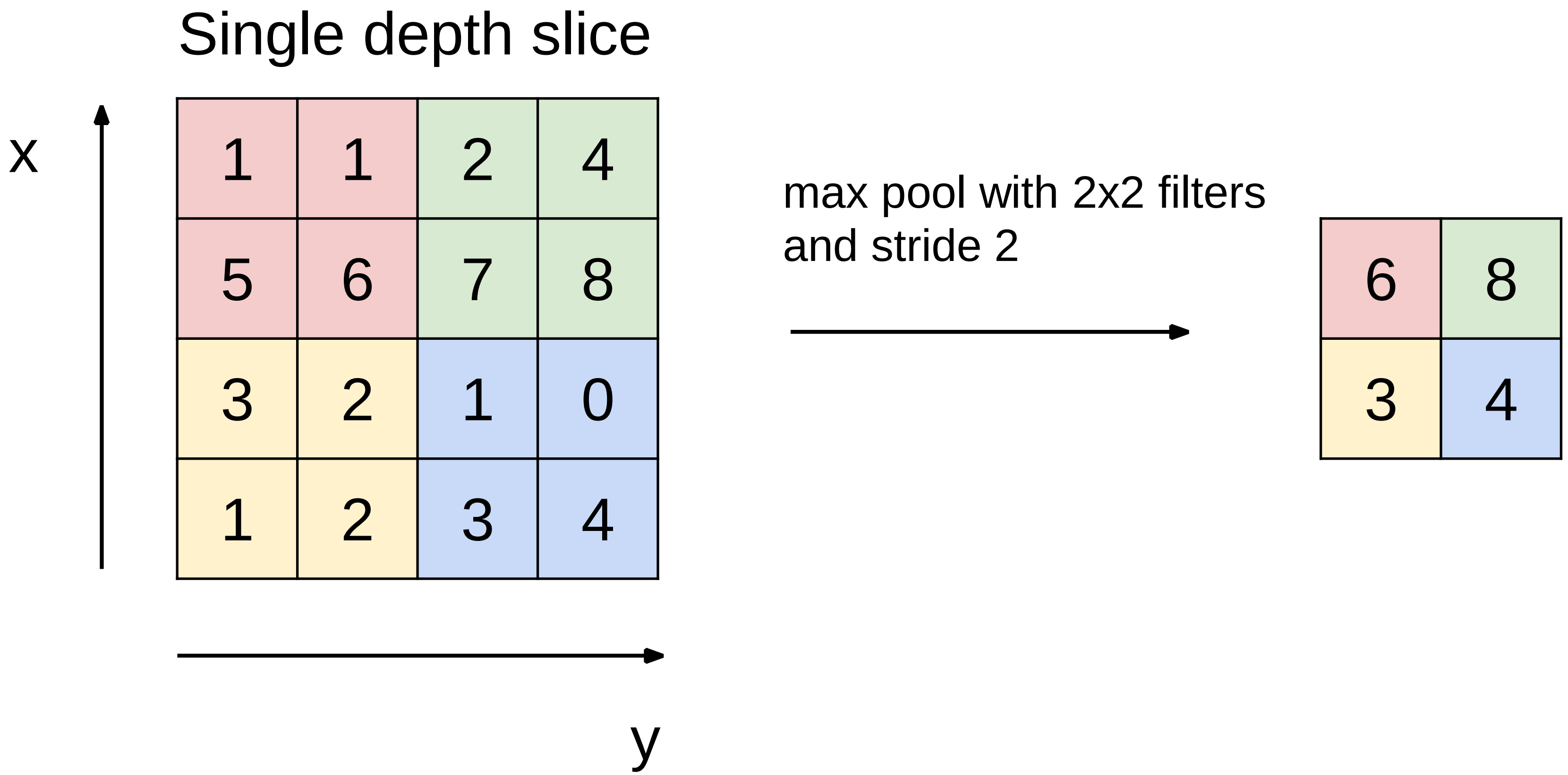
example: image 5x5, filter 3x3

На прошлом занятии: Convolutional Layer



We stack these up to get a “new image” of size 28x28x6!

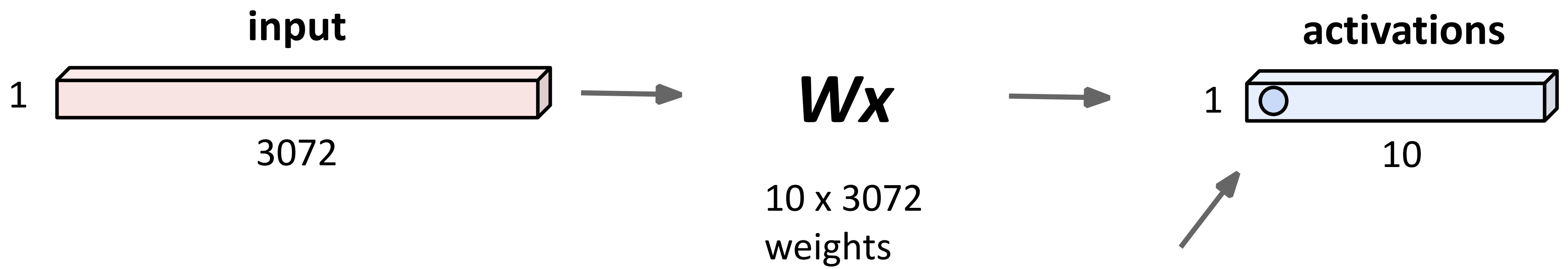
MAX POOLING


Single depth slice
x
1 1 2 4
5 6 7 8
3 2 1 0
1 2 3 4
y
max pool with 2x2 filters
and stride 2
6 8
3 4

На прошлом занятии: **Fully Connected Layer**

32x32x3 image -> stretch to 3072 x 1

Each neuron
looks at the full
input volume

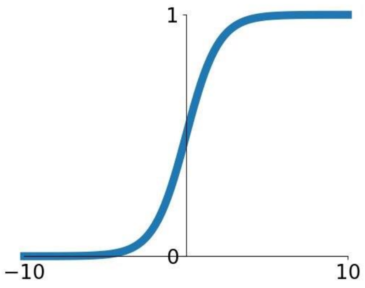


1 number:
the result of taking a dot product
between a row of W and the input
(a 3072-dimensional dot product)

На прошлом занятии: **Активационные функции**

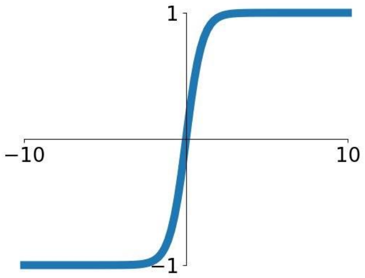
Sigmoid

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



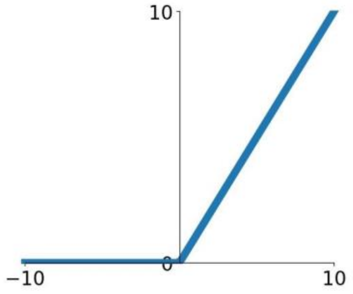
tanh

$$\tanh(x)$$



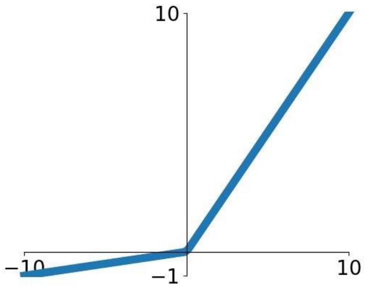
ReLU

$$\max(0, x)$$



Leaky ReLU

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

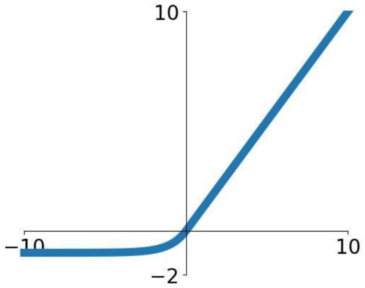


Maxout neuron

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

ELU

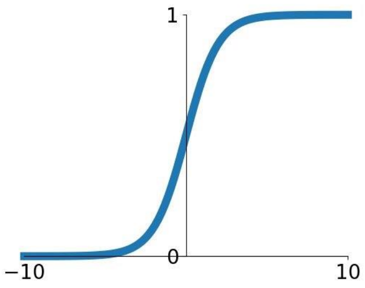
$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



На прошлом занятии: **Активационные функции**

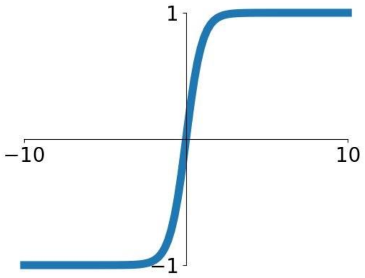
Sigmoid

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



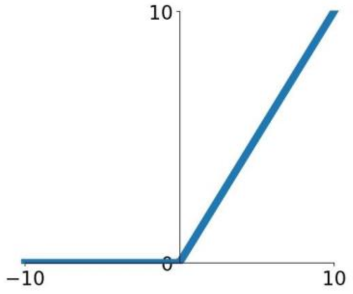
tanh

$$\tanh(x)$$



ReLU

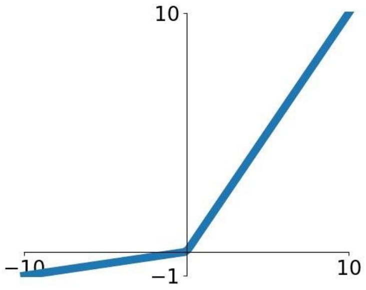
$$\max(0, x)$$



Хороший выбор

Leaky ReLU

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

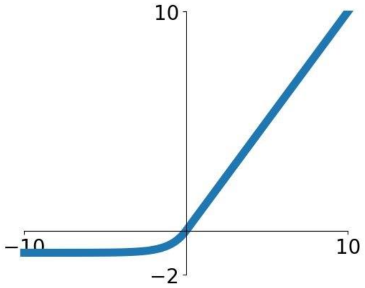


Maxout neuron

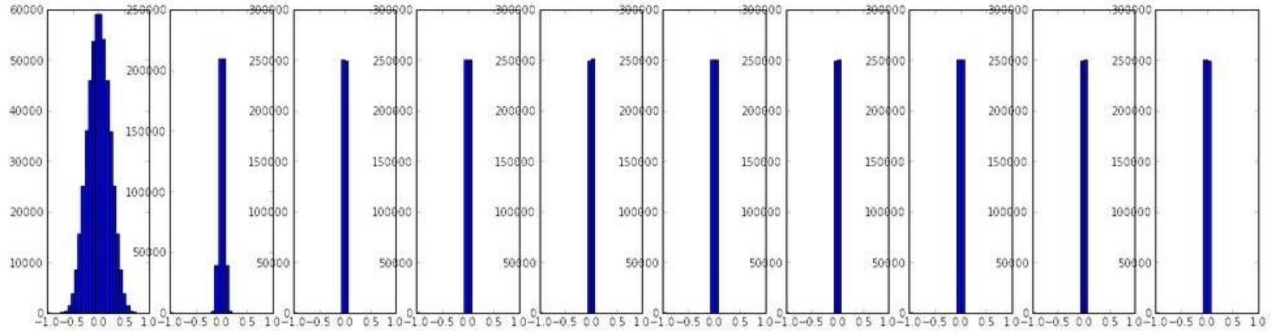
$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

ELU

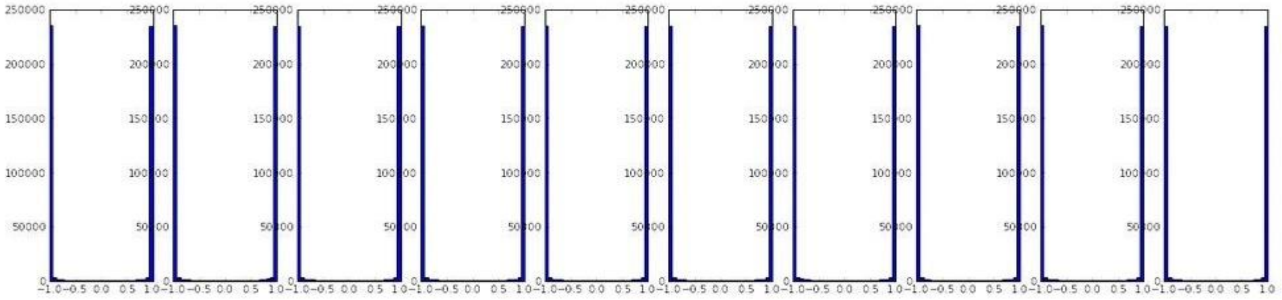
$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



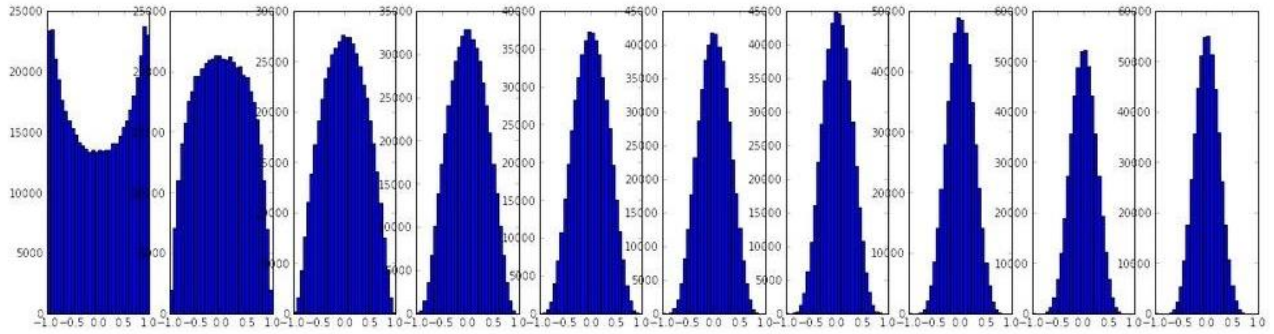
На прошлом занятии: **Инициализация весов**



Инициализация маленькими значениями
Активации нулевые, градиенты нулевые. Нет обучения.

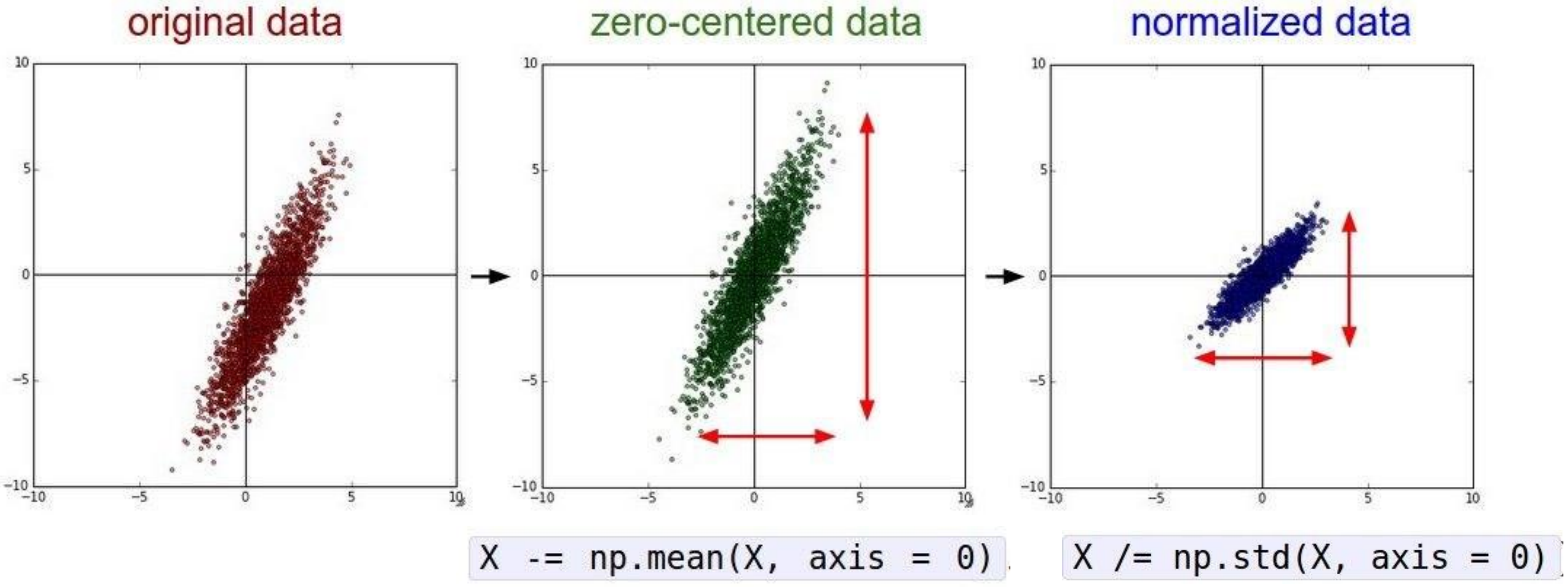


Инициализация большими значениями
Активации в насыщении (tanh), градиенты нулевые. Нет обучения.



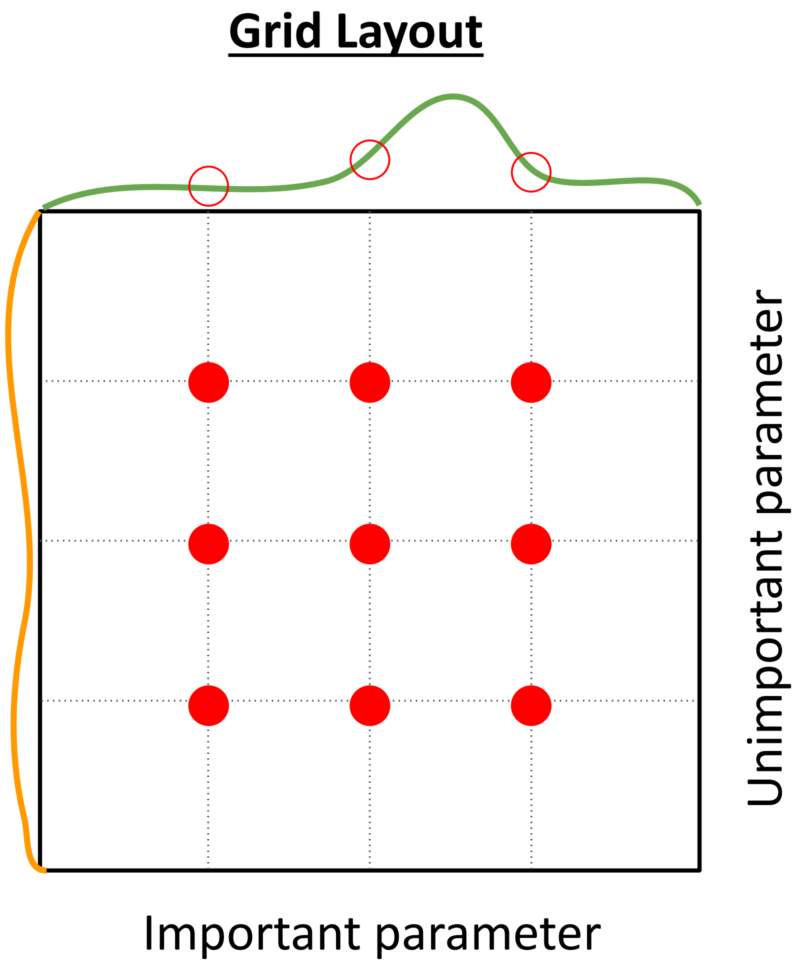
Инициализация верными значениями (Xavier)
Активации имеют хорошее распределение. Есть обучение.

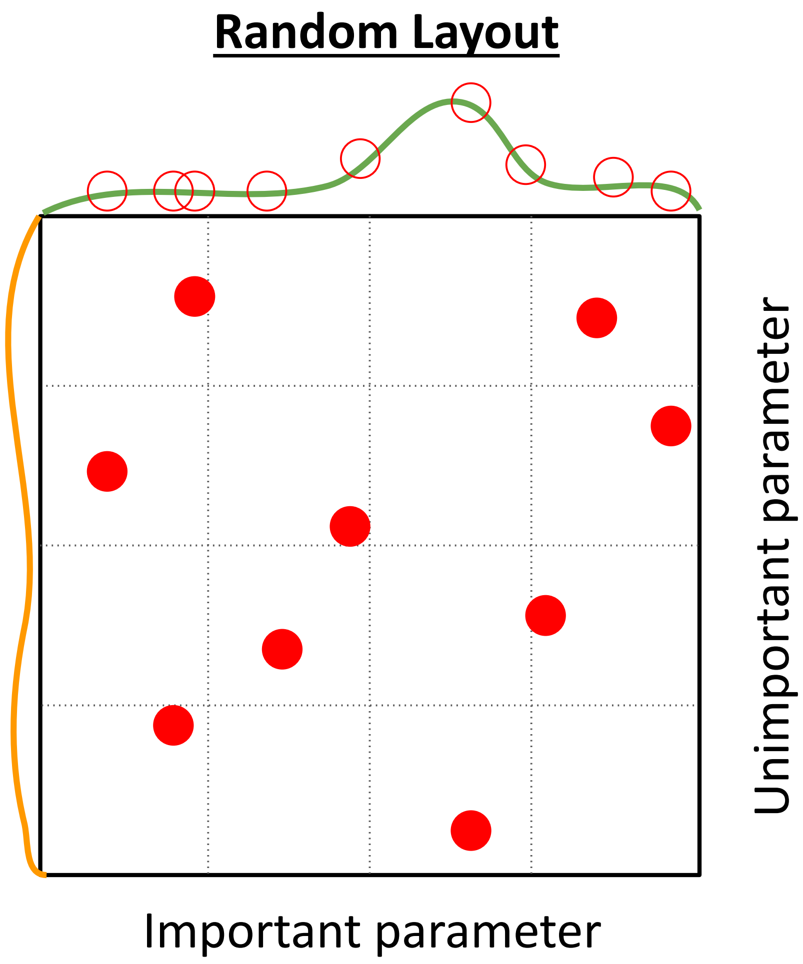
На прошлом занятии: **Предобработка данных**



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

На прошлом занятии:
Random Search vs. Grid Search





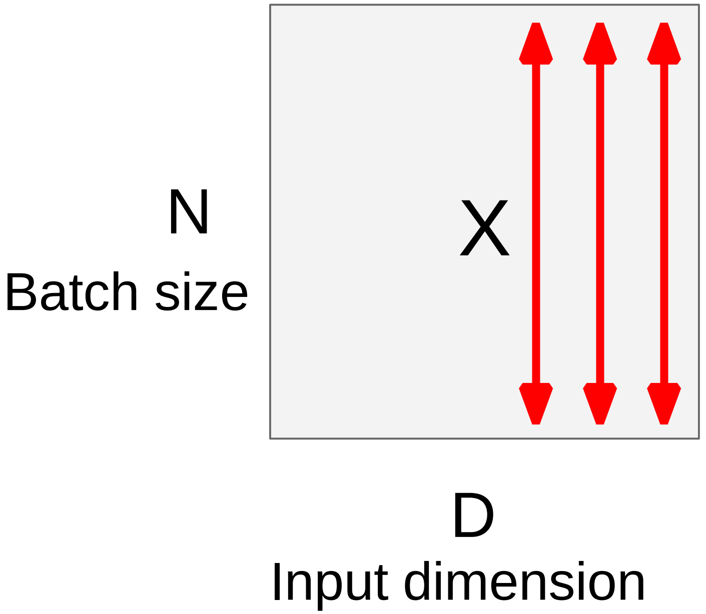
Random Search for Hyper-Parameter Optimization Bergstra and Bengio, 2012

Сегодня

- Стратегии изменения весов нейронной сети (Momentum, Adam)
- Методы регуляризации (Dropout)
- Knowledge transfer - перенос знаний из одной нейронной сети в другую
- Архитектуры CNN

На прошлом занятии: **Batch Normalization**

"you want unit gaussian activations? just make them so."



1. compute the empirical mean and variance independently for each neuron.

2. Normalize

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

[Ioffe and Szegedy, 2015]

На прошлом занятии: **Batch Normalization**

Input: $x : N \times D$

Learnable params: $\gamma, \beta : D$

Intermediates: $\mu, \sigma : D$

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

Output: $y : N \times D$

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

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

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

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

Batch Normalization: **Test Time**

Use estimated mean and variance during training

Input: $x : N \times D$

Learnable params: $\gamma, \beta : D$

Intermediates: $\mu, \sigma : D$

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

Output: $y : N \times D$

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

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

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

Batch Normalization: **Test Time**

Input: $x : N \times D$

Learnable params: $\gamma, \beta : D$

Intermediates: $\mu, \sigma : D$

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

Output: $y : N \times D$

$\mu_j =$ (Running) average of values seen during training

$\sigma_j^2 =$ (Running) average of values seen during training

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

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

Batch Normalization for ConvNets

Batch Normalization for **fully-connected** networks

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

Normalize

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

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

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

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

Layer Normalization

Batch Normalization for fully-connected networks

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

Normalize

$$\mu,\sigma: 1 \times D$$

$$\gamma,\beta: 1 \times D$$

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

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

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

Normalize

$$\mu,\sigma: N \times 1$$

$$\gamma,\beta: 1 \times D$$

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

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

Instance Normalization

Batch Normalization for **convolutional** networks

$$\boxed{\begin{array}{ll} \mathbf{x}: & N \times C \times H \times W \\ \text{Normalize} & \downarrow_N \quad \downarrow_H \; \downarrow_W \\ \mu, \sigma: & 1 \times C \times 1 \times 1 \end{array}}$$

$$\gamma, \beta: \; 1 \times C \times 1 \times 1$$

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

Instance Normalization for convolutional networks

Same behaviour at train / test!

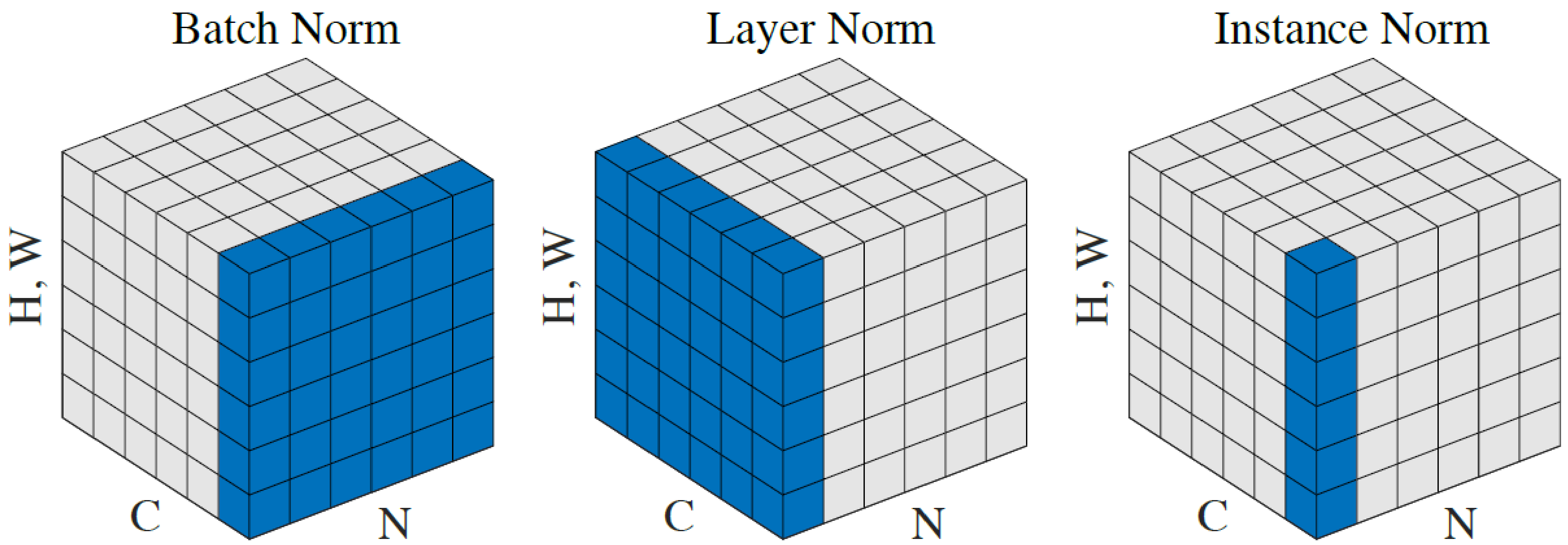
$$\boxed{\begin{array}{ll} \mathbf{x}: & N \times C \times H \times W \\ \text{Normalize} & \downarrow_H \; \downarrow_W \\ \mu, \sigma: & N \times C \times 1 \times 1 \end{array}}$$

$$\gamma, \beta: \; 1 \times C \times 1 \times 1$$

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

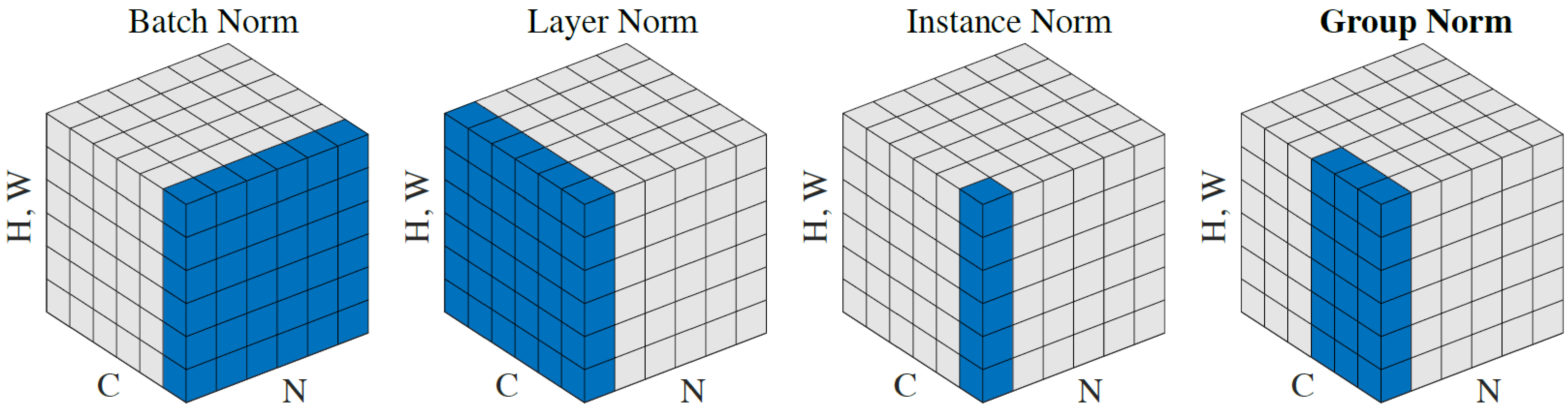
Ulyanov et al, Improved Texture Networks: Maximizing Quality and Diversity in Feed-forward Stylization and Texture Synthesis, CVPR 2017

Comparison of Normalization Layers



Wu and He, "Group Normalization", arXiv 2018

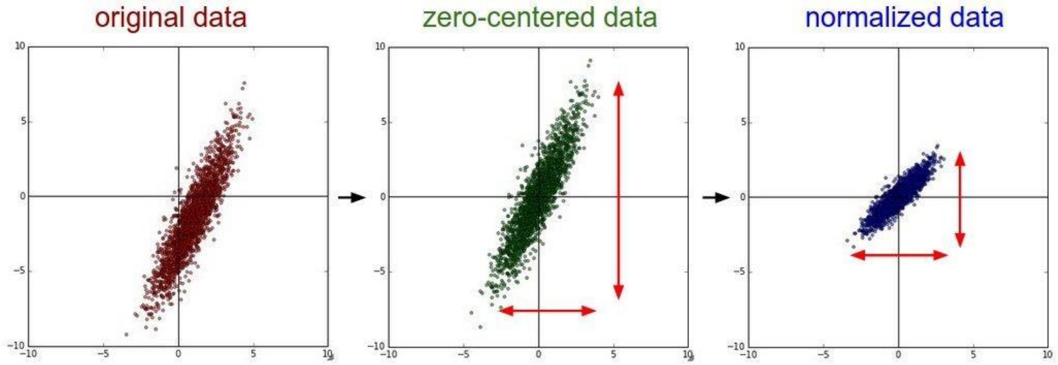
Group Normalization



Wu and He, “Group Normalization”, arXiv 2018

Decorrelated Batch Normalization

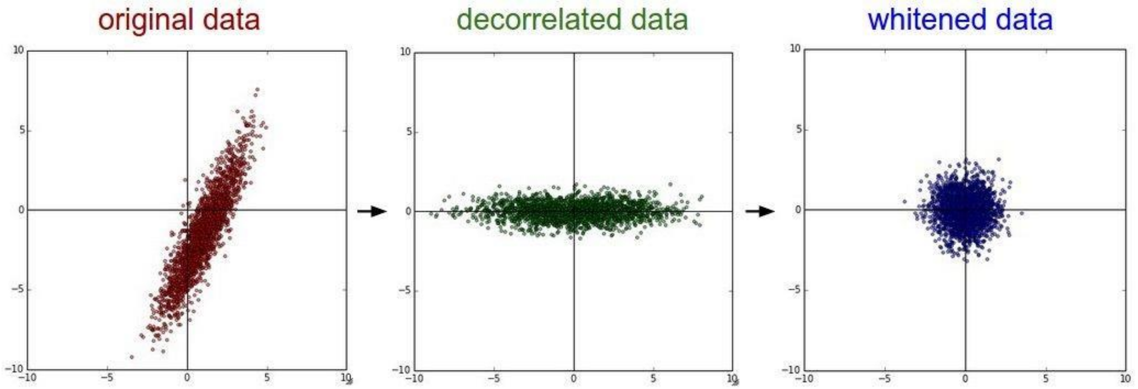
Batch Normalization



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

BatchNorm normalizes the data, but cannot correct for correlations among the input features

Decorrelated Batch Normalization



$$\hat{x}_i = \Sigma^{-\frac{1}{2}}(x_i - \mu)$$

DBN **whitens** the data using the full covariance matrix of the minibatch; this corrects for correlations

Huang et al, “Decorrelated Batch Normalization”, arXiv 2018

ECCV 2018: Normalization Methods for Training Deep Neural Networks: Theory and Practice

- Feature normalization ← BatchNorm, LayerNorm, InstanceNorm, GroupNorm

ECCV 2018: Normalization Methods for Training Deep Neural Networks: Theory and Practice

- Feature normalization
- Weight normalization
- Gradient normalization

Presentations:

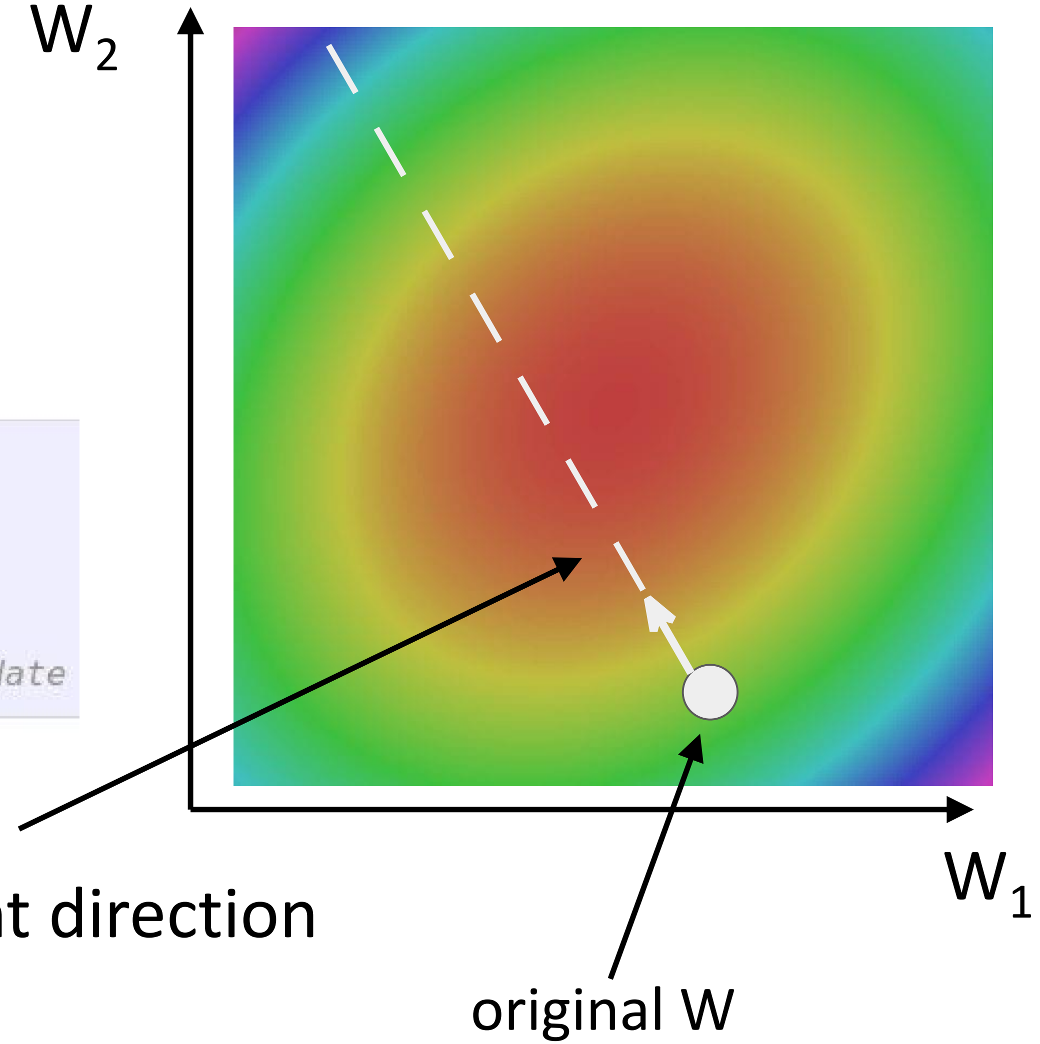
https://sites.google.com/view/normalization-eccv18/schedule

Source code:
https://sites.google.com/view/normalization-eccv18/resources/software

Оптимизация: Stochastic Gradient Descent (SGD)

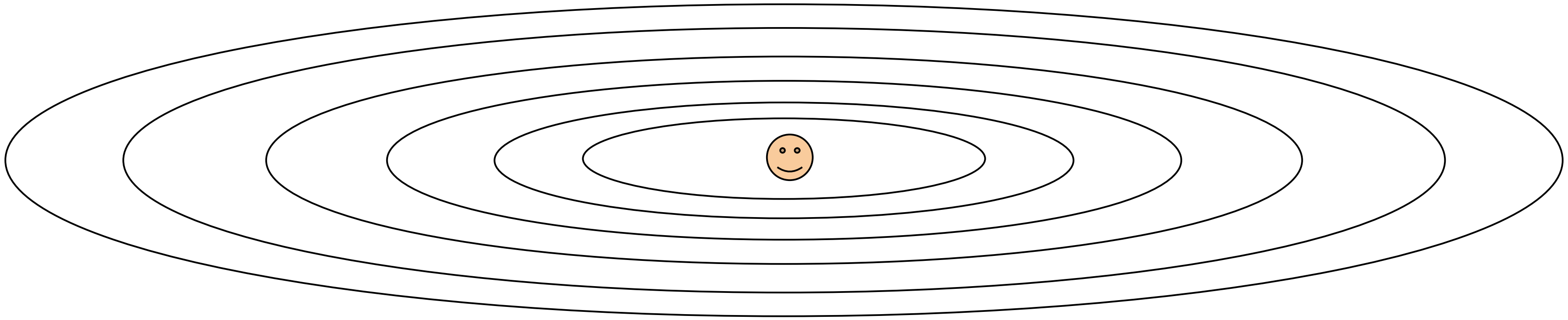
```
# Vanilla Gradient Descent

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



Оптмизация: **Problems with SGD**

Suppose our loss changes quickly in one direction and slowly in another.
What does gradient descent do?
Q: What is the trajectory along which we converge towards the minimum with SGD?



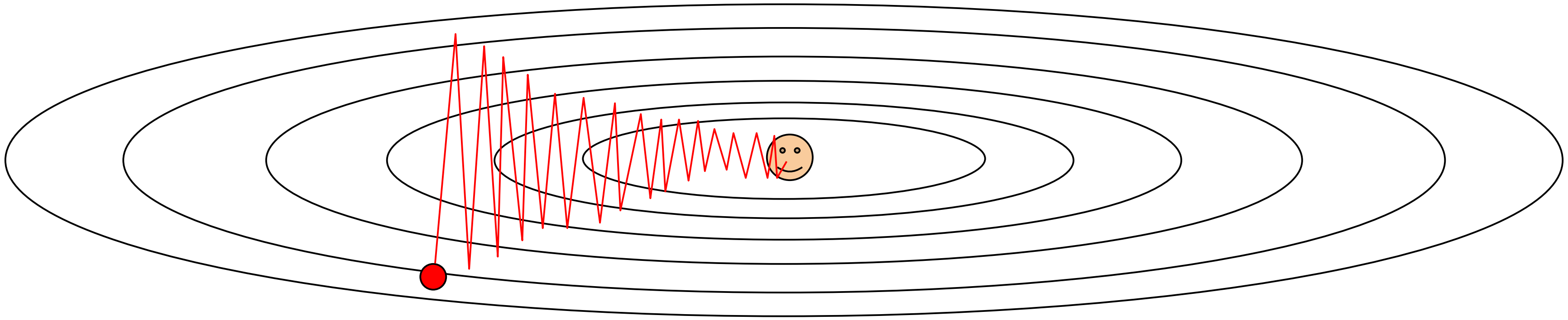
Loss function has high **condition number**: ratio of largest to smallest singular value of the Hessian matrix is large

Оптмизация: Problems with SGD

Suppose our loss changes quickly in one direction and slowly in another.
What does gradient descent do?

Q: What is the trajectory along which we converge towards the minimum with SGD?

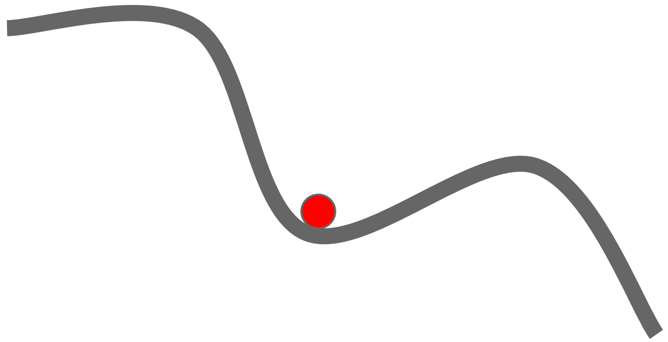
Very slow progress along shallow dimension, jitter along steep direction

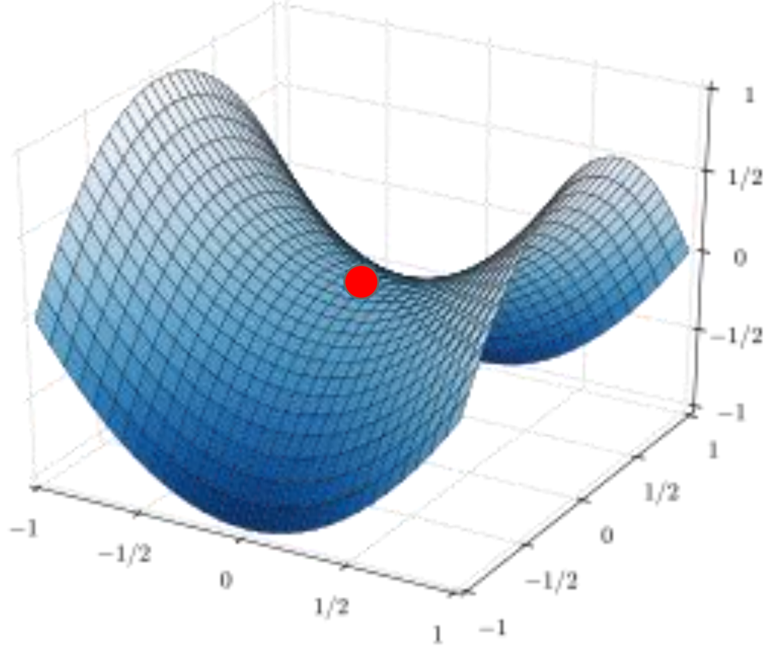


Loss function has high **condition number**: ratio of largest to smallest singular value of the Hessian matrix is large

Оптмизация: Problems with SGD

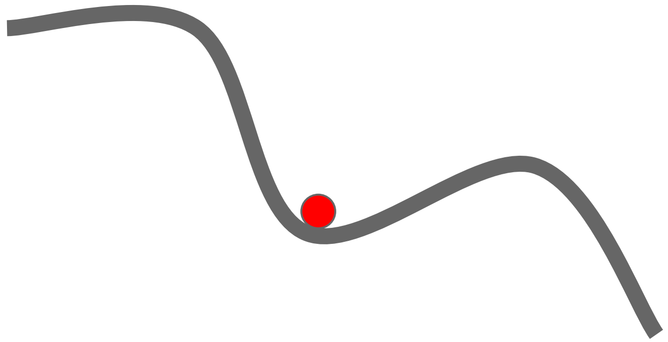
What if the loss function has a **local minima** or **saddle point**?

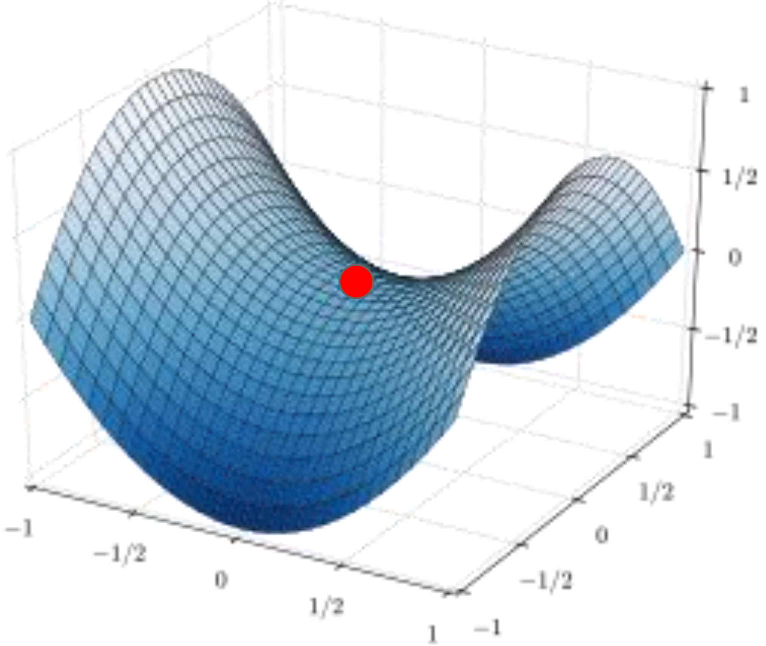




Оптмизация: **Problems with SGD**

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



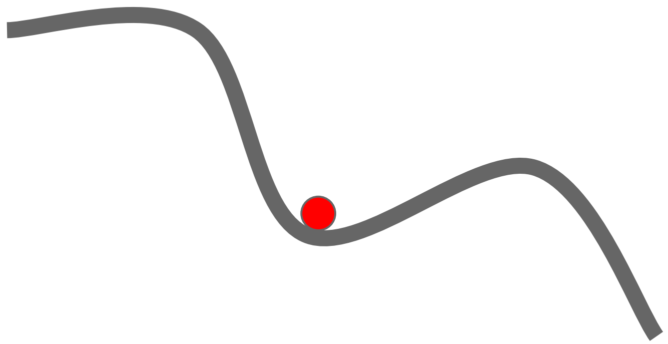


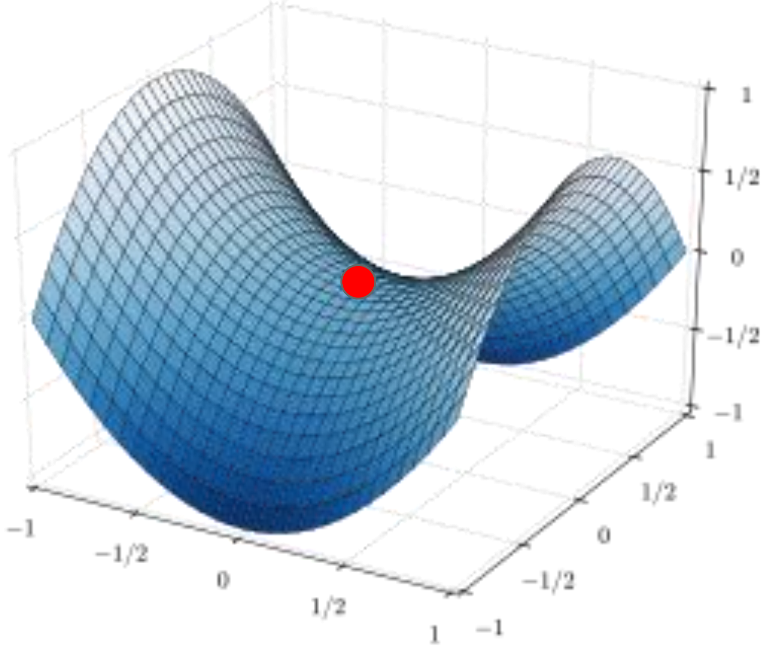
Zero gradient, gradient descent gets stuck

Оптмизация: **Problems with SGD**

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

Saddle points much more common in high dimension



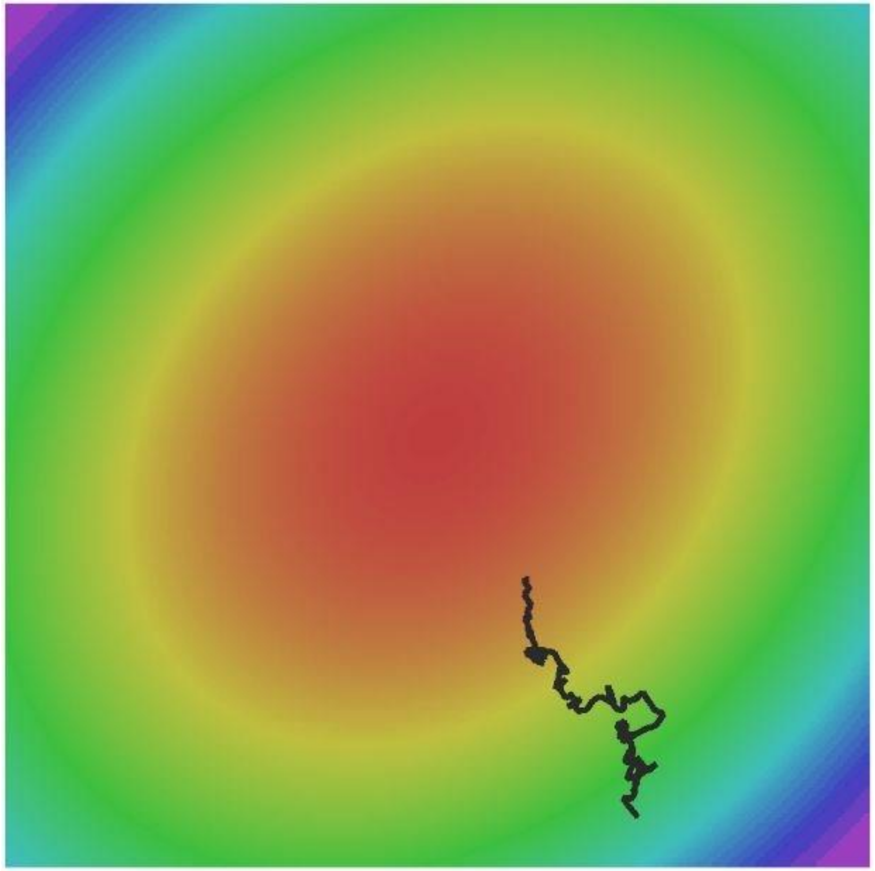


Оптмизация: Problems with SGD

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

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

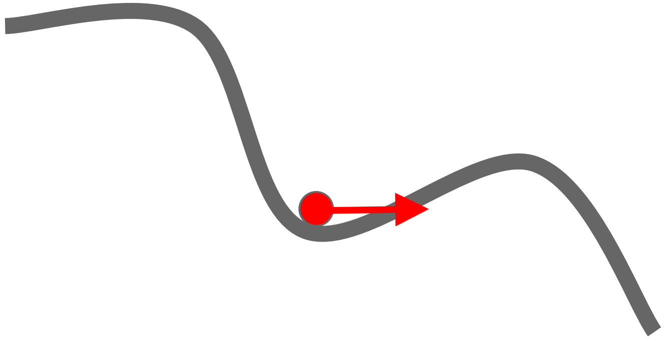
$$x_{t+1} = x_t - \alpha v_{t+1}$$

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

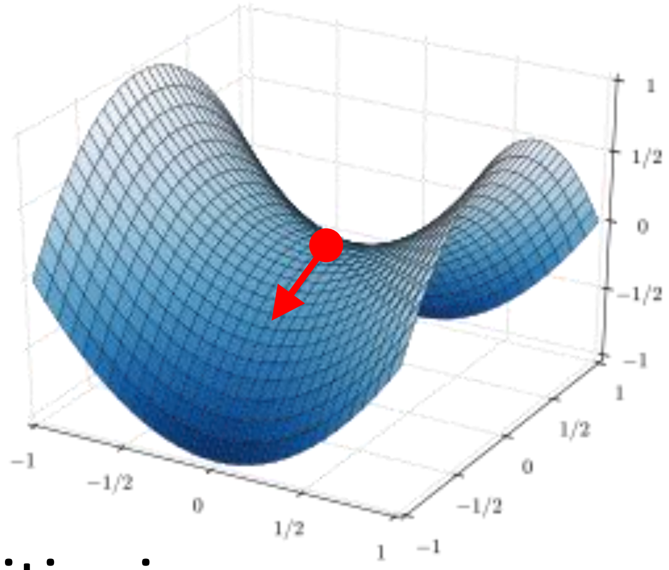
- Build up "velocity" as a running mean of gradients
- Rho gives "friction"; typically rho=0.9 or 0.99

SGD + Momentum

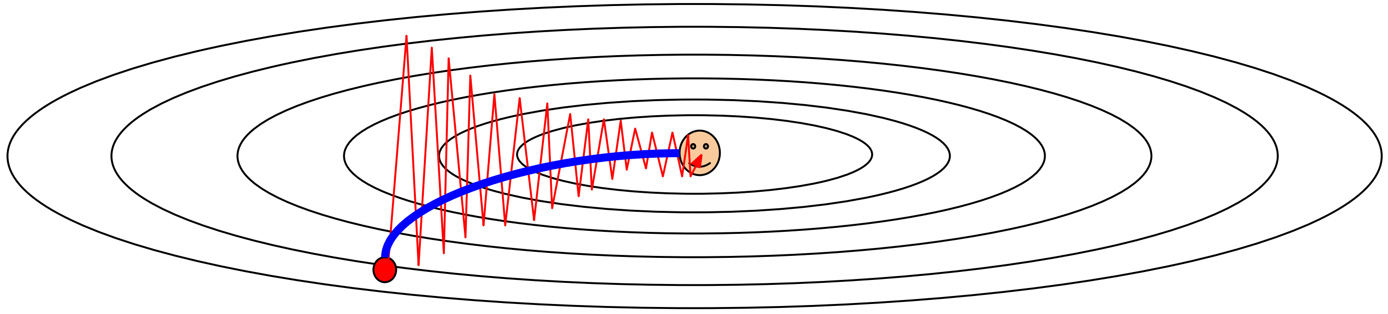
Local Minima



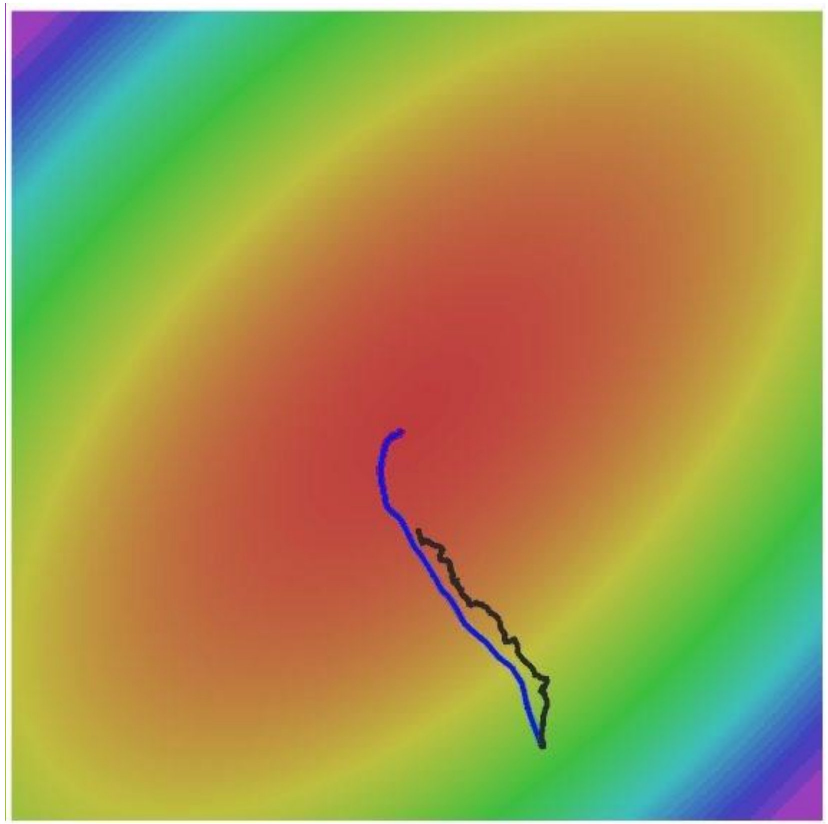
Saddle point



Poor Conditioning

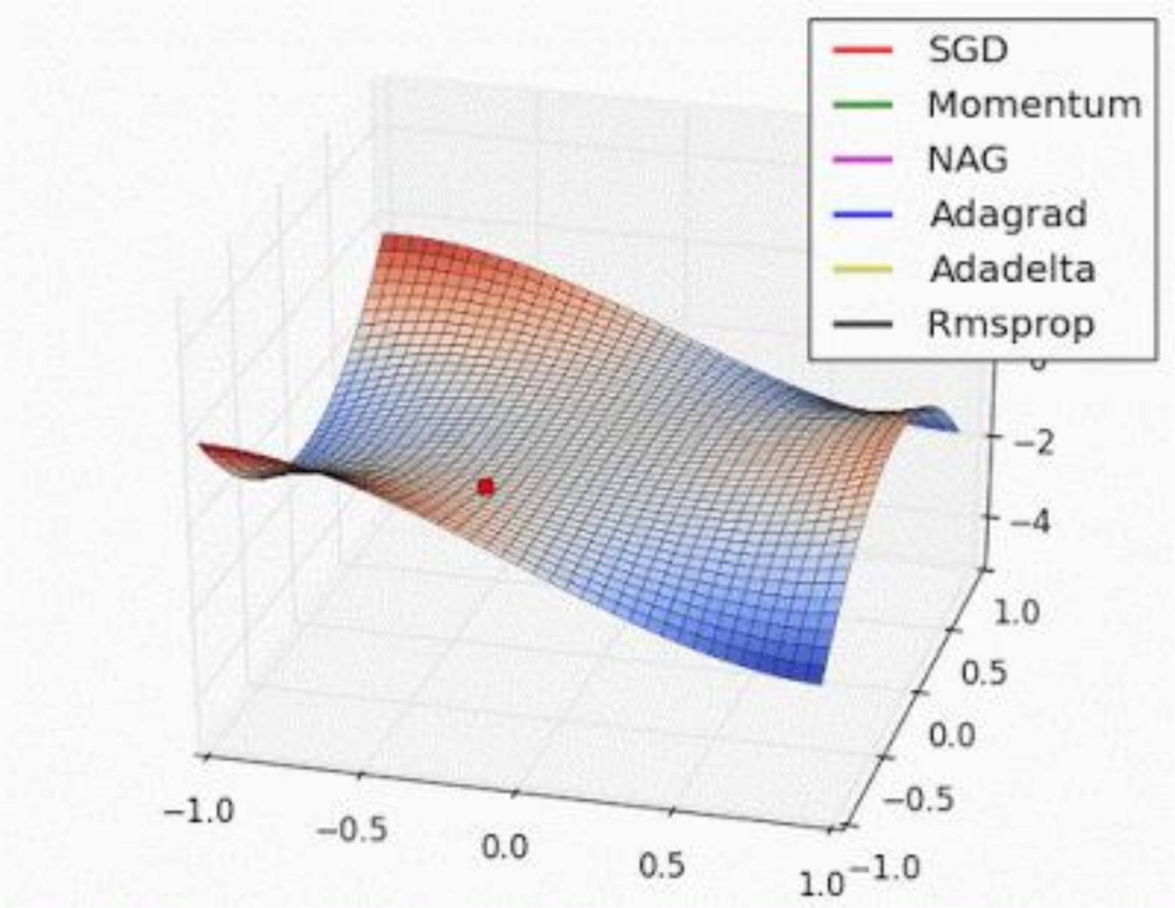


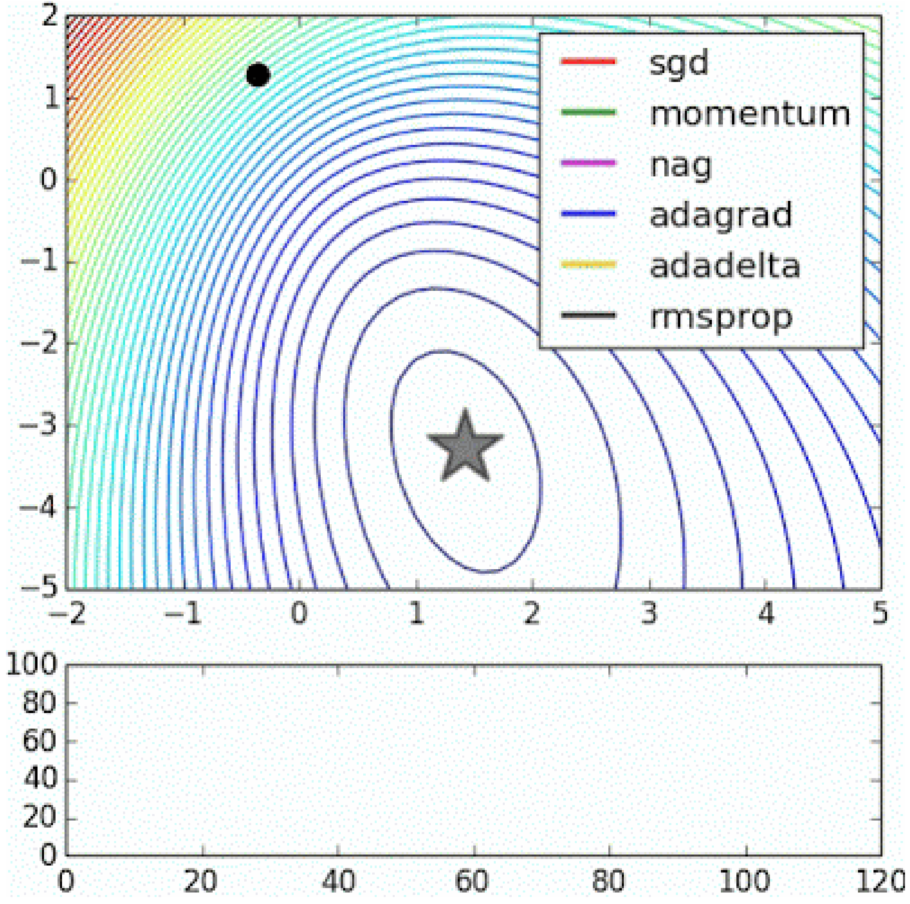
Gradient Noise



SGD

SGD + Momentum





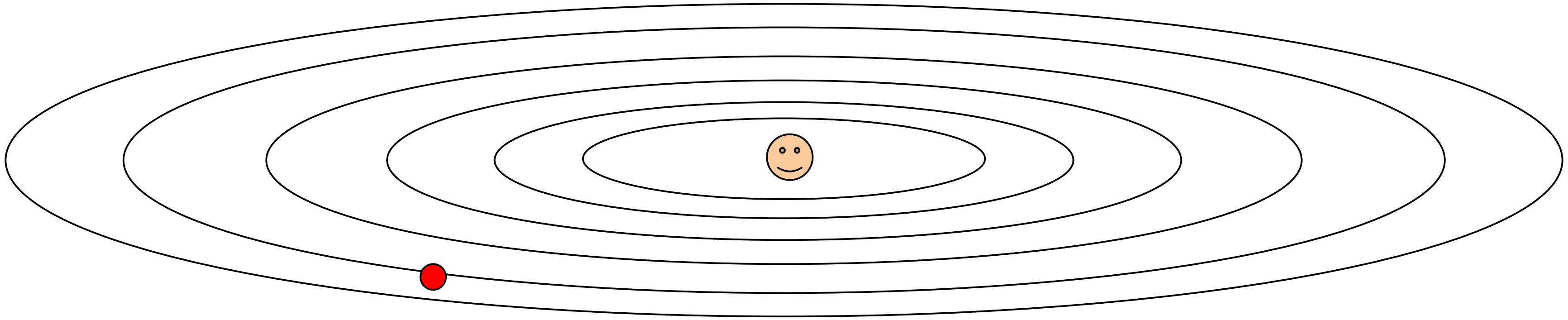
AdaGrad

```
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

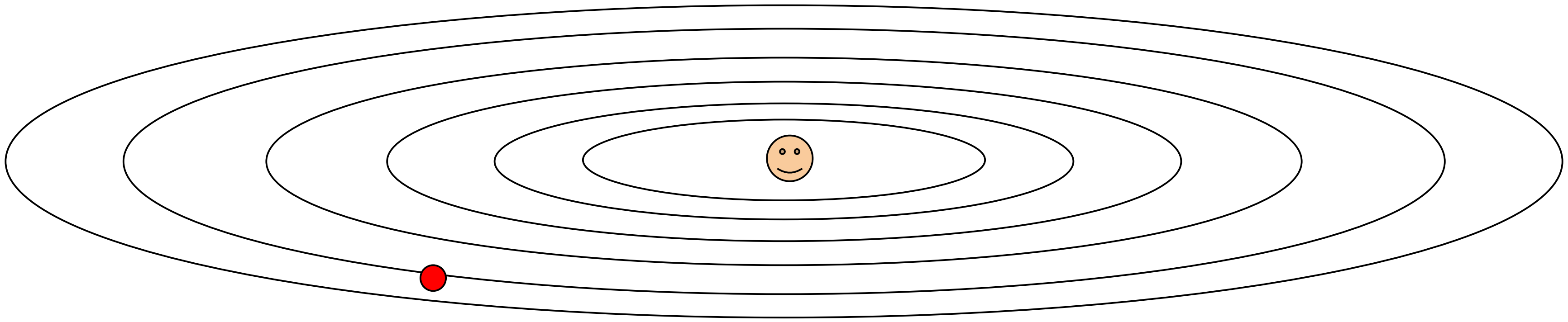
```
grad_squared = 0
while True:
  dx = compute_gradient(x)
  grad_squared += dx * dx
  x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```



Q: What happens with AdaGrad ?
(in situation with high condition number)

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

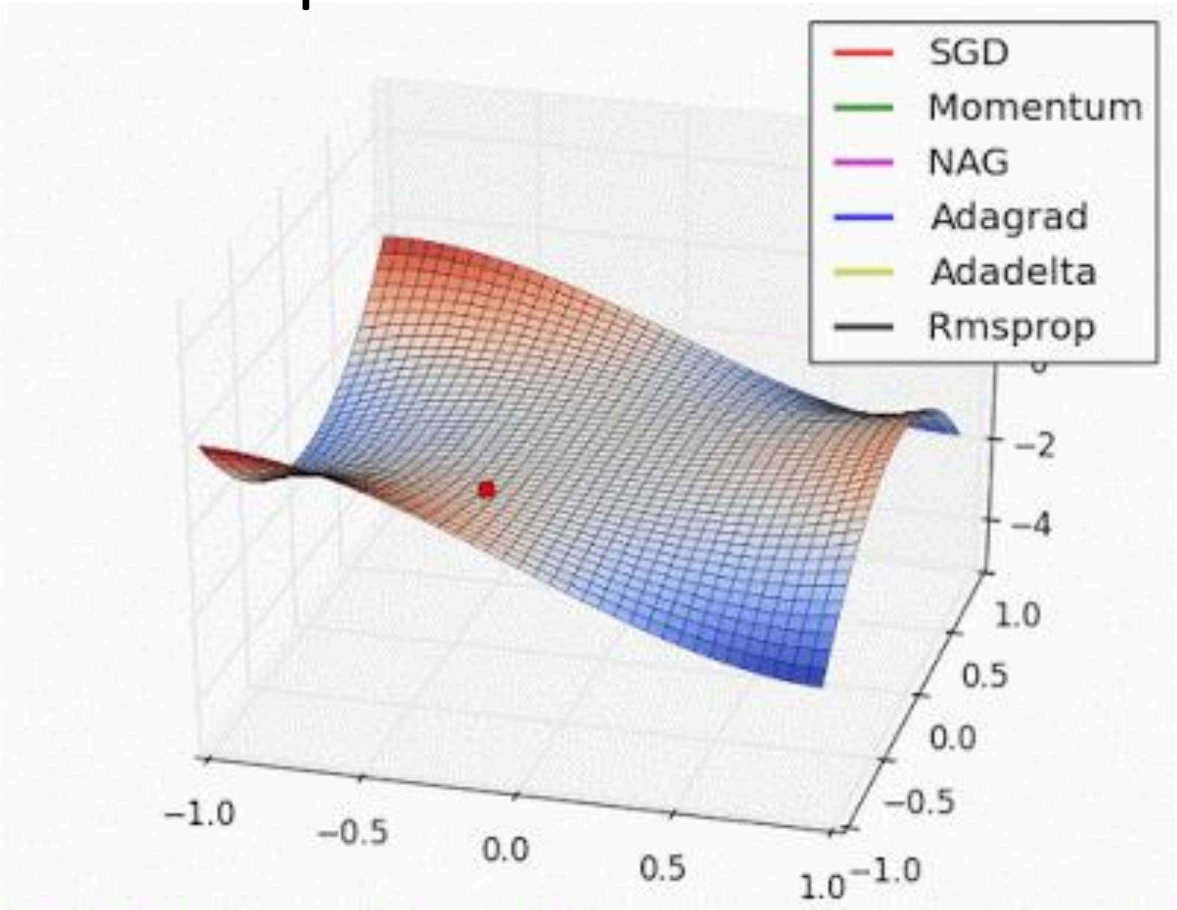
AdaGrad

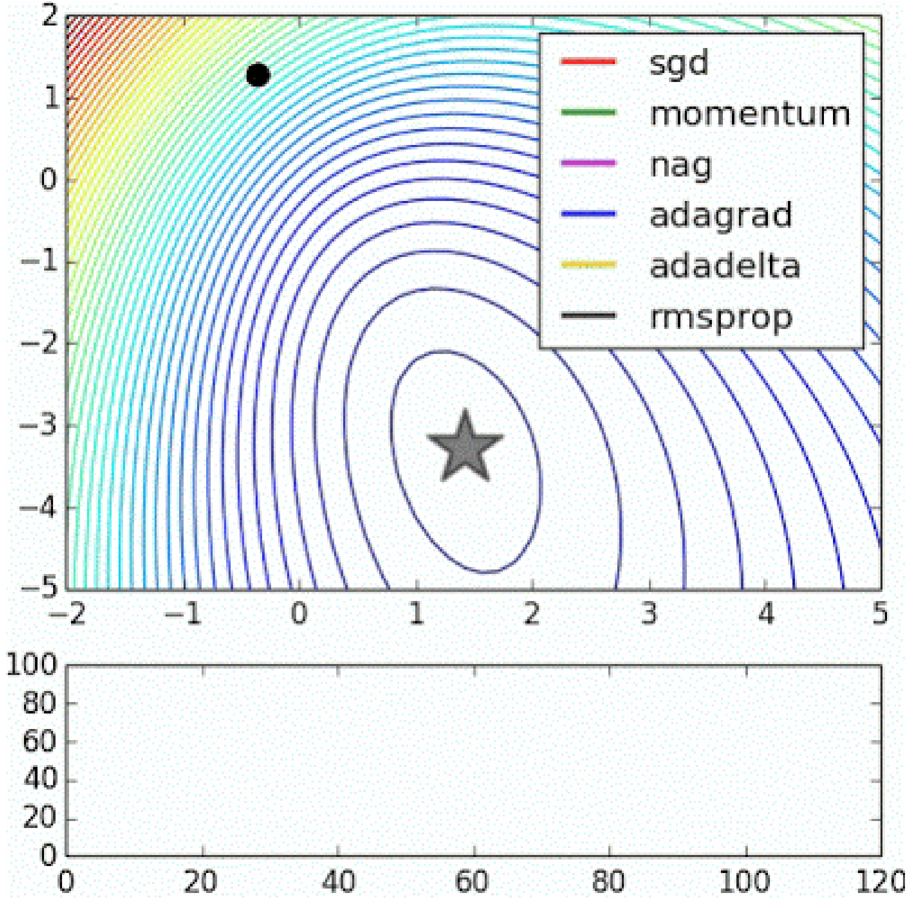
```
grad_squared = 0
while True:
  dx = compute_gradient(x)
  grad_squared += dx * dx
  x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```

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)
```

AdaGrad
RMSProp





Adam – adaptive moment estimation

```
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:
mean dx

RMSProp:
mean dx*dx

For each derivative we calculate
first_moment
second_moment

Adam

```
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:
mean dx

RMSProp:
mean dx*dx

beta1 is about 0.9
beta2 is about 0.995...0.999

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

AdaGrad / RMSProp

Bias correction

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

Adam (full form)

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

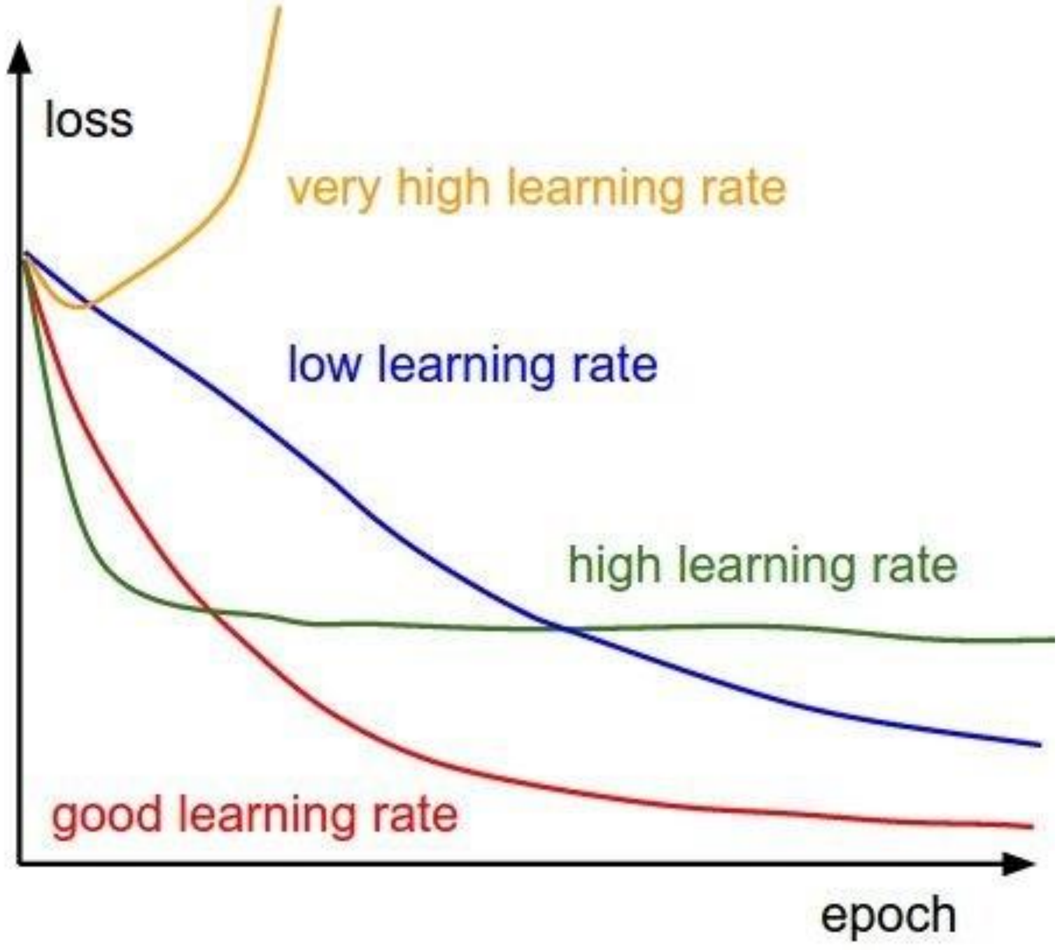
AdaGrad / RMSProp

Bias correction

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!

SGD+Momentum, 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.

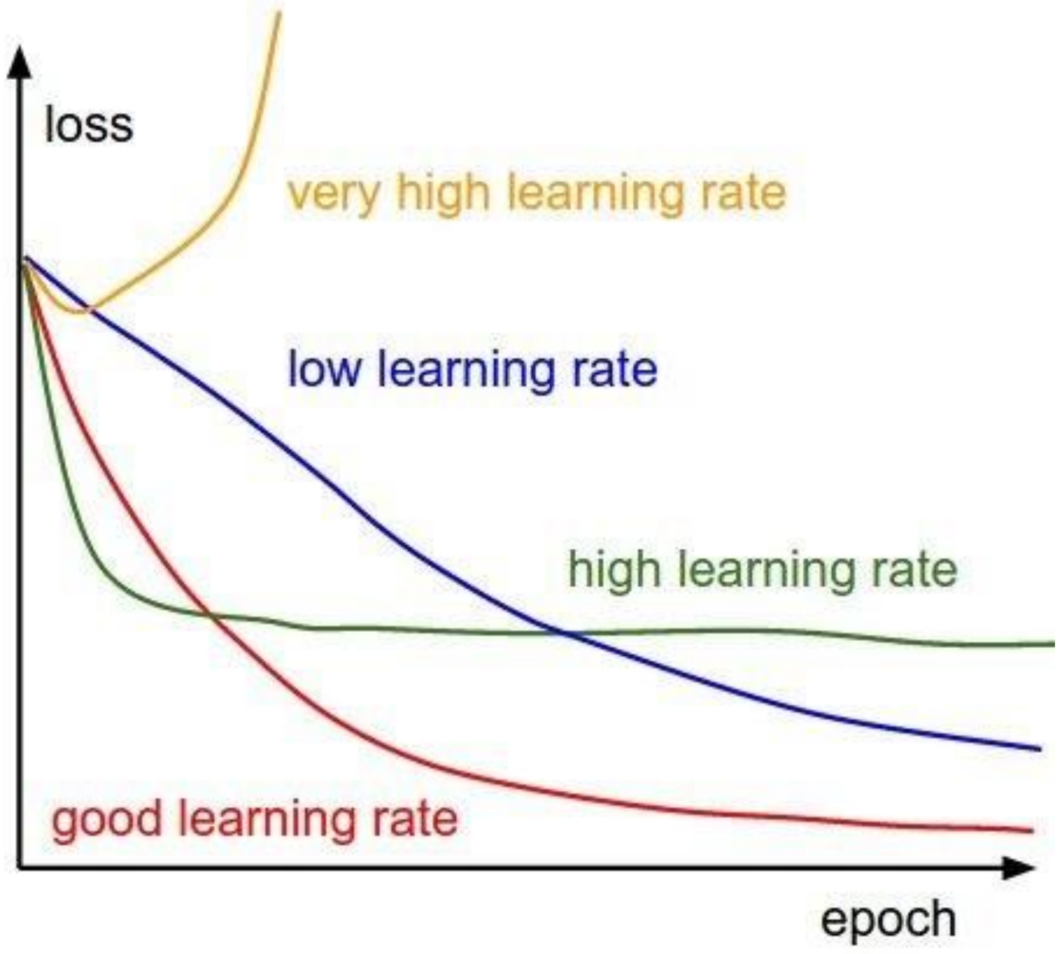
exponential decay:

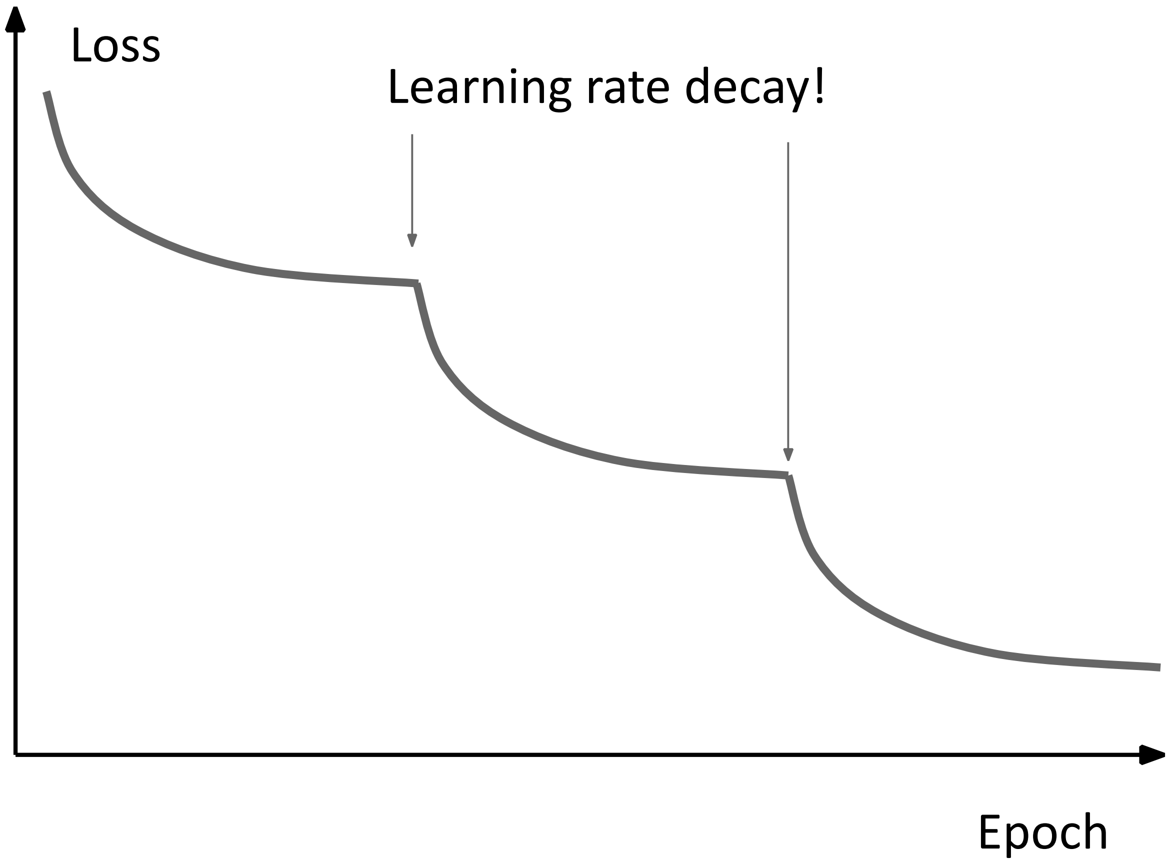
$$\alpha = \alpha_0 e^{-kt}$$

1/t decay:

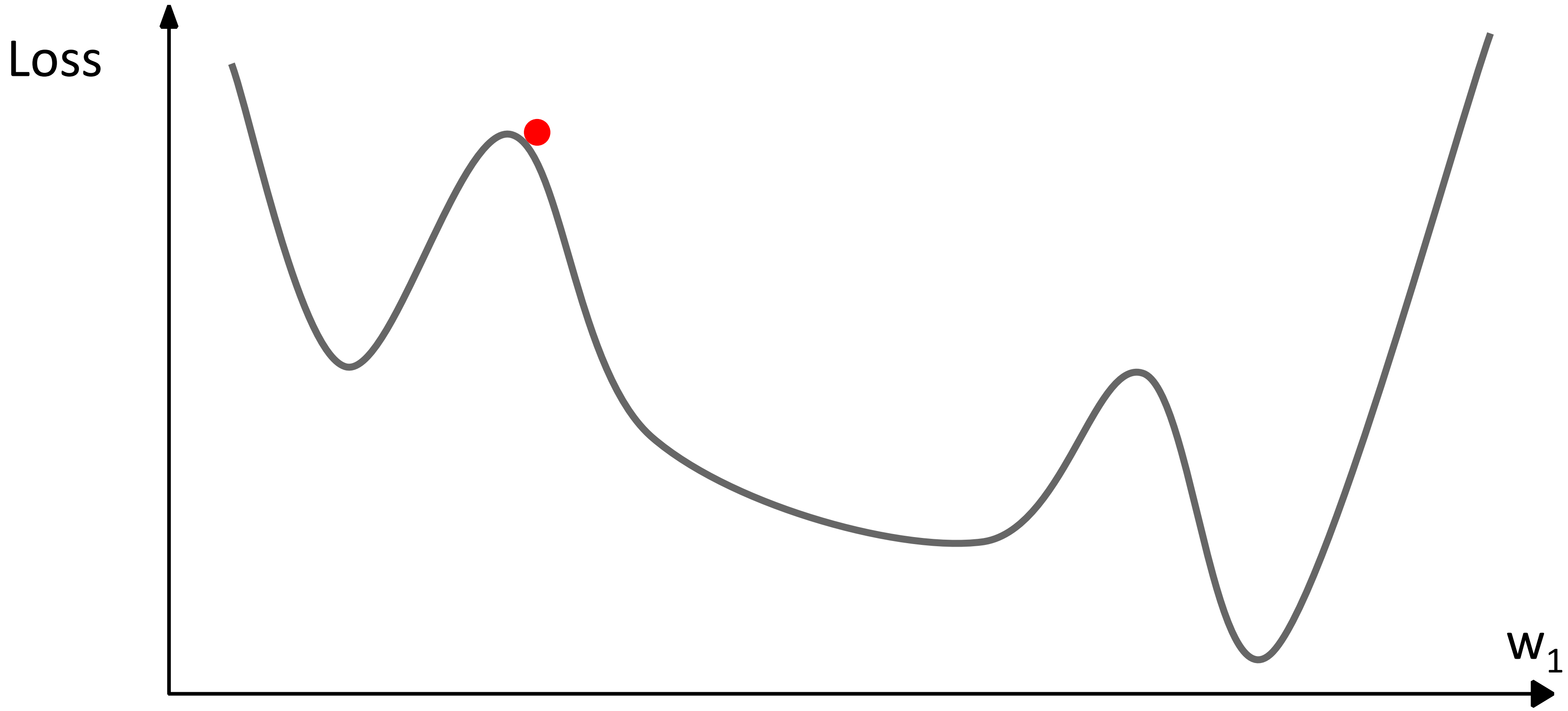
$$\alpha = \alpha_0 / (1 + kt)$$

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



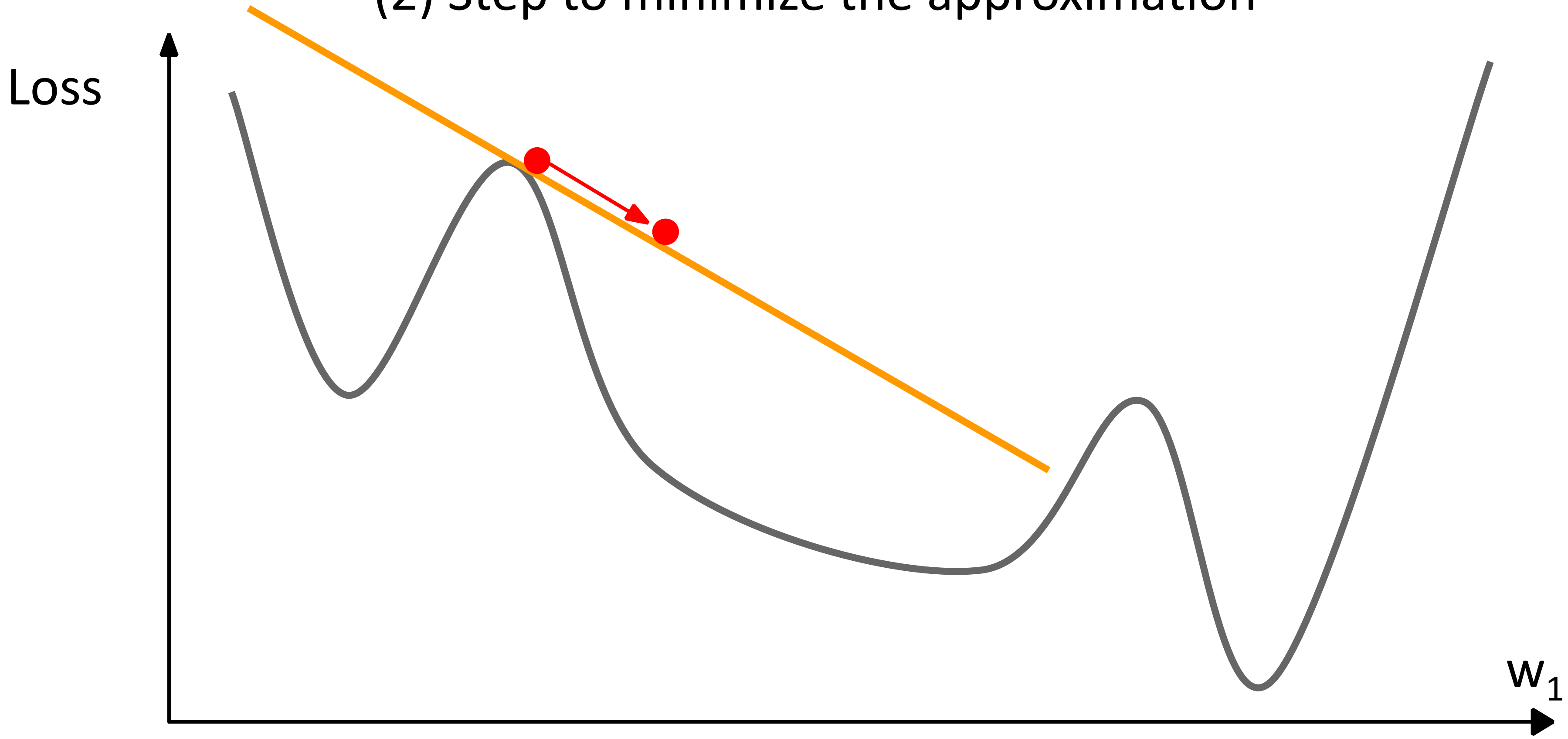


First-Order Optimization



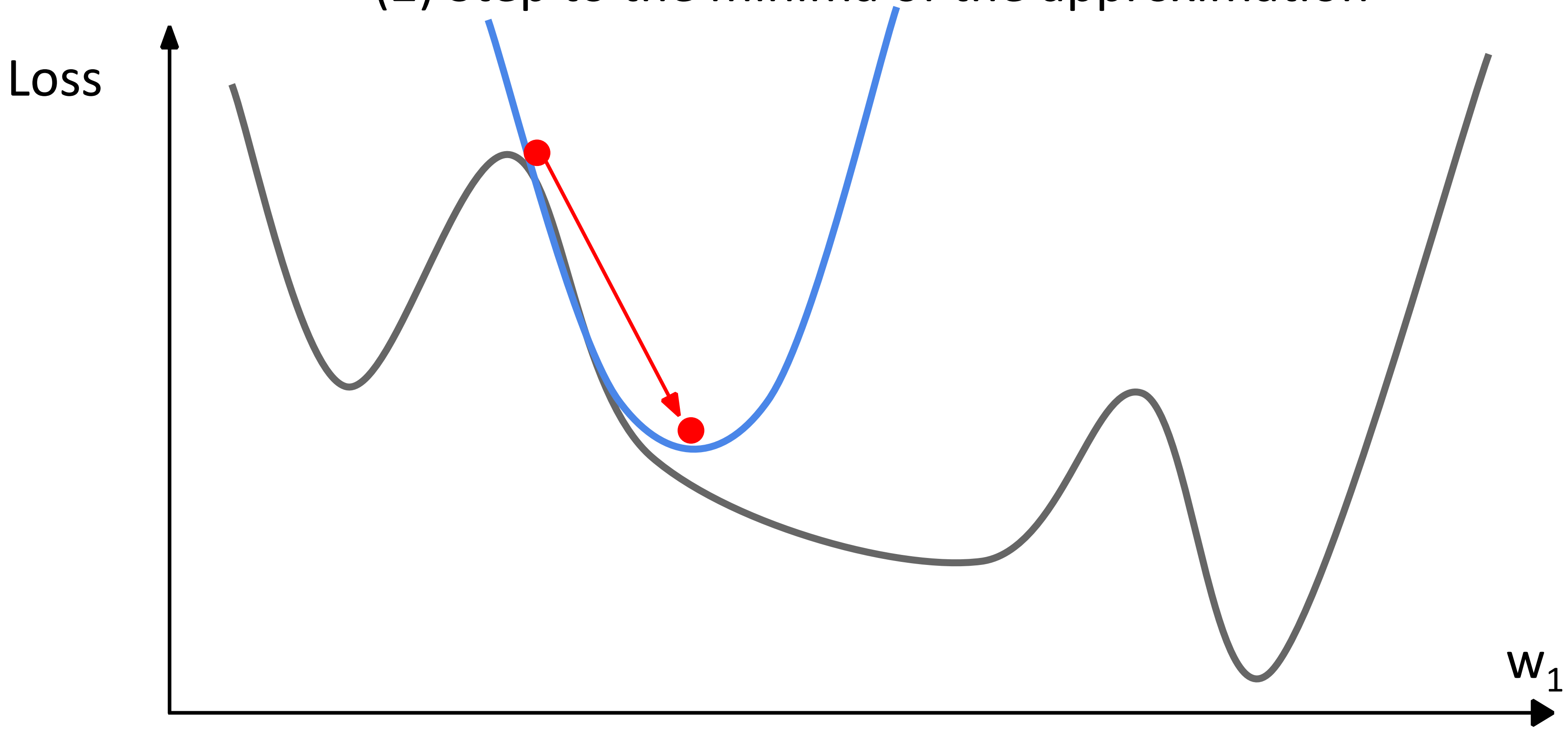
First-Order Optimization

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



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:

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

Hessian matrix

$$H_{i,j} = \frac{\partial^2 J}{\partial \theta_i \partial \theta_j}$$

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

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

Q: What is nice about this update?

Second-Order Optimization

second-order Taylor expansion:

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

Hessian matrix

$$H_{i,j} = \frac{\partial^2 J}{\partial \theta_i \partial \theta_j}$$

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

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

Q: What is nice about this update?

No hyperparameters!

No learning rate!

Second-Order Optimization

second-order Taylor expansion:

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

Hessian matrix

$$H_{i,j} = \frac{\partial^2 J}{\partial \theta_i \partial \theta_j}$$

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

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

Q2: Why is this bad for deep learning?

Second-Order Optimization

second-order Taylor expansion:

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

Hessian matrix

$$H_{i,j} = \frac{\partial^2 J}{\partial \theta_i \partial \theta_j}$$

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

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

Q2: Why is this bad for deep learning?

Hessian has O(N^2) elements

Inverting takes O(N^3)

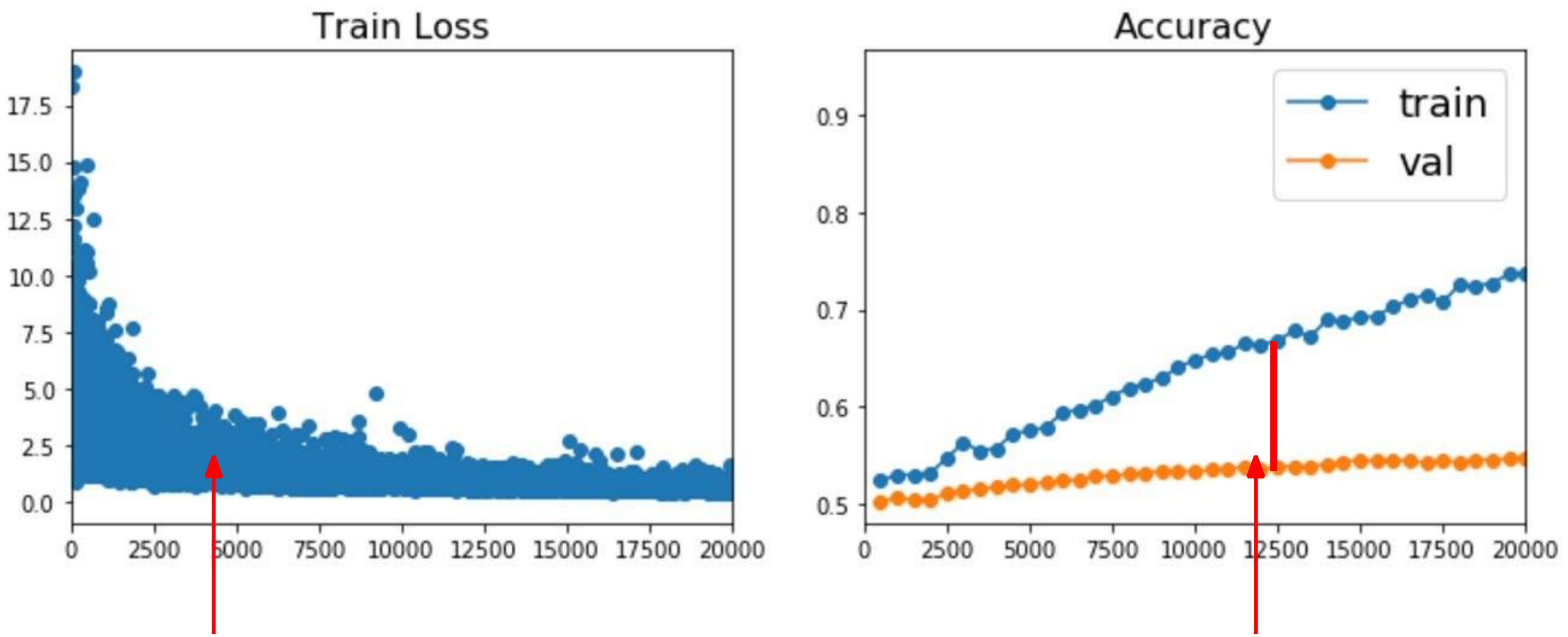
N = (Tens or Hundreds of) Millions

In practice:

Use **Adam**. It is a good default choice in most cases!

For State of the Art results use **SGD+Momentum**.

Beyond Training Error



Better optimization algorithms and stronger model help reduce training loss

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

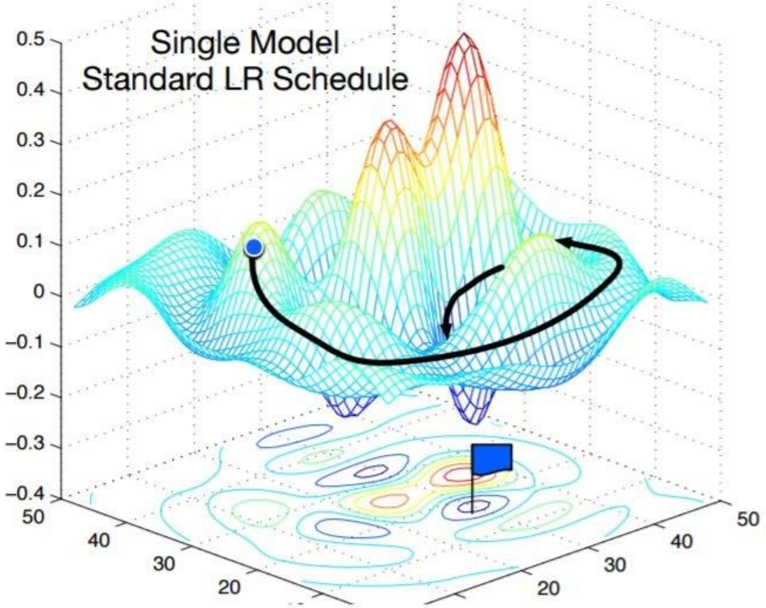
Model Ensembles

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

Enjoy 2% extra performance

Model Ensembles: Tips and Tricks

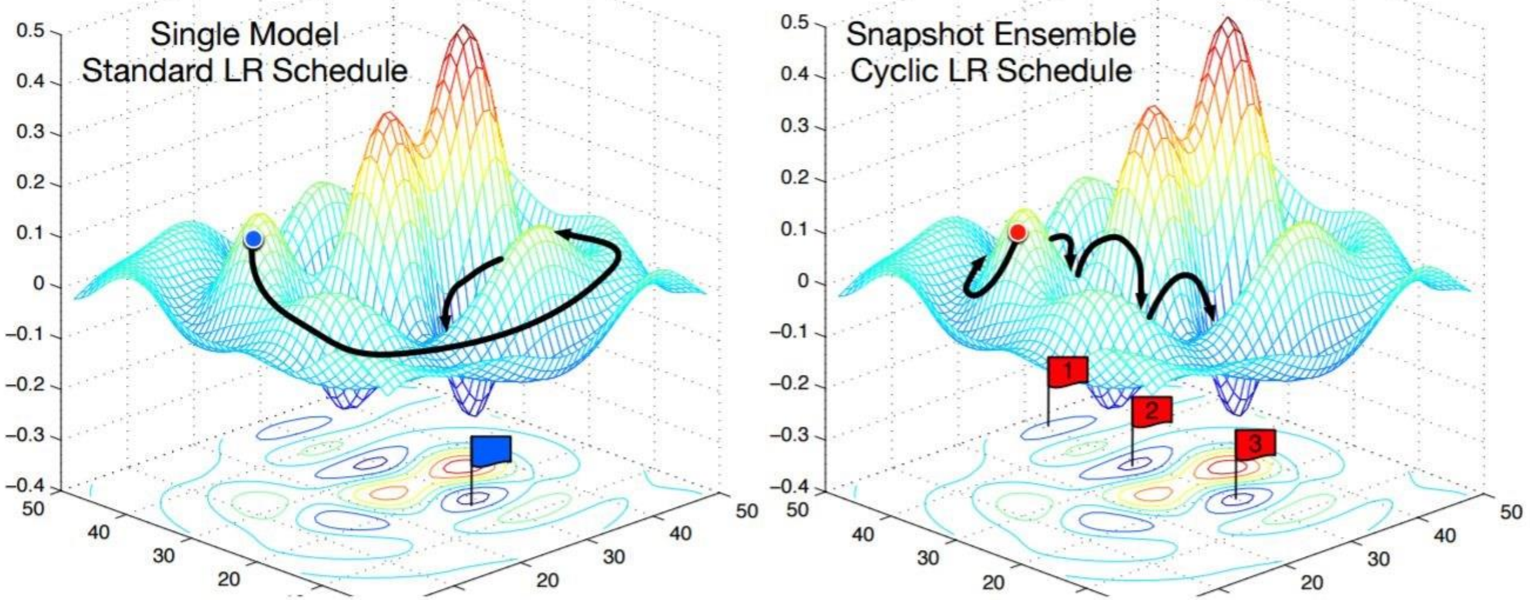
Instead of training independent models, use multiple snapshots of a single model during training!

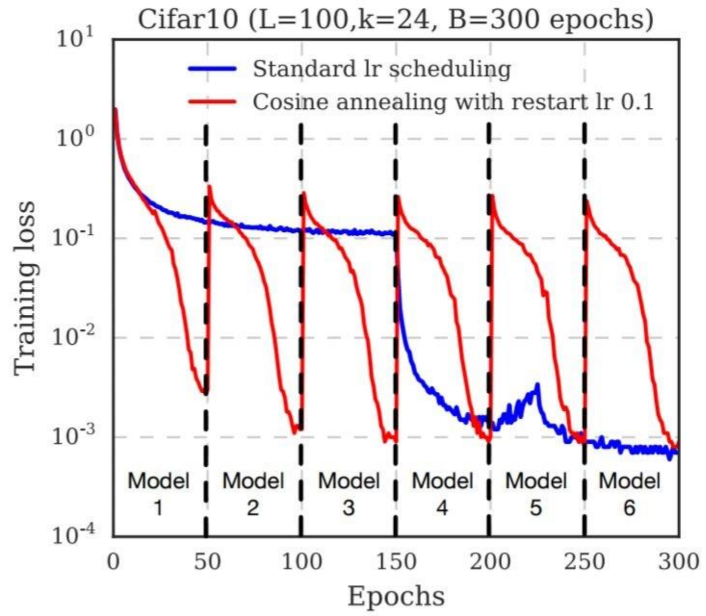


Loshchilov and Hutter, “SGDR: Stochastic gradient descent with restarts”, arXiv 2016
Huang et al, “Snapshot ensembles: train 1, get M for free”, ICLR 2017

Model Ensembles: Tips and Tricks

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

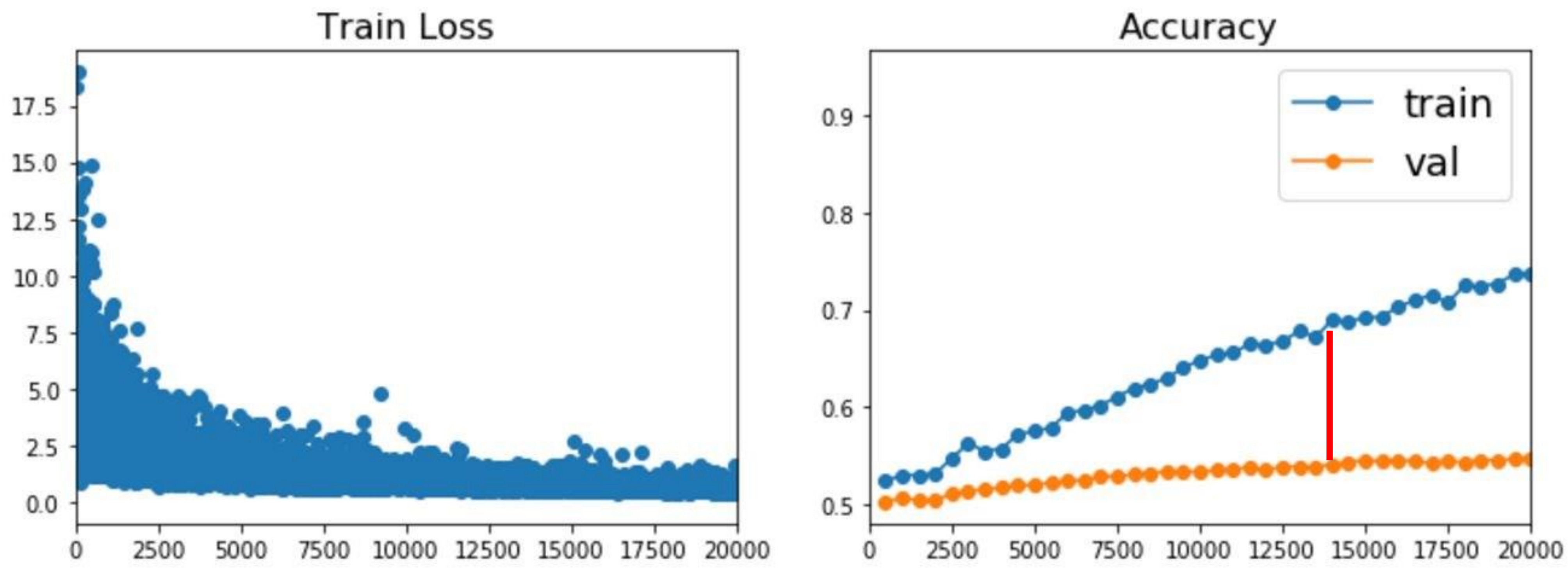




Loshchilov and Hutter, “SGDR: Stochastic gradient descent with restarts”, arXiv 2016
Huang et al, “Snapshot ensembles: train 1, get M for free”, ICLR 2017

Cyclic learning rate schedules can make single model even better!

How to improve single-model performance?



Regularization

Regularization: **Add weight decay**

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

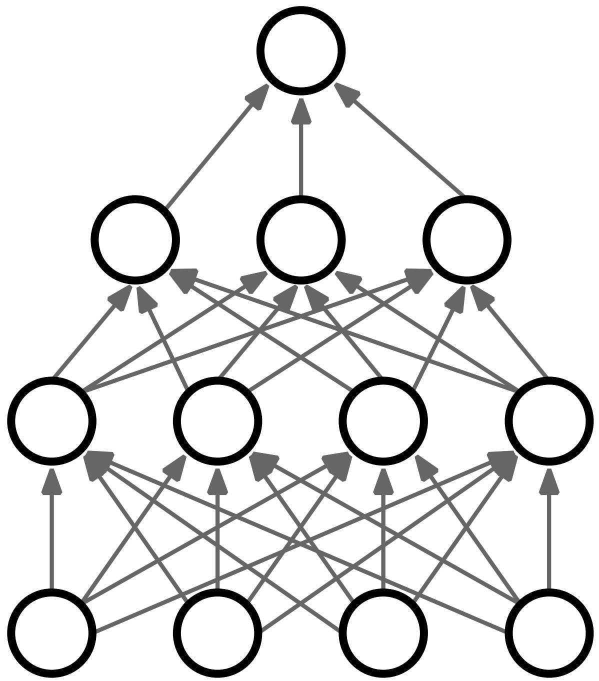
L2 regularization $R(W) = \sum_k \sum_l W_{k,l}^2$

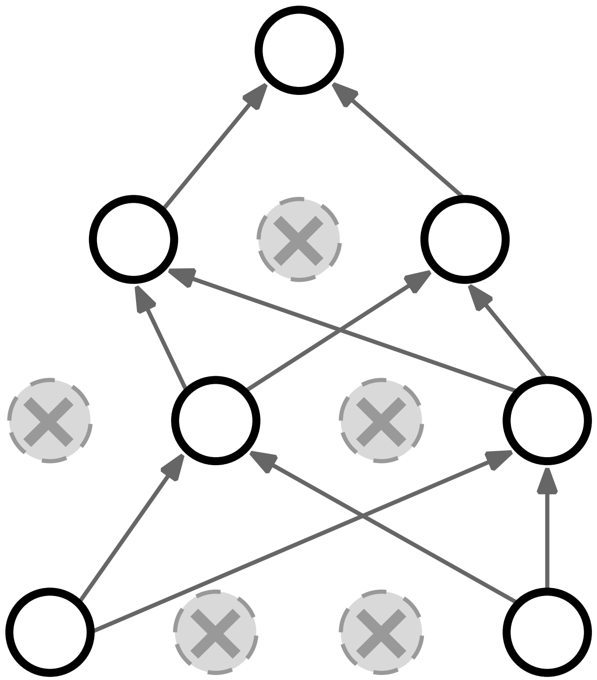
L1 regularization $R(W) = \sum_k \sum_l |W_{k,l}|$

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

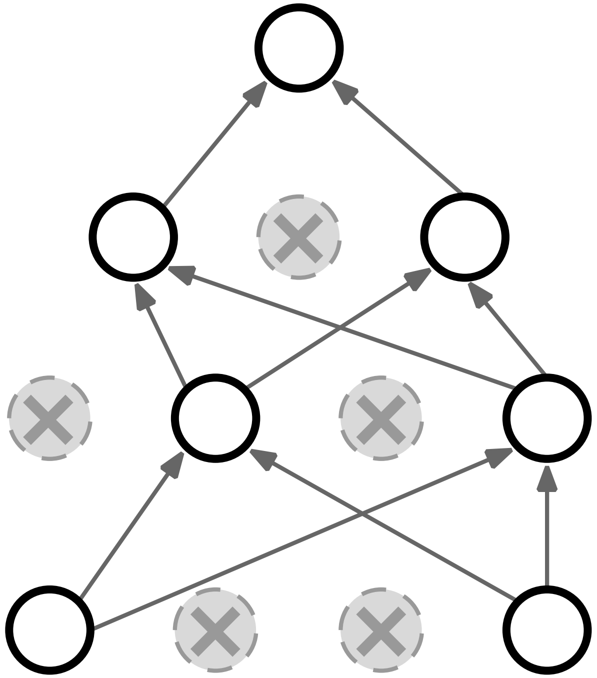




Srivastava et al, “Dropout: A simple way to prevent neural networks from overfitting”, JMLR 2014

Regularization: **Dropout**

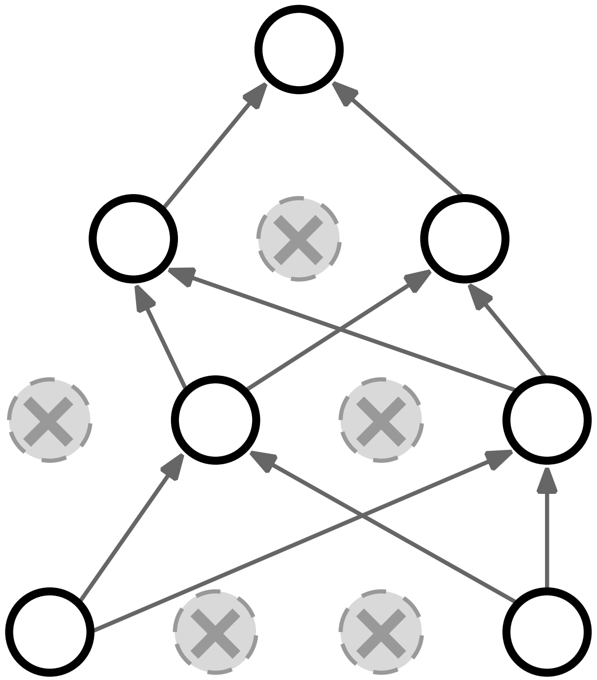
How can this possibly be a good idea?



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

Regularization: **Dropout**

How can this possibly be a good idea?



Another interpretation:

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

Each binary mask is one model

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

Dropout: Test time

Dropout makes our output random!

Output (label)

Input (image)

$$\boxed{y} = f_W(\boxed{x}, \boxed{z})$$

Random mask

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

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

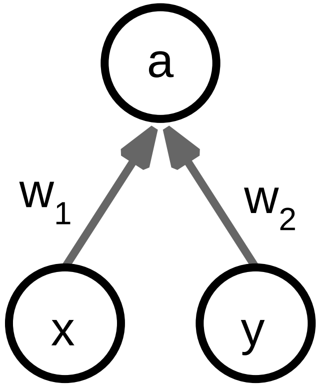
But this integral seems hard …

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.

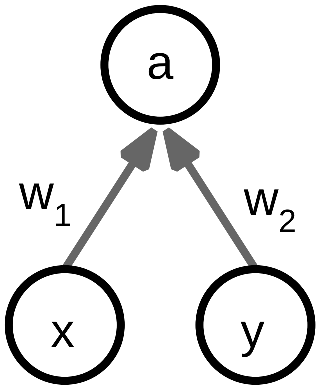


Dropout: Test time

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.



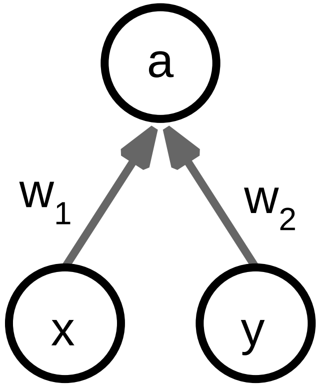
At test time we have: $E\big[a\big] = w_1 x + w_2 y$

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:

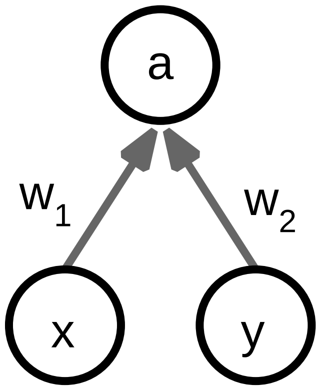
$$\begin{aligned} E\big[a\big] =& \frac{1}{4}(w_1 x + w_2 y) + \frac{1}{4}(w_1 x + 0y) \\ &+ \frac{1}{4}(0x + 0y) + \frac{1}{4}(0x + w_2 y) \\ =& \frac{1}{2}(w_1 x + w_2 y) \end{aligned}$$

Dropout: Test time

Want to approximate the integral

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

Consider a single neuron.



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

During training we have:

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

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

Dropout: Test time

```python
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 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

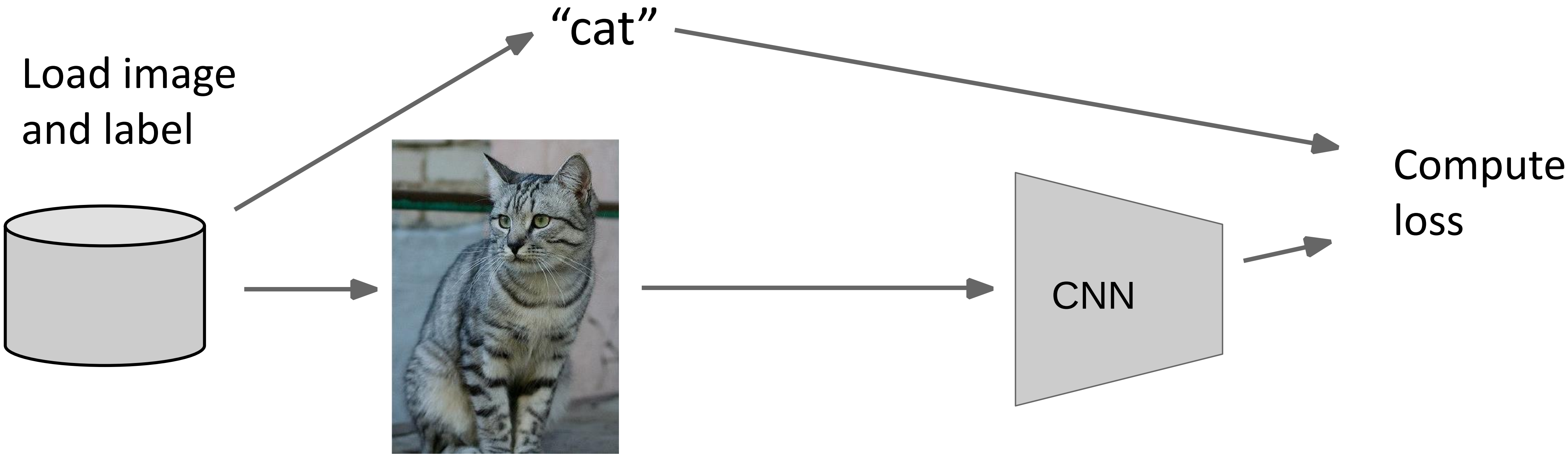
Training: Add some kind of randomness

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

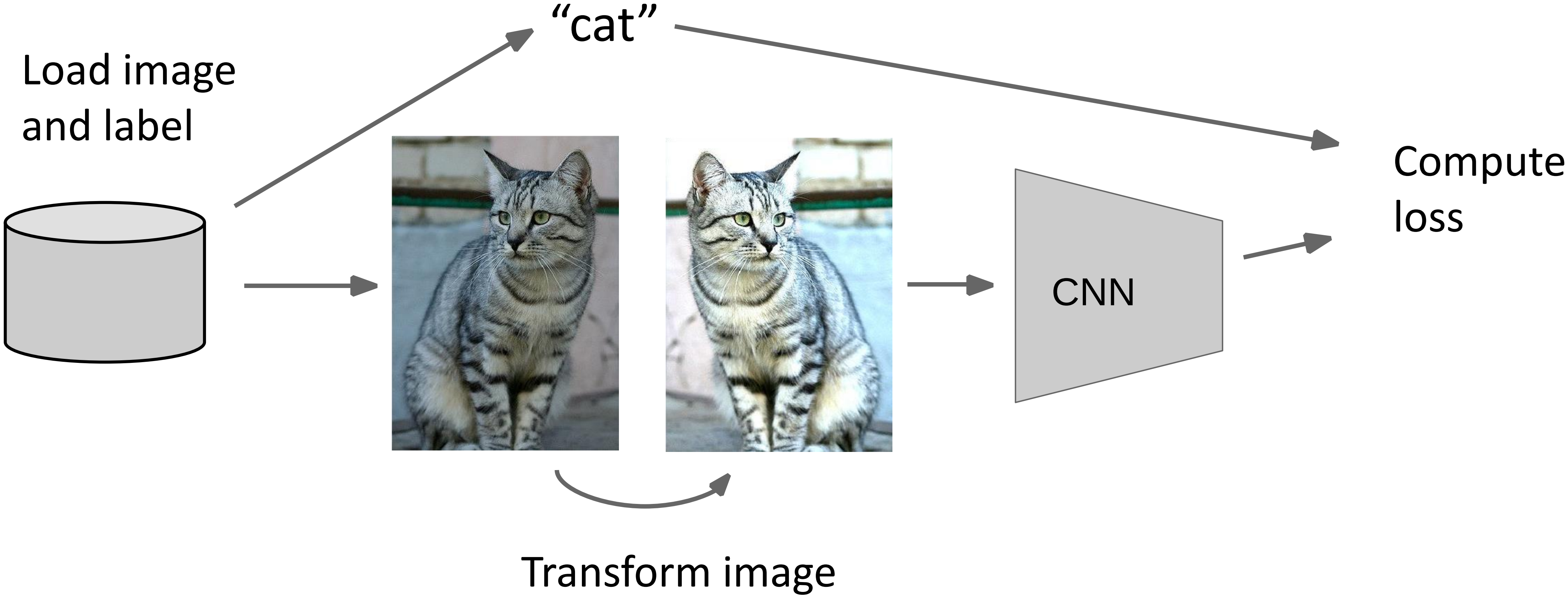
Testing: Average out randomness (sometimes approximate)

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

Regularization: Data Augmentation

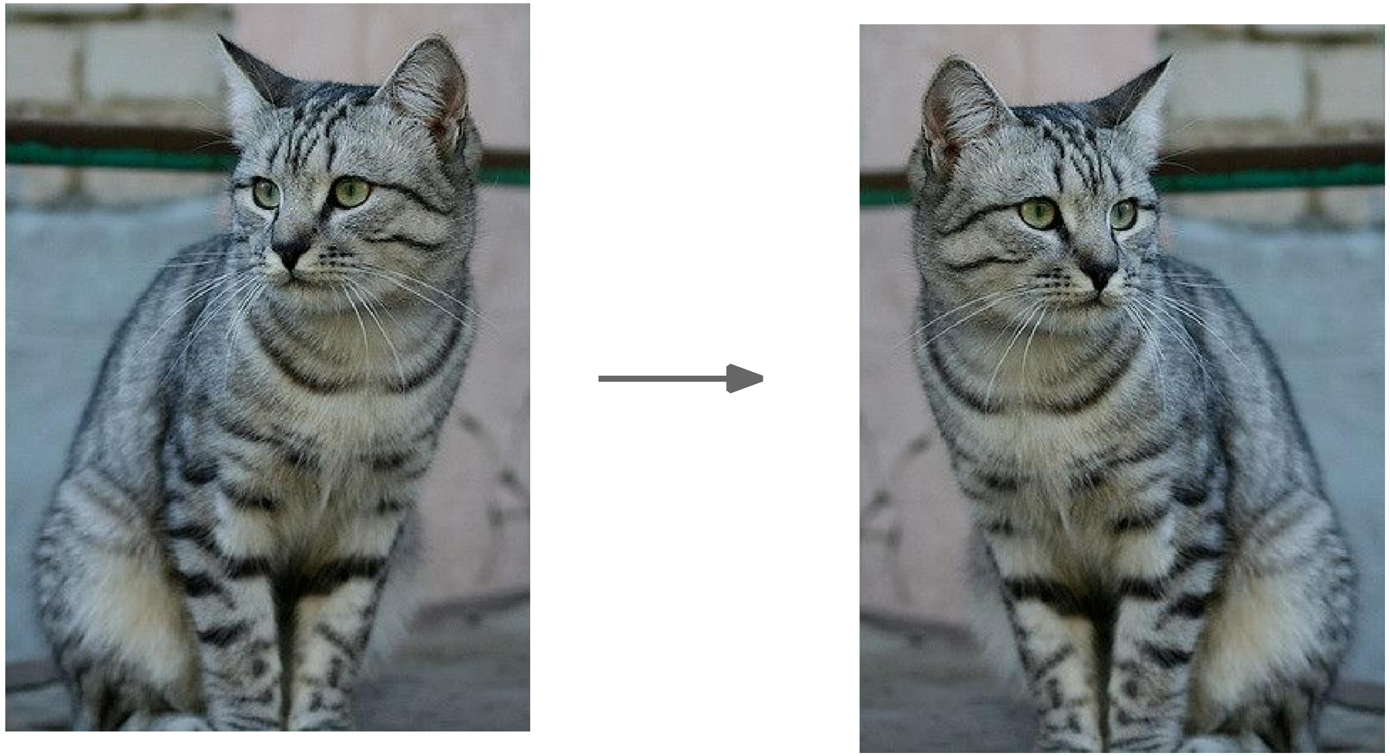


Regularization: Data Augmentation



Data Augmentation

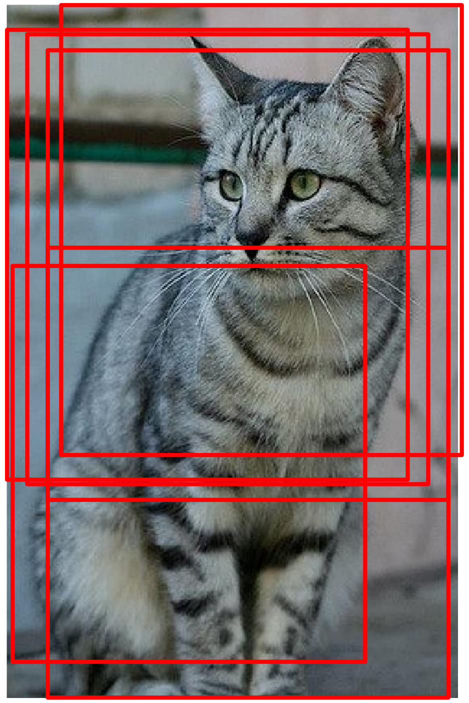
Horizontal Flips



Data Augmentation

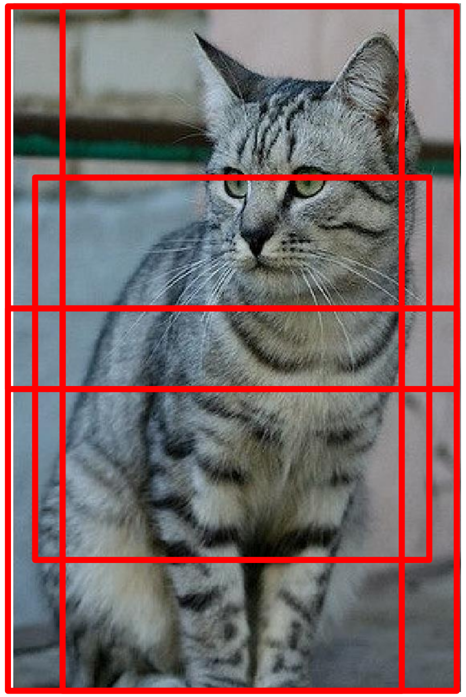
Random crops and scales

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



Data Augmentation

Random crops and scales



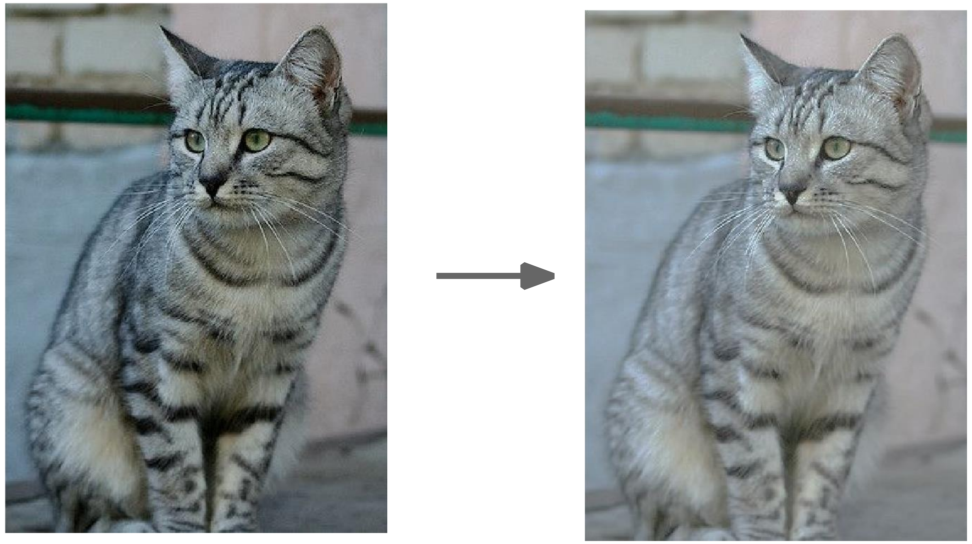
Training: sample random crops / scales
ResNet:
1. Pick random L in range [256, 480]
2. Resize training image, short side = L
3. Sample random 224 x 224 patch

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

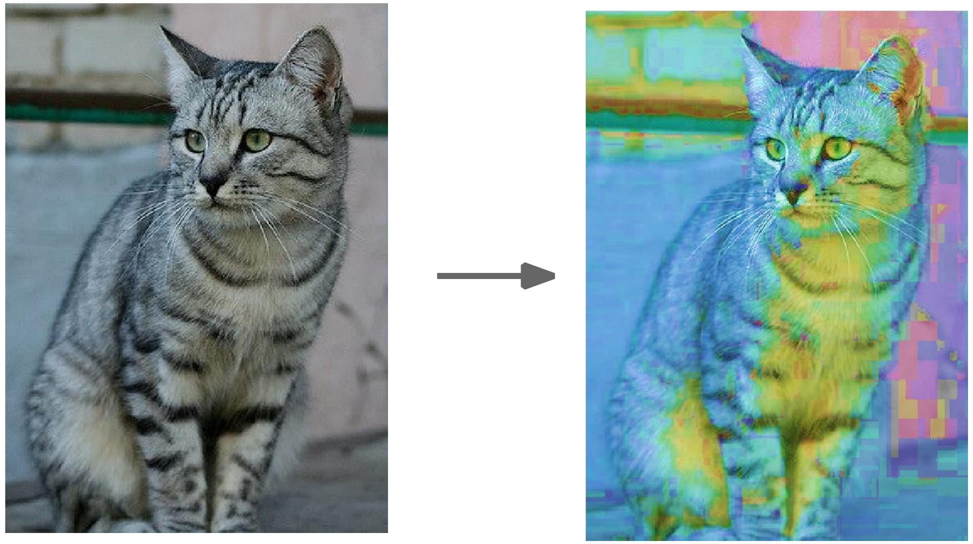
Data Augmentation

Color Jitter

Randomize
contrast and brightness



Randomize
color rotation



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

Transfer Learning

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

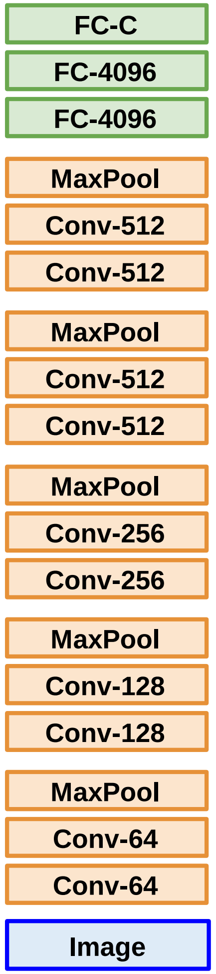
Transfer Learning

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

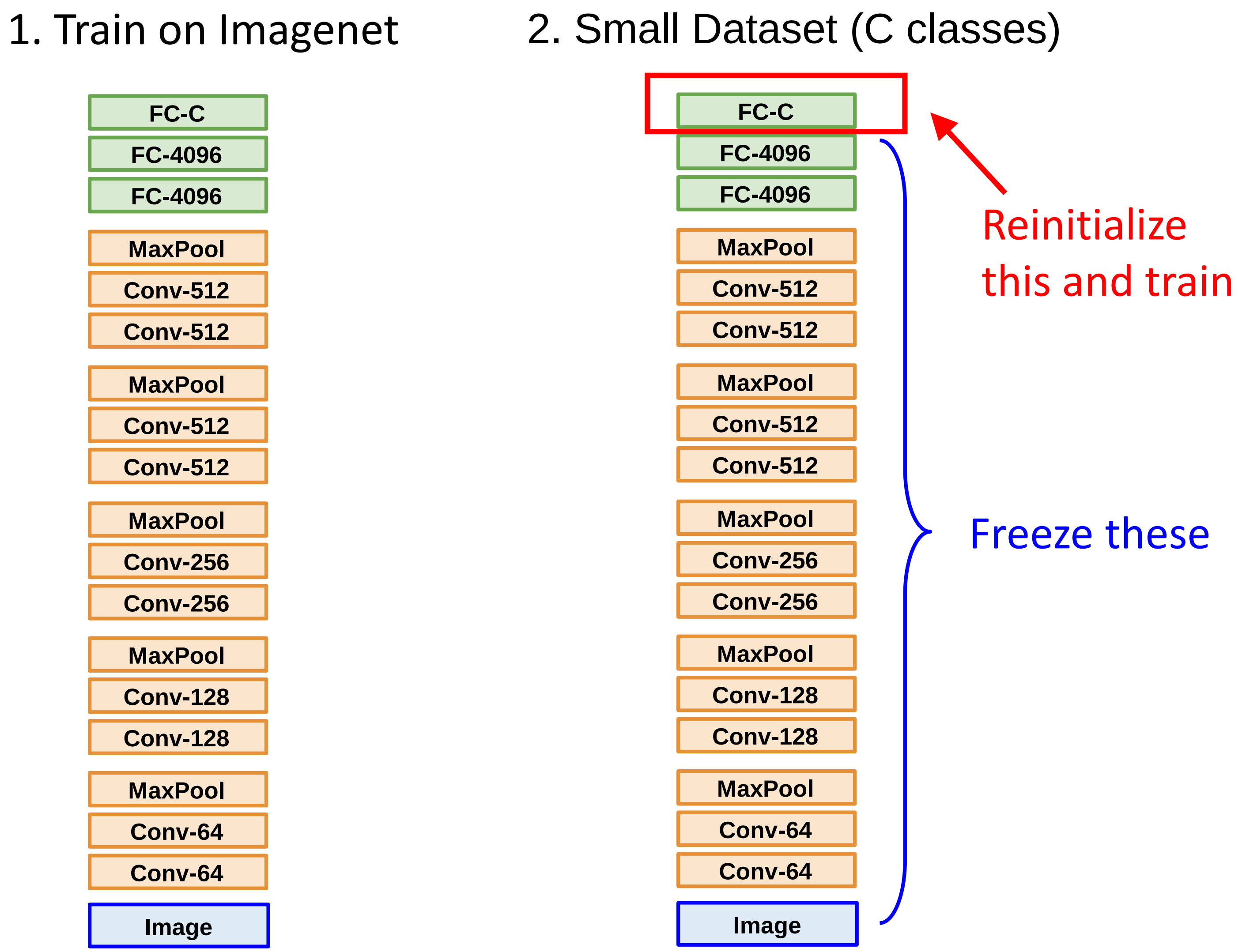
BUSTED

Transfer Learning with CNNs

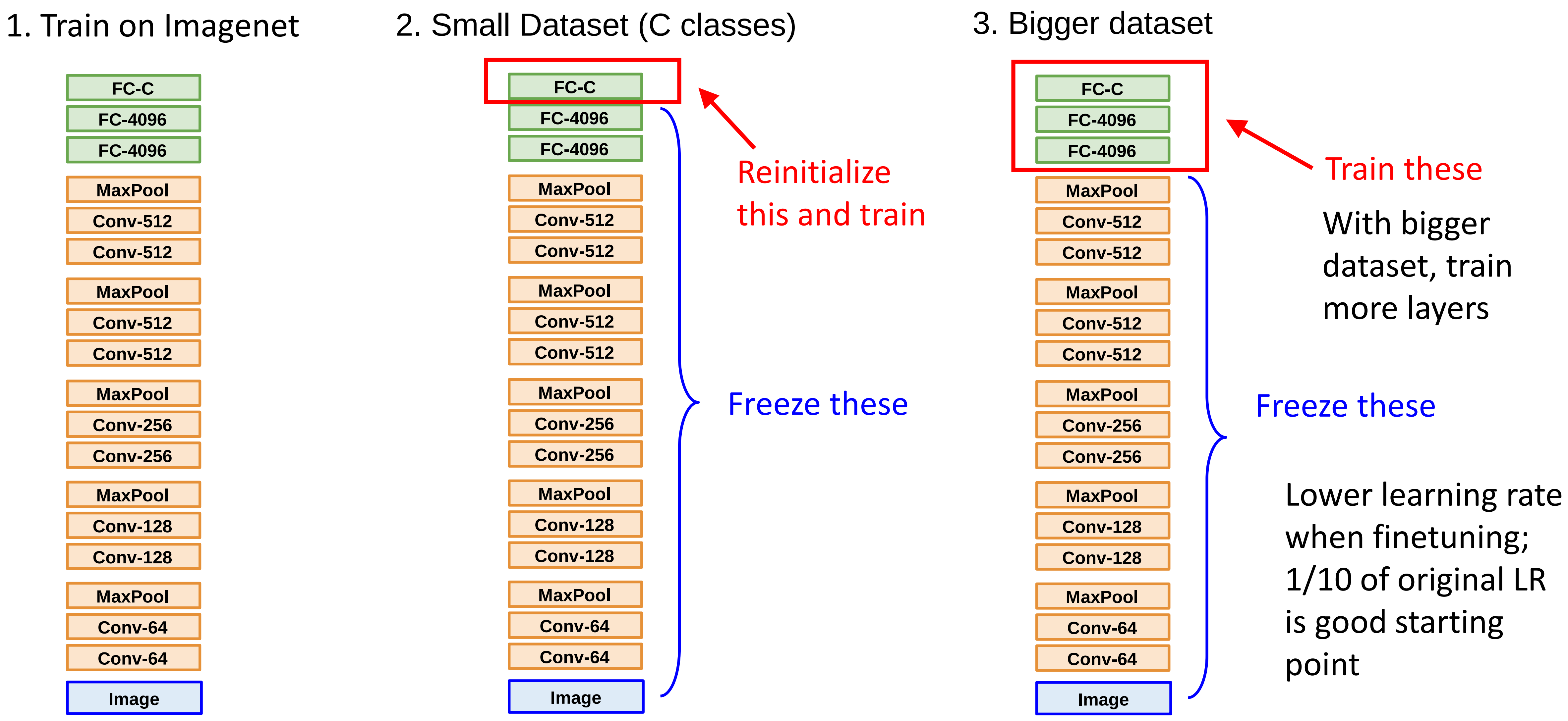
1. Train on Imagenet



Transfer Learning with CNNs



Transfer Learning with CNNs



Transfer learning with CNNs is common

(it's the norm, not an exception)

Object Detection
(Fast R-CNN)

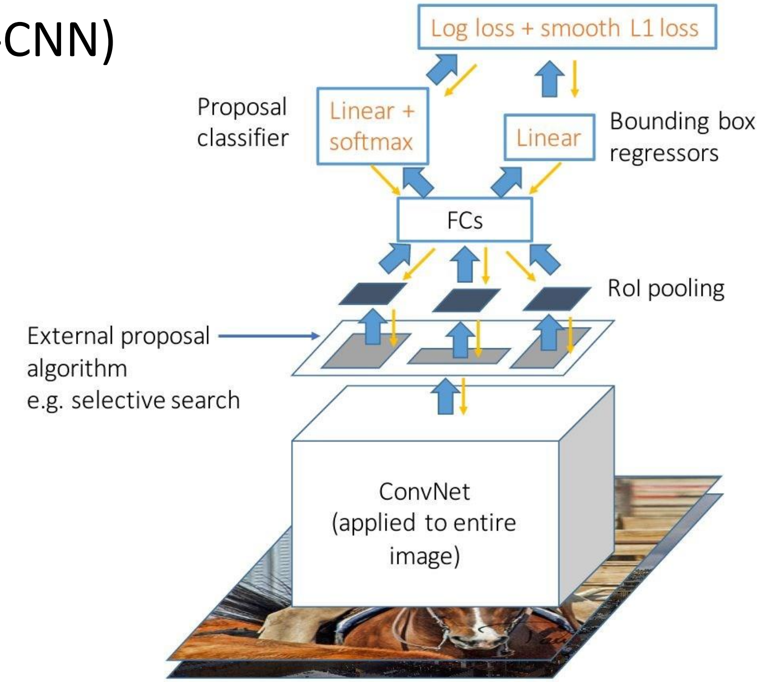
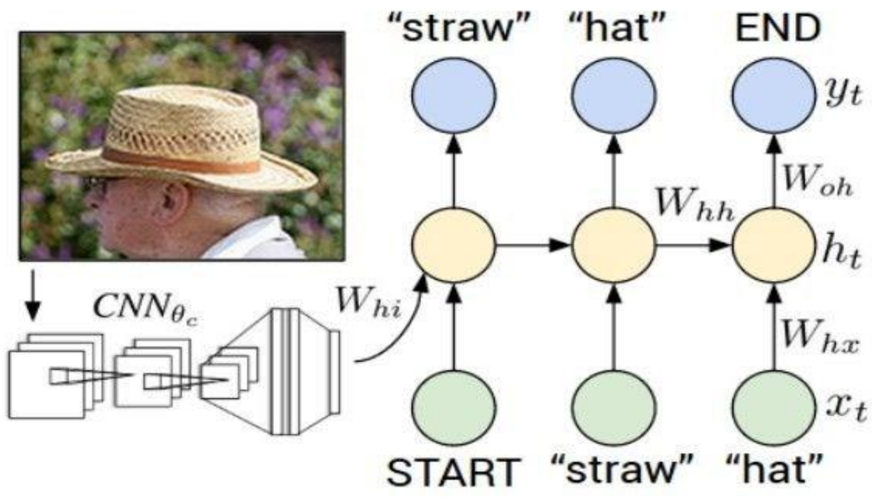


Image Captioning: CNN + RNN



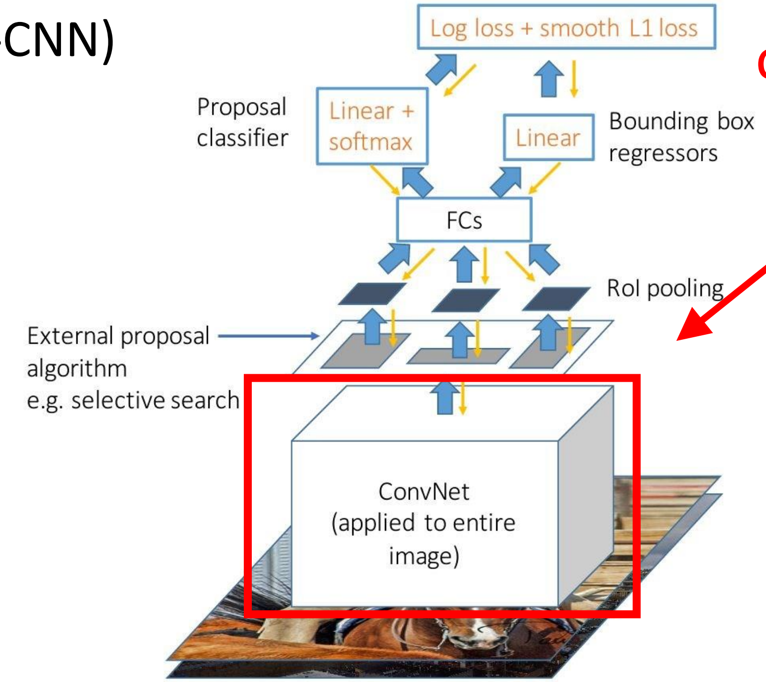
Girshick, "Fast R-CNN", ICCV 2015

Karpathy and Fei-Fei, "Deep Visual-Semantic Alignments for Generating Image Descriptions", CVPR 2015

Transfer learning with CNNs is common

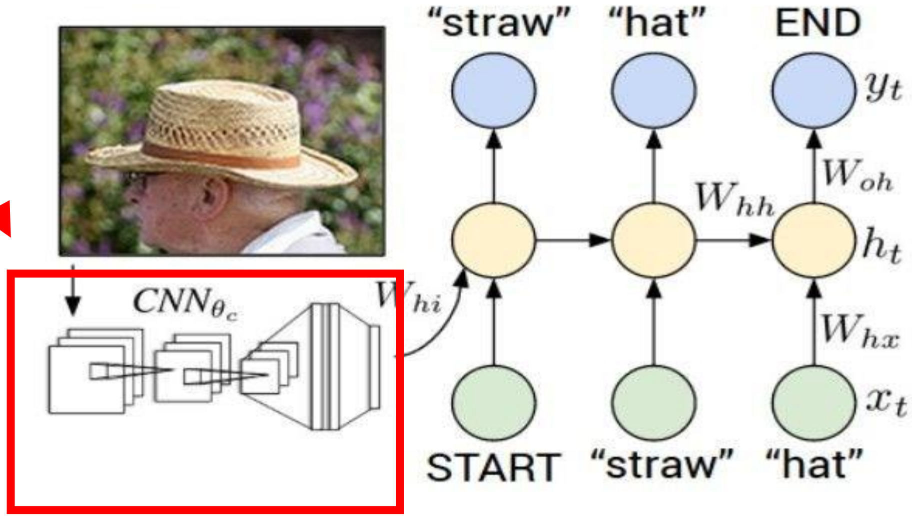
(it's the norm, not an exception)

Object Detection
(Fast R-CNN)



CNN pretrained on ImageNet

Image Captioning: CNN + RNN



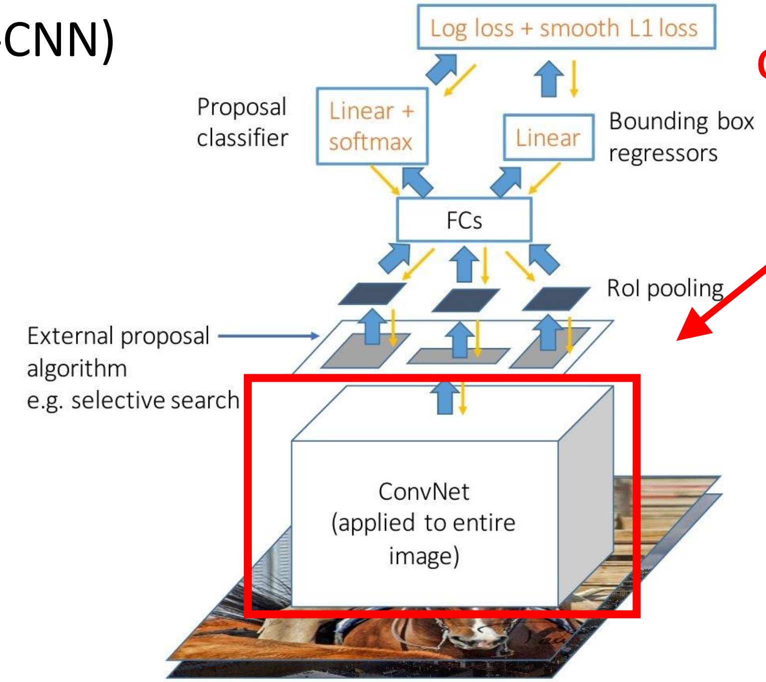
Girshick, "Fast R-CNN", ICCV 2015

Karpathy and Fei-Fei, "Deep Visual-Semantic Alignments for Generating Image Descriptions", CVPR 2015

Transfer learning with CNNs is common

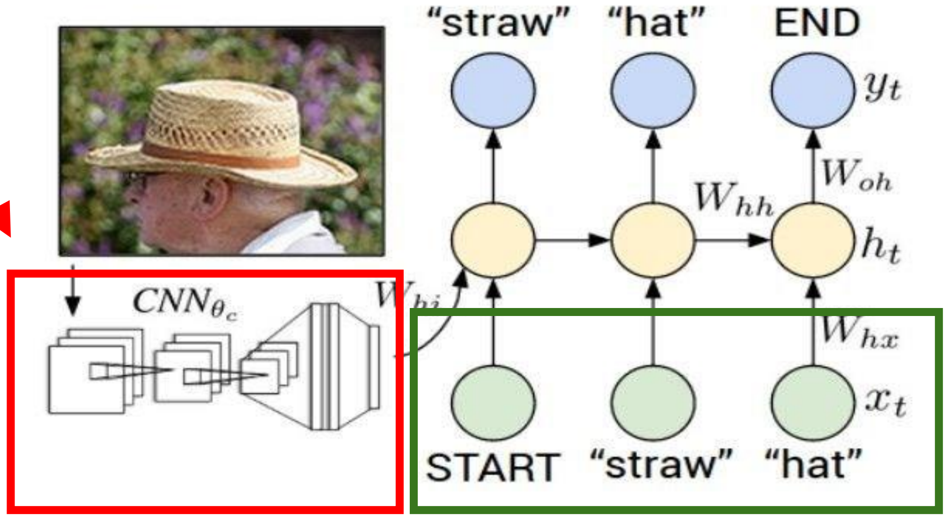
(it's the norm, not an exception)

Object Detection
(Fast R-CNN)



CNN pretrained on ImageNet

Image Captioning: CNN + RNN



Word vectors pretrained with word2vec

Girshick, "Fast R-CNN", ICCV 2015

Karpathy and Fei-Fei, "Deep Visual-Semantic Alignments for Generating Image Descriptions", CVPR 2015