Lecture 7: CNNs II – Model Training and Optimization

Shenghua Gao

Outline

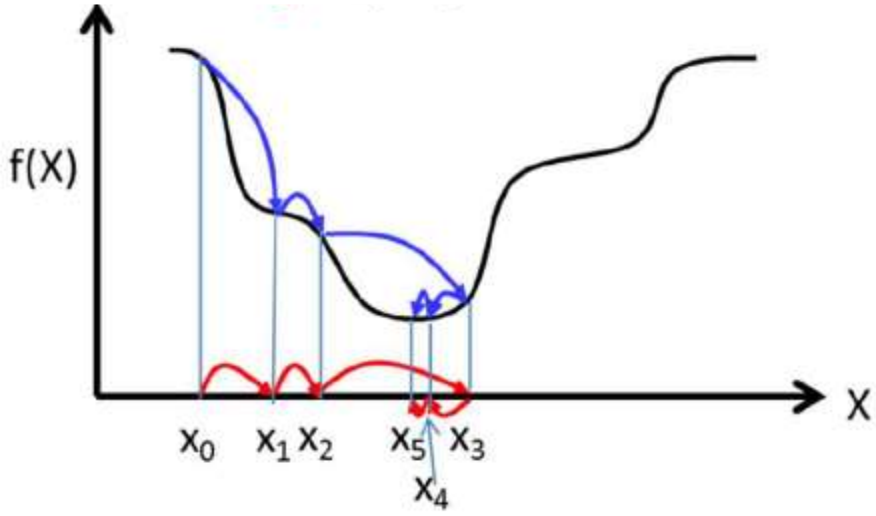
- Overview of CNN training
- CNN training as optimization
  - Data preprocessing
  - Weight initialization
  - Parameter update
  - Batch normalization

Acknowledgement: UofT, CMU & Feifei Li's cs231n notes

Training overview

- Supervised learning paradigm
- Mini-batch SGD

  Loop:
  - Sample a (mini-)batch of data
  - Forward propagation it through the network, compute loss
  - Backpropagation to calculate the gradients
  - Update the parameters using the gradient



Training overview

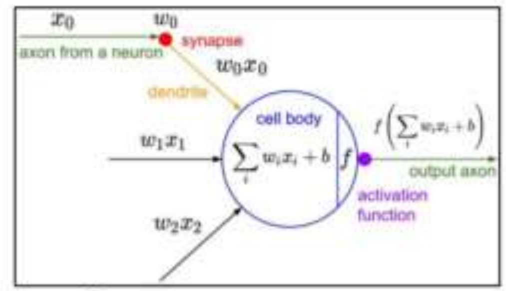
- Two aspects of training networks
  - Optimization
    - How do we minimize the loss function effectively?
  - Generalization
    - How do we avoid overfitting?
- CNN training pipeline
  - Data processing
  - Weight initialization
  - Parameter updates
  - Batch normalization

Data Preprocessing

- Motivation

Consider what happens when the input to a neuron is always positive...

$$f\left(\sum_i w_i x_i + b\right)$$



What can we say about the gradients on **w**?

We know that local gradient of sigmoid is always positive

We are assuming x is always positive

So!! Sign of gradient **for all w$_i$** is the same as the sign of upstream scalar gradient!

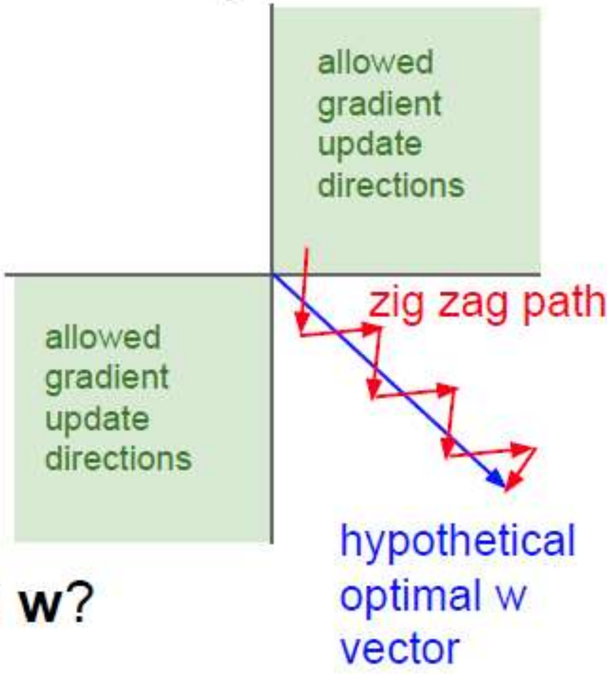
$$\boxed{\frac{\partial L}{\partial w}} = \boxed{\sigma\left(\sum_i w_i x_i + b\right)\left(1-\sigma\left(\sum_i w_i x_i + b\right)\right)}\boxed{x} \times \boxed{upstream\_gradient}$$

Data Preprocessing

- Motivation

Remember: Consider what happens when the input to a neuron is always positive...

$$f\left(\sum_i w_i x_i + b\right)$$



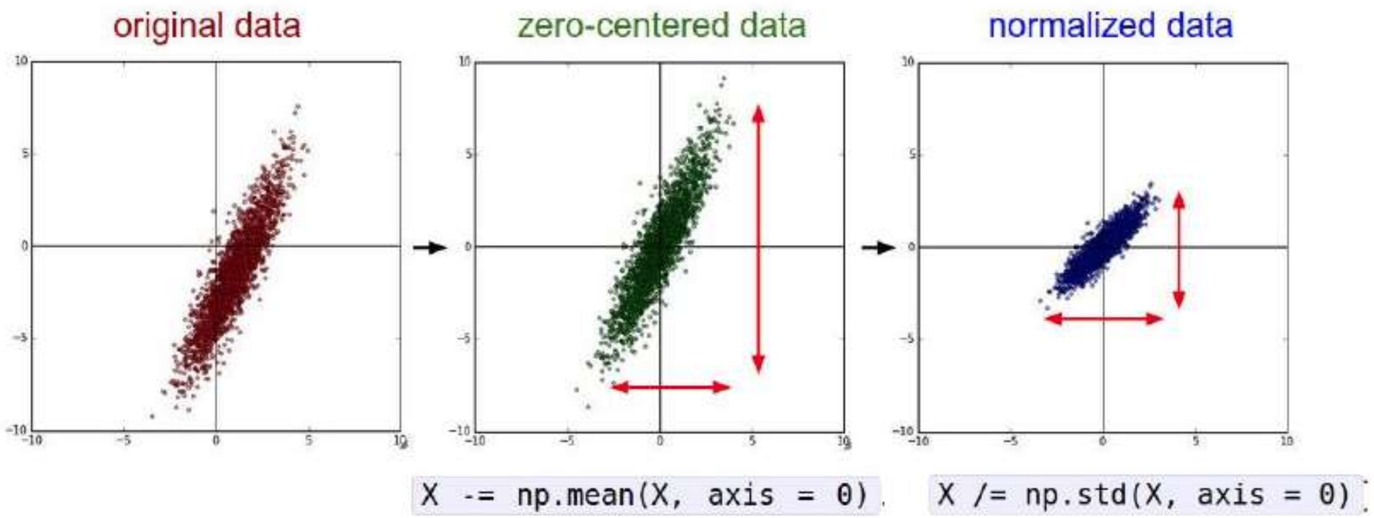
What can we say about the gradients on **w**?

Always all positive or all negative :(

(this is also why you want zero-mean data!)

Data Preprocessing

- Data normalization
  - Center your inputs to zero mean and unit variance

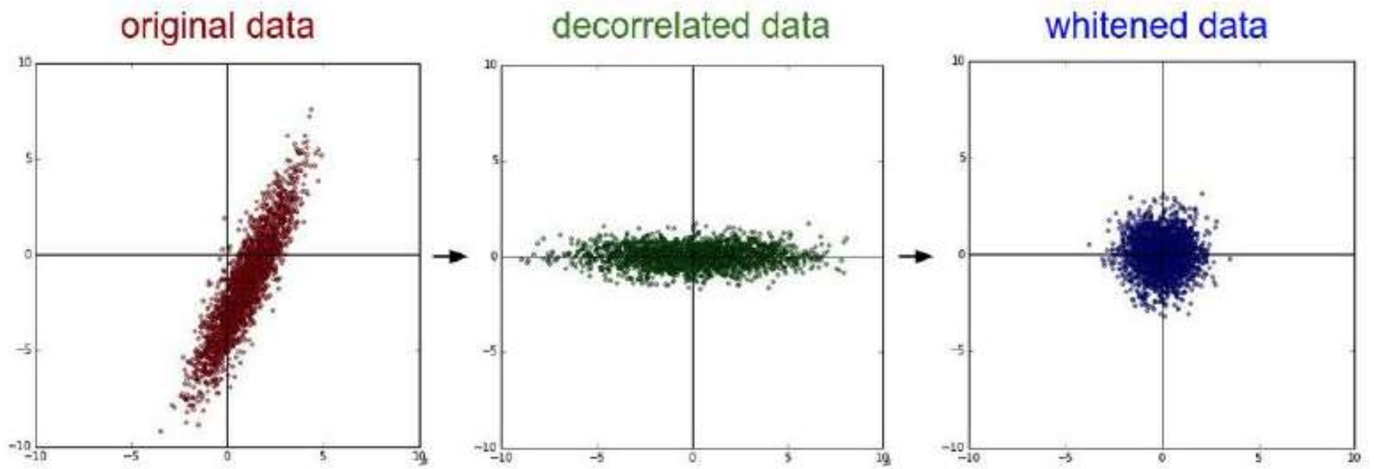


(Assume X [NxD] is data matrix, each example in a row)

Data Preprocessing

- More advanced methods

In practice, you may also see **PCA** and **Whitening** of the data



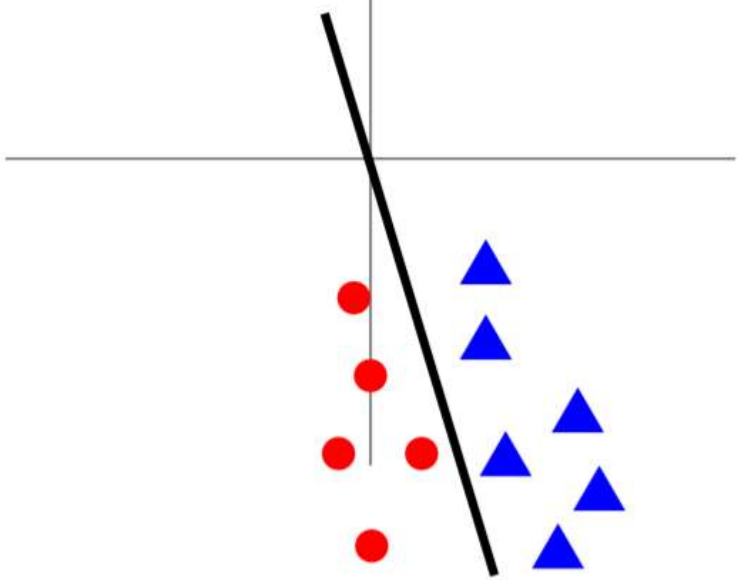
(data has diagonal covariance matrix)

(covariance matrix is the identity matrix)

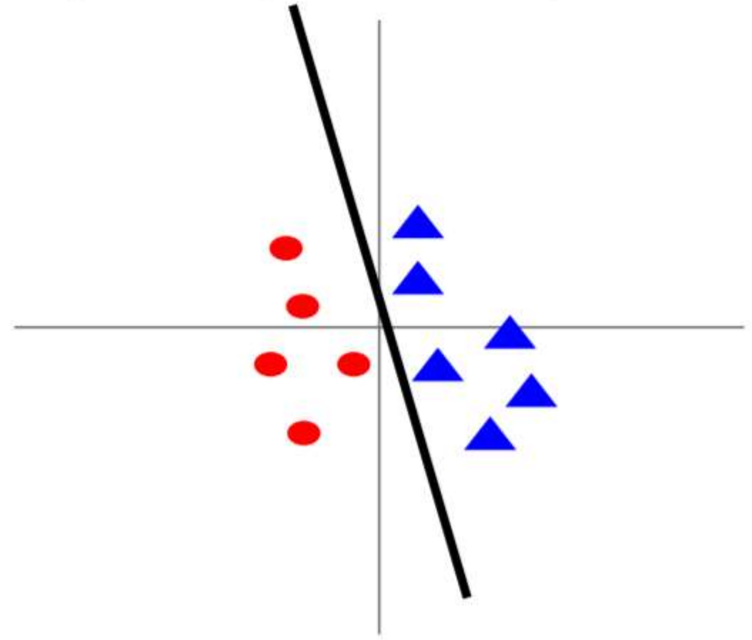
The whitening operation takes the data in the eigenbasis and divides every dimension by the eigenvalue to normalize the scale. The geometric interpretation of this transformation is that if the input data is a multivariable gaussian, then the whitened data will be a gaussian with zero mean and identity covariance matrix.

Data Preprocessing

Before normalization: classification loss very sensitive to changes in weight matrix; hard to optimize



After normalization: less sensitive to small changes in weights; easier to optimize



Data Preprocessing

- For visual recognition tasks
  - In practice for images: centering only
  - Not common to do PCA or whitening
- For example, CIFAR-10
  - Subtract the mean image (e.g. AlexNet)
    (mean image = [32,32,3] array)
  - Subtract per-channel mean (e.g. VGGNet)
    (mean along each channel = 3 numbers)
  - Subtract per-channel mean and Divide by per-channel std (e.g. ResNet)
    (mean along each channel = 3 numbers)

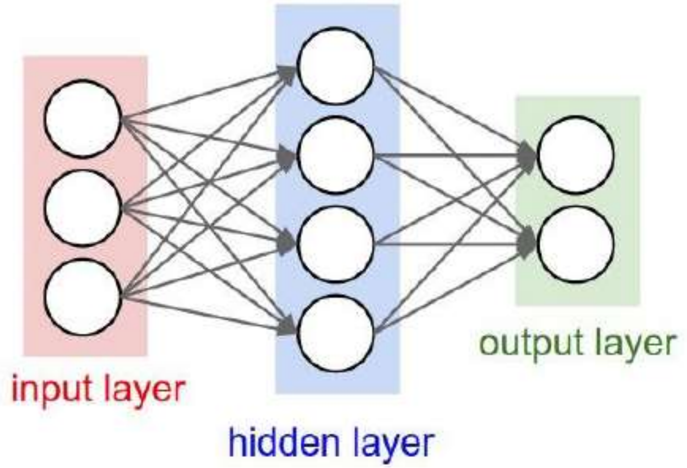
Outline

- Overview of CNN training
- CNN training as optimization
  - Data preprocessing
  - Weight initialization
  - Parameter update
  - Batch normalization

Acknowledgement: UofT, CMU & Feifei Li's cs231n notes

Weight Initialization

- Non-convex objective functions
  - Neural nets have a weight symmetry: permute all the hidden units in a given layer and obtain an equivalent solution.
  - Q: What happens when W=0 initialization is used?



Weight Initialization

- First idea: Small random numbers
  - Gaussian with zero mean and 1e-2 std

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

  - Simpler models to start
  - Outputs are close to uniform for classification

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

Weight Initialization

- Motivating example
  - Look at some activation statistics
  - E.g., 6-layer net with 4096 neurons on each layer using tanh non-linearities.

```
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

- Motivating example

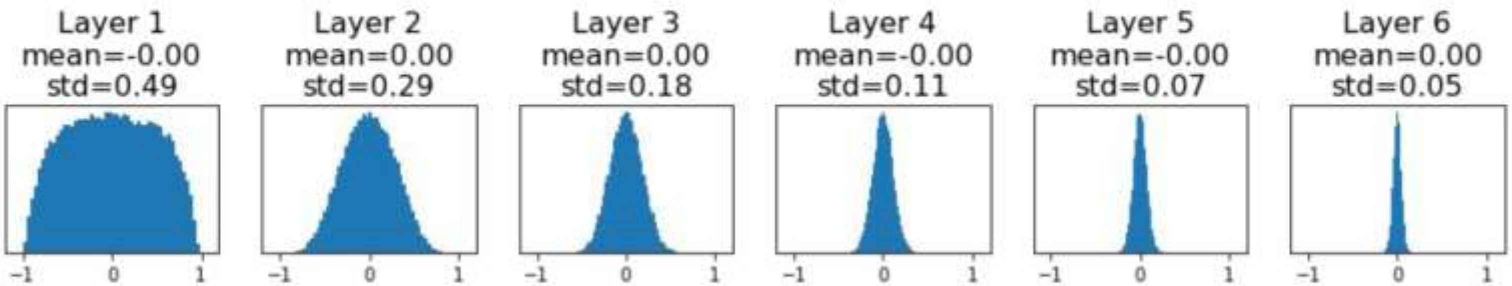
Weight Initialization: Activation statistics

```
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?



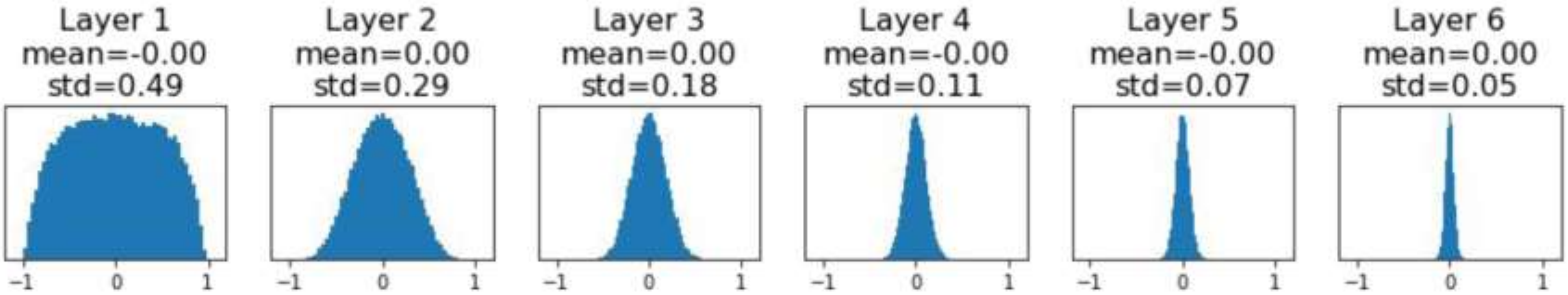
```
dims = [4096] * 7
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = 0.01 * np.random.randn(Din, Dout)
    x = np.tanh(x.dot(W))
    hs.append(x)
```

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

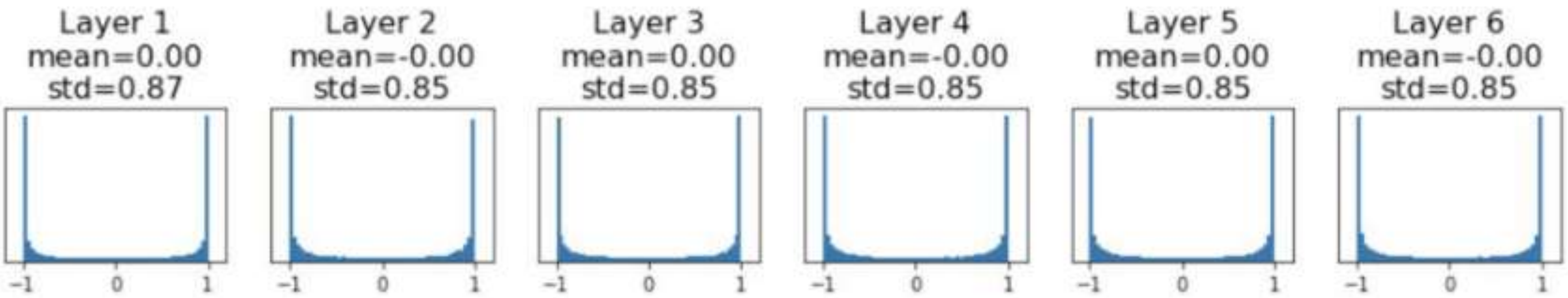
- Motivating example

```
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

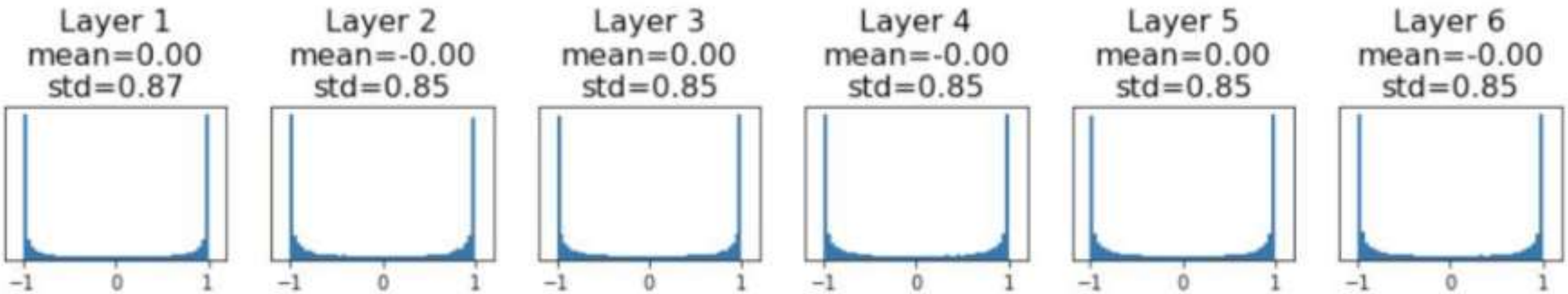
```
dims = [4096] * 7
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = 0.05 * np.random.randn(Din, Dout)
    x = np.tanh(x.dot(W))
    hs.append(x)
```

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 =(



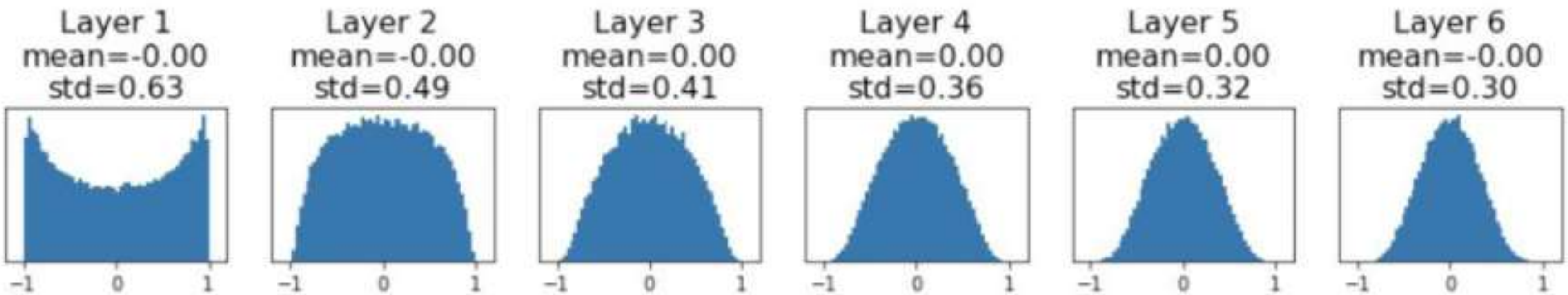
Weight Initialization

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

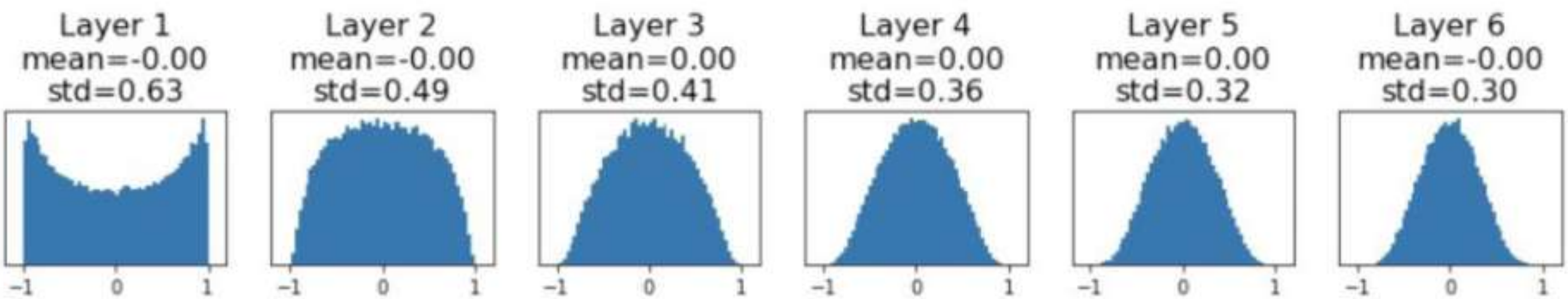
Weight Initialization

```python
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 filter_size^2 * input_channels



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

Weight Initialization

- Theoretic analysis

Suppose we have an input X with n components and a fully connected layer (also denoted linear or dense) with random weights W that outputs a number Y such that

$$Y = W_1X_1 + W_2X_2 + \ldots + W_nX_n$$

To make sure that the weights remain in a reasonable range, we expect that $Var(Y) = Var(X_i)_{i \in [1,n]}$

We also know how to compute the variance of the product of two random variables. Therefore

$$Var(W_iX_i) = E[X_i]^2 Var(W_i) + E[W_i]^2 Var(X_i) + Var(W_i)Var(X_i)$$

Both our inputs and weights have a mean 0. It simplifies to

$$Var(W_iX_i) = Var(W_i)Var(X_i)$$

Now we make a further assumption that the X_i and W_i are all independent and identically distributed (iid).

$$Var(Y) = Var(W_1X_1 + W_2X_2 + \ldots + W_nX_n) = nVar(W_i)Var(X_i)$$

It turns that, if we want to have $Var(Y) = Var(X_i)$, we must enforce the condition $nVar(W_i) = 1$.

$$Var(W_i) = \frac{1}{n} = \frac{1}{n_{in}}$$

Weight Initialization

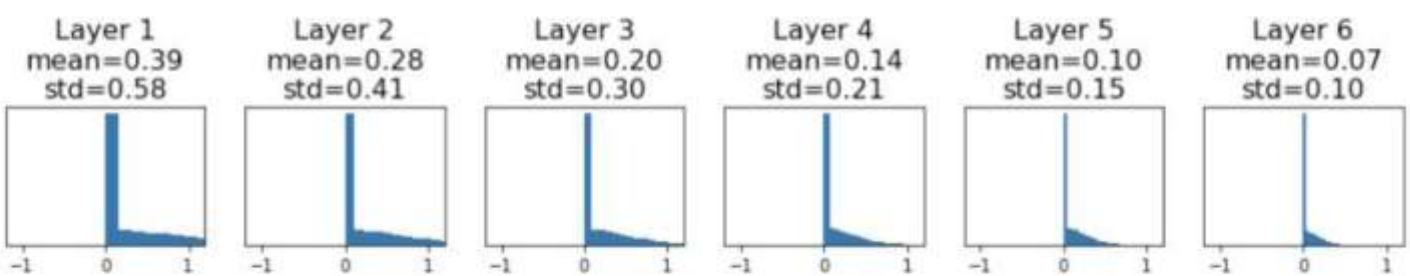
- Problems with ReLU activation
  - Xavier initialization assumes zero centered activation function, and hence breaks under ReLU

```
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 =(



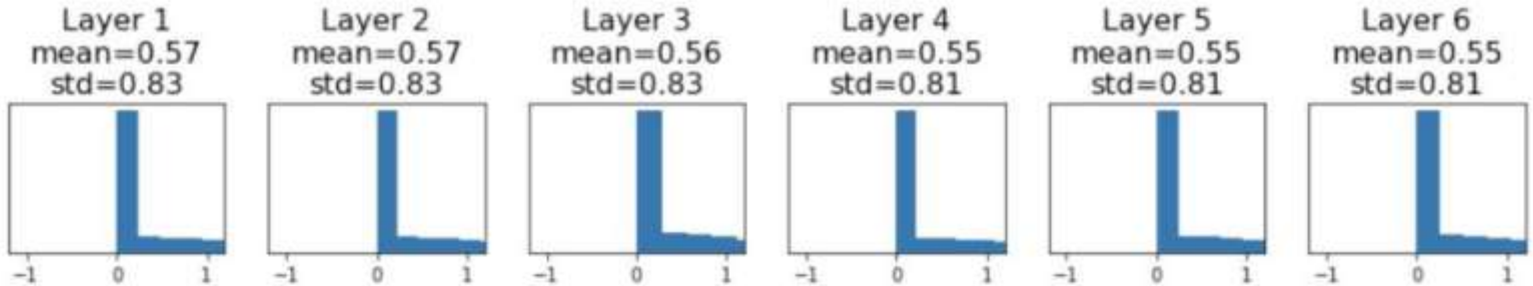
Weight Initialization

- Initialization for CNNs with ReLU [He et al., 2015]
  - MSRA Initialization: std = sqrt(2/fan_in)

```
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(2/Din)
    x = np.maximum(0, x.dot(W))
    hs.append(x)
```

ReLU correction: std = sqrt(2 / Din)

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



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

Weight Initialization

- Weight initialization is an active are of research…
  - Understanding the difficulty of training deep feedforward neural networks *by Glorot and Bengio, 2010*
  - Exact solutions to the nonlinear dynamics of learning in deep linear neural networks *by Saxe et al, 2013*
  - Random walk initialization for training very deep feedforward networks *by Sussillo and Abbott, 2014*
  - Delving deep into rectifiers: Surpassing human-level performance on ImageNet classification *by He et al., 2015*
  - Data-dependent Initializations of Convolutional Neural Networks *by Krähenbühl et al., 2015*
  - All you need is a good init, *Mishkin and Matas, 2015*
  - Fixup Initialization: Residual Learning Without Normalization, *Zhang et al, 2019*
  - The Lottery Ticket Hypothesis: Finding Sparse, Trainable Neural Networks, *Frankle and Carbin, 2019*

Outline

- Overview of CNN training
- CNN training as optimization
  - Data preprocessing
  - Weight initialization
  - Parameter update
  - Batch normalization

Acknowledgement: UofT, CMU & Feifei Li's cs231n notes

Optimization

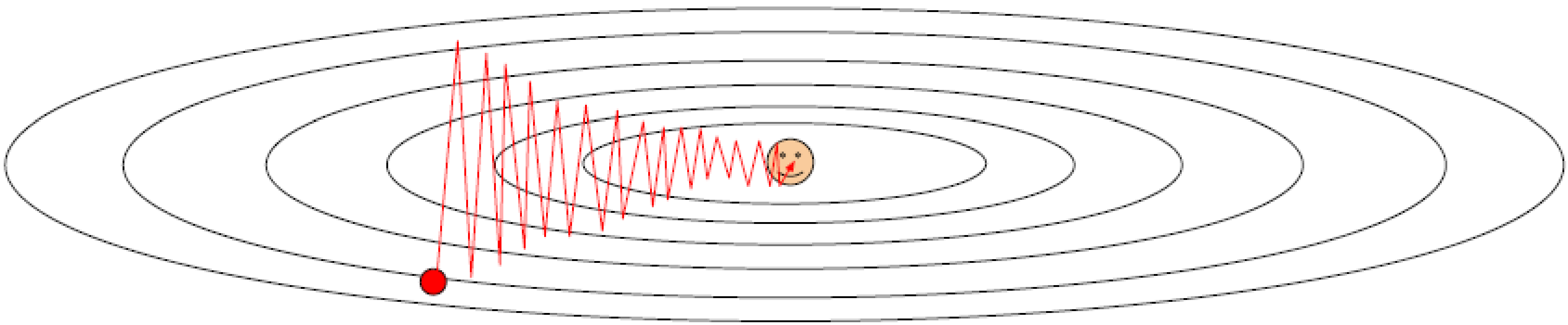
- Stochastic Gradient Descent

```
# Vanilla Gradient Descent

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

Optimization

- Problems with SGD

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



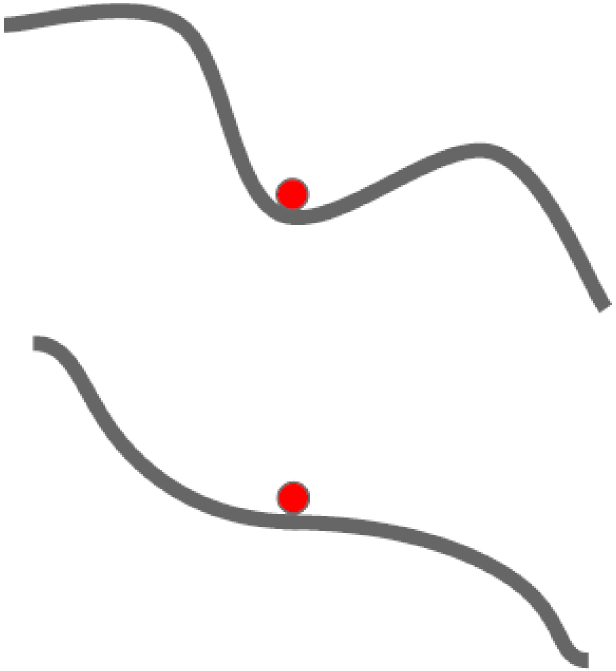
Loss function has high **condition number**: ratio of largest to smallest singular value of the Hessian matrix is large

Optimization

- Problems with SGD

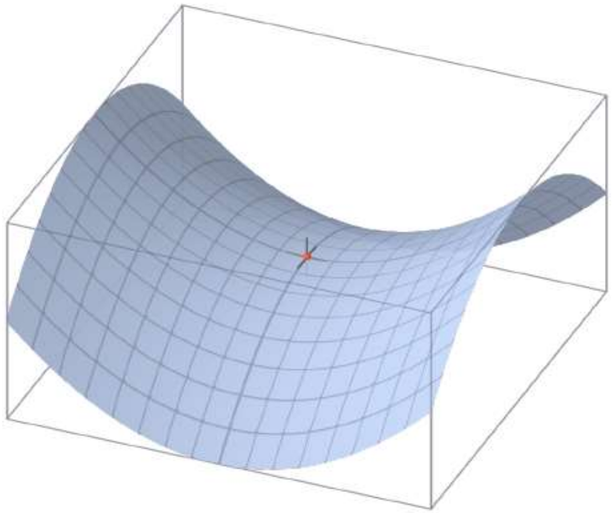
What if the loss function has a **local minima** or **saddle point**?

Zero gradient, gradient descent gets stuck



Optimization

- Problems with SGD
  - Saddle points are more common in high-dim space



At a saddle point $\frac{\partial \mathcal{E}}{\partial \boldsymbol{\theta}} = 0$, even though we are not at a minimum. Some directions curve upwards, and others curve downwards.

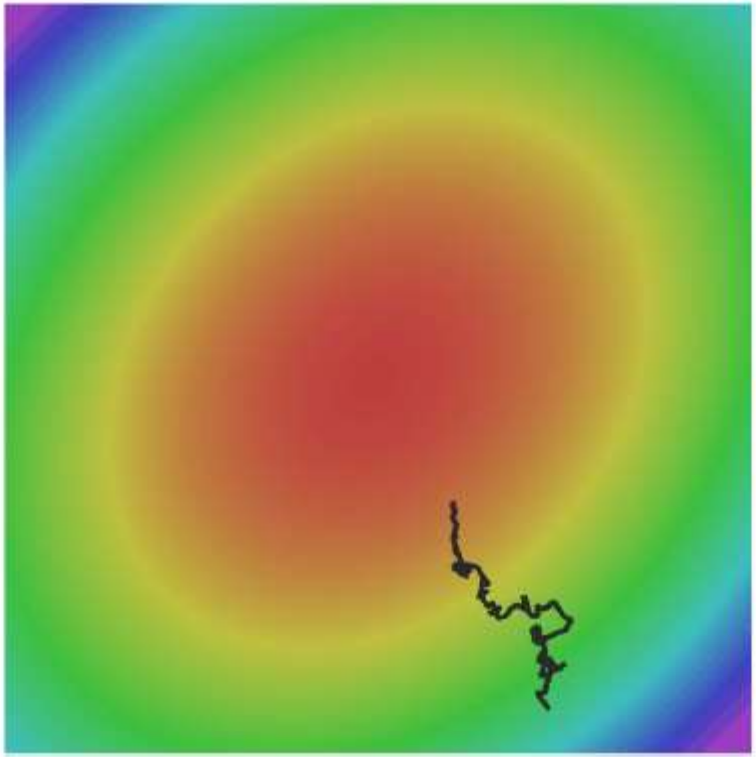
Optimization

- Problems with SGD

Our gradients come from minibatches so they can be noisy!

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

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



Optimization

- 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

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

Optimization

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

SGD+Momentum

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

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

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

SGD+Momentum

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

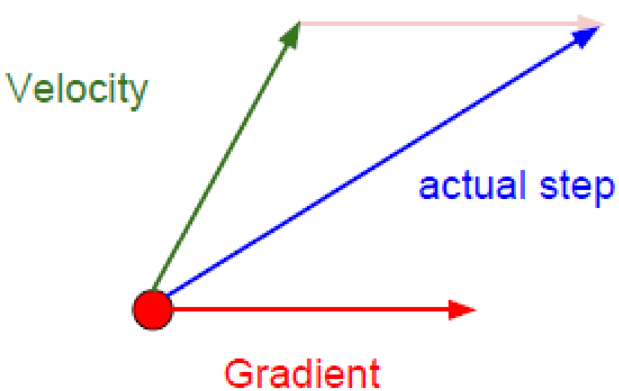
$$x_{t+1} = x_t - \alpha v_{t+1}$$

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

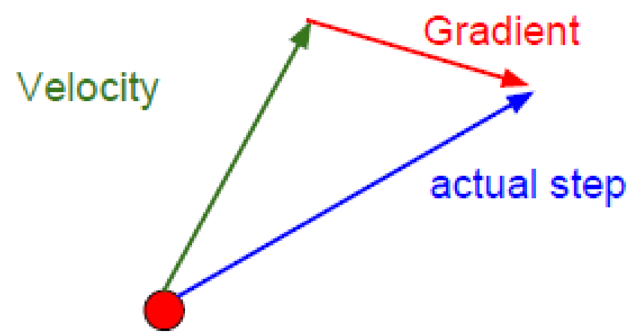
Optimization

- Nesterov Momentum
  - “Look ahead” to the point where updating using velocity would take us;
  - Compute gradient there and mix it with velocity to get actual update direction

Momentum update:



Nesterov Momentum



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

Nesterov, “A method of solving a convex programming problem with convergence rate O(1/k^2)”, 1983
Nesterov, “Introductory lectures on convex optimization: a basic course”, 2004
Sutskever et al, “On the importance of initialization and momentum in deel learning”, ICML 2013

Optimization

- Nesterov Momentum

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

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

Optimization

- Nesterov Momentum

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

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

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

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

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

Optimization

- AdaGrad

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

Added element-wise scaling of the gradient based on the historical sum of squares in each dimension

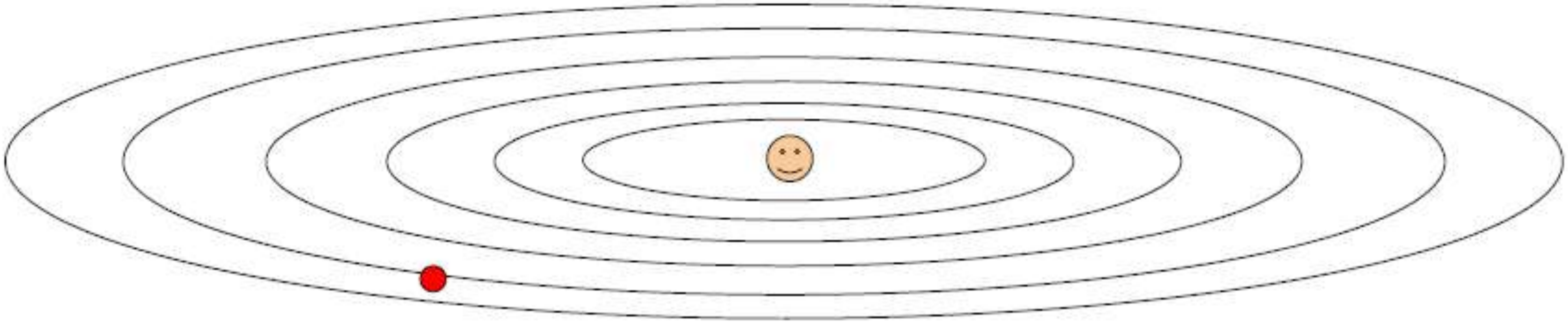
“Per-parameter learning rates” or “adaptive learning rates”

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

Optimization

- AdaGrad

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



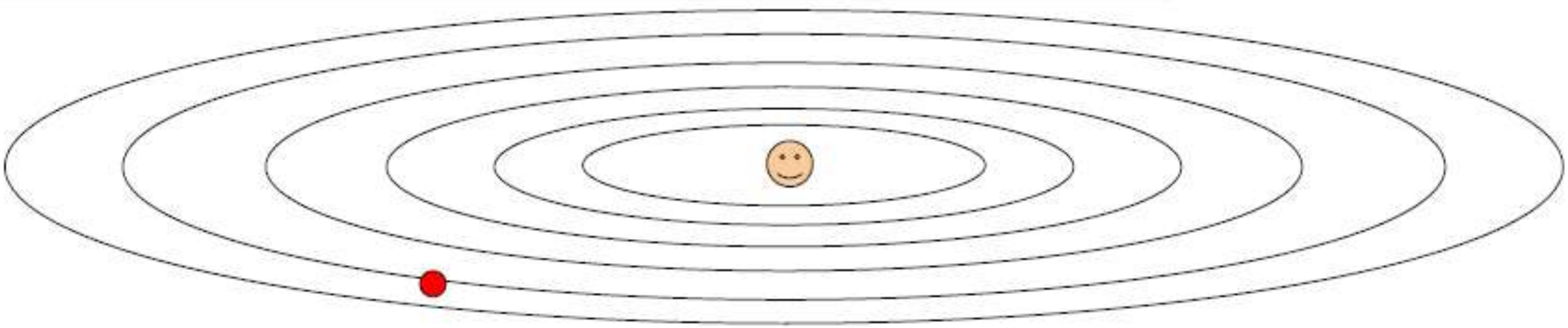
Q: What happens with AdaGrad?

Progress along “steep” directions is damped;
progress along “flat” directions is accelerated

Optimization

- AdaGrad

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



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

Decays to zero

Optimization

- RMSProp: smoothed version

AdaGrad

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

RMSProp

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

Tieleman and Hinton, 2012

Optimization

- Adam (almost)

```
first_moment = 0
second_moment = 0
while True:
  dx = compute_gradient(x)
  first_moment = beta1 * first_moment  + (1 - beta1) * dx
  second_moment = beta2 * second_moment + (1 - beta2) * dx * dx
  x -= learning_rate * first_moment / (np.sqrt(second_moment) + 1e-7))
```

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

Optimization

- Adam (almost)

```
first_moment = 0
second_moment = 0
while True:
  dx = compute_gradient(x)
  first_moment = beta1 * first_moment  + (1 - beta1) * dx
  second_moment = beta2 * second_moment + (1 - beta2) * dx * dx
  x -= learning_rate * first_moment / (np.sqrt(second_moment) + 1e-7))
```

Momentum

AdaGrad / RMSProp

Sort of like RMSProp with momentum

Q: What happens at first timestep?

Optimization

- Adam (full form)

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

Momentum

Bias correction

AdaGrad / RMSProp

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

Optimization

- Adam (full form)

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

Momentum

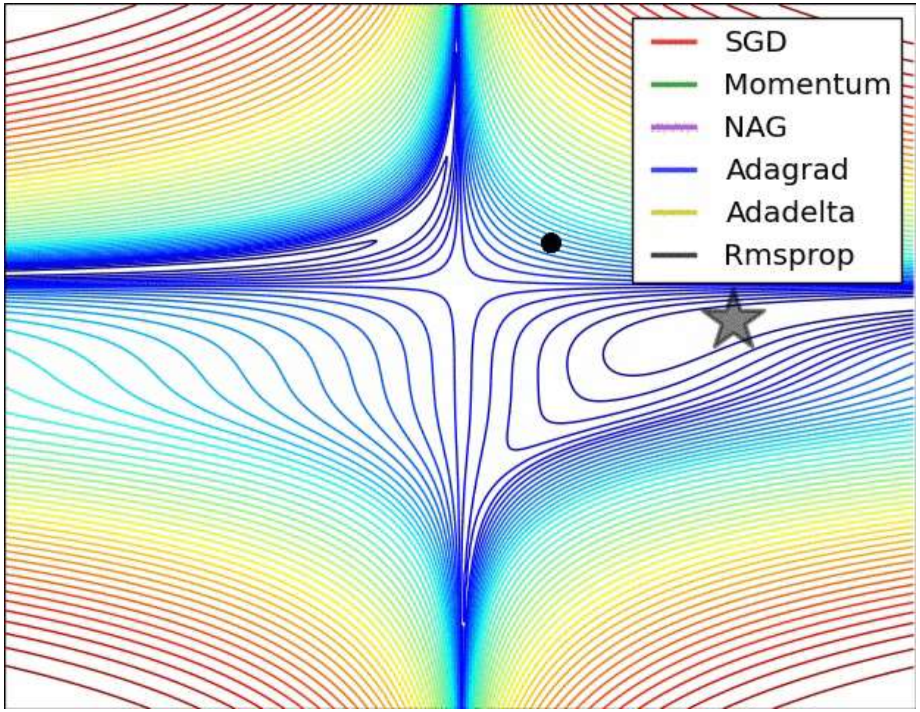
Bias correction

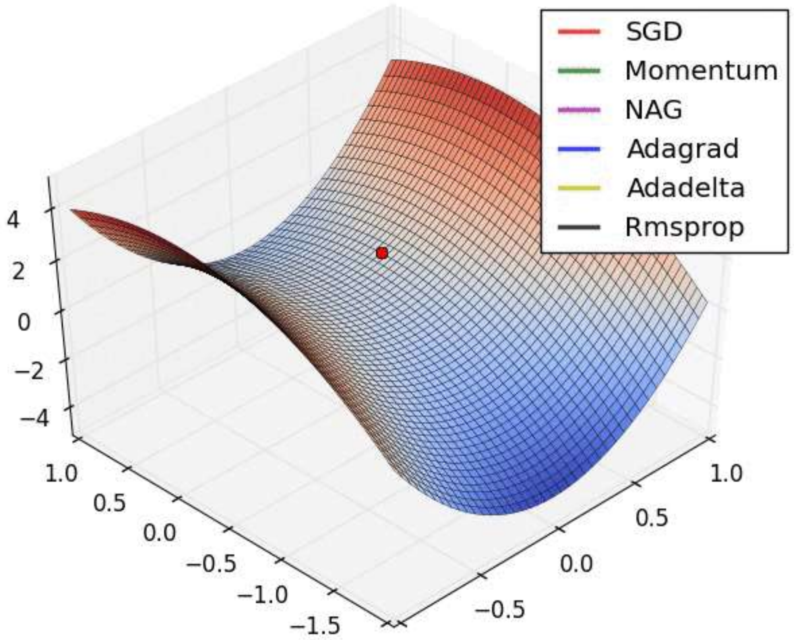
AdaGrad / RMSProp

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

Adam with beta1 = 0.9, beta2 = 0.999, and learning_rate = 1e-3 or 5e-4 is a great starting point for many models!

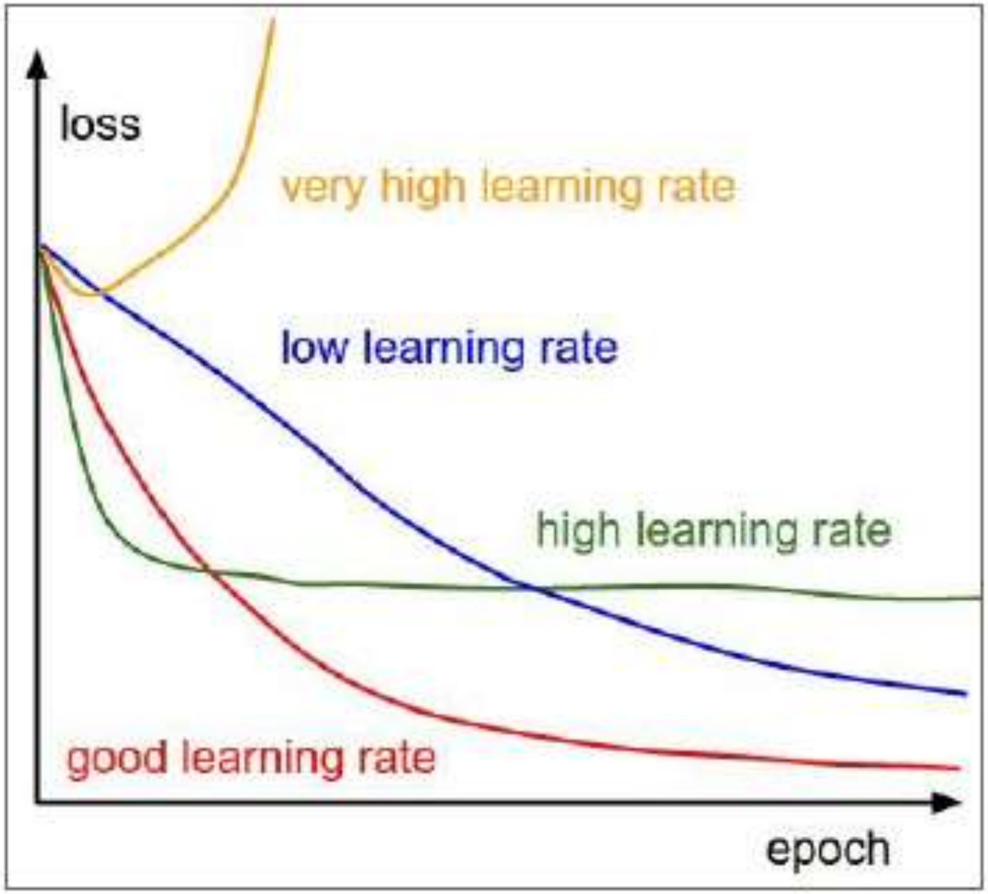
Optimization





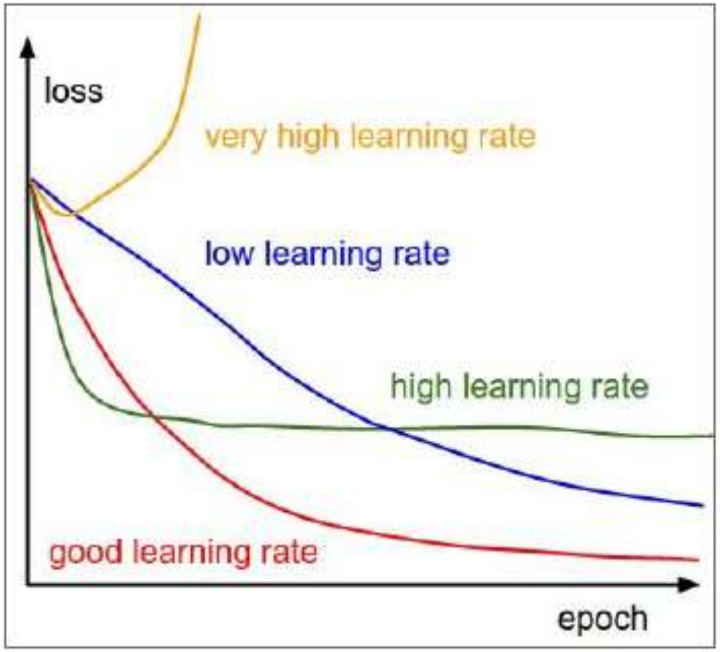
Learning rate

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



Learning rate

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



=> Learning rate decay over time!

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

exponential decay:

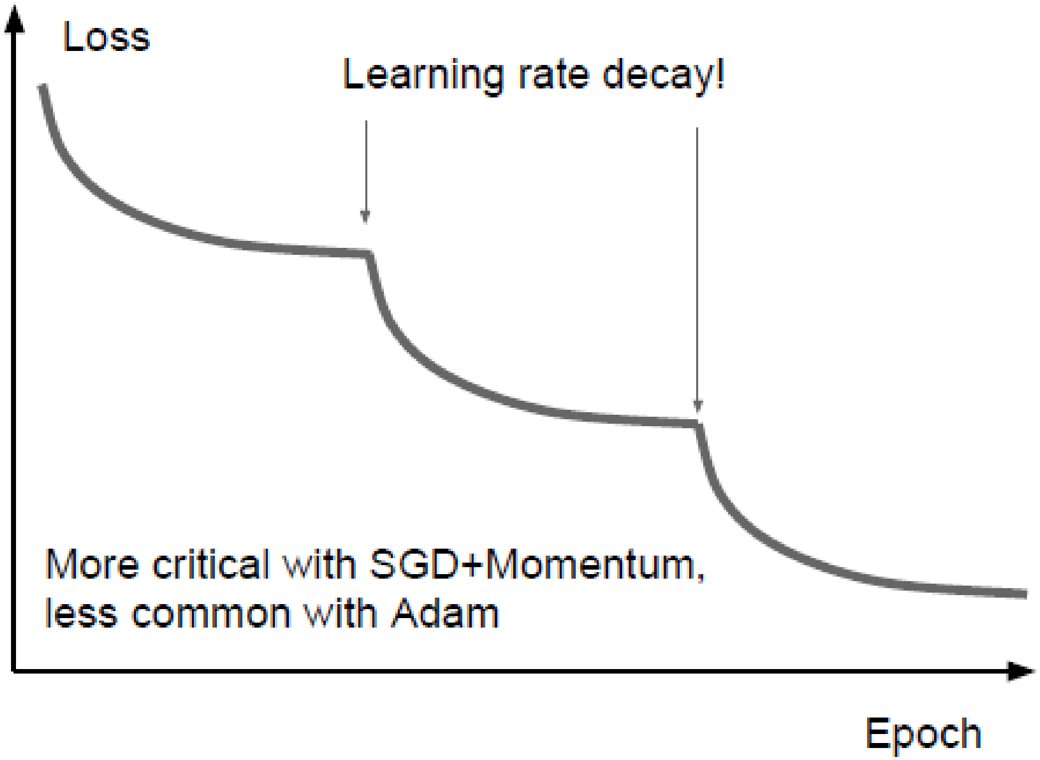
$$\alpha = \alpha_0 e^{-kt}$$

1/t decay:

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

Learning rate decay

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



Learning rate decay

- Cosine

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

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

- Linear

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

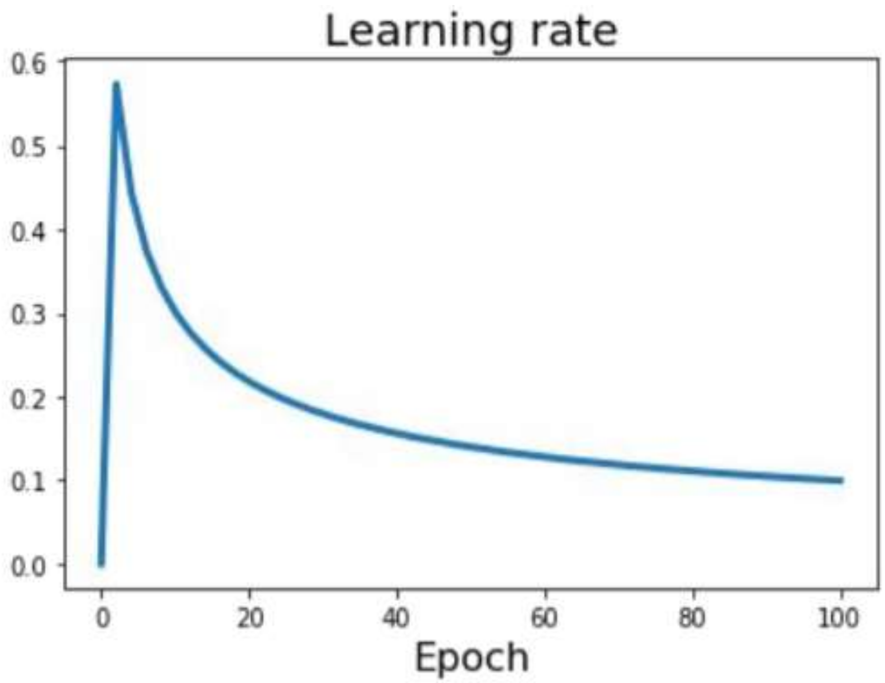
- Inverse sqrt

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

Loshchilov and Hutter, “SGDR: Stochastic Gradient Descent with Warm Restarts”, ICLR 2017
Radford et al, “Improving Language Understanding by Generative Pre-Training”, 2018
Feichtenhofer et al, “SlowFast Networks for Video Recognition”, arXiv 2018
Child at al, “Generating Long Sequences with Sparse Transformers”, arXiv 2019
Devlin et al, “BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding”, 2018
Vaswani et al, “Attention is all you need”, NIPS 2017

Learning rate decay

- Linear warmup



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

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

Goyal et al, “Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour”, arXiv 2017

What can we find

- Popular hypothesis
  - In large networks, saddle points are far more common than local minima
  - Gradient descent algorithms often get “stuck” in saddle points
  - Most local minima are equivalent and close to global minimum

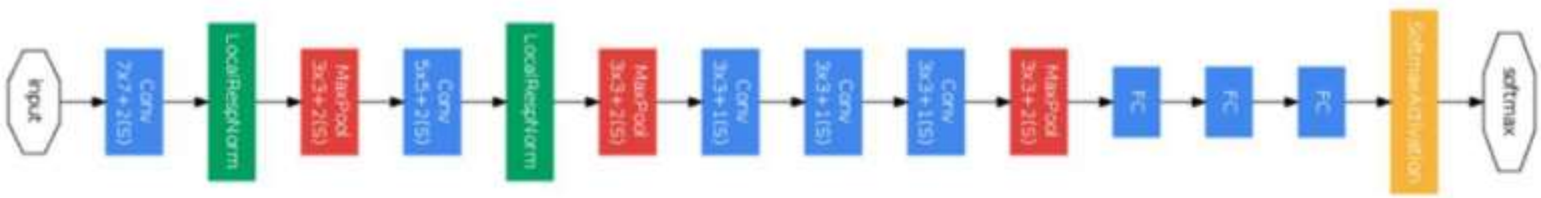
Outline

- Overview of CNN training
- CNN training as optimization
  - Data preprocessing
  - Weight initialization
  - Parameter update
  - Batch normalization

Acknowledgement: UofT, CMU & Feifei Li's cs231n notes

Batch Normalization

- Problem in deep network learning



$$\ell = F_2(F_1(\mathrm{u}, \Theta_1), \Theta_2)$$

  - Change of distribution in activation across layers

Batch Normalization

- Normalize the inputs to a layer:

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

consider a batch of activations at some layer.
To make each dimension unit gaussian, apply:

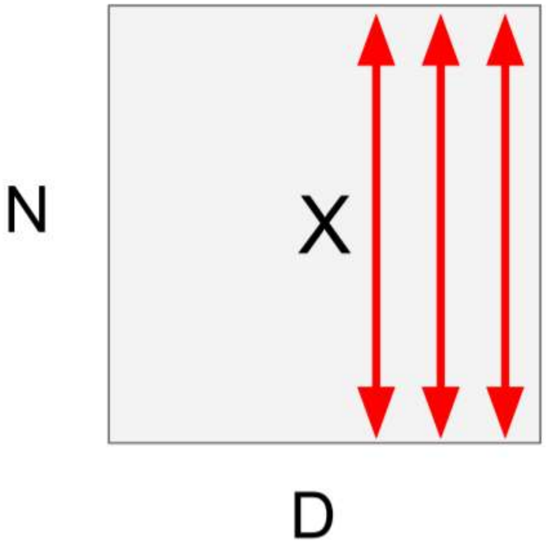
$$\widehat{x}^{(k)} = \frac{x^{(k)} - \mathrm{E}[x^{(k)}]}{\sqrt{\mathrm{Var}[x^{(k)}]}}$$

this is a vanilla
differentiable function...

Batch Normalization

- Layer details

Input: $x : N \times D$



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

Per-channel mean, shape is D

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

Per-channel var, shape is D

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

Normalized x, Shape is N x D

Batch Normalization

- Extra capacity:

Input: $x : N \times D$

Learnable scale and shift parameters:

$$\gamma, \beta : D$$

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

$$\mu_j = \frac{1}{N}\sum_{i=1}^{N} x_{i,j}$$ Per-channel mean, shape is D

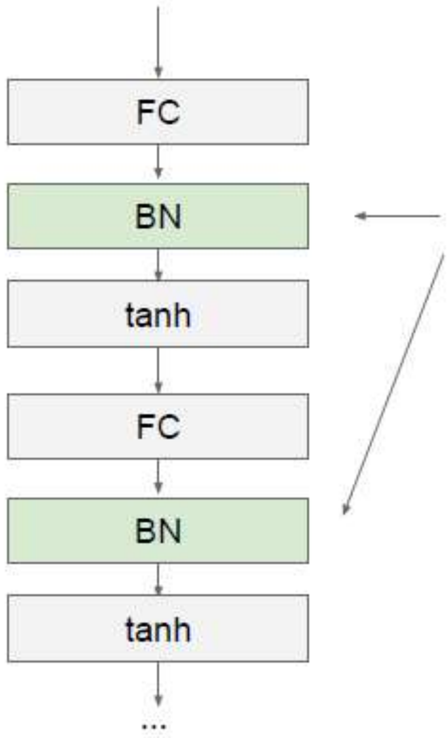
$$\sigma_j^2 = \frac{1}{N}\sum_{i=1}^{N}(x_{i,j} - \mu_j)^2$$ Per-channel var, shape is D

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

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

Batch Normalization

- Layer details



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

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

Batch Normalization

- Algorithm

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1\ldots m}\}$;
Parameters to be learned: γ, β

Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m} (x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma \widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

- Improves gradient flow through the network
- Allows higher learning rates
- Reduces the strong dependence on initialization
- Acts as a form of regularization in a funny way, and slightly reduces the need for dropout, maybe

Batch Normalization

Batch Normalization: Test-Time

Estimates depend on minibatch; can't do this at test-time!

Input: $x : N \times D$

Learnable scale and shift parameters:

$\gamma, \beta : D$

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

$$\boxed{\begin{aligned} \mu_j &= \frac{1}{N}\sum_{i=1}^{N} x_{i,j} && \text{Per-channel mean, shape is D} \\ \sigma_j^2 &= \frac{1}{N}\sum_{i=1}^{N} (x_{i,j} - \mu_j)^2 && \text{Per-channel var, shape is D} \end{aligned}}$$

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

Normalized x, Shape is N x D

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

Output, Shape is N x D

Batch Normalization

- Test time

Input: $x : N \times D$

Learnable scale and shift parameters:

$$\gamma, \beta : D$$

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

$\mu_j =$ (Running) average of values seen during training — Per-channel mean, shape is D

$\sigma_j^2 =$ (Running) average of values seen during training — Per-channel var, shape is D

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

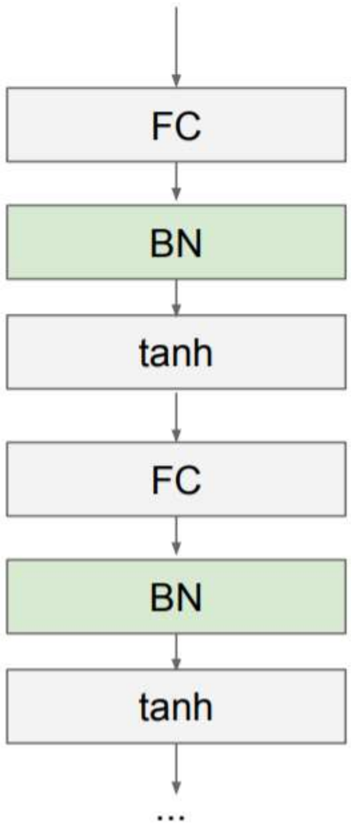
Normalized x, Shape is N x D

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

Output, Shape is N x D

Batch Normalization

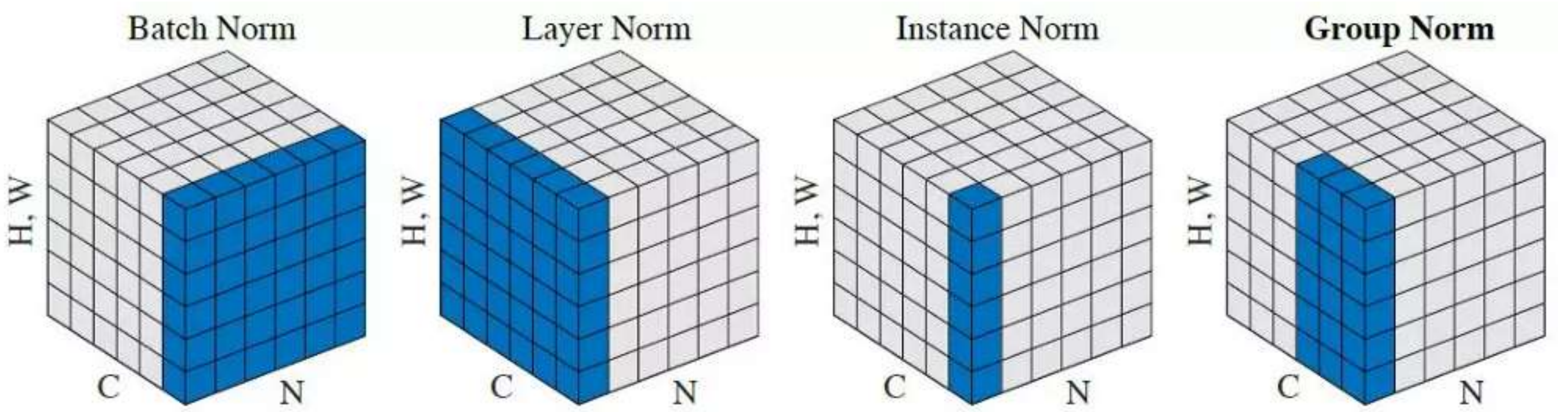
- Benefits



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

Batch-like Normalization

- Layer normalization (Ba, Kiros, Hinton, 2016)
- Instance normalization (Ulyanov, Vedaldi, Lempitsky, 2016)
- Group normalization (Wu and He, 2018)



Batch Normalization

- ConvNets

Batch Normalization for **fully-connected** networks

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

Normalize

$$\boldsymbol{\mu}, \boldsymbol{\sigma}: \ 1 \times D$$

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

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

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

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

Normalize

$$\boldsymbol{\mu}, \boldsymbol{\sigma}: \ 1 \times C \times 1 \times 1$$

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

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

Layer Normalization

- Batch Normalization vs. Layer Normalization

Batch Normalization for fully-connected networks

$x: N \times D$

Normalize

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

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

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

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

$x: N \times D$

Normalize

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

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

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

Ba, Kiros, and Hinton, “Layer Normalization”, arXiv 2016

Instance Normalization

Batch Normalization for convolutional networks

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

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

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

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

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

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

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

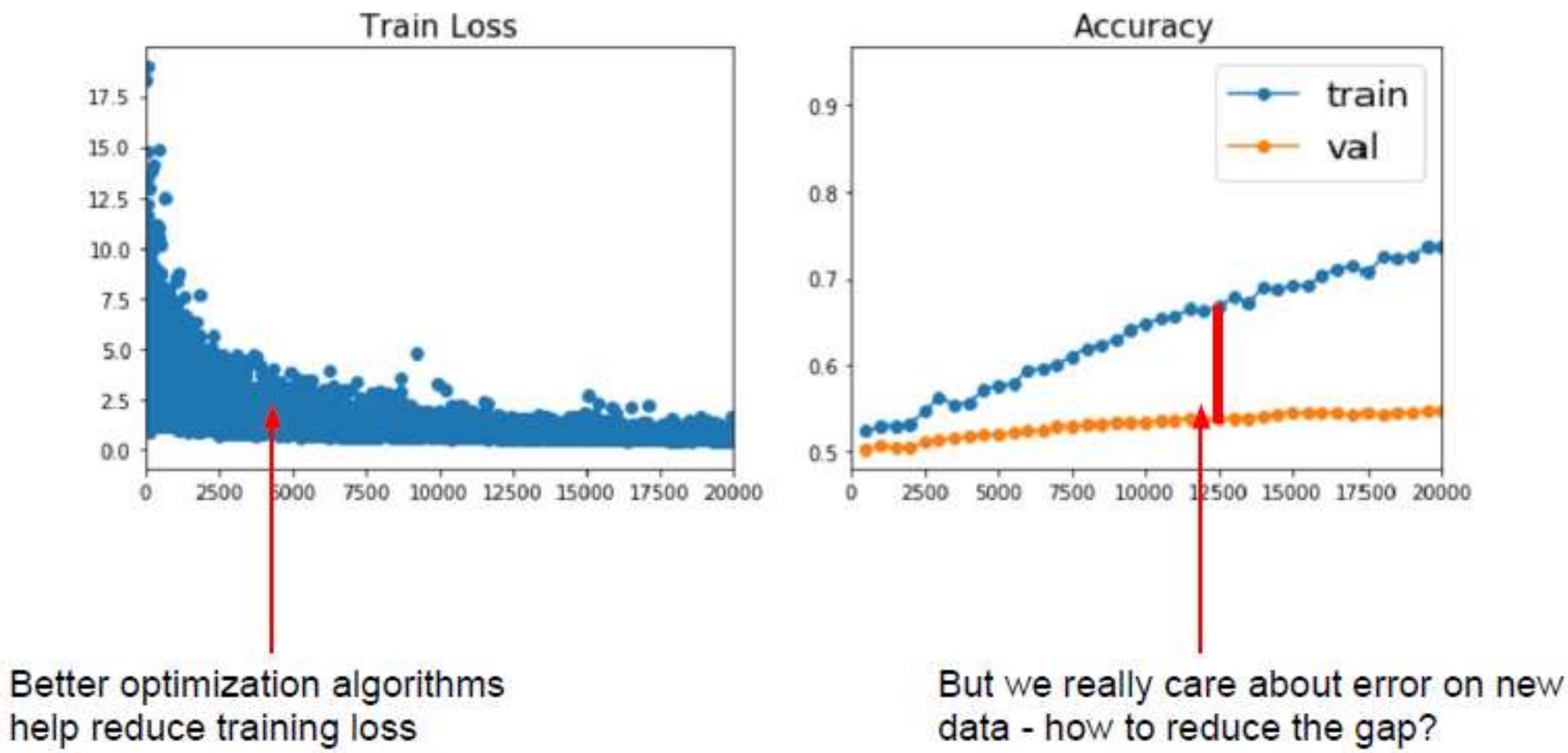
Ulyanov et al, Improved Texture Networks: Maximizing Quality and Diversity in Feed-forward Stylization and Texture Synthesis, CVPR 2017

Training overview

- Two aspects of training networks
  - Optimization
    - How do we minimize the loss function effectively?
  - Generalization
    - How do we avoid overfitting?
- CNN training pipeline
  - Data processing
  - Weight initialization
  - Parameter updates
  - Batch normalization
- Avoid overfitting: Regularization

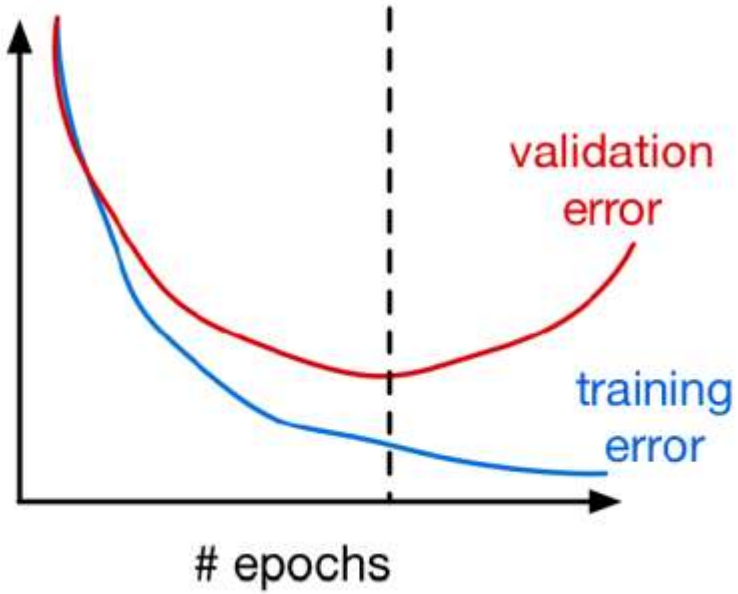
Beyond Training Error

- How do we generalize to unseen data?
  - Well studied but still poorly understood



Early Stopping

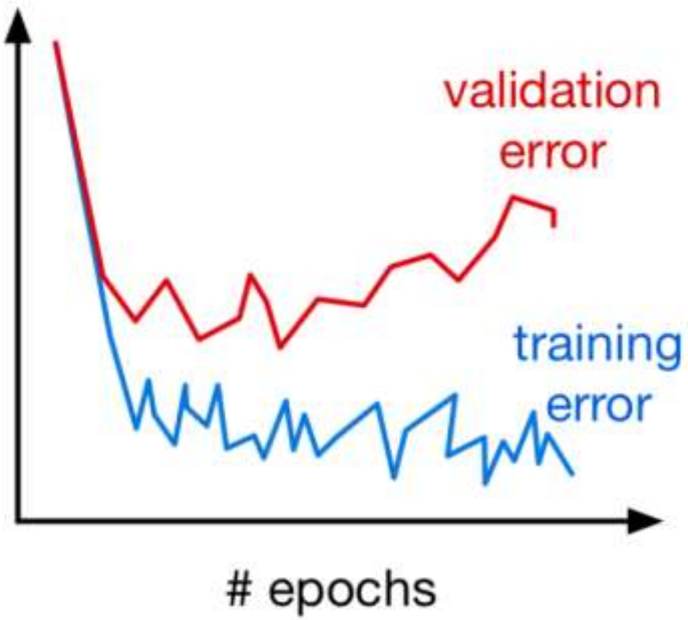
- Early stopping: monitor performance on a validation set, stop training when the validation error starts going up.
  - We don't always want to find a global (or even local) optimum of our cost function.



  - Weights start out small, so it takes time for them to grow large. Therefore, it has a similar effect to weight decay.

Early Stopping

- A slight catch: validation error fluctuates because of stochasticity in the updates.
  - Determining when the validation error has actually leveled off can be tricky.
  - May use temporal smoothing



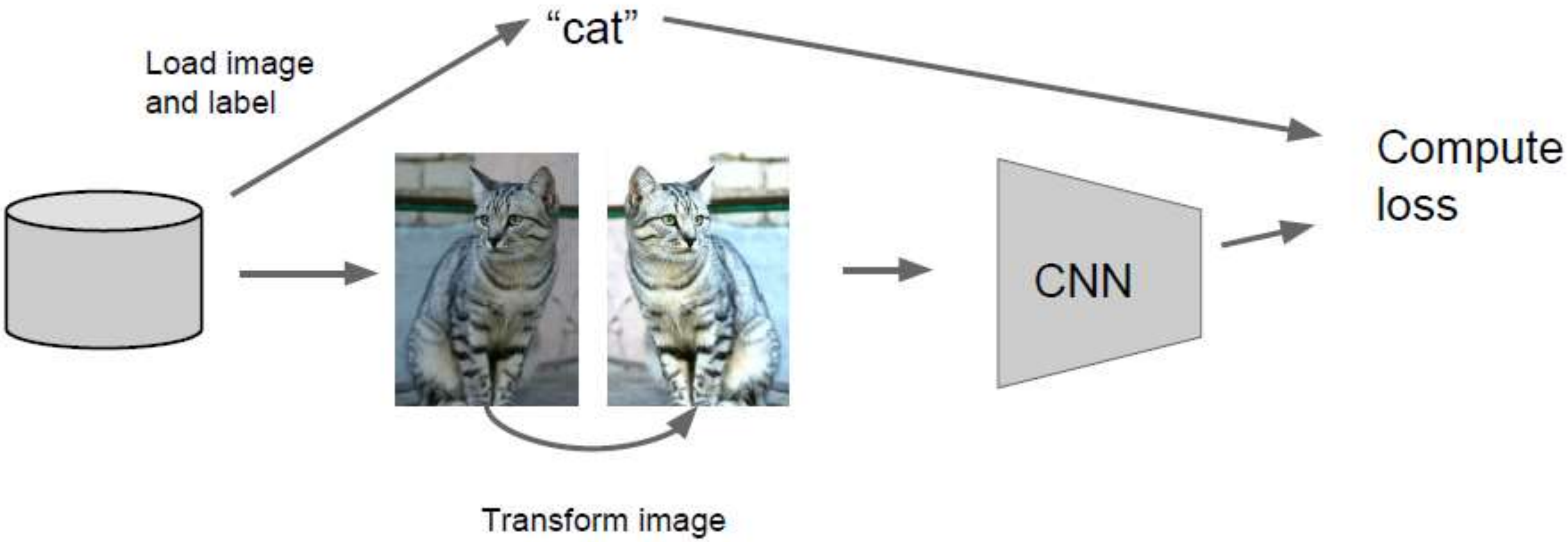
Outline

- Regularization in CNN training
  - Data Augmentation
  - Weight Regularization & Transfer Learning
  - Stochastic Regularization
  - Hyper-parameter optimization

Acknowledgement: Feifei Li's cs231n notes

Data Augmentation

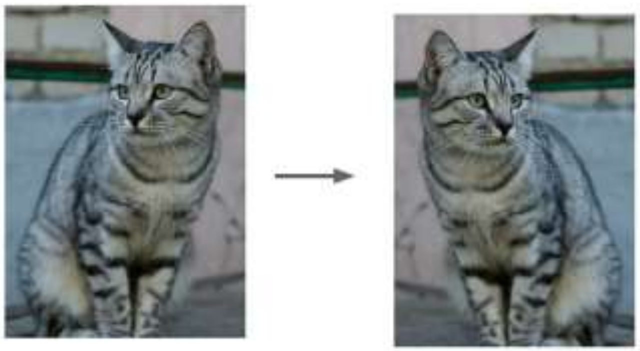
- Create more data for regularization



Data Augmentation

- Create more data for regularization

Horizontal Flips



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

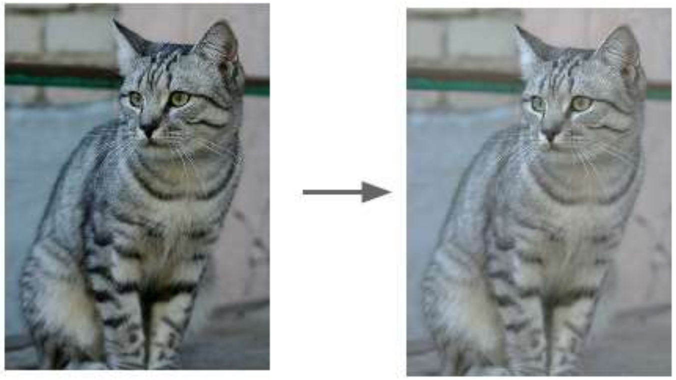
Testing: average a fixed set of crops
ResNet:
1. Resize image at 5 scales: {224, 256, 384, 480, 640}
2. For each size, use 10 224 x 224 crops: 4 corners + center, + flips

Data Augmentation

- Create more data for regularization

Color Jitter

Simple: Randomize contrast and brightness



More Complex:

1. Apply PCA to all [R, G, B] pixels in training set
2. Sample a “color offset” along principal component directions
3. Add offset to all pixels of a training image

(As seen in *[Krizhevsky et al. 2012]*, ResNet, etc)

Data Augmentation

- Create more data for regularization
- Examples (for visual recognition)
  - translation
  - horizontal or vertical
  - flip
  - rotation
  - smooth warping
  - noise (e.g. flip random pixels)
- The choice of transformations depends on the task.
  - E.g. horizontal flip for object recognition, but not handwritten digit recognition.

Data Augmentation

- AutoAugment (Cubuk et al, Arxiv 2018)
  - An automatic way to design custom data augmentation policies for computer vision datasets,
  - Selecting an optimal policy from a search space of 2.9×10^{32} image transformation possibilities.
    - E.g., guiding the selection of basic image transformation operations, such as flipping an image horizontally/vertically, rotating an image, changing the color of an image, etc.
  - Using reinforcement learning strategy (More later…)
- Results
  - New state of the art: ImageNet: 83.54% top1 accuracy; SVHN: error rate 1.02%.
  - AutoAugment policies are found to be transferable to other vision datasets.

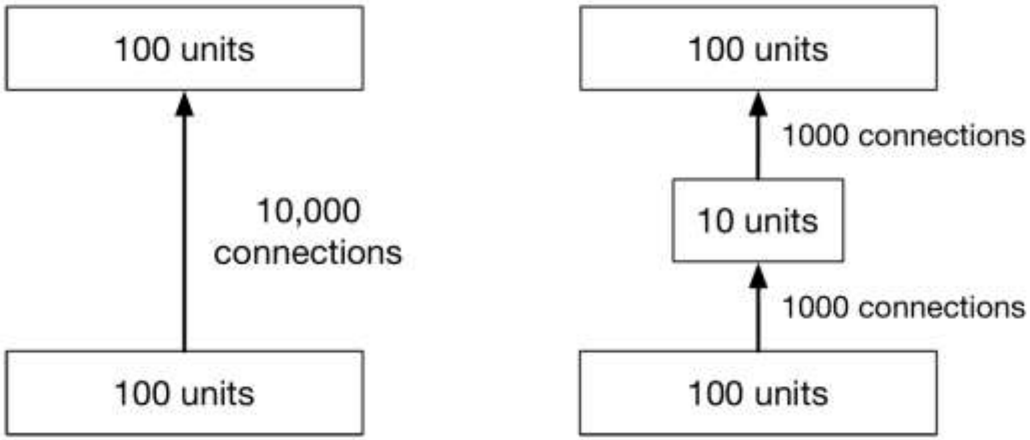
Outline

- Regularization in CNN training
  - Data Augmentation
  - Weight Regularization & Transfer Learning
  - Stochastic Regularization
  - Hyper-parameter optimization

Acknowledgement: Feifei Li's cs231n notes

Reducing # of Parameters

- Reducing the number of layers or the number of parameters per layer.
- Adding a linear bottleneck layer:



  - The first network is strictly more expressive than the second (i.e. it can represent a strictly larger class of functions). (Why?)
  - Remember how linear layers don't make a network more expressive? They might still improve generalization.

Weight Regularization

- L_2 regularization / weight decay
  - Encouraging the weights to be small in magnitude

$$\mathcal{E}_{\text{reg}} = \mathcal{E} + \lambda\mathcal{R} = \mathcal{E} + \frac{\lambda}{2}\sum_j w_j^2$$

  - The gradient update can be interpreted as weight decay

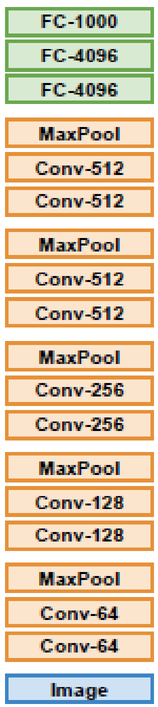
$$\begin{aligned}\mathbf{w} &\leftarrow \mathbf{w} - \alpha\left(\frac{\partial\mathcal{E}}{\partial\mathbf{w}} + \lambda\frac{\partial\mathcal{R}}{\partial\mathbf{w}}\right)\\ &= \mathbf{w} - \alpha\left(\frac{\partial\mathcal{E}}{\partial\mathbf{w}} + \lambda\mathbf{w}\right)\\ &= (1-\alpha\lambda)\mathbf{w} - \alpha\frac{\partial\mathcal{E}}{\partial\mathbf{w}}\end{aligned}$$

Transfer Learning

Transfer Learning with CNNs

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

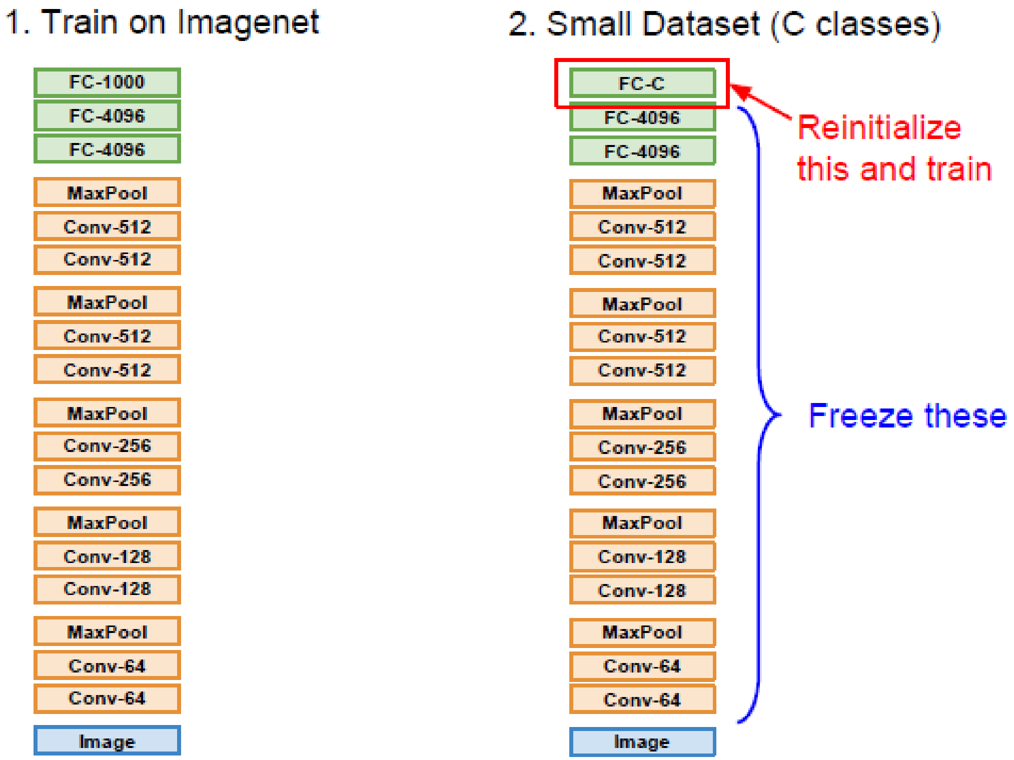
1. Train on Imagenet



Transfer Learning

Transfer Learning with CNNs

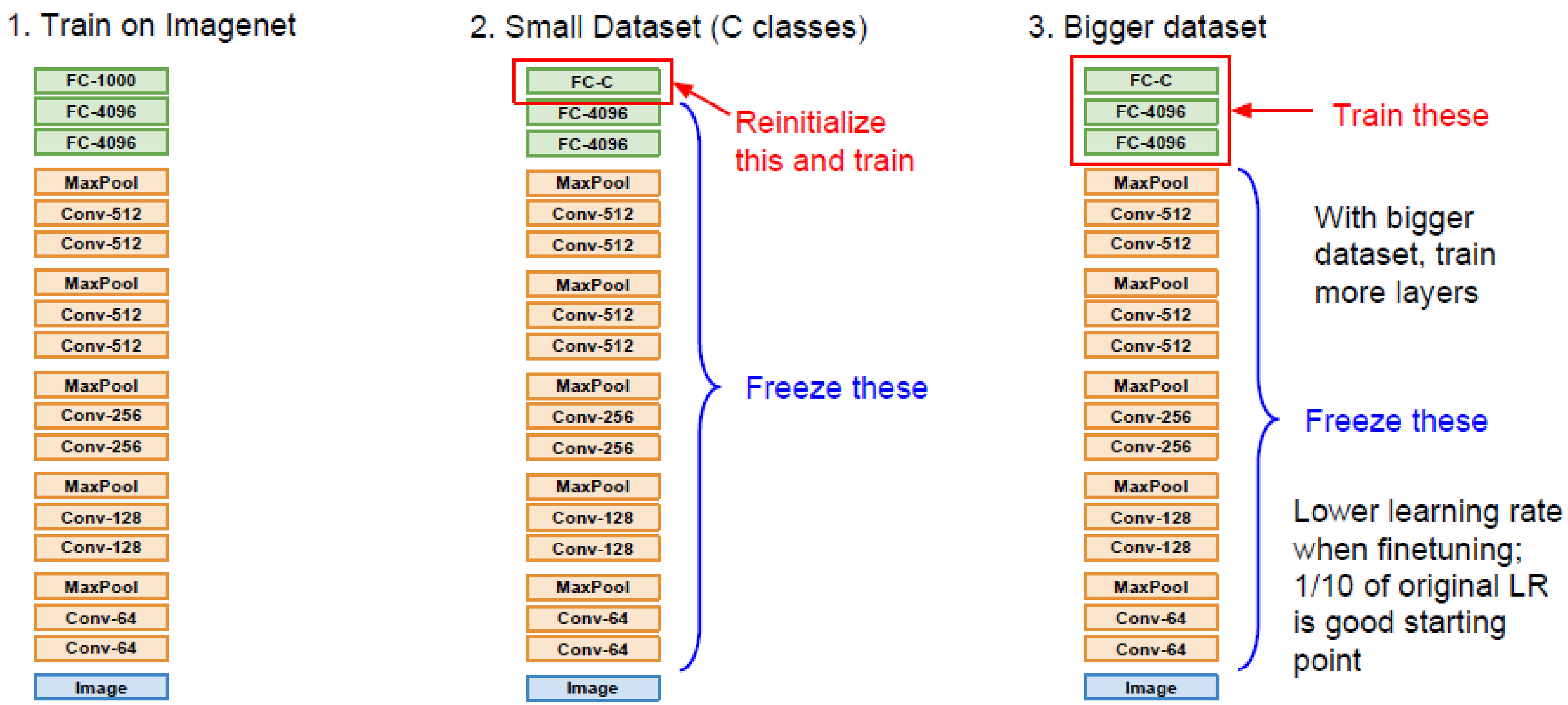
Donahue et al, "DeCAF: A Deep Convolutional Activation Feature for Generic Visual Recognition", ICML 2014
Razavian et al, "CNN Features Off-the-Shelf: An Astounding Baseline for Recognition", CVPR Workshops 2014



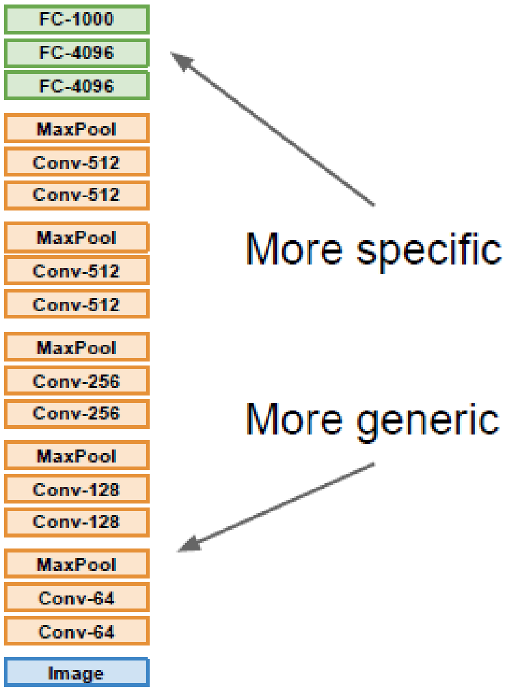
Transfer Learning

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

Transfer Learning with CNNs



Transfer Learning



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

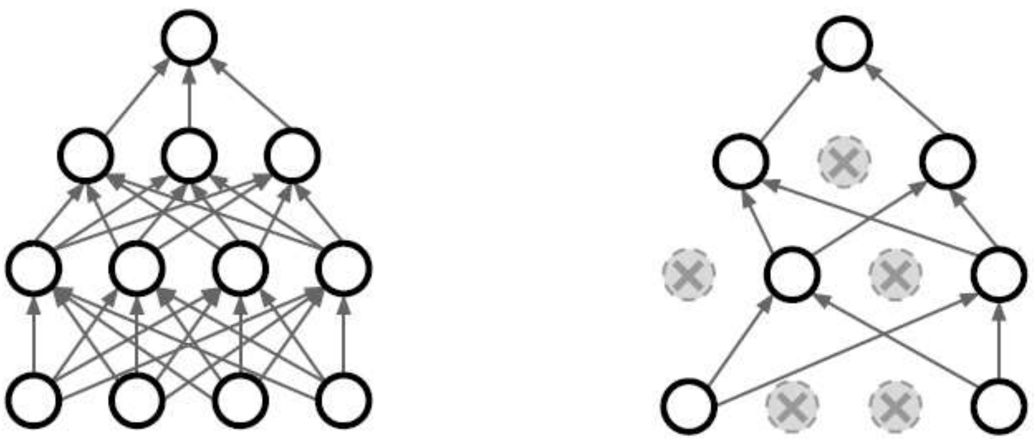
Outline

- Regularization in CNN training
  - Data Augmentation
  - Weight Regularization & Transfer Learning
  - Stochastic Regularization
  - Hyper-parameter optimization
- Network Architectures

Acknowledgement: Feifei Li's cs231n notes

Stochastic Regularization

- For a network to overfit, its computations need to be really precise. This suggests regularizing them by injecting noise into the computations, a strategy known as stochastic regularization.
- Dropout is a stochastic regularizer which randomly deactivates a subset of the units



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

Dropout

- Operations

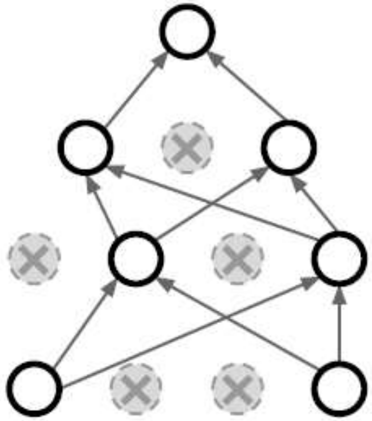
$$h_i = m_i \cdot \phi(z_i),$$

where m_i is a Bernoulli random variable, independent for each hidden unit.

Regularization: Dropout

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

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

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

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

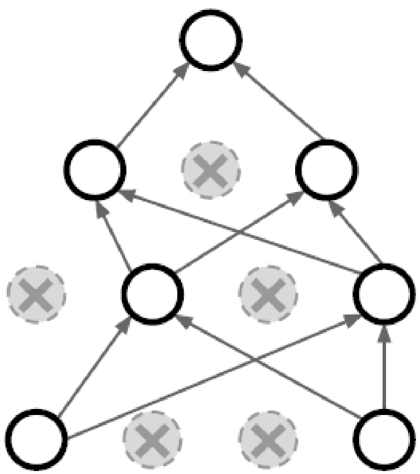
Example forward pass with a 3-layer network using dropout



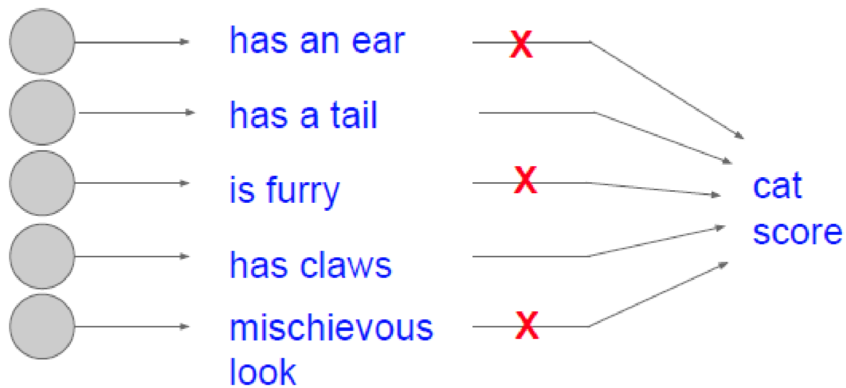
Understanding Dropout

Regularization: Dropout

How can this possibly be a good idea?

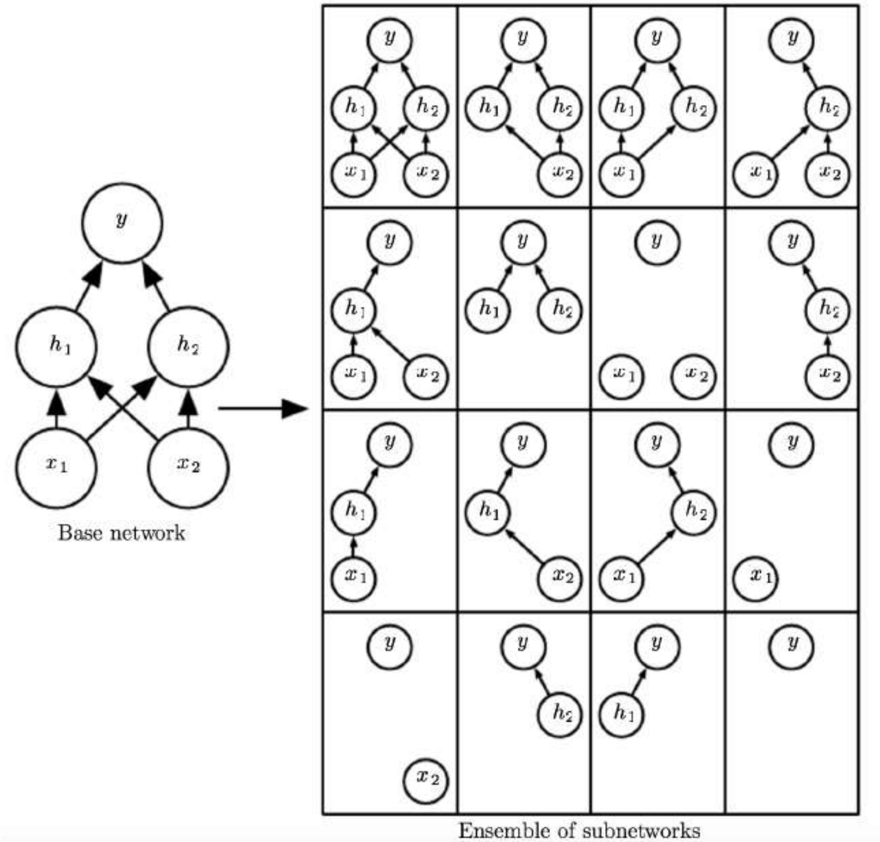


Forces the network to have a redundant representation;
Prevents co-adaptation of features



Understanding Dropout

- Dropout can be seen as training an ensemble of 2^D different architectures with shared weights (where D is the number of units):



— Goodfellow et al., *Deep Learning*

Dropout

- Dropout at test time

Output (label)

Input (image)

Dropout makes our output random! $\boxed{y} = f_W(\boxed{x}, \boxed{z})$

Random mask

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

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

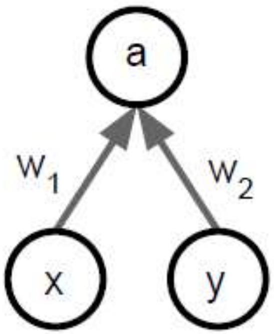
But this integral seems hard …

Dropout

- Dropout at 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.

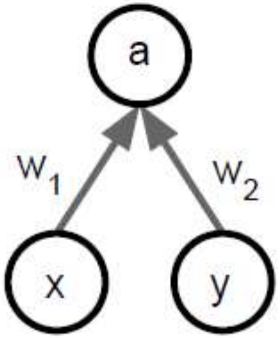
At test time we have: $E\big[a\big] = w_1x + w_2y$

Dropout

- Dropout at 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: $E\left[a\right] = w_1 x + w_2 y$

During training we have:

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

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

Dropout

- Dropout at test time

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

At test time all neurons are active always
=> We must scale the activations so that for each neuron:
<u>output at test time</u> = <u>expected output at training time</u>

Dropout

- Implementation: Inverted dropout

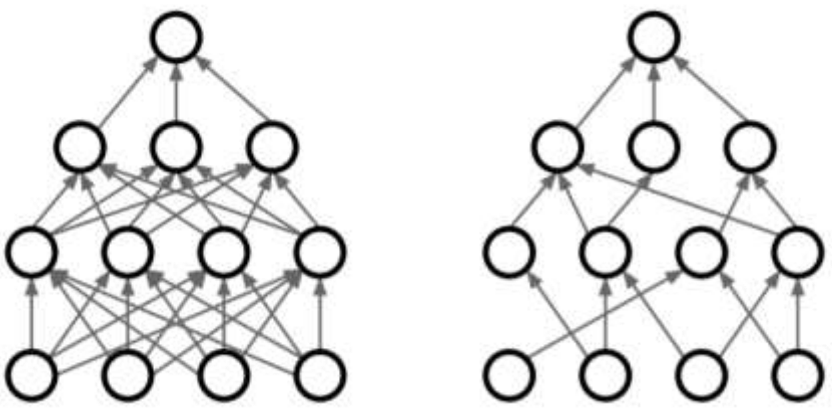
```
p = 0.5 # probability of keeping a unit active. higher = less dropout

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!

Stochastic Regularization

- Lots of other stochastic regularizers have been proposed:
  - DropConnect drops connections instead of activations.

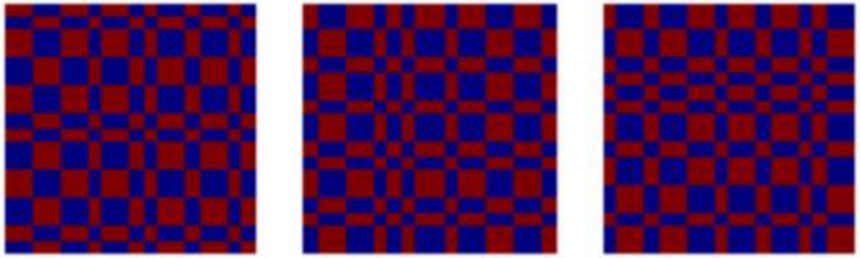
    - Training: Drop connections between neurons (set weights to 0)
    - Testing: Use all the connections



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

Stochastic Regularization

- Lots of other stochastic regularizers have been proposed:
  - Fractional Pooling
    - Training: Use randomized pooling regions
    - Testing: Average predictions from several regions

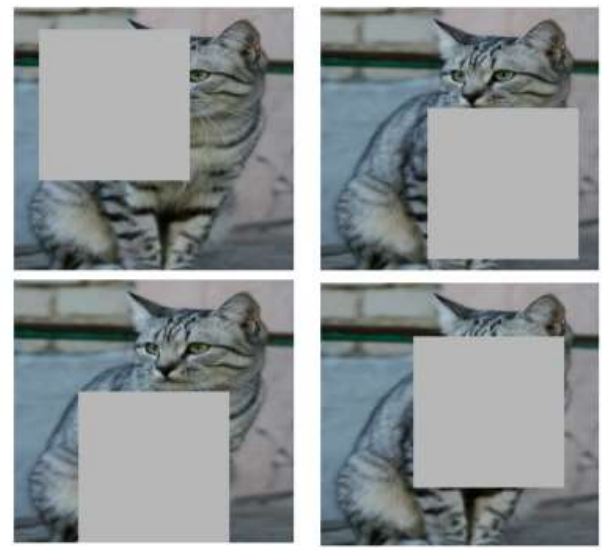


Graham, “Fractional Max Pooling”, arXiv 2014

Stochastic Regularization

- Lots of other stochastic regularizers have been proposed:
  - Cutout

    - Training: Set random image regions to zero
    - Testing: Use full image predictions from several regions



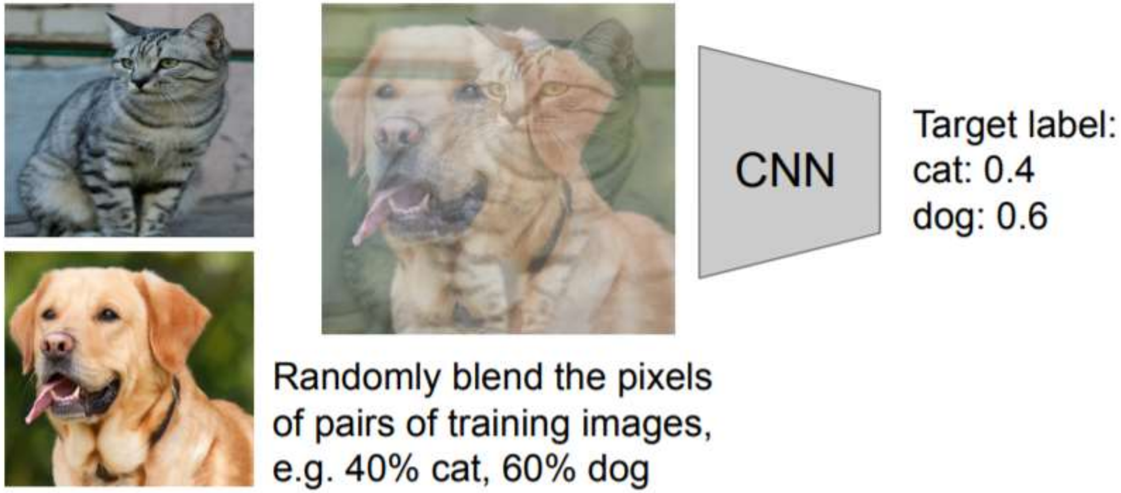
Works very well for small datasets like CIFAR, less common for large datasets like ImageNet

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

Stochastic Regularization

- Lots of other stochastic regularizers have been proposed:
  - Mixup

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



Zhang et al, “mixup: Beyond Empirical Risk Minimization”, ICLR 2018

Stochastic Regularization

- Lots of other stochastic regularizers have been proposed:
  - Training: Add random noise
  - Testing: Marginalize over the noise
- In practice
  - Consider dropout for large fully-connected layers
  - Batch normalization and data augmentation almost always a good idea
  - Try cutout and mixup especially for small classification datasets

Outline

- Regularization in CNN training
  - Data Augmentation
  - Weight Regularization & Transfer Learning
  - Stochastic Regularization
  - Hyper-parameter optimization

Acknowledgement: Feifei Li's cs231n notes

Hyperparameter optimization

- (Cross-)validation strategy

coarse -> fine cross-validation in stages

First stage: only a few epochs to get rough idea of what params work
Second stage: longer running time, finer search
… (repeat as necessary)

Tip for detecting explosions in the solver:
If the cost is ever > 3 * original cost, break out early

Hyperparameter optimization

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

Hyperparameter optimization

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

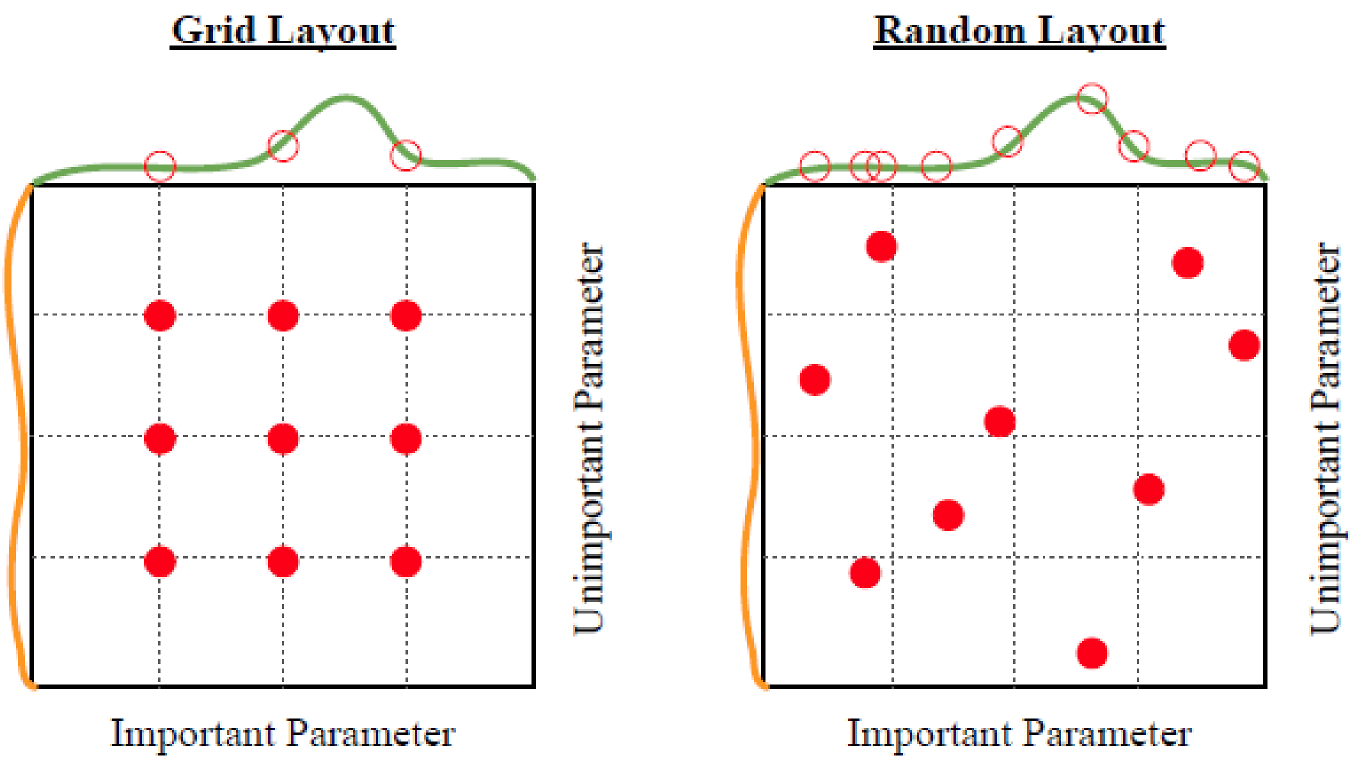
```
max_count = 100
for count in xrange(max_count):
      reg = 10**uniform(-4, 0)
      lr = 10**uniform(-3, -4)
```

```
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.

Hyperparameter optimization

- Random search vs. Grid search



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

Hyperparameter optimization

- **Hyperparameters to play with:**
  - network architecture
  - learning rate, its decay schedule, update type
  - regularization (L2/Dropout strength)

- Other hyperparameter optimization methods
  - Shahriari, et al. "Taking the human out of the loop: A review of Bayesian optimization." Proceedings of the IEEE 104.1 (2016): 148-175.

Summary

- CNN training as optimization task
  - Non-convex and local minimal
  - Overcoming ravines in loss surfaces
  - Data pre-processing + weight initialization + first-order update
  - Batch normalization
- Bag of tricks for improving generalization
  - Pros: you have a toolbox to use
  - Cons: many trial and error, tedious process
- Seeking fully automatic approaches to model selection
  - Bayesian optimization
  - Reinforcement learning