Machine Learning for Image Analysis

Lecture 4: Convolutional Neural Networks: Training

Dagmar Kainmueller, 07.05.2024

Course Outline

- Introduction to Image Analysis
- Basics: Neural Networks
- **Convolutional Neural Networks**
- Transformers
- Model Interpretability
- Self-supervised Learning
- Generative Models (GANs, Diffusion)

Convolutional Neural Networks (CNNs)

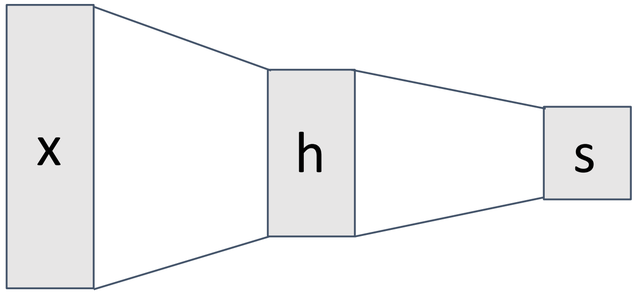
- A bit of CNN history
- CNN Layers
    - Convolutions, Pooling, Fully Connected Layers, Normalization
- Training CNNs
    - Advanced Optimization
    - Weight initialization
    - Regularization: Dropout
    - Extended training data: Augmentation, Transfer Learning
    - Sanity-checking the learning process; Hyperparameter tuning strategies
- Famous CNN architectures
- CNNs for image segmentation

Convolutional Neural Networks (CNNs)

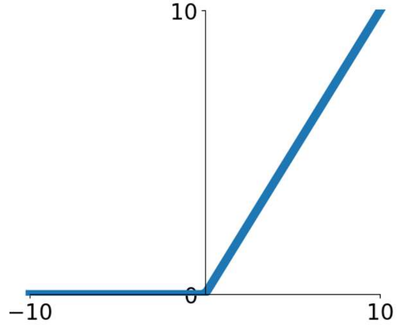
- A bit of CNN history
- CNN Layers
    - Convolutions, Pooling, Fully Connected Layers, Normalization
- Training CNNs
    - Advanced Optimization
    - Weight initialization
    - Regularization: Dropout
    - Extended training data: Augmentation, Transfer Learning
    - Sanity-checking the learning process; Hyperparameter tuning strategies
- Famous CNN architectures
- CNNs for image segmentation

Recap: Convolutional Networks

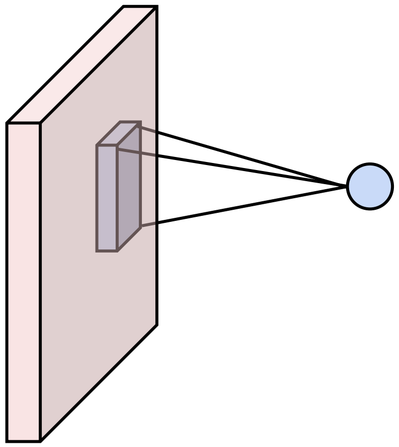
Fully-Connected Layers



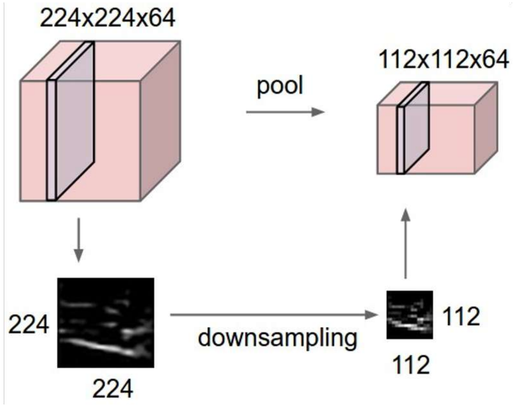
Activation Function



Convolution Layers



Pooling Layers



Normalization

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

Credit: Justin Johnson, U Michigan, Deep Learning for Computer Vision 2019

Convolution Layer: Summary

➡ Accepts an input of dimension $width_{input} \times height_{input} \times depth_{input}$

➡ Four hyper-parameters:

- Number of filters K
- Filter spatial extent F
- Stride S
- Padding P

➡ Produces of output of dimension $width_{output} \times height_{output} \times depth_{output}$ where

- $width_{output} = (width_{input} - F + 2P)/S + 1$
- $height_{output} = (height_{input} - F + 2P)/S + 1$
- $depth_{output} = K$

Convolution Layer: Summary

➡ $F \times F \times depth_{input}$ weights and 1 bias per filter

➡ Total of $(F \times F \times depth_{input}) \times K$ weights and K biases

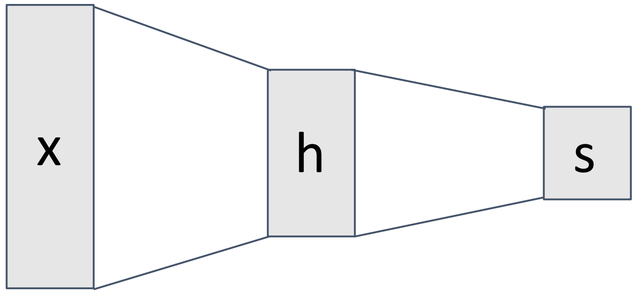
➡ In the output, the nth slice (of size $width_{output} \times height_{output}$) results from convolution of the input with nth filter with stride S and then offset by the nth bias.

➡ Note: usually K = (powers of 2, e.g: 32,64,128,256) “just to be nice”

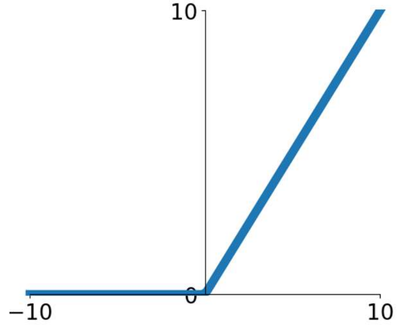
- F = 3, S = 1, P = 1
- F = 5, S = 1, P = 2
- F = 5, S = 2, P = whatever fits
- F = 1, S = 1, P = 0

Components of Convolutional Networks

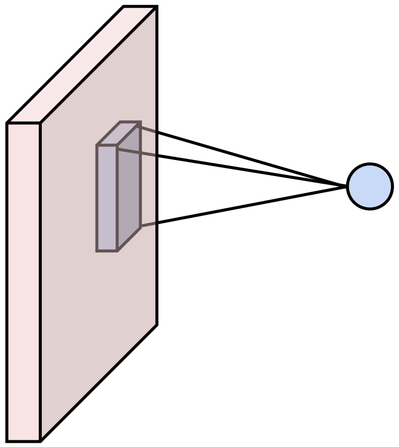
Fully-Connected Layers



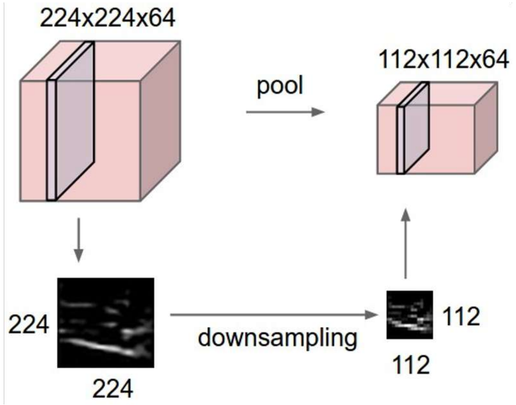
Activation Function



Convolution Layers



Pooling Layers

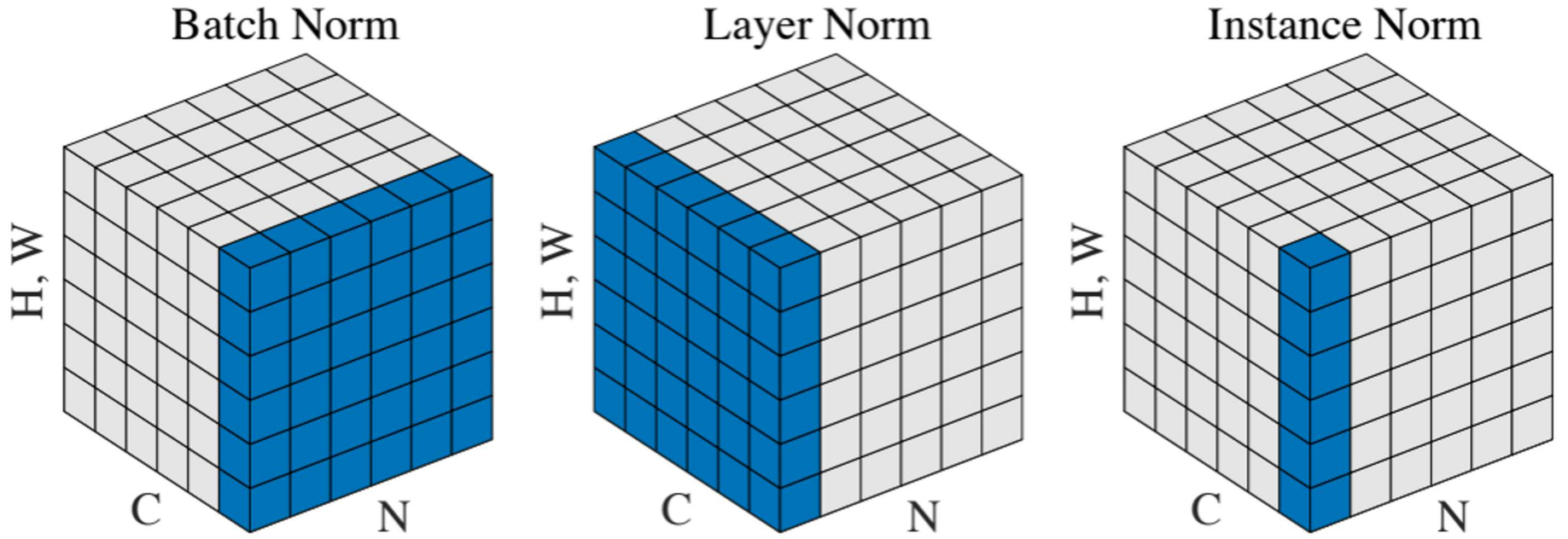


Normalization

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

Credit: Justin Johnson, U Michigan, Deep Learning for Computer Vision 2019

Comparison of Normalization Layers



Wu and He, “Group Normalization”, ECCV 2018

Instance Normalization

Batch Normalization for convolutional networks

$$x : N \times C \times H \times W$$

Normalize

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

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

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

Instance Normalization for convolutional networks

$$x : N \times C \times H \times W$$

Normalize

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

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

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

Convolutional Neural Networks: Summary

- CNNs stack conv-, pooling-, fully connected layers, and normalization
- What is the right way to combine these components?
- Classical classification architectures look like:

  [(conv,ReLU)*N, Pool]*M, (FC,ReLU)*K, Softmax

  Typical M ∈ {5, …, 50}, N ∈ {2, 3}, and K ∈ {0, 1, 2}

- Batch-, Layer- or InstanceNorm after conv or FC for faster convergence

Convolutional Neural Networks (CNNs)

- A bit of CNN history
- CNN Layers
    - Convolutions, Pooling, Fully Connected Layers, Normalization
- Training CNNs
    - Advanced Optimization
    - Weight initialization
    - Regularization: Dropout
    - Extended training data: Augmentation, Transfer Learning
    - Sanity-checking the learning process; Hyperparameter tuning strategies
- Famous CNN architectures
- CNNs for image segmentation

Advanced Optimization

Stochastic Gradient Descent

Momentum

Adam

Which of the following about SGD are true? Check all that apply

- When the training set size m is very large, SGD can be much faster than gradient descent
- The cost function $J_{train}(\theta) = \frac{1}{2m}\sum_{i=q}^{m}(h_\theta(x^{(i)} - y^{(i)})^2$ should go down with every iteration of batch gradient descent (assuming appropriate α) but not necessarily with SGD
- SGD is applicable only to linear regression but not to other models (like logistic regression or neural networks)
- Before beginning the main loop of SGD, it is a good idea to “shuffle” your training data into a random order.

Credits to Andrew Ng (Stanford Machine Learning, 2017, cs229)

Which of the following about SGD are true? Check all that apply

- When the training set size m is very large, SGD can be much faster than gradient descent
- The cost function $J_{train}(\theta) = \frac{1}{2m}\sum_{i=q}^{m}(h_\theta(x^{(i)} - y^{(i)})^2$ should go down with every iteration of batch gradient descent (assuming appropriate α) but not necessarily with SGD
- SGD is applicable only to linear regression but not to other models (like logistic regression or neural networks)
- Before beginning the main loop of SGD, it is a good idea to “shuffle” your training data into a random order.

Credits to Andrew Ng (Stanford Machine Learning, 2017, cs229)

Which of the following about SGD are true? Check all that apply

- When the training set size m is very large, SGD can be much faster than gradient descent
- The cost function $J_{train}(\theta) = \frac{1}{2m}\sum_{i=q}^{m}(h_\theta(x^{(i)} - y^{(i)})^2$ should go down with every iteration of batch gradient descent (assuming appropriate α) but not necessarily with SGD
- SGD is applicable only to linear regression but not to other models (like logistic regression or neural networks)
- Before beginning the main loop of SGD, it is a good idea to “shuffle” your training data into a random order.

Credits to Andrew Ng (Stanford Machine Learning, 2017, cs229)

Which of the following about SGD are true? Check all that apply

- When the training set size m is very large, SGD can be much faster than gradient descent
- The cost function $J_{train}(\theta) = \frac{1}{2m}\sum_{i=q}^{m}(h_\theta(x^{(i)} - y^{(i)})^2$ should go down with every iteration of batch gradient descent (assuming appropriate α) but not necessarily with SGD
- SGD is applicable only to linear regression but not to other models (like logistic regression or neural networks)
- Before beginning the main loop of SGD, it is a good idea to “shuffle” your training data into a random order.

Credits to Andrew Ng (Stanford Machine Learning, 2017, cs229)

Mini-batch Gradient Descent

Batch gradient descent: Use all *m* examples in each iteration
Stochastic gradient descent: Use 1 example in each iteration
Mini-batch gradient descent: Use b examples in each iteration

b = mini-batch size e.g., b= 10

$$\theta_j := \theta_j - \alpha \frac{1}{b} \sum_{i=k \cdot b}^{(k+1) \cdot b - 1} (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)}$$

vectorized computation, parallelize

Credits to Andrew Ng (Stanford Machine Learning, 2017, cs229)

- Suppose you use mini-batch gradient descent on a training set of size m, and you use a mini-batch size of B. The algorithm becomes the same as batch gradient descent if:

  - B = 1
  - B = m/2
  - B = m
  - NOTA

Credits to Andrew Ng (Stanford Machine Learning, 2017, cs229)

- Suppose you use mini-batch gradient descent on a training set of size m, and you use a mini-batch size of B. The algorithm becomes the same as batch gradient descent if:

  - B = 1
  - B = m/2
  - B = m 
  - NOTA

Credits to Andrew Ng (Stanford Machine Learning, 2017, cs229)

Stochastic gradient descent convergence

How to set the learning rate?

Batch gradient descent:

Plot loss as a function of the number of gradient steps

Stochastic gradient descent:

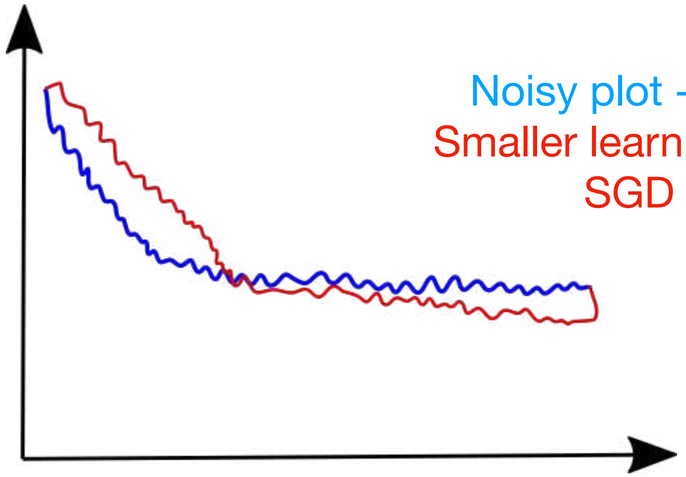
Compute loss contribution of current sample in each gradient step

Every 100 iterations, plot losses averaged over the last 100 samples

Credits to Andrew Ng (Stanford Machine Learning, 2017, cs229)

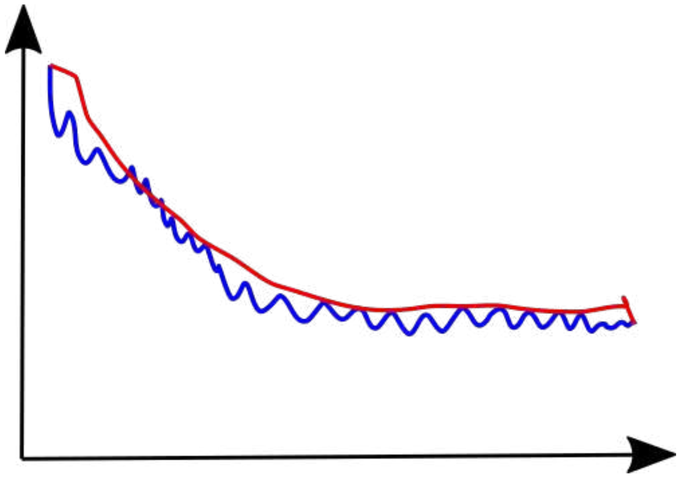
Checking for convergence

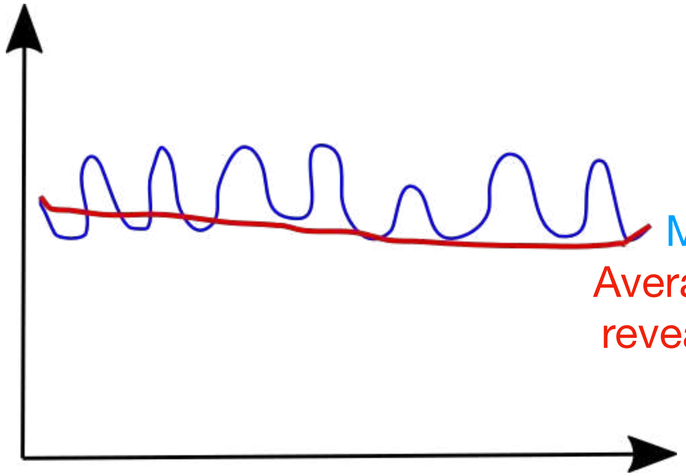
Plot loss averaged over last (e.g.) 1000 samples



Noisy plot - cost is going down
Smaller learning rate -learns slowly
SGD oscillates less

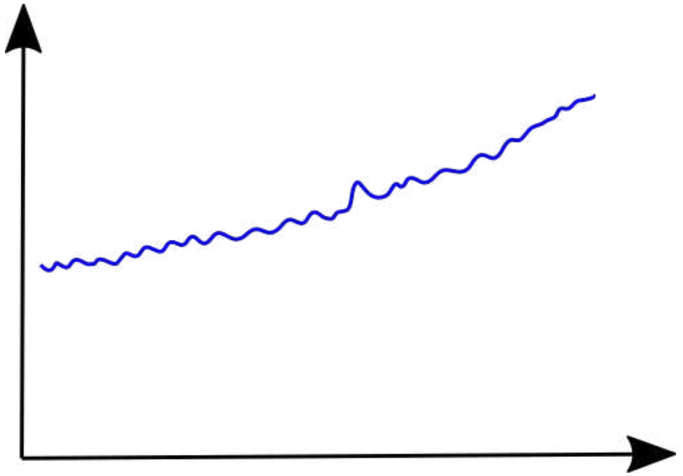
Smoother curve by averaging over more samples





Model is not learning?
Averaging over more samples reveals that cost going down

Diverging model
Decrease learning rate



Credits to Andrew Ng (Stanford Machine Learning, 2017, cs229)

Which of the following about SGD are true? Check all that apply

- Picking a learning rate α that is very small has no disadvantage and can only speed up learning
- If we reduce the learning rate α (and run SGD), its possible that we may find a set of better parameters than with larger α
- If we want SGD to converge to a (local) minimum rather than wander or "oscillate" around it, we should slowly increase α over time.
- If we plot $cost(\theta, (x^{(i)}, y^{(i)}))$ (averaged over the last 1000 examples) and SGD does not seem to be reducing the cost, one possible problem may be that the learning rate α is poorly tuned

Credits to Andrew Ng (Stanford Machine Learning, 2017, cs229)

Which of the following about SGD are true? Check all that apply

- Picking a learning rate α that is very small has no disadvantage and can only speed up learning
- If we reduce the learning rate α (and run SGD), its possible that we may find a set of better parameters than with larger α 
- If we want SGD to converge to a (local) minimum rather than wander or "oscillate" around it, we should slowly increase α over time.
- If we plot $cost(\theta, (x^{(i)}, y^{(i)}))$ (averaged over the last 1000 examples) and SGD does not seem to be reducing the cost, one possible problem may be that the learning rate α is poorly tuned

Credits to Andrew Ng (Stanford Machine Learning, 2017, cs229)

Which of the following about SGD are true? Check all that apply

- Picking a learning rate α that is very small has no disadvantage and can only speed up learning
- If we reduce the learning rate α (and run SGD), its possible that we may find a set of better parameters than with larger α 
- If we want SGD to converge to a (local) minimum rather than wander or "oscillate" around it, we should slowly increase α over time.
- If we plot $cost(\theta, (x^{(i)}, y^{(i)}))$ (averaged over the last 1000 examples) and SGD does not seem to be reducing the cost, one possible problem may be that the learning rate α is poorly tuned 

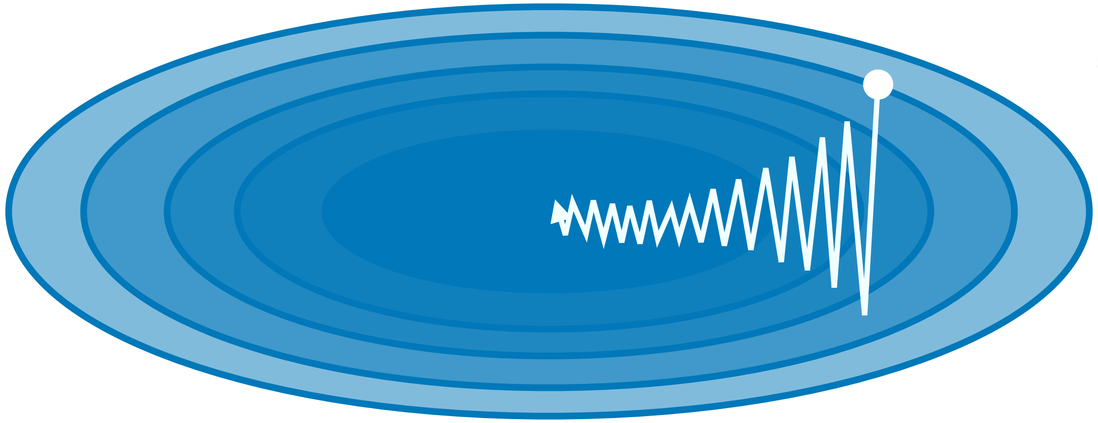
Credits to Andrew Ng (Stanford Machine Learning, 2017, cs229)

Problems of SGD

#1 Jitter

Large gradient in one direction, small gradient in other

⟶ *Slow progress, jitter*

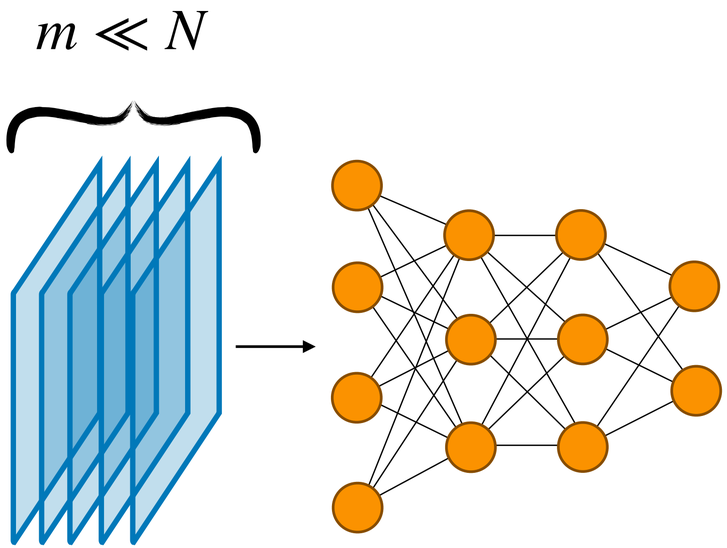


Loss function

High ratio between smallest and largest value in Hessian matrix - loss function has high condition number

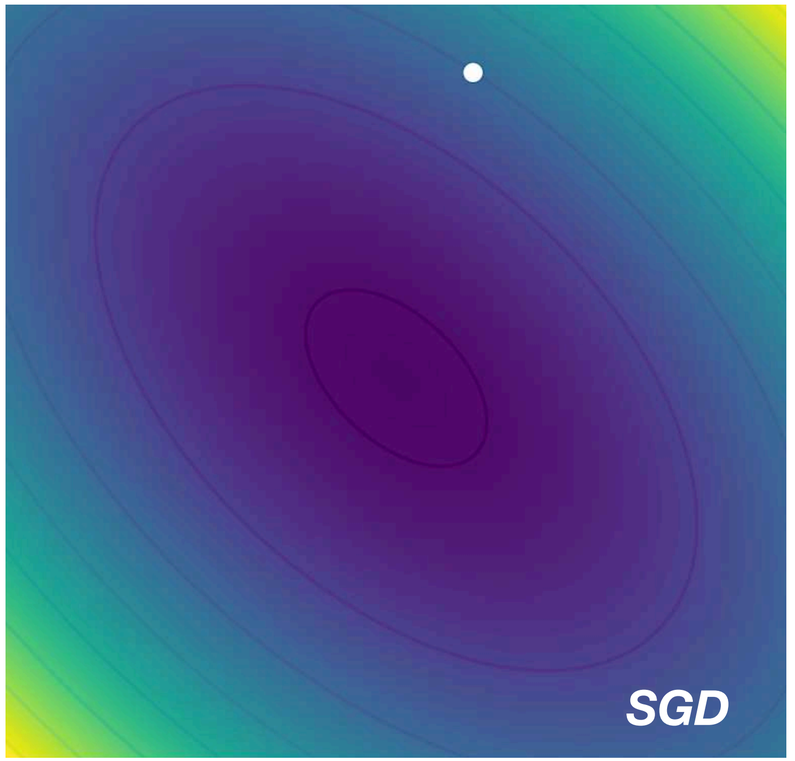
Problems of SGD

#2 noisy gradients



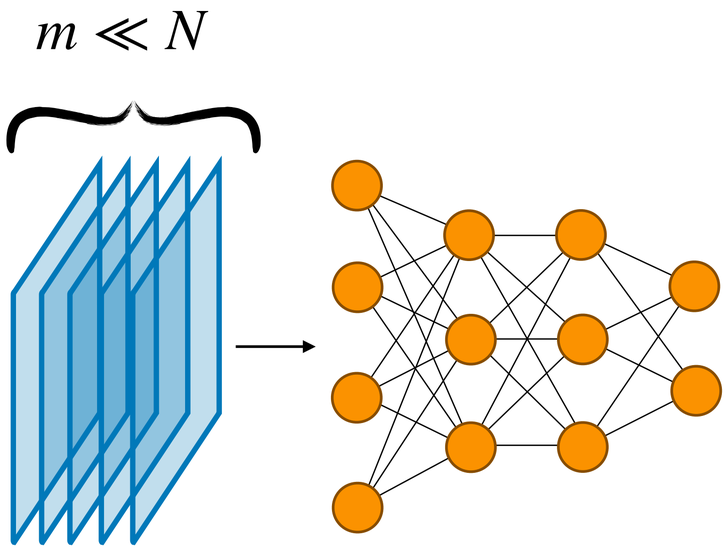
Gradients come from mini-batches

(average gradient of mini-batch)



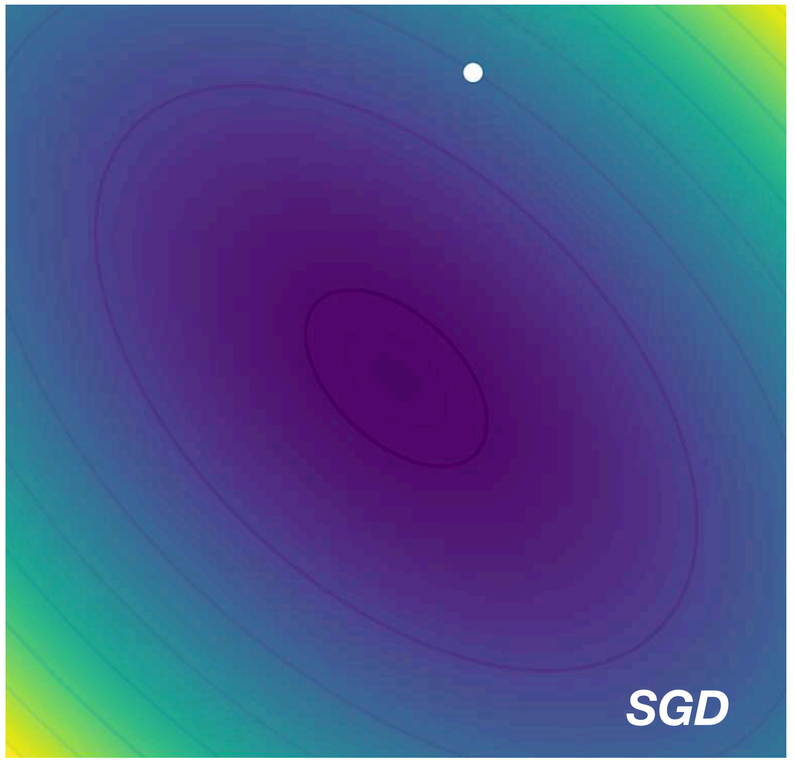
Problems of SGD

#2 noisy gradients



Gradients come from mini-batches

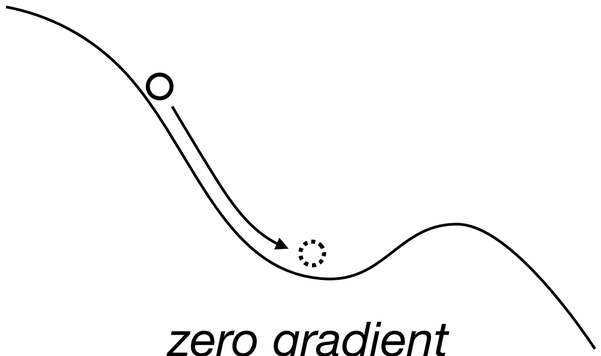
(average gradient of mini-batch)



Problems of SGD

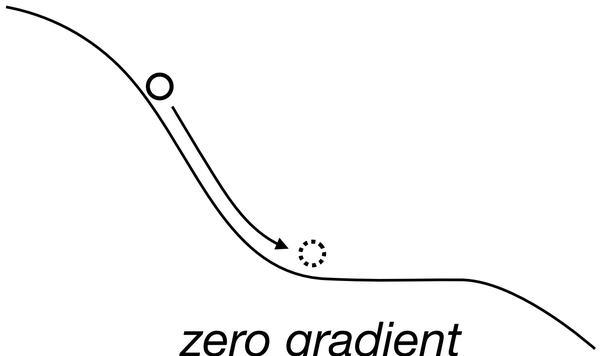
#3 local minima and saddle points

local minimum



zero gradient

saddle point

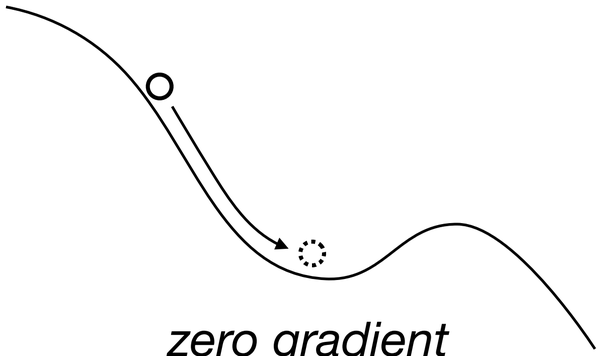


zero gradient

Problems of SGD

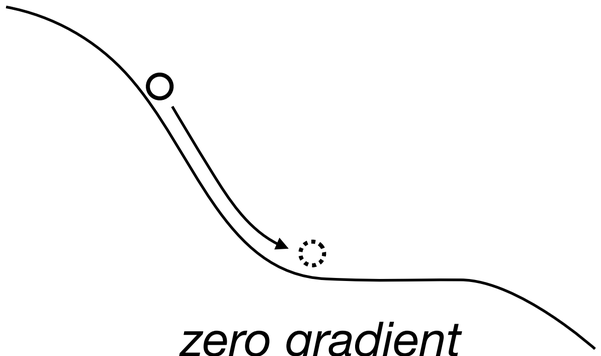
#3 local minima and saddle points

local minimum



zero gradient

saddle point → *More common in high dimensions*

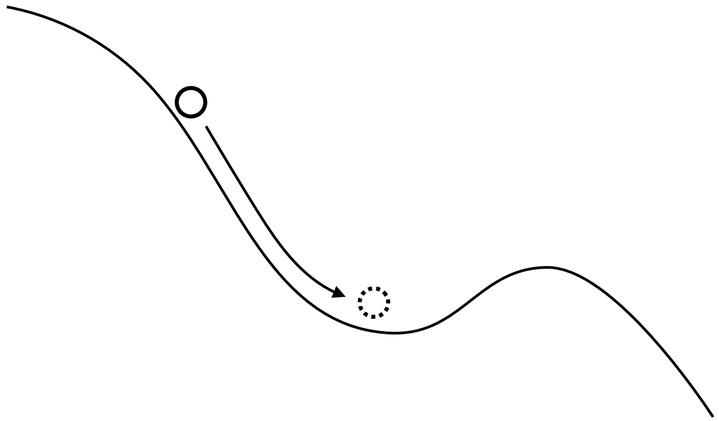


zero gradient

Introducing a momentum

potential picture

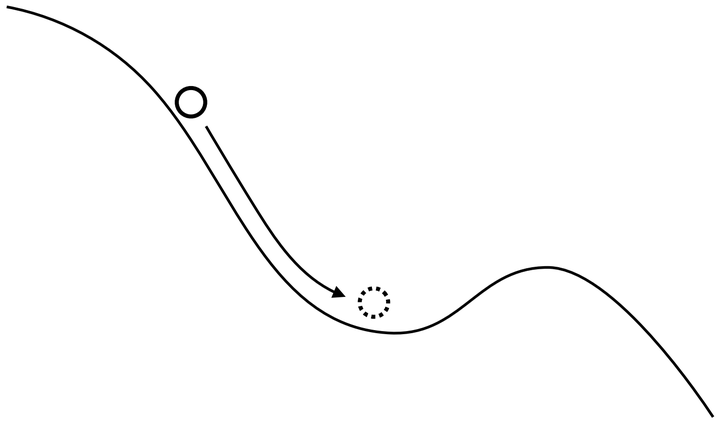
SGD: Weightless particle



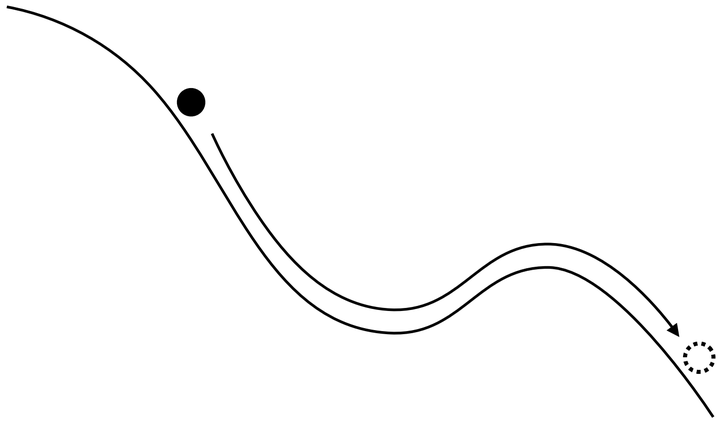
Introducing a momentum

potential picture

SGD: Weightless particle



Introduce mass (momentum)



Introducing a momentum

Algorithm

SGD

$$\theta_{t+1} = \theta_t - \alpha \frac{\partial \varepsilon(\theta)}{\partial \theta}$$

```
for trainingSteps:

  weights += -learning_rate*gradient
```

SGD + Momentum

$$v_{t+1} = \rho v_t + (1-\rho)\frac{\partial \varepsilon(\theta)}{\partial \theta}$$

$$\theta_{t+1} = \theta_t - \alpha v_{t+1}$$

```
vel = 0

for trainingSteps:

  vel      = rho*vel + (1-rho)*gradient
  weights += -learning_rate*vel
```

Introducing a momentum

Algorithm

SGD

$$\theta_{t+1} = \theta_t - \alpha \frac{\partial \varepsilon(\theta)}{\partial \theta}$$

```
for trainingSteps:

  weights += -learning_rate*gradient
```

SGD + Momentum

$$v_{t+1} = \rho v_t + (1-\rho)\frac{\partial \varepsilon(\theta)}{\partial \theta}$$

$$\theta_{t+1} = \theta_t - \alpha v_{t+1}$$

```
vel = 0

for trainingSteps:

  vel      = rho*vel + (1-rho)*gradient
  weights += -learning_rate*vel
```

Friction ρ *typically set to 0.9 or 0.99*

Introducing a momentum

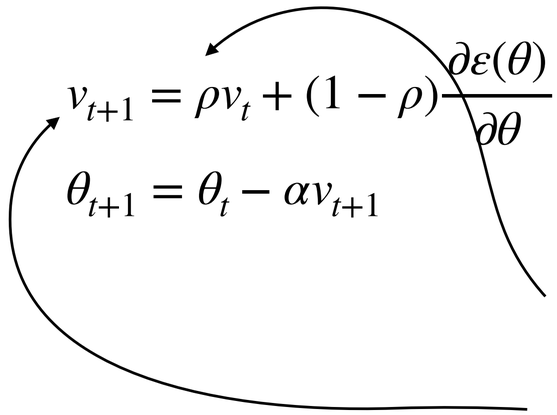
Algorithm

SGD

$$\theta_{t+1} = \theta_t - \alpha \frac{\partial \varepsilon(\theta)}{\partial \theta}$$

```
for trainingSteps:

  weights += -learning_rate*gradient
```

SGD + Momentum




```
vel = 0

for trainingSteps:

  vel      = rho*vel + (1-rho)*gradient
  weights += -learning_rate*vel
```

Friction ρ *typically set to 0.9 or 0.99*

(negative) ***Velocity*** v *is an accumulation of gradients (exponential decaying influence)*

Introducing a momentum

Algorithm

SGD

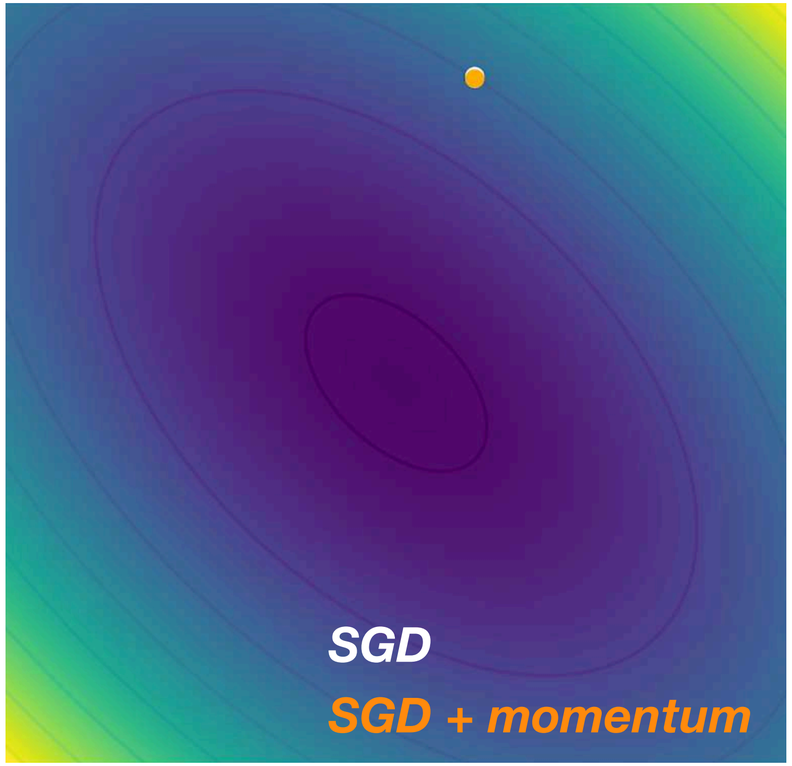
$$\theta_{t+1} = \theta_t - \alpha \frac{\partial \varepsilon(\theta)}{\partial \theta}$$

SGD + Momentum

$$v_{t+1} = \rho v_t + (1 - \rho) \frac{\partial \varepsilon(\theta)}{\partial \theta}$$

$$\theta_{t+1} = \theta_t - \alpha v_{t+1}$$

Tackles previous problems (#1 jitter, #2 noisy gradients, #3 local minima / saddle points)



Introducing a momentum

Algorithm

SGD

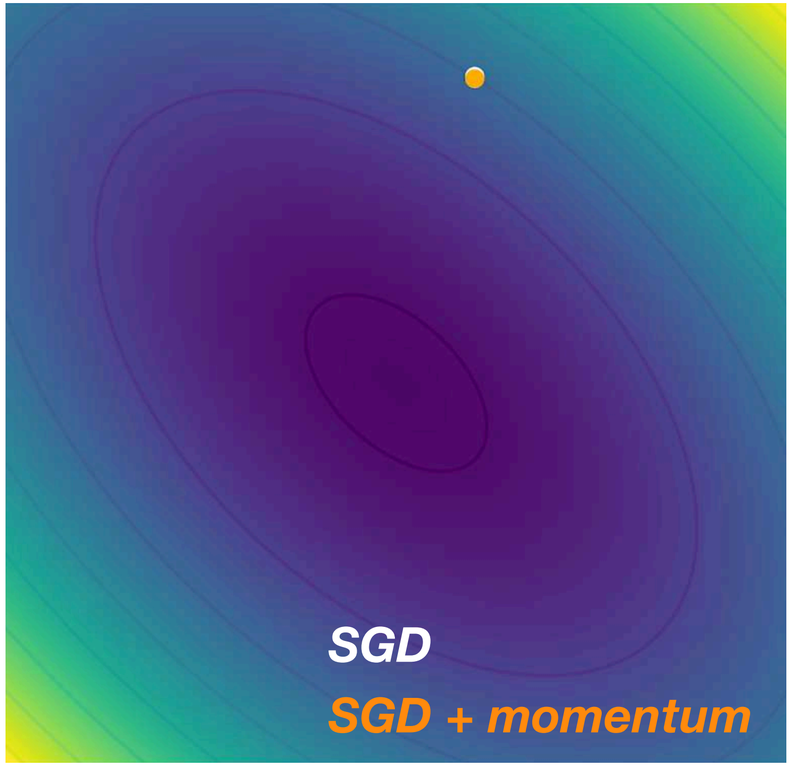
$$\theta_{t+1} = \theta_t - \alpha \frac{\partial \varepsilon(\theta)}{\partial \theta}$$

SGD + Momentum

$$v_{t+1} = \rho v_t + (1 - \rho) \frac{\partial \varepsilon(\theta)}{\partial \theta}$$

$$\theta_{t+1} = \theta_t - \alpha v_{t+1}$$

Tackles previous problems (#1 jitter, #2 noisy gradients, #3 local minima / saddle points)

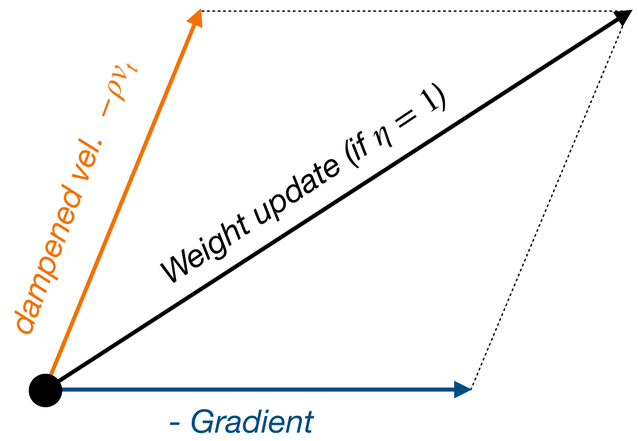


Nesterow Momentum

Momentum:

$$v_{t+1} = \rho v_t + (1-\rho)\nabla\varepsilon(\theta)$$

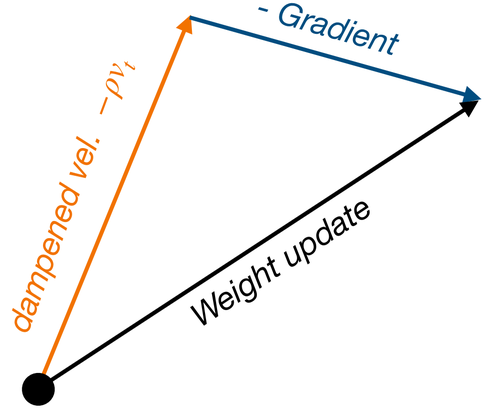
$$\theta_{t+1} = \theta_t - \eta v_{t+1}$$



Nesterov Momentum:

$$v_{t+1} = \rho v_t + (1-\rho)\nabla\varepsilon(\theta_t - \eta\rho v_t)$$

$$\theta_{t+1} = \theta_t - \eta v_{t+1}$$



AdaGrad

SGD

```
for _ in range(training_steps):

  grad = compute_gradients(weights)
  weights += -learning_rate * grad
```

AdaGrad

```
grad_squared = 0

for _ in range(training_steps):

  grad = compute_gradients(weights)
  grad_squared += grad * grad
  weights += -learning_rate * grad / (np.sqrt(grad_squared) + 1e-7)
```

AdaGrad

SGD

```
for _ in range(training_steps):

  grad = compute_gradients(weights)
  weights += -learning_rate * grad
```

AdaGrad

```
grad_squared = 0

for _ in range(training_steps):

  grad = compute_gradients(weights)
  grad_squared += grad * grad
  weights += -learning_rate * grad / (np.sqrt(grad_squared) + 1e-7)
```

Scaling gradients in each dimension by accumulation of squared gradients

RMSprop

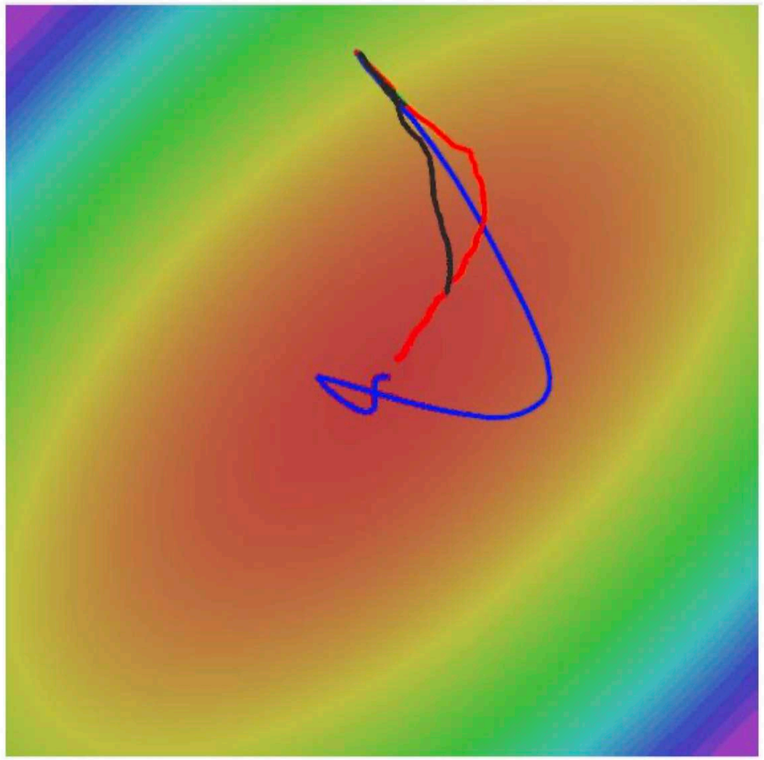
AdaGrad

```
grad_squared = 0

for _ in range(training_steps):

  grad = compute_gradients(weights)
  grad_squared += grad * grad
  weights += -learning_rate * grad / (np.sqrt(grad_squared) + 1e-7)
```

RMSProp

```
grad_squared = 0

for _ in range(training_steps):

  grad = compute_gradients(weights)
  grad_squared += decay_rate * grad_squared + (1 - decay_rate) * grad * grad
  weights += -learning_rate * grad / (np.sqrt(grad_squared) + 1e-7)
```

RMSprop



Adam

Derivation

```
first_moment = 0
second_moment = 0

for _ in range(training_steps):

  grad = compute_gradients(weights)

  first_moment  = beta1 * first_moment  + (1 - beta1) * grad
  second_moment = beta2 * second_moment + (1 - beta2) * grad * grad

  weights -= learning_rate * first_moment / (np.sqrt(second_moment) + 1e-7)
```

Adam

Derivation

```
first_moment = 0
second_moment = 0

for _ in range(training_steps):

  grad = compute_gradients(weights)

  first_moment  = beta1 * first_moment  + (1 - beta1) * grad
  second_moment = beta2 * second_moment + (1 - beta2) * grad * grad

  weights -= learning_rate * first_moment / (np.sqrt(second_moment) + 1e-7)
```

Momentum

Adam

Derivation

```
first_moment = 0
second_moment = 0

for _ in range(training_steps):

  grad = compute_gradients(weights)

  first_moment  = beta1 * first_moment  + (1 - beta1) * grad
  second_moment = beta2 * second_moment + (1 - beta2) * grad * grad

  weights -= learning_rate * first_moment / (np.sqrt(second_moment) + 1e-7)
```

Momentum

AdaGrad / RMSProp

Adam

Derivation

```
first_moment = 0
second_moment = 0

for _ in range(training_steps):

  grad = compute_gradients(weights)

  first_moment  = beta1 * first_moment  + (1 - beta1) * grad
  second_moment = beta2 * second_moment + (1 - beta2) * grad * grad

  weights -= learning_rate * first_moment / (np.sqrt(second_moment) + 1e-7)
```

Momentum

AdaGrad / RMSProp

Q: What happens at first timestep?

Adam

Algorithm

```
first_moment = 0
second_moment = 0

for t in range(1, training_steps + 1):

  grad = compute_gradients(weights)

  first_moment  = beta1 * first_moment  + (1 - beta1) * grad
  second_moment = beta2 * second_moment + (1 - beta2) * grad * grad

  first_unbias  = first_moment  / (1 - beta1 ** t)
  second_unbias = second_moment / (1 - beta2 ** t)

  weights -= learning_rate * first_unbias / (np.sqrt(second_unbias)  +1e-7)
```

Momentum

AdaGrad / RMSProp

Adam

Algorithm

```
first_moment = 0
second_moment = 0

for t in range(1, training_steps + 1):

  grad = compute_gradients(weights)

  first_moment  = beta1 * first_moment  + (1 - beta1) * grad
  second_moment = beta2 * second_moment + (1 - beta2) * grad * grad

  first_unbias  = first_moment  / (1 - beta1 ** t)
  second_unbias = second_moment / (1 - beta2 ** t)

  weights -= learning_rate * first_unbias / (np.sqrt(second_unbias)  +1e-7)
```

Momentum

Bias correction *accounting for first / second moment estimates starting at zero*

AdaGrad / RMSProp

Adam

Algorithm

```
first_moment = 0
second_moment = 0

for t in range(1, training_steps + 1):

  grad = compute_gradients(weights)

  first_moment  = beta1 * first_moment  + (1 - beta1) * grad
  second_moment = beta2 * second_moment + (1 - beta2) * grad * grad

  first_unbias  = first_moment  / (1 - beta1 ** t)
  second_unbias = second_moment / (1 - beta2 ** t)

  weights -= learning_rate * first_unbias / (np.sqrt(second_unbias)  +1e-7)
```

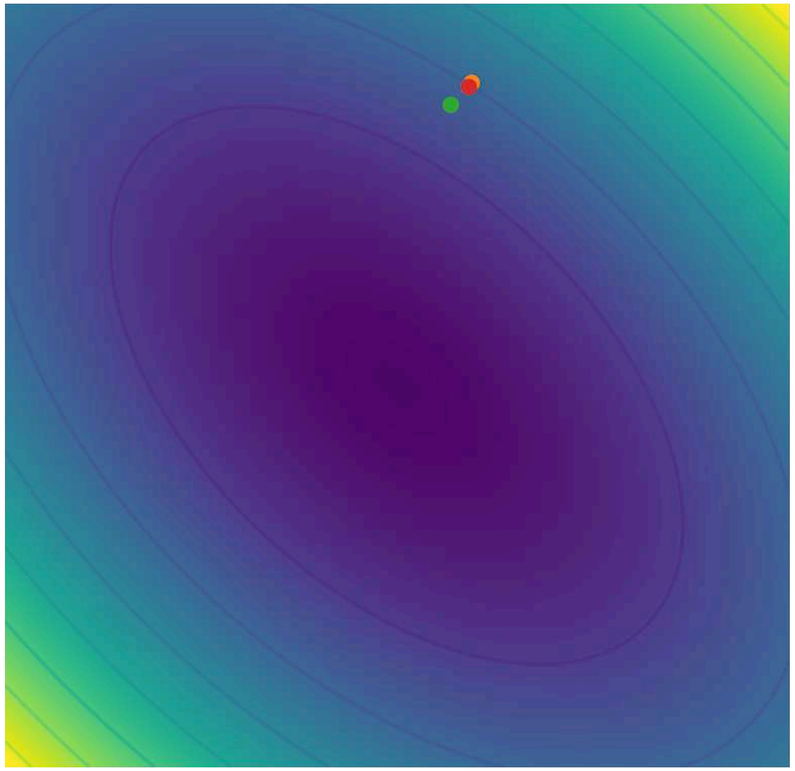
Momentum

Bias correction *accounting for first / second moment estimates starting at zero*

AdaGrad / RMSProp

Practical tip: *Adam with beta1 = 0.9 / 0.999 and learning_rate = 1e-3 / 5e-4 is good first choice for many models.*

Adam



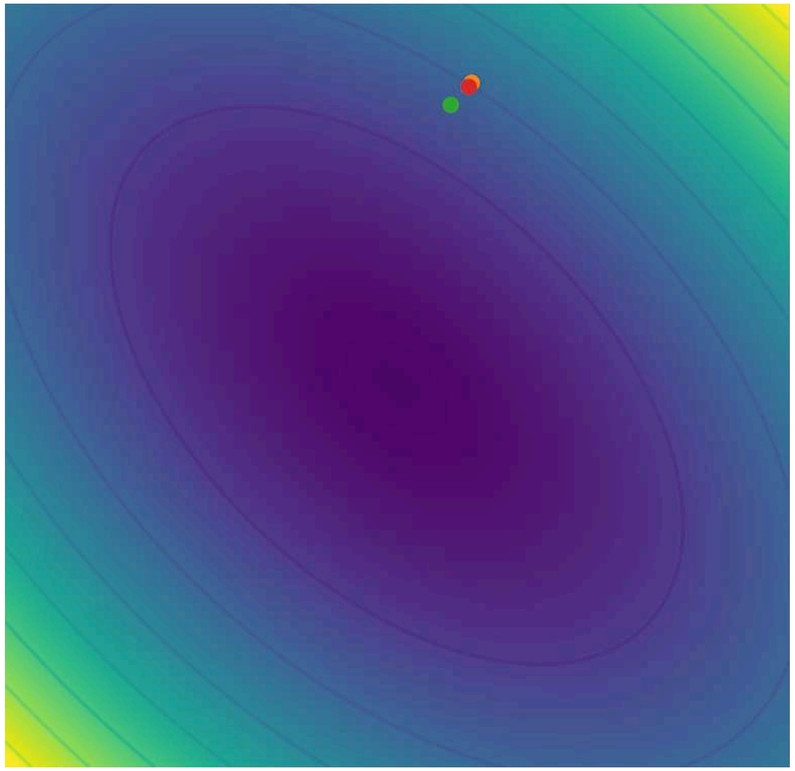
SGD

SGD + momentum

RMSProp

Adam

Adam



SGD

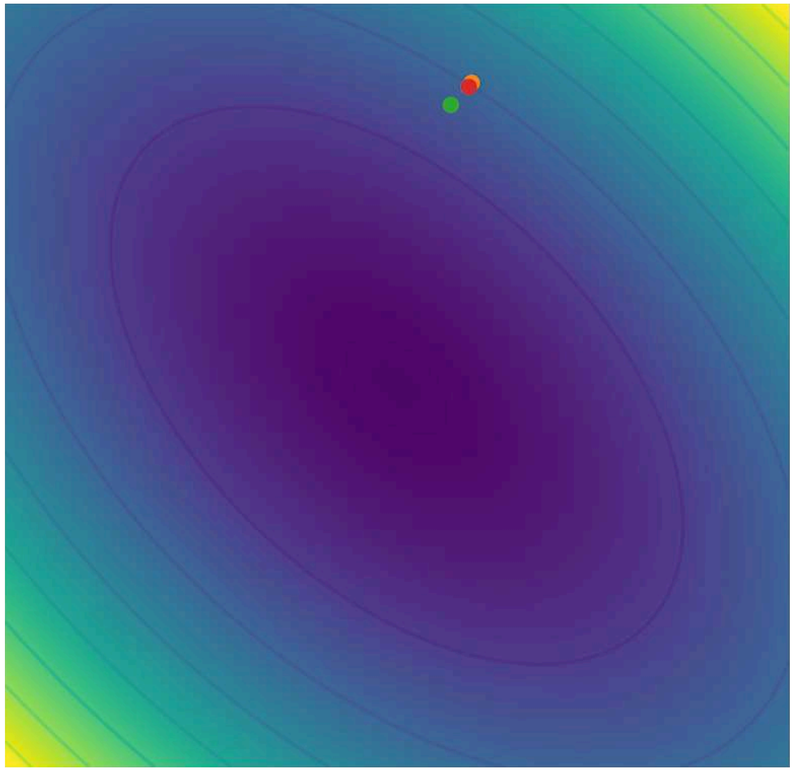
SGD + momentum

RMSProp

Adam

Adam

AdamW: https://arxiv.org/pdf/1711.05101



SGD

SGD + momentum

RMSProp

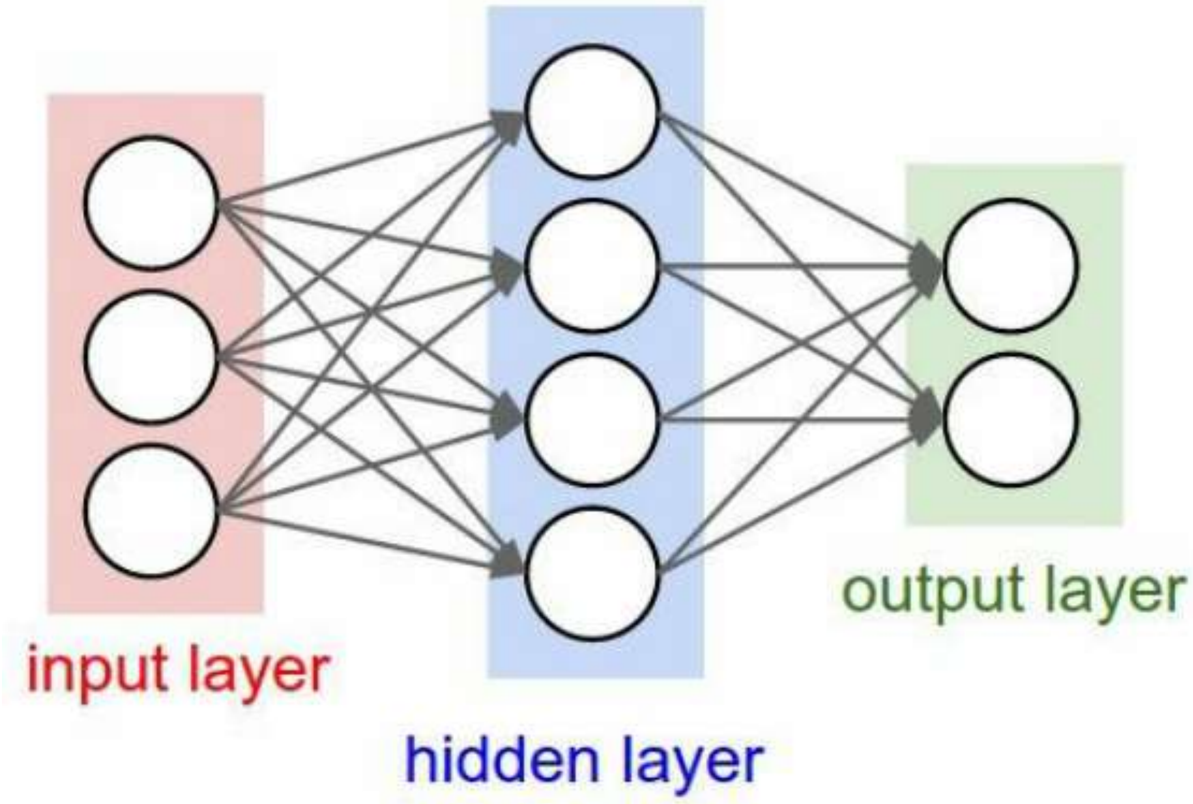
Adam

Convolutional Neural Networks (CNNs)

- A bit of CNN history
- CNN Layers
    - Convolutions, Pooling, Fully Connected Layers, Normalization
- Training CNNs
    - Advanced Optimization
    - Weight initialization
    - Regularization: Dropout
    - Extended training data: Augmentation, Transfer Learning
    - Sanity-checking the learning process; Hyperparameter tuning strategies
- Famous CNN architectures
- CNNs for image segmentation

Weight Initialization

Q: What happens when $\theta \equiv$ const initialization is used? E.g., $\theta \equiv 0$?



Credits to Fei-Fei Li, Justin Johnson & Serena Yeung (Stanford CNNs for Visual Recognition, 2017, cs231)

First idea: **Small random numbers**

(gaussian with zero mean and 0.01 standard deviation)

```
W = 0.01* np.random.randn(D,H)
```

Works ~okay for small networks, but problems with deeper networks.

Weight Initialization: Activation Statistics

```
dims = [4096] * 7
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = 0.01 * np.random.randn(Din, Dout)
    x = np.tanh(x.dot(W))
    hs.append(x)
```

Forward pass for a 6-layer net with hidden size 4096

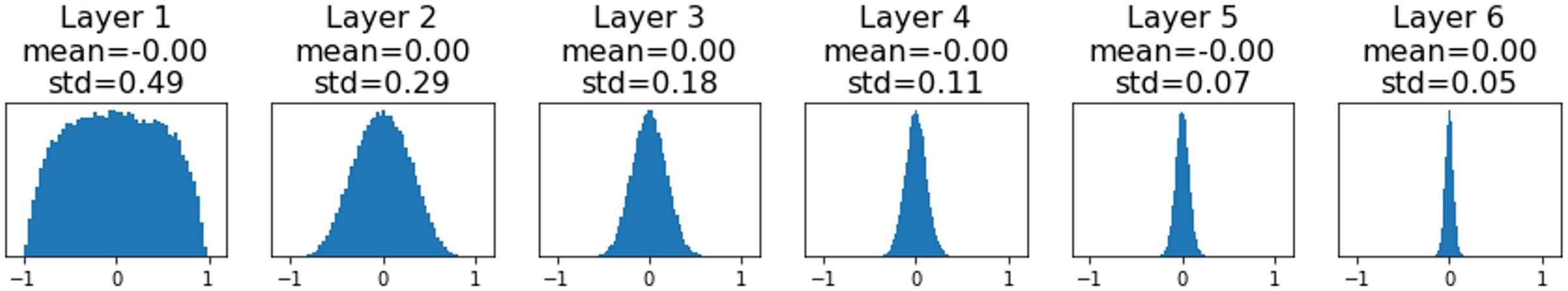
Weight Initialization: Activation Statistics

```
dims = [4096] * 7
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = 0.01 * np.random.randn(Din, Dout)
    x = np.tanh(x.dot(W))
    hs.append(x)
```

Forward pass for a 6-layer net with hidden size 4096

All activations tend to zero for deeper network layers

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



Weight Initialization: Activation Statistics

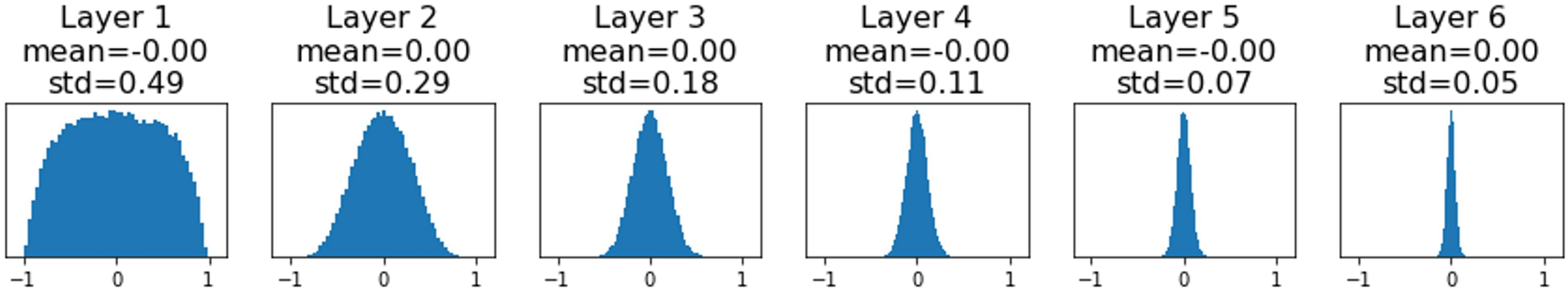
```
dims = [4096] * 7
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = 0.01 * np.random.randn(Din, Dout)
    x = np.tanh(x.dot(W))
    hs.append(x)
```

Forward pass for a 6-layer net with hidden size 4096

All activations tend to zero for deeper network layers

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

A: All zero, no learning =(



Weight Initialization: Activation Statistics

```
dims = [4096] * 7
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = 0.05 * np.random.randn(Din, Dout)
    x = np.tanh(x.dot(W))
    hs.append(x)
```

Increase std of initial weights from 0.01 to 0.05

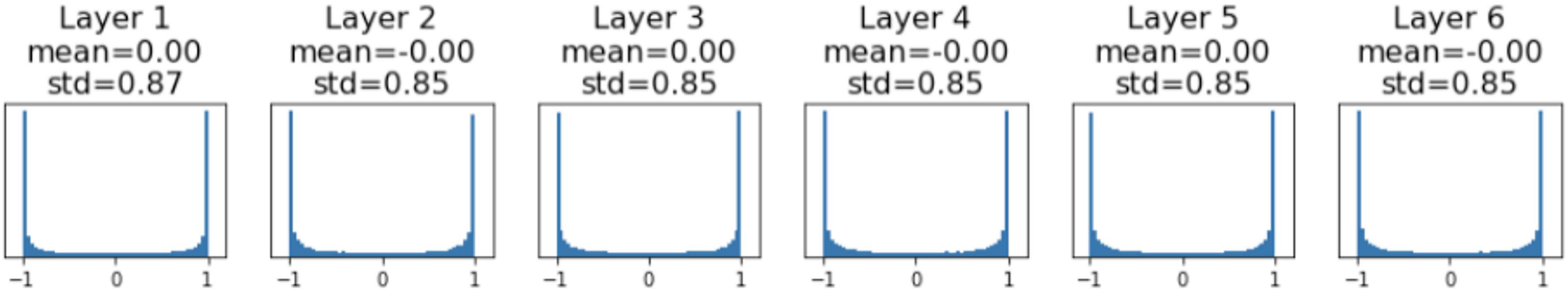
Weight Initialization: Activation Statistics

```
dims = [4096] * 7
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = 0.05 * np.random.randn(Din, Dout)
    x = np.tanh(x.dot(W))
    hs.append(x)
```

Increase std of initial weights from 0.01 to 0.05

All activations saturate

Q: What do the gradients look like?



Weight Initialization: Activation Statistics

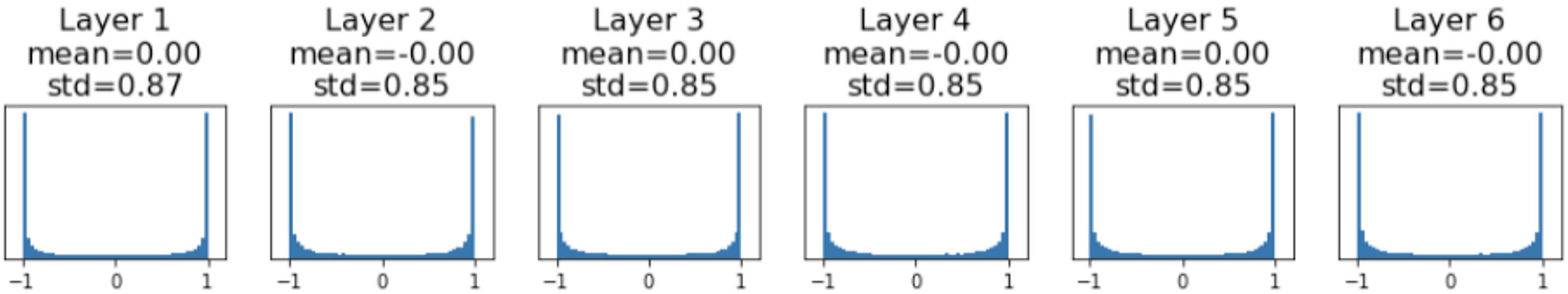
```
dims = [4096] * 7
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = 0.05 * np.random.randn(Din, Dout)
    x = np.tanh(x.dot(W))
    hs.append(x)
```

Increase std of initial weights from 0.01 to 0.05

All activations saturate

Q: What do the gradients look like?

A: Local gradients all zero, no learning =(



Weight Initialization: Xavier Initialization

```
dims = [4096] * 7
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = np.random.randn(Din, Dout) / np.sqrt(Din)
    x = np.tanh(x.dot(W))
    hs.append(x)
```

"Xavier" initialization:
std = 1/sqrt(Din)

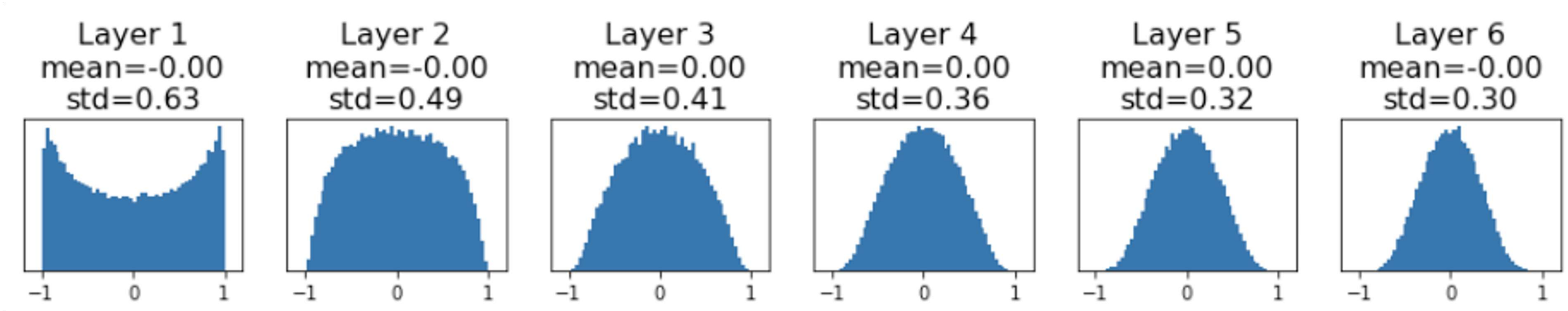
Glorot and Bengio, "Understanding the difficulty of training deep feedforward neural networks", AISTAT 2010

Weight Initialization: Xavier Initialization

```
dims = [4096] * 7
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = np.random.randn(Din, Dout) / np.sqrt(Din)
    x = np.tanh(x.dot(W))
    hs.append(x)
```

"Xavier" initialization:
std = 1/sqrt(Din)

"Just right": Activations are nicely scaled for all layers!



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

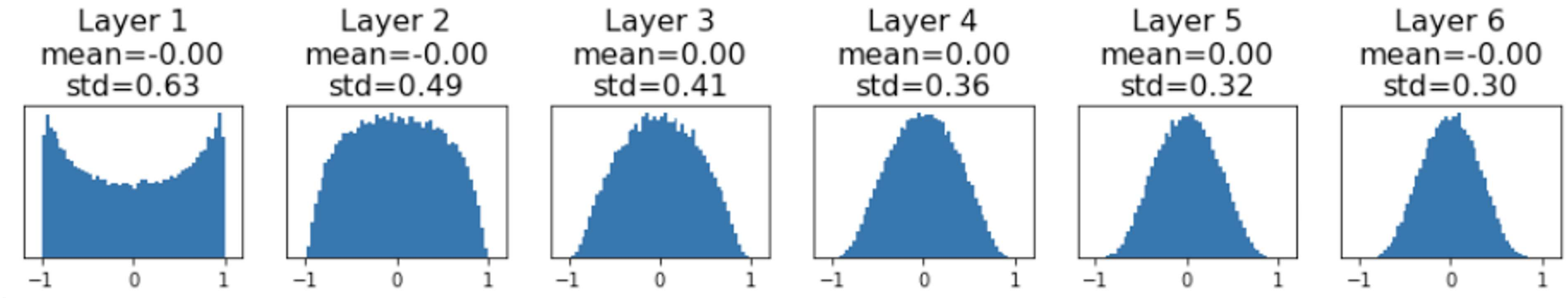
Weight Initialization: Xavier Initialization

```
dims = [4096] * 7
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = np.random.randn(Din, Dout) / np.sqrt(Din)
    x = np.tanh(x.dot(W))
    hs.append(x)
```

"Xavier" initialization:
std = 1/sqrt(Din)

"Just right": Activations are nicely scaled for all layers!

For conv layers, Din is kernel_size^2 * input_channels



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

Weight Initialization: Xavier Initialization

"Xavier" initialization:
std = 1/sqrt(Din)

Derivation: Variance of output = Variance of input

y = Wx

$$y_i = \sum_{j=1}^{Din} x_j w_j$$

Weight Initialization: Xavier Initialization

"Xavier" initialization:
std = 1/sqrt(Din)

Derivation: Variance of output = Variance of input

$y = Wx$

$$y_i = \sum_{j=1}^{Din} x_j w_j$$

$Var(y_i) = Din * Var(x_i w_i)$

[Assume x, w are iid]

Weight Initialization: Xavier Initialization

"Xavier" initialization:
std = 1/sqrt(Din)

Derivation: Variance of output = Variance of input

$y = Wx$

$$y_i = \sum_{j=1}^{Din} x_j w_j$$

$Var(y_i) = Din * Var(x_i w_i)$

[Assume x, w are iid]

$= Din * Var(x_i) * Var(w_i)$

[Assume x, w zero-centered,
assume x^2, w^2 uncorrelated]

Weight Initialization: Xavier Initialization

"Xavier" initialization:
std = 1/sqrt(Din)

Derivation: Variance of output = Variance of input

$y = Wx$

$$y_i = \sum_{j=1}^{Din} x_j w_j$$

$Var(y_i) = Din * Var(x_i w_i)$

[Assume x, w are iid]

$= Din * Var(x_i) * Var(w_i)$

[Assume x, w zero-centered,
assume x^2, w^2 uncorrelated]

If $Var(w_i) = 1/Din$ then $Var(y_i) = Var(x_i)$

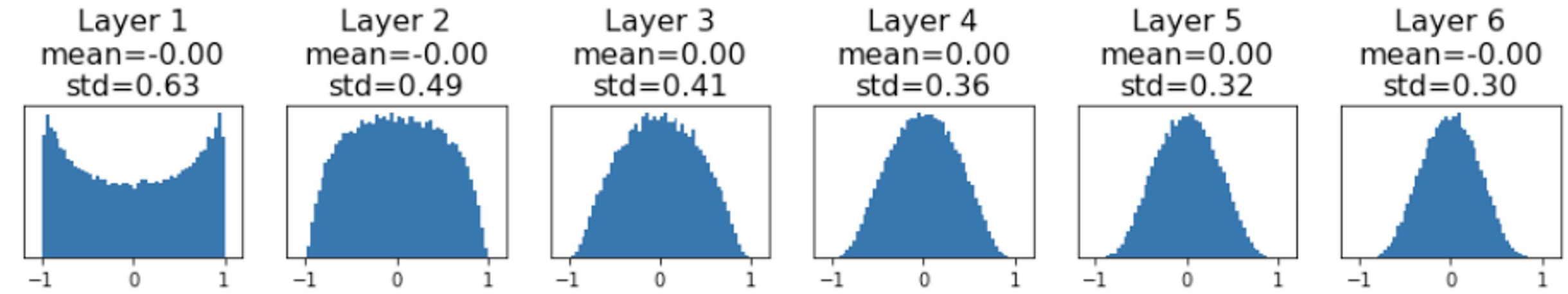
Weight Initialization: Xavier Initialization

```
dims = [4096] * 7
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = np.random.randn(Din, Dout) / np.sqrt(Din)
    x = np.tanh(x.dot(W))
    hs.append(x)
```

"Xavier" initialization:
std = 1/sqrt(Din)

"Just right": Activations are nicely scaled for all layers!

For conv layers, Din is kernel_size^2 * input_channels



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

Weight Initialization: What about ReLU?

```
dims = [4096] * 7
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = np.random.randn(Din, Dout) / np.sqrt(Din)
    x = np.maximum(0, x.dot(W))
    hs.append(x)
```

Change from tanh to ReLU

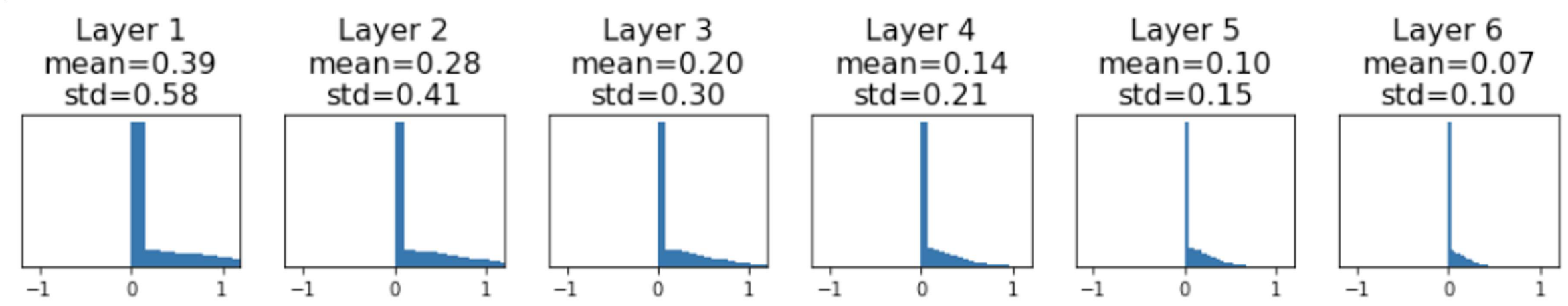
Weight Initialization: What about ReLU?

```
dims = [4096] * 7
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = np.random.randn(Din, Dout) / np.sqrt(Din)
    x = np.maximum(0, x.dot(W))
    hs.append(x)
```

Change from tanh to ReLU

Xavier assumes zero centered activation function

Activations collapse to zero again, no learning =(

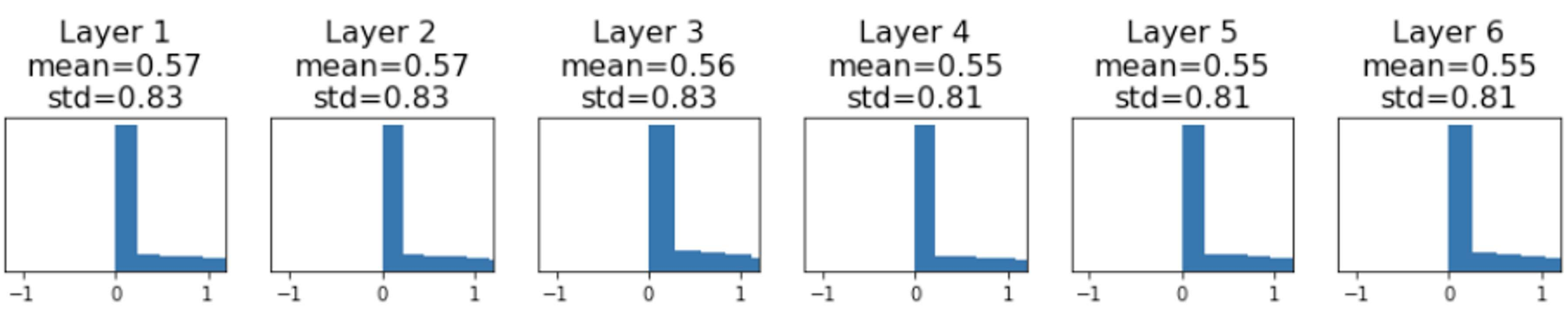


Weight Initialization: Kaiming / MSRA Initialization

```
dims = [4096] * 7
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = np.random.randn(Din, Dout) / np.sqrt(Din)
    x = np.maximum(0, x.dot(W))
    hs.append(x)
```

ReLU correction: std = sqrt(2 / Din)

”Just right” – activations nicely scaled for all layers



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

Convolutional Neural Networks (CNNs)

- A bit of CNN history
- CNN Layers
    - Convolutions, Pooling, Fully Connected Layers, Normalization
- Training CNNs
    - Advanced Optimization
    - Weight initialization
    - Regularization: Dropout
    - Extended training data: Augmentation, Transfer Learning
    - Sanity-checking the learning process; Hyperparameter tuning strategies
- Famous CNN architectures
- CNNs for image segmentation

Convolutional Neural Networks (CNNs)

- A bit of CNN history
- CNN Layers
    - Convolutions, Pooling, Fully Connected Layers, Normalization
- Training CNNs
    - Advanced Optimization
    - Weight initialization
    - Regularization: Dropout
    - Extended training data: Augmentation, Transfer Learning
    - Sanity-checking the learning process; Hyperparameter tuning strategies
- Famous CNN architectures
- CNNs for image segmentation

Advanced Regularization

Dropout, Ensembles

Recap: Additional Term in Loss

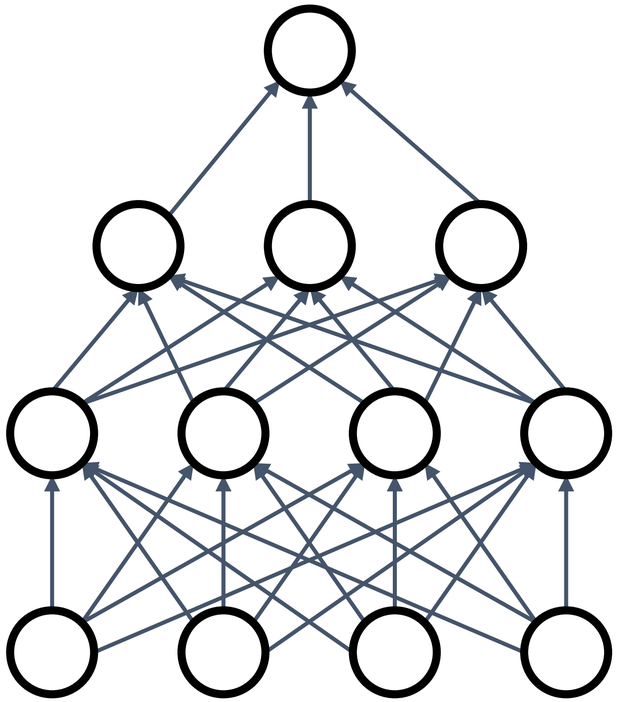
$$L = \frac{1}{N}\sum_{i=0}^{N} cost(h(x_i), y_i, \theta) + \underbrace{\lambda R(\theta)}$$

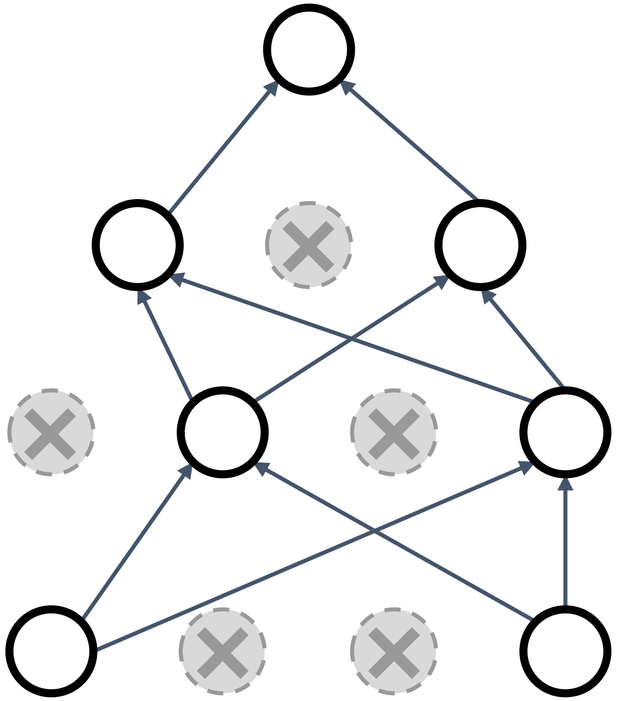
L2 regularization: $R(\theta) = \sum_{i,j} \theta_{i,j}^2$ (weight decay)

L1 regularization: $R(\theta) = \sum_{i,j} |\theta_{i,j}|$

Regularization: Dropout

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



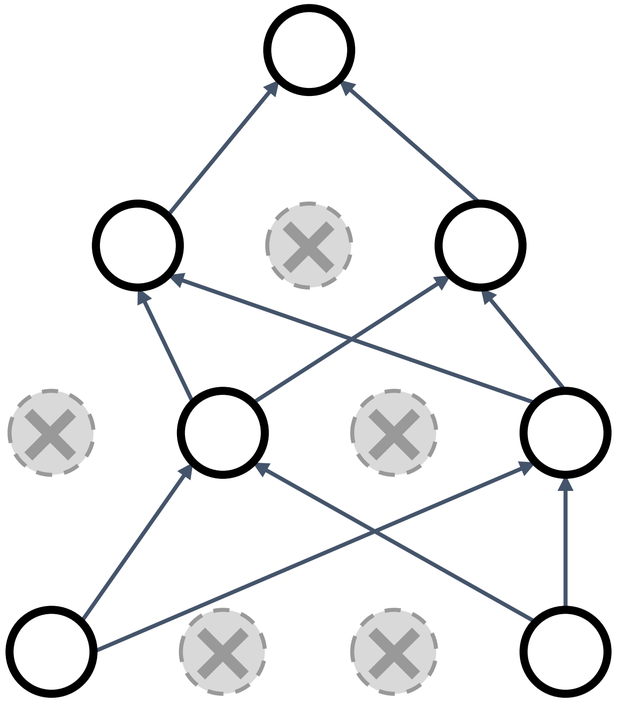


Srivastava et al, “Dropout: A simple way to prevent neural networks from overfitting”, JMLR 2014

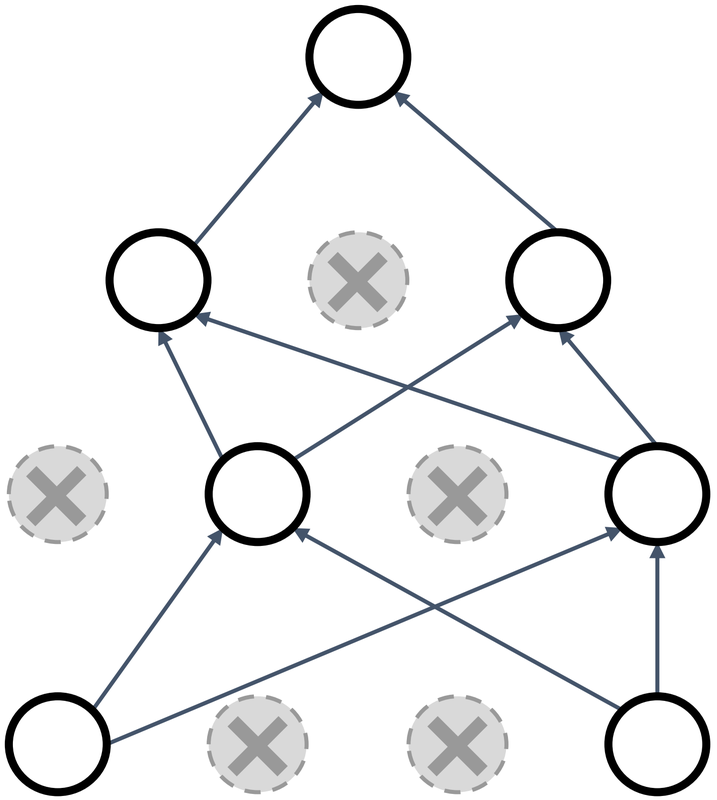
Regularization: Dropout

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

def train_step(X):
  """ X contains the data """

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

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

Example forward pass with a 3-layer network using dropout

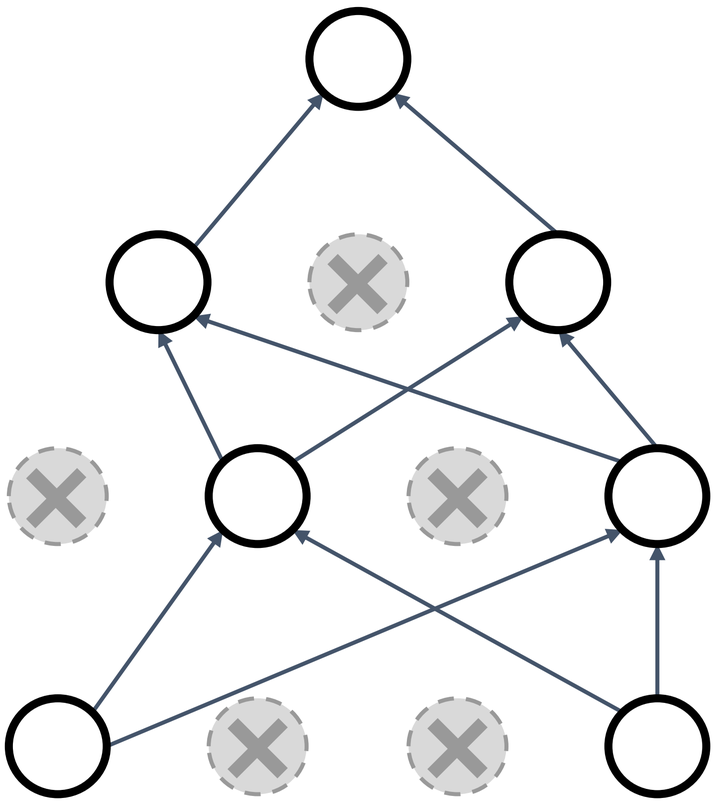


Regularization: Dropout



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

Regularization: Dropout



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)

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

Random mask

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

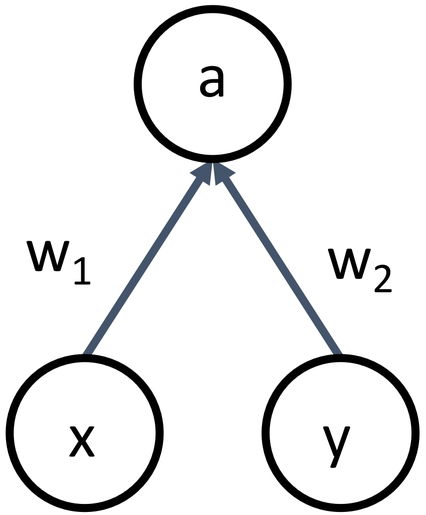
$$y = f(x) = E_z[f(x, z)] = \int p(z) f(x, z) dz$$

But this integral seems hard …

Dropout: Test Time

Want to approximate the integral

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



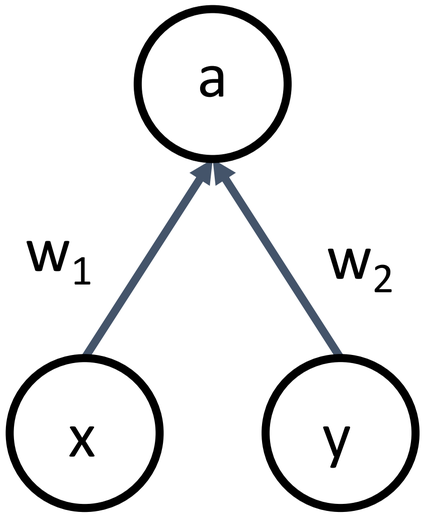
Consider a single neuron:

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

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: $a = w_1 x + w_2 y$

During training we have:

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

At test time, drop nothing and **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”

```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
```

Drop and scale during training

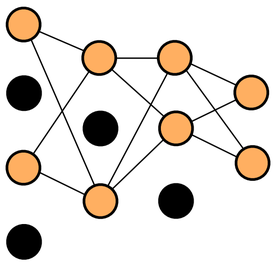
test time is unchanged!

Regularization: A common pattern

Training: add randomness to network

Testing: average over distribution

Dropout

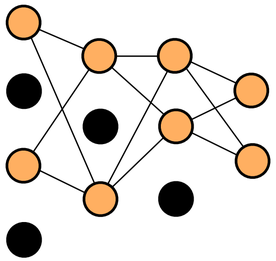


Regularization: A common pattern

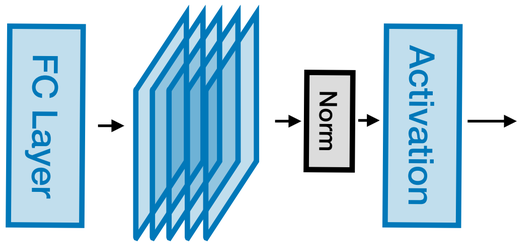
Training: add randomness to network

Testing: average over distribution

Dropout



Batch Normalization

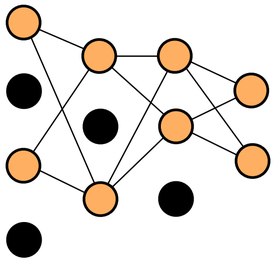


Regularization: A common pattern

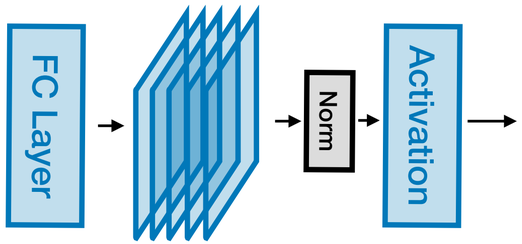
Training: add randomness to network

Testing: average over distribution

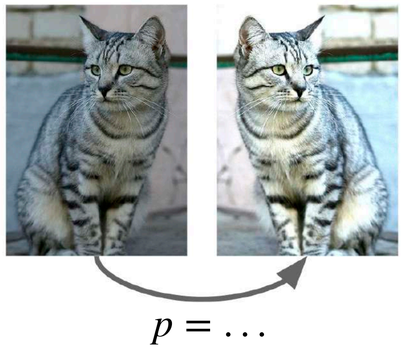
Dropout



Batch Normalization



Data augmentation

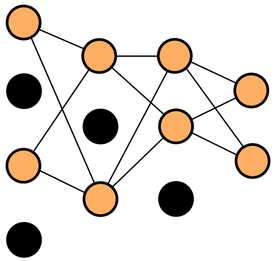


Regularization: A common pattern

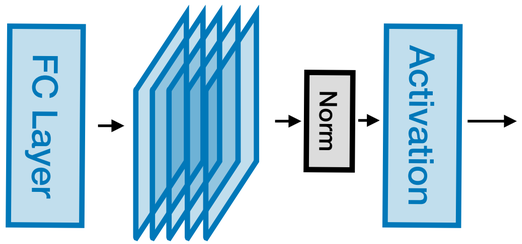
Training: add randomness to network

Testing: average over distribution

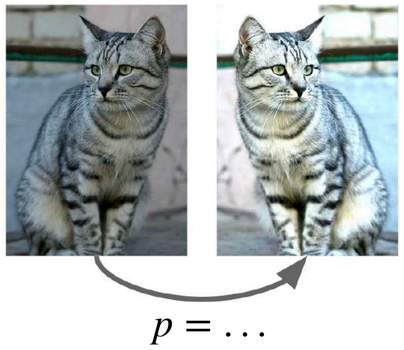
Dropout



Batch Normalization



Data augmentation



And many more:

DropConnect,
Stochastic Depth,
Fractional Max pooling…

Convolutional Neural Networks (CNNs)

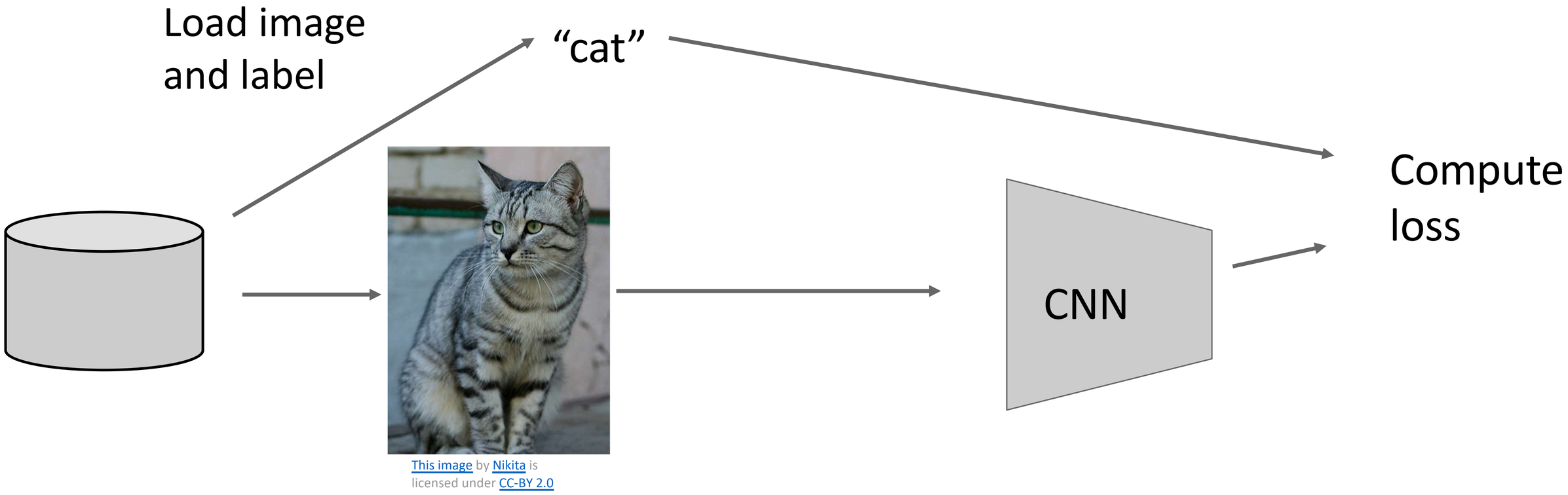
- A bit of CNN history
- CNN Layers
    - Convolutions, Pooling, Fully Connected Layers, Normalization
- Training CNNs
    - Advanced Optimization
    - Weight initialization
    - Regularization: Dropout
    - Extended training data: Augmentation, Transfer Learning
    - Sanity-checking the learning process; Hyperparameter tuning strategies
- Famous CNN architectures
- CNNs for image segmentation

Convolutional Neural Networks (CNNs)

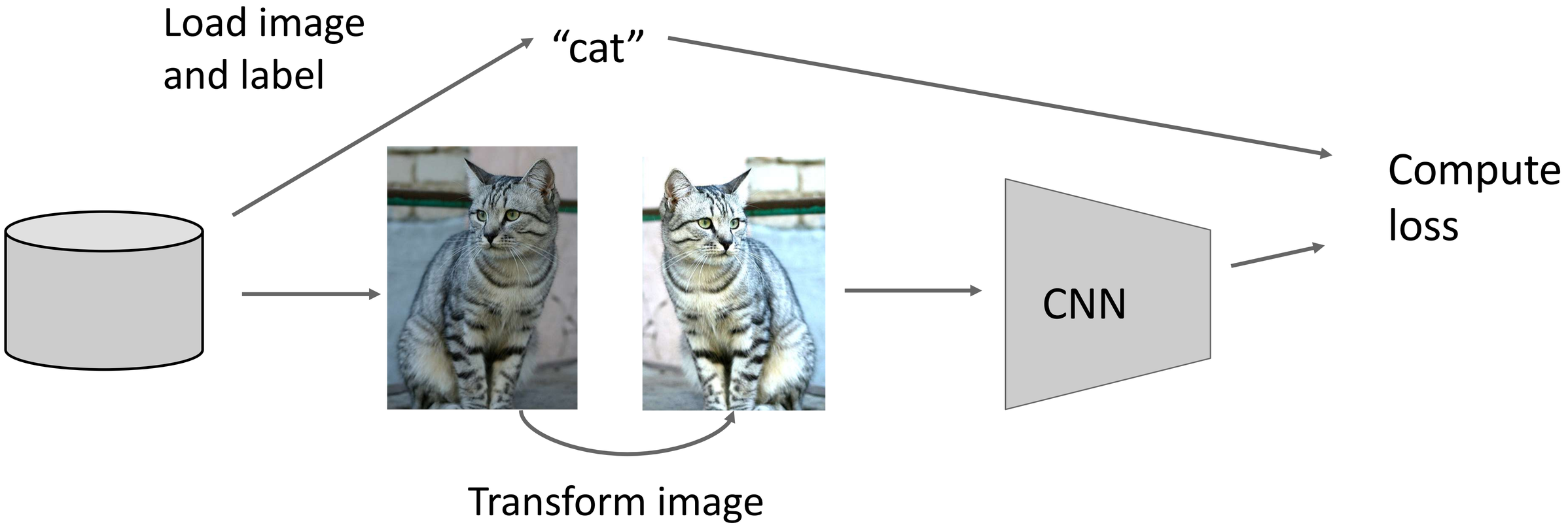
- A bit of CNN history
- CNN Layers
    - Convolutions, Pooling, Fully Connected Layers, Normalization
- Training CNNs
    - Advanced Optimization
    - Weight initialization
    - Regularization: Dropout
    - Extended training data: Augmentation, Transfer Learning
    - Sanity-checking the learning process; Hyperparameter tuning strategies
- Famous CNN architectures
- CNNs for image segmentation

Data Augmentation

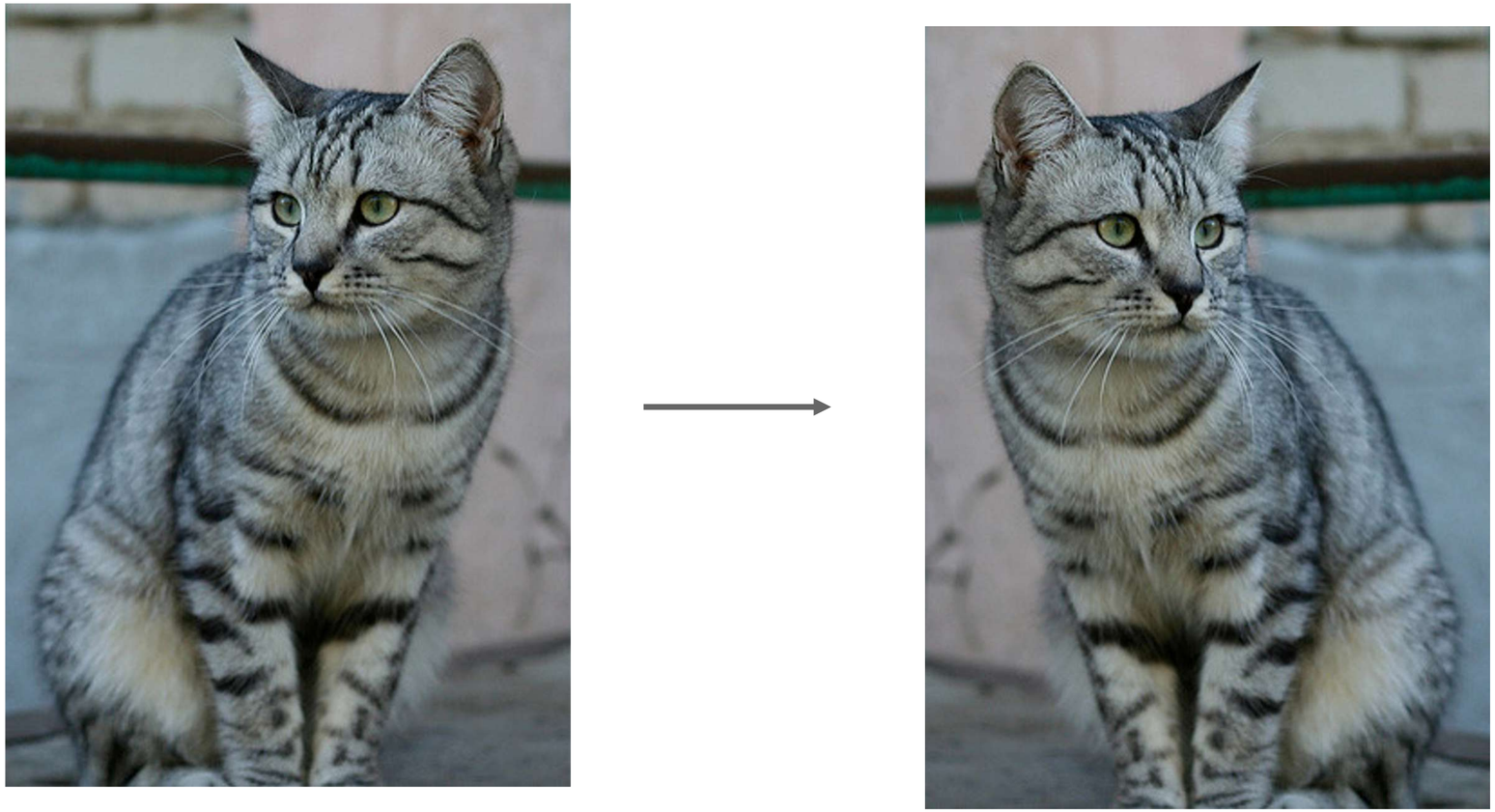
Data Augmentation



Data Augmentation



Data Augmentation: Horizontal Flips



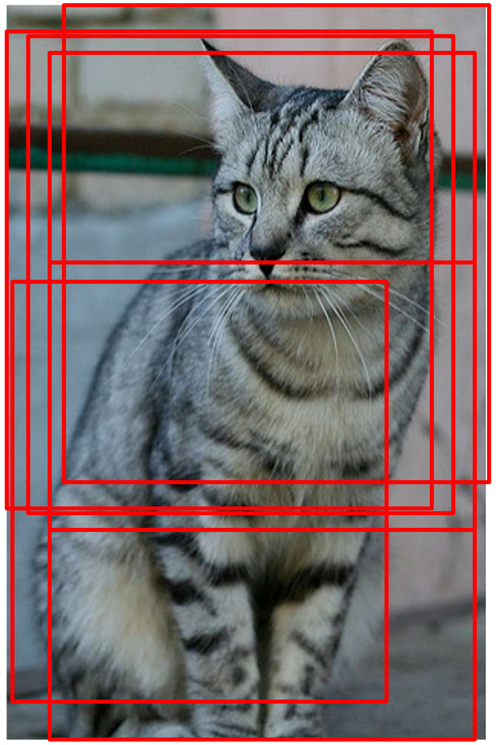
Data Augmentation: Horizontal Flips

Vertical flips?

Data Augmentation: Horizontal Flips

Vertical flips?
Rotations?

Data Augmentation: Random Crops and Scales



Data Augmentation: Color Jitter

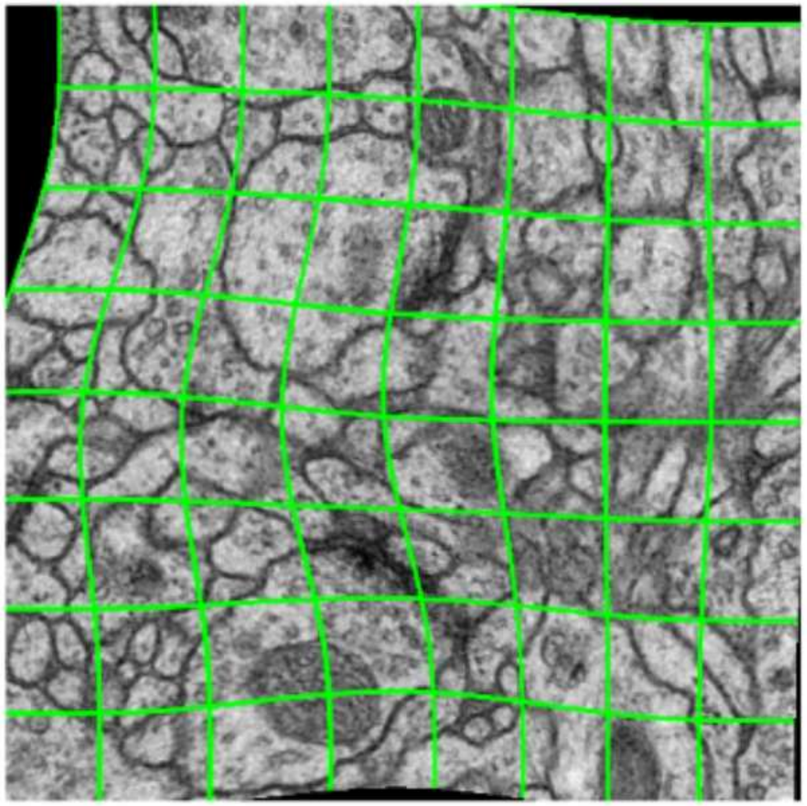
Simple: Randomize contrast and brightness

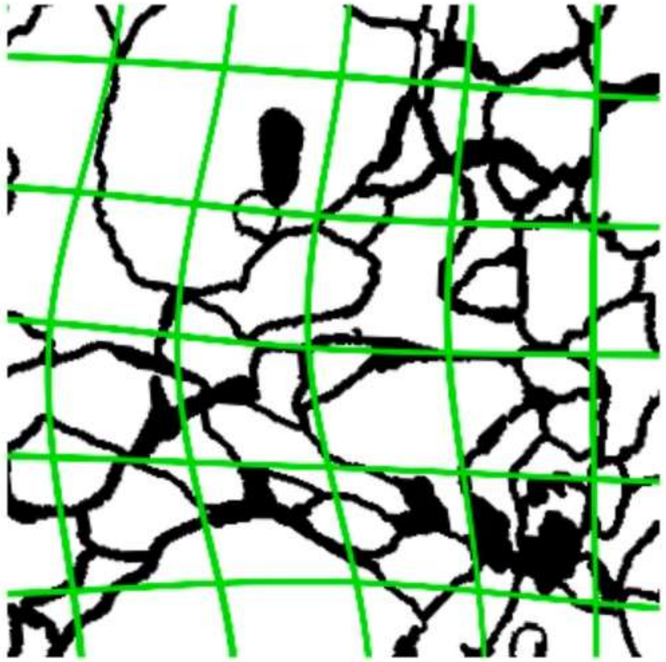
More Complex:

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

(Used in AlexNet, ResNet, etc)

Data Augmentation: Elastic Deformation





1. Specify:
   - Grid size
   - Distribution of offsets
2. Sample offsets for grid points
3. Interpolate in-between

Image from Ronneberger et al., MICCAI 2015

Data Augmentation: Test-time Augmentation

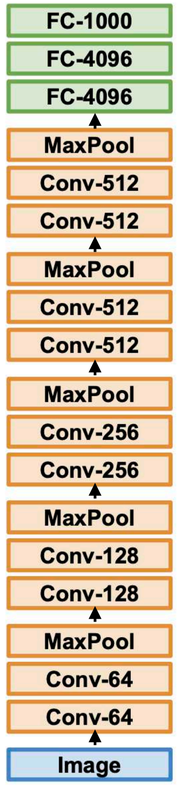
1. Create a set of random augmented versions of your test image, e.g. n=8
2. Average the predictions

=> A different kind of ensemble / randomness
=> often yields improved predictions
=> at the expense of increased computational cost

Transfer Learning

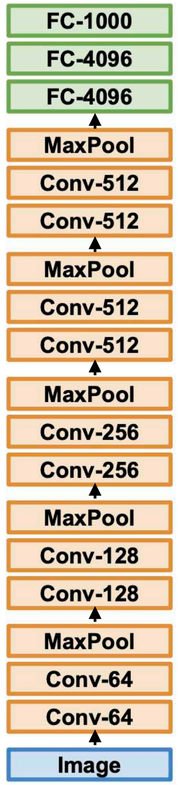
Transfer Learning with CNNs

#1 Train on Imagenet

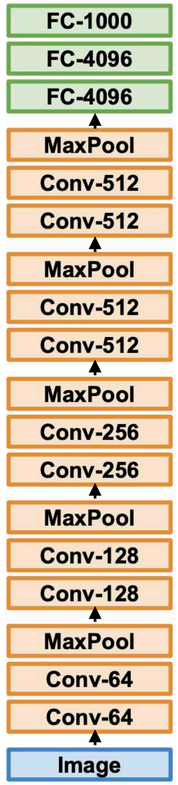


Transfer Learning with CNNs

#1 Train on Imagenet

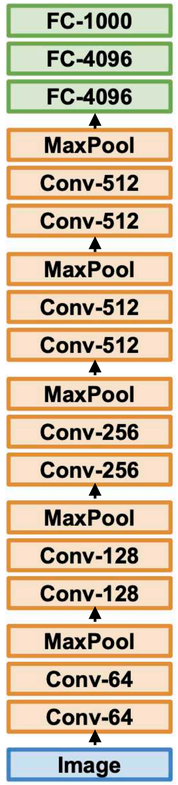


#2 Small Dataset (C classes)

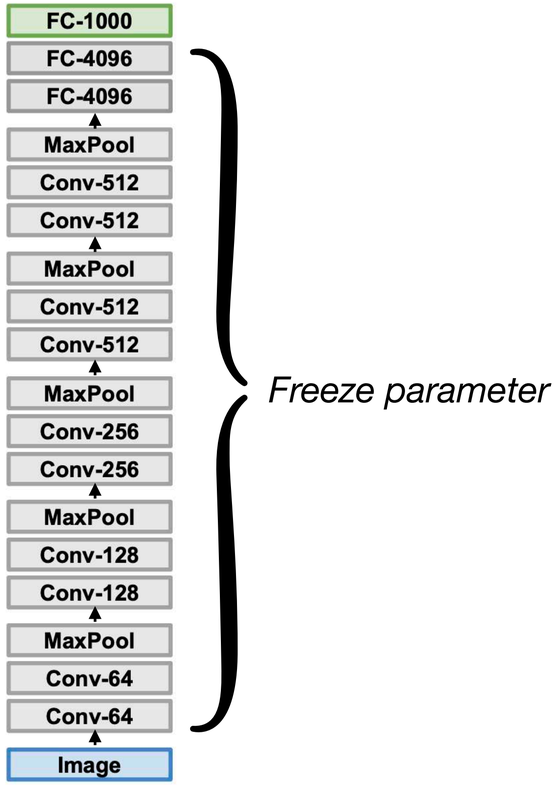


Transfer Learning with CNNs

#1 Train on Imagenet

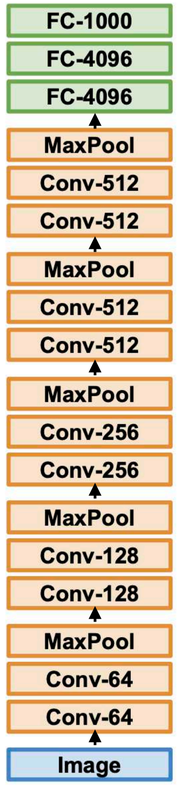


#2 Small Dataset (C classes)

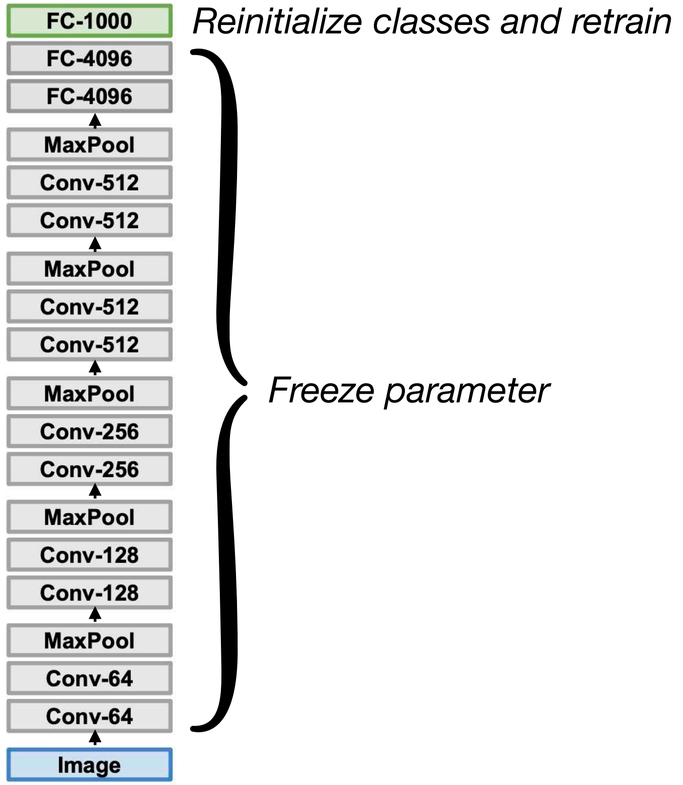


Transfer Learning with CNNs

#1 Train on Imagenet

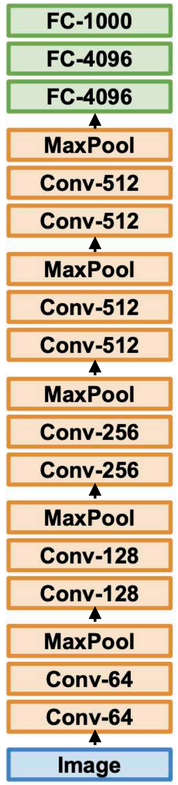


#2 Small Dataset (C classes)

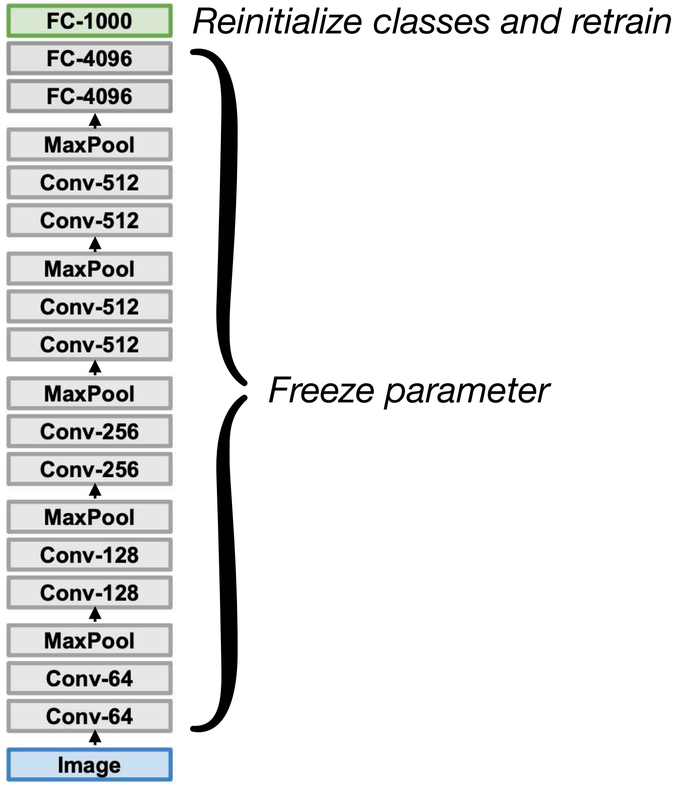


Transfer Learning with CNNs

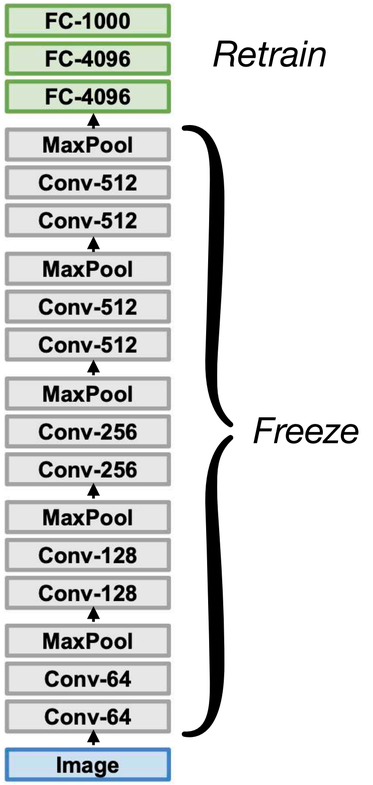
#1 Train on Imagenet



#2 Small Dataset (C classes)



#3 Bigger Dataset



Credits to FeiFei Li & Justin Johnson & Serena Yeung (Stanford CNNs for Visual Recognition, 2017, cs231)

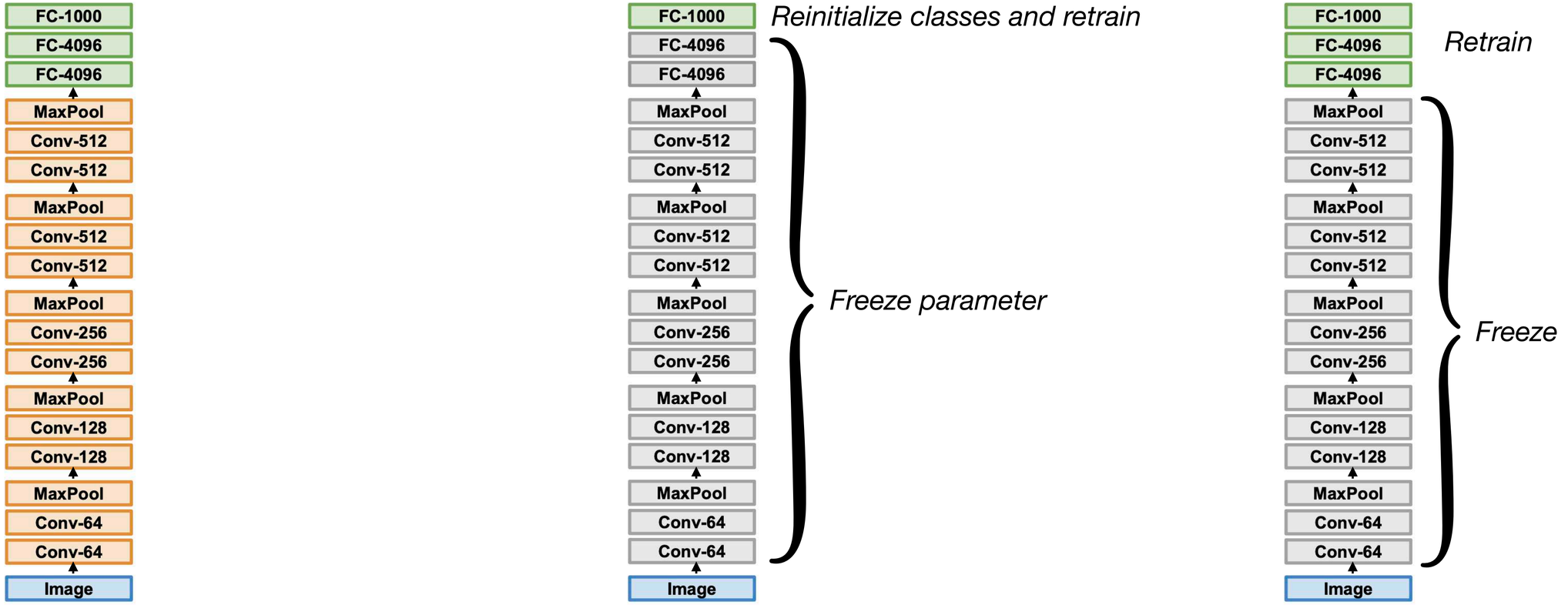
Transfer Learning with CNNs

Try smaller learning rate, e.g. 0.1 of original learning rate

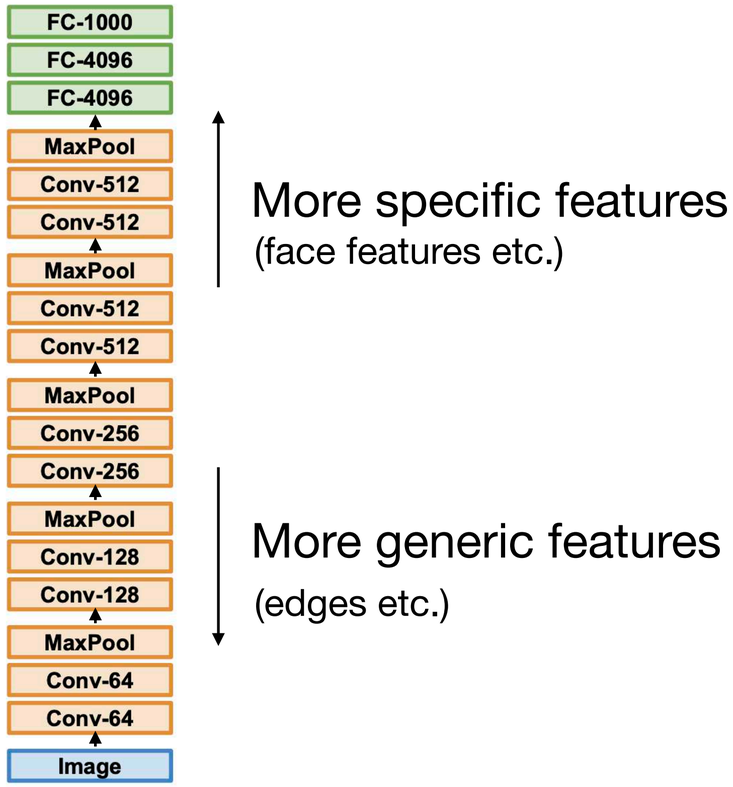
#1 Train on Imagenet

#2 Small Dataset (C classes)

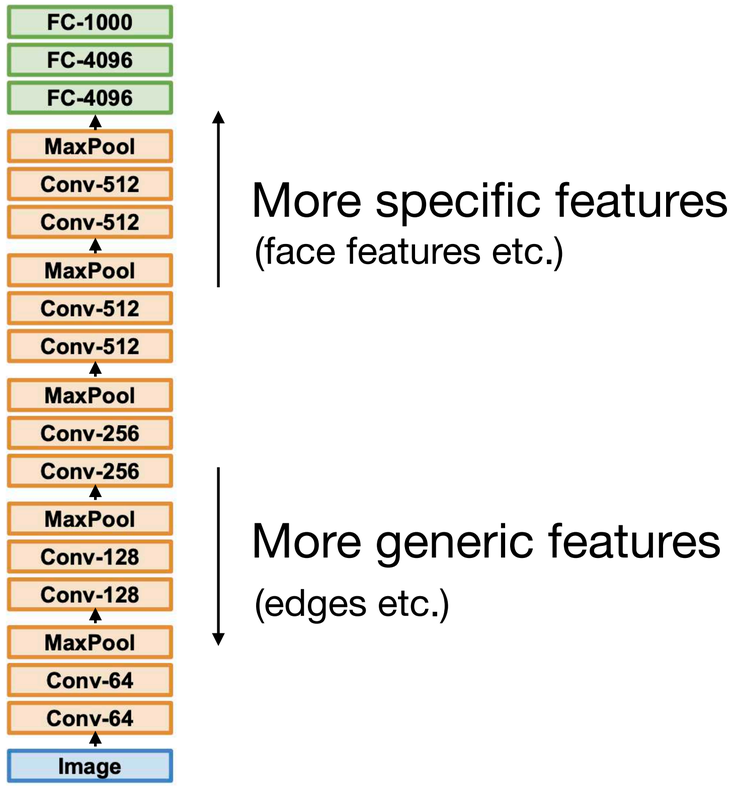
#3 Bigger Dataset



Transfer Learning with CNNs



Transfer Learning with CNNs

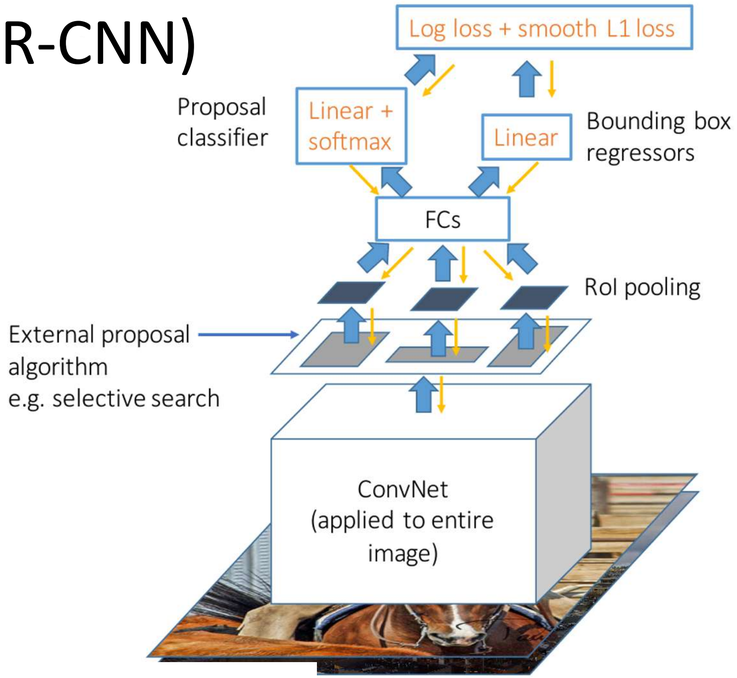


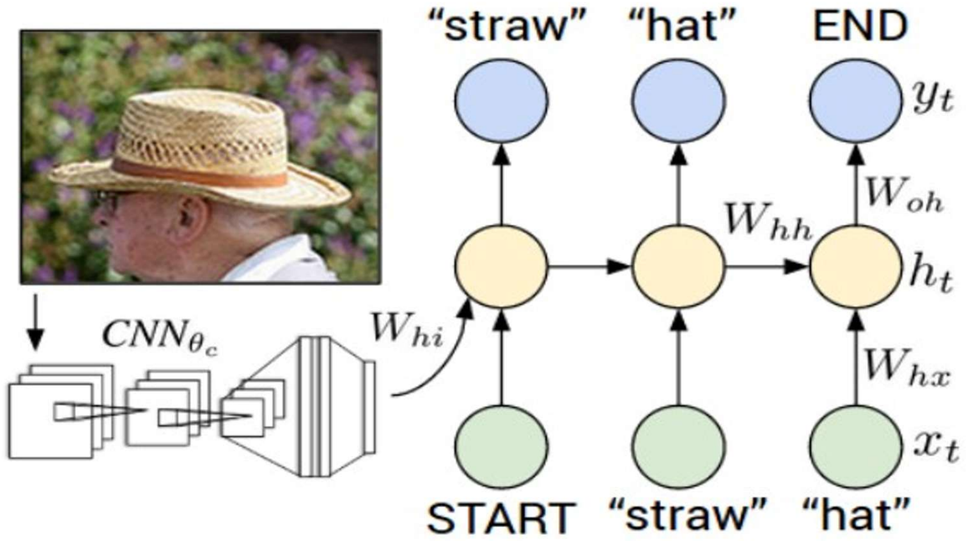
	very similar dataset	**very different dataset**
very little data	Use Linear Classifier on top layer	You're in trouble… Try linear classifier from different stages
quite a lot of data	Finetune a few layers	Finetune a larger number of layers

Credits to FeiFei Li & Justin Johnson & Serena Yeung (Stanford CNNs for Visual Recognition, 2017, cs231)

Transfer learning is pervasive!
It's the norm, not the exception

Object
Detection
(Fast R-CNN)





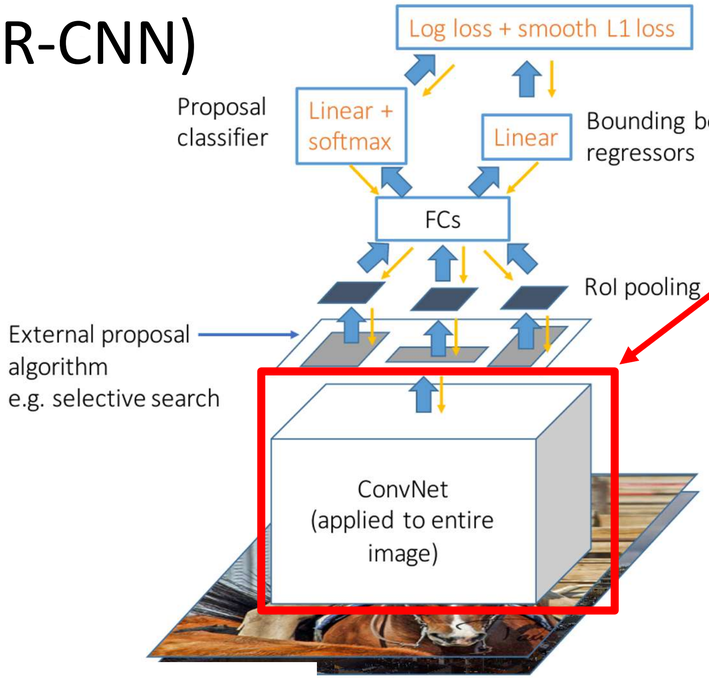
Girshick, "Fast R-CNN", ICCV 2015
Figure copyright Ross Girshick, 2015. Reproduced with permission.

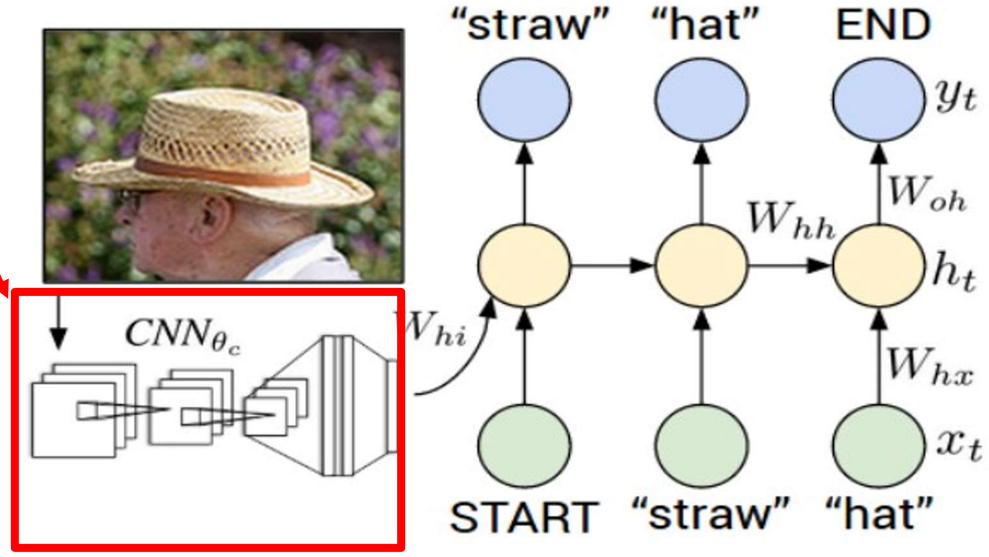
Karpathy and Fei-Fei, "Deep Visual-Semantic Alignments for Generating Image Descriptions", CVPR 2015

Transfer learning is pervasive!
It's the norm, not the exception

Object Detection (Fast R-CNN)

CNN pretrained on ImageNet



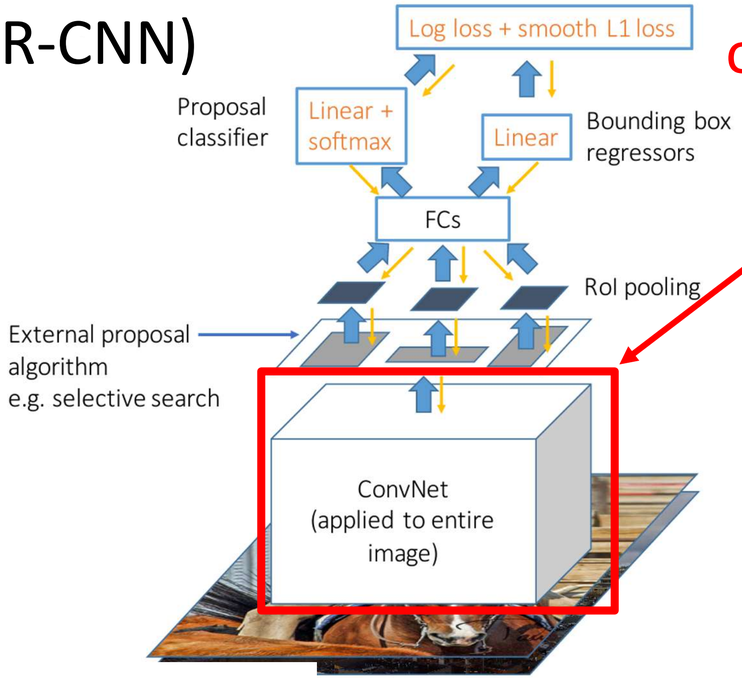


Girshick, "Fast R-CNN", ICCV 2015
Figure copyright Ross Girshick, 2015. Reproduced with permission.

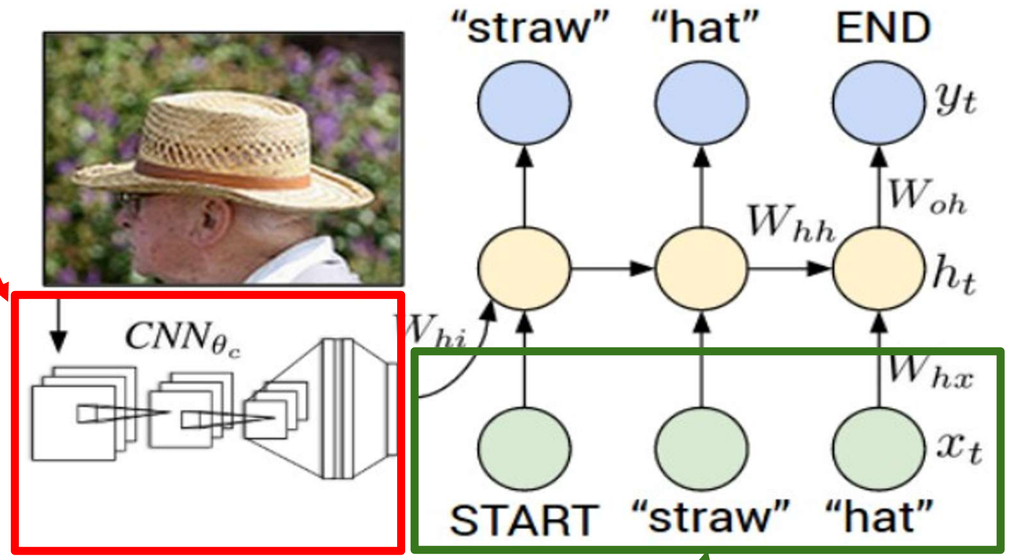
Karpathy and Fei-Fei, "Deep Visual-Semantic Alignments for Generating Image Descriptions", CVPR 2015

Transfer learning is pervasive!
It's the norm, not the exception

Object Detection (Fast R-CNN)



CNN pretrained on ImageNet



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

Convolutional Neural Networks (CNNs)

- A bit of CNN history
- CNN Layers
    - Convolutions, Pooling, Fully Connected Layers, Normalization
- Training CNNs
    - Advanced Optimization
    - Weight initialization
    - Regularization: Dropout
    - Extended training data: Augmentation, Transfer Learning
    - Sanity-checking the learning process; Hyperparameter tuning strategies
- Famous CNN architectures
- CNNs for image segmentation

Convolutional Neural Networks (CNNs)

- A bit of CNN history
- CNN Layers
    - Convolutions, Pooling, Fully Connected Layers, Normalization
- Training CNNs
    - Advanced Optimization
    - Weight initialization
    - Regularization: Dropout
    - Extended training data: Augmentation, Transfer Learning
    - Sanity-checking the learning process; Hyperparameter tuning strategies
- Famous CNN architectures
- CNNs for image segmentation

Babysitting the learning process

Double check that the loss is reasonable:

```
def init_two_layer_model(input_size, hidden_size, output_size):
  # initialize a model
  model = {}
  model['W1'] = 0.0001 * np.random.randn(input_size, hidden_size)
  model['b1'] = np.zeros(hidden_size)
  model['W2'] = 0.0001 * np.random.randn(hidden_size, output_size)
  model['b2'] = np.zeros(output_size)
  return model
```

```
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
loss, grad = two_layer_net(X_train, model, y_train, 0.0)
print loss
```

disable regularization

What output would you expect?

returns the loss and the gradient for all parameters

Double check that the loss is reasonable:

```
def init_two_layer_model(input_size, hidden_size, output_size):
  # initialize a model
  model = {}
  model['W1'] = 0.0001 * np.random.randn(input_size, hidden_size)
  model['b1'] = np.zeros(hidden_size)
  model['W2'] = 0.0001 * np.random.randn(hidden_size, output_size)
  model['b2'] = np.zeros(output_size)
  return model
```

```
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
loss, grad = two_layer_net(X_train, model, y_train, 0.0)
print loss
```

disable regularization

```
2.30261216167
```

loss ~2.3.
"correct " for
10 classes

returns the loss and the
gradient for all parameters

Double check that the loss is reasonable:

```
def init_two_layer_model(input_size, hidden_size, output_size):
  # initialize a model
  model = {}
  model['W1'] = 0.0001 * np.random.randn(input_size, hidden_size)
  model['b1'] = np.zeros(hidden_size)
  model['W2'] = 0.0001 * np.random.randn(hidden_size, output_size)
  model['b2'] = np.zeros(output_size)
  return model
```

```
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
loss, grad = two_layer_net(X_train, model, y_train, 1e3)
print loss
```

crank up regularization

```
3.06859716482
```

loss went up, good. (sanity check)

Let’s try to train now…

1st check: Make sure that you can overfit very small portion of the training data

```
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
trainer = ClassifierTrainer()
X_tiny = X_train[:20] # take 20 examples
y_tiny = y_train[:20]
best_model, stats = trainer.train(X_tiny, y_tiny, X_tiny, y_tiny,
                                  model, two_layer_net,
                                  num_epochs=200, reg=0.0,
                                  update='sgd', learning_rate_decay=1,
                                  sample_batches = False,
                                  learning_rate=1e-3, verbose=True)
```

The above code:

- take the first 20 examples from CIFAR-10
- turn off regularization (reg = 0.0)
- use standard “sgd”

Let’s try to train now…

1st check: Make sure that you can overfit very small portion of the training data

Very small loss, train accuracy 1.00, nice!

```
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
trainer = ClassifierTrainer()
X_tiny = X_train[:20] # take 20 examples
y_tiny = y_train[:20]
best_model, stats = trainer.train(X_tiny, y_tiny, X_tiny, y_tiny,
                                  model, two_layer_net,
                                  num_epochs=200, reg=0.0,
                                  update='sgd', learning_rate_decay=1,
                                  sample_batches = False,
                                  learning_rate=1e-3, verbose=True)
```

```
Finished epoch 1 / 200: cost 2.302603, train: 0.400000, val 0.400000, lr 1.000000e-03
Finished epoch 2 / 200: cost 2.302258, train: 0.450000, val 0.450000, lr 1.000000e-03
Finished epoch 3 / 200: cost 2.301849, train: 0.600000, val 0.600000, lr 1.000000e-03
Finished epoch 4 / 200: cost 2.301196, train: 0.650000, val 0.650000, lr 1.000000e-03
Finished epoch 5 / 200: cost 2.300044, train: 0.650000, val 0.650000, lr 1.000000e-03
Finished epoch 6 / 200: cost 2.297864, train: 0.550000, val 0.550000, lr 1.000000e-03
Finished epoch 7 / 200: cost 2.293595, train: 0.600000, val 0.600000, lr 1.000000e-03
Finished epoch 8 / 200: cost 2.285096, train: 0.550000, val 0.550000, lr 1.000000e-03
Finished epoch 9 / 200: cost 2.268094, train: 0.550000, val 0.550000, lr 1.000000e-03
Finished epoch 10 / 200: cost 2.234787, train: 0.500000, val 0.500000, lr 1.000000e-03
Finished epoch 11 / 200: cost 2.173187, train: 0.500000, val 0.500000, lr 1.000000e-03
Finished epoch 12 / 200: cost 2.076862, train: 0.500000, val 0.500000, lr 1.000000e-03
Finished epoch 13 / 200: cost 1.974090, train: 0.400000, val 0.400000, lr 1.000000e-03
Finished epoch 14 / 200: cost 1.895885, train: 0.400000, val 0.400000, lr 1.000000e-03
Finished epoch 15 / 200: cost 1.820876, train: 0.450000, val 0.450000, lr 1.000000e-03
Finished epoch 16 / 200: cost 1.737430, train: 0.450000, val 0.450000, lr 1.000000e-03
Finished epoch 17 / 200: cost 1.642356, train: 0.500000, val 0.500000, lr 1.000000e-03
Finished epoch 18 / 200: cost 1.535239, train: 0.600000, val 0.600000, lr 1.000000e-03
Finished epoch 19 / 200: cost 1.421527, train: 0.600000, val 0.600000, lr 1.000000e-03
...
Finished epoch 195 / 200: cost 0.002694, train: 1.000000, val 1.000000, lr 1.000000e-03
Finished epoch 196 / 200: cost 0.002674, train: 1.000000, val 1.000000, lr 1.000000e-03
Finished epoch 197 / 200: cost 0.002655, train: 1.000000, val 1.000000, lr 1.000000e-03
Finished epoch 198 / 200: cost 0.002635, train: 1.000000, val 1.000000, lr 1.000000e-03
Finished epoch 199 / 200: cost 0.002617, train: 1.000000, val 1.000000, lr 1.000000e-03
Finished epoch 200 / 200: cost 0.002597, train: 1.000000, val 1.000000, lr 1.000000e-03
finished optimization. best validation accuracy: 1.000000
```

Let’s try to train now…

Start with small regularization and find learning rate that makes the loss go down.

```
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
trainer = ClassifierTrainer()
best_model, stats = trainer.train(X_train, y_train, X_val, y_val,
                                  model, two_layer_net,
                                  num_epochs=10, reg=0.000001,
                                  update='sgd', learning_rate_decay=1,
                                  sample_batches = True,
                                  learning_rate=1e-6, verbose=True)
```

Let’s try to train now…

Start with small regularization and find learning rate that makes the loss go down.

loss not going down:
learning rate too low

```
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
trainer = ClassifierTrainer()
best_model, stats = trainer.train(X_train, y_train, X_val, y_val,
                                  model, two_layer_net,
                                  num_epochs=10, reg=0.000001,
                                  update='sgd', learning_rate_decay=1,
                                  sample_batches = True,
                                  learning_rate=1e-6, verbose=True)
```

```
Finished epoch 1 / 10: cost 2.302576, train: 0.080000, val 0.103000, lr 1.000000e-06
Finished epoch 2 / 10: cost 2.302582, train: 0.121000, val 0.124000, lr 1.000000e-06
Finished epoch 3 / 10: cost 2.302558, train: 0.119000, val 0.138000, lr 1.000000e-06
Finished epoch 4 / 10: cost 2.302519, train: 0.127000, val 0.151000, lr 1.000000e-06
Finished epoch 5 / 10: cost 2.302517, train: 0.158000, val 0.171000, lr 1.000000e-06
Finished epoch 6 / 10: cost 2.302518, train: 0.179000, val 0.172000, lr 1.000000e-06
Finished epoch 7 / 10: cost 2.302466, train: 0.180000, val 0.176000, lr 1.000000e-06
Finished epoch 8 / 10: cost 2.302452, train: 0.175000, val 0.185000, lr 1.000000e-06
Finished epoch 9 / 10: cost 2.302459, train: 0.206000, val 0.192000, lr 1.000000e-06
Finished epoch 10 / 10: cost 2.302420, train: 0.190000, val 0.192000, lr 1.000000e-06
finished optimization. best validation accuracy: 0.192000
```

Loss barely changing:
Learning rate is probably too low

Notice train/val accuracy goes to 20% though, what’s up with that? (remember this is softmax)

Let’s try to train now…

Start with small regularization and find learning rate that makes the loss go down.

loss not going down:
learning rate too low

```
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
trainer = ClassifierTrainer()
best_model, stats = trainer.train(X_train, y_train, X_val, y_val,
                                  model, two_layer_net,
                                  num_epochs=10, reg=0.000001,
                                  update='sgd', learning_rate_decay=1,
                                  sample_batches = True,
```

Now let’s try learning rate 1e6.

Let’s try to train now…

Start with small regularization and find learning rate that makes the loss go down.

loss not going down:
learning rate too low

loss exploding:
learning rate too high

```
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
trainer = ClassifierTrainer()
best_model, stats = trainer.train(X_train, y_train, X_val, y_val,
                                  model, two_layer_net,
                                  num_epochs=10, reg=0.000001,
                                  update='sgd', learning_rate_decay=1,
                                  sample_batches = True,
                                  learning_rate=1e6, verbose=True)
```

```
/home/karpathy/cs231n/code/cs231n/classifiers/neural_net.py:50: RuntimeWarning: divide by zero en
countered in log
  data_loss = -np.sum(np.log(probs[range(N), y])) / N
/home/karpathy/cs231n/code/cs231n/classifiers/neural_net.py:48: RuntimeWarning: invalid value enc
ountered in subtract
  probs = np.exp(scores - np.max(scores, axis=1, keepdims=True))
Finished epoch 1 / 10: cost nan, train: 0.091000, val 0.087000, lr 1.000000e+06
Finished epoch 2 / 10: cost nan, train: 0.095000, val 0.087000, lr 1.000000e+06
Finished epoch 3 / 10: cost nan, train: 0.100000, val 0.087000, lr 1.000000e+06
```

loss: NaN often means learning rate too high…

Let’s try to train now…

Start with small regularization and find learning rate that makes the loss go down.

loss not going down: learning rate too low

loss exploding: learning rate too high

```
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
trainer = ClassifierTrainer()
best_model, stats = trainer.train(X_train, y_train, X_val, y_val,
                                  model, two_layer_net,
                                  num_epochs=10, reg=0.000001,
                                  update='sgd', learning_rate_decay=1,
                                  sample_batches = True,
                                  learning_rate=3e-3, verbose=True)
```

```
Finished epoch 1 / 10: cost 2.186654, train: 0.308000, val 0.306000, lr 3.000000e-03
Finished epoch 2 / 10: cost 2.176230, train: 0.330000, val 0.350000, lr 3.000000e-03
Finished epoch 3 / 10: cost 1.942257, train: 0.376000, val 0.352000, lr 3.000000e-03
Finished epoch 4 / 10: cost 1.827868, train: 0.329000, val 0.310000, lr 3.000000e-03
Finished epoch 5 / 10: cost inf, train: 0.128000, val 0.128000, lr 3.000000e-03
Finished epoch 6 / 10: cost inf, train: 0.144000, val 0.147000, lr 3.000000e-03
```

3e-3 is still too high. Cost explodes…

➡ Rough range for learning rate we should be validating is somewhere [1e-3 … 1e-5]

Hyperparameter Optimization

Random Search vs. Grid Search

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

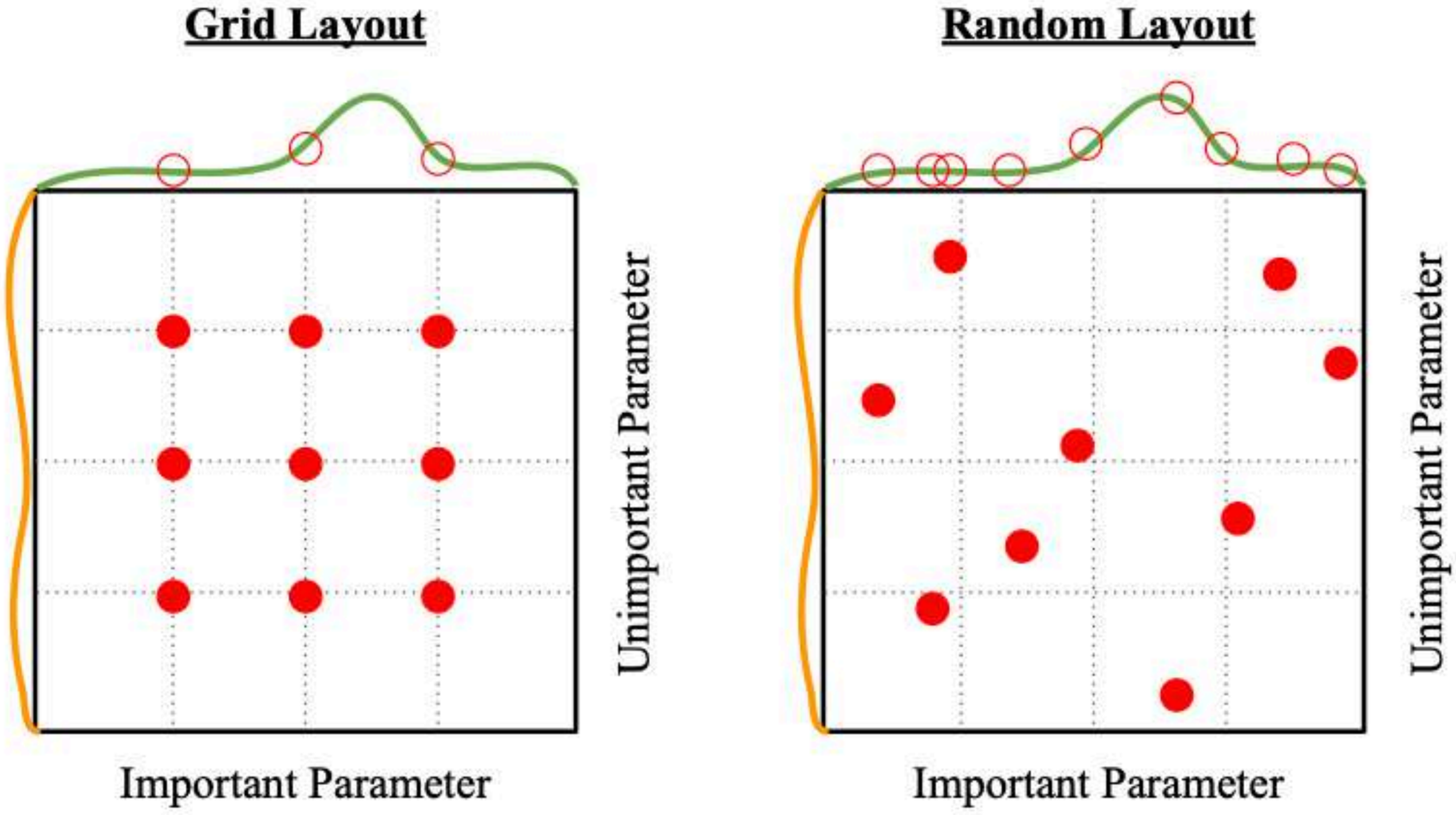


Illustration of Bergstra et al., 2012 by Shayne Longpre, copyright CS231n 2017

Validation strategy

coarse → fine validation in stages

First stage: only a few epochs to get rough idea of what params works

Second stage: longer running time, finer search

… (repeat as necessary)

For example: run coarse search for 5 epochs

```
max_count = 100
for count in xrange(max_count):
      reg = 10**uniform(-5, 5)
      lr = 10**uniform(-3, -6)

      trainer = ClassifierTrainer()
      model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
      trainer = ClassifierTrainer()
      best_model_local, stats = trainer.train(X_train, y_train, X_val, y_val,
                                              model, two_layer_net,
                                              num_epochs=5, reg=reg,
                                              update='momentum', learning_rate_decay=0.9,
                                              sample_batches = True, batch_size = 100,
                                              learning_rate=lr, verbose=False)
```

note it's best to optimize in log space!

```
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)
```

nice

Now run finer search …

```
max_count = 100
for count in xrange(max_count):
      reg = 10**uniform(-5, 5)
      lr = 10**uniform(-3, -6)
```

adjust range →

```
max_count = 100
for count in xrange(max_count):
      reg = 10**uniform(-4, 0)
      lr = 10**uniform(-3, -4)
```

```
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)
```

53% - relatively good for a 2-layer neural net with 50 hidden neurons.

← But this best cross-validation result is worrying. Why?

Convolutional Neural Networks (CNNs)

- A bit of CNN history
- CNN Layers
    - Convolutions, Pooling, Fully Connected Layers, Normalization
- Training CNNs
    - Advanced Optimization
    - Weight initialization
    - Regularization: Dropout
    - Extended training data: Augmentation, Transfer Learning
    - Sanity-checking the learning process; Hyperparameter tuning strategies
- Famous CNN architectures
- CNNs for image segmentation

Course Outline

- Introduction to Image Analysis
- Basics: Neural Networks
- **Convolutional Neural Networks**
- Transformers
- Model Interpretability
- Self-supervised Learning
- Generative Models (GANs, Diffusion)