Lecture 05: CNNs II – Model Training and Optimization

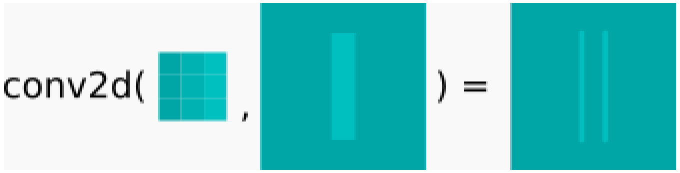
Lan Xu
SIST, ShanghaiTech
Fall, 2022

Summary of CNNs

- CNN properties [Bronstein et al., 2018]
  - Convolutional (Translation invariance)
  - Scale Separation (Compositionality)
  - Filters localized in space (Deformation Stability)
  - O(1) parameters per filter (independent of input image size n)
  - O(n) complexity per layer (filtering done in the spatial domain)
  - O(log n) layers in classification tasks

Math Properties of CNNs

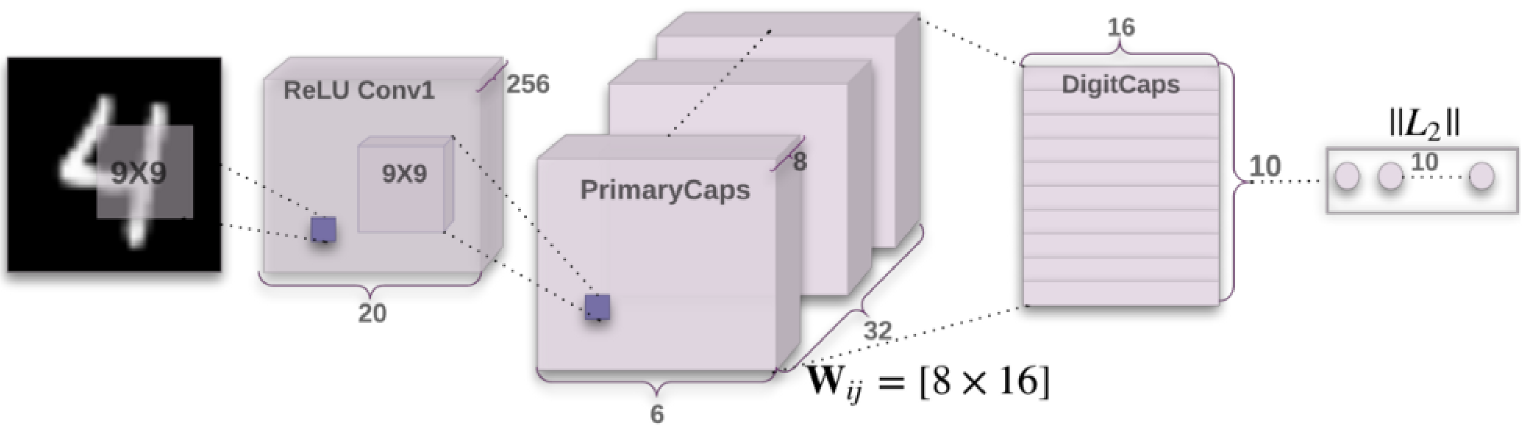
- Recent results on convolution layers
  - Convolutions are equivariant to translation
  - Convolutions are not equivariant to other isometries of the sampling lattice, e.g., rotation



  - What if a CNN learns rotated copies of the same filter?
    - The stack of feature maps is equivariant to rotation.

Math Properties of CNNs

- Recent results on convolution layers
  - Ordinary CNNs can be generalized to Group Equivariant Networks (Cohen and Welling ICML'16, Kondor and Trivedi ICML'18)
    - Redefining the convolution and pooling operations
    - Equivariant to more general transformation from some group *G*
  - Replacing pooling by other network designs
    - Capsule network (Sabour et al, 2017) https://arxiv.org/abs/1710.09829



Outline

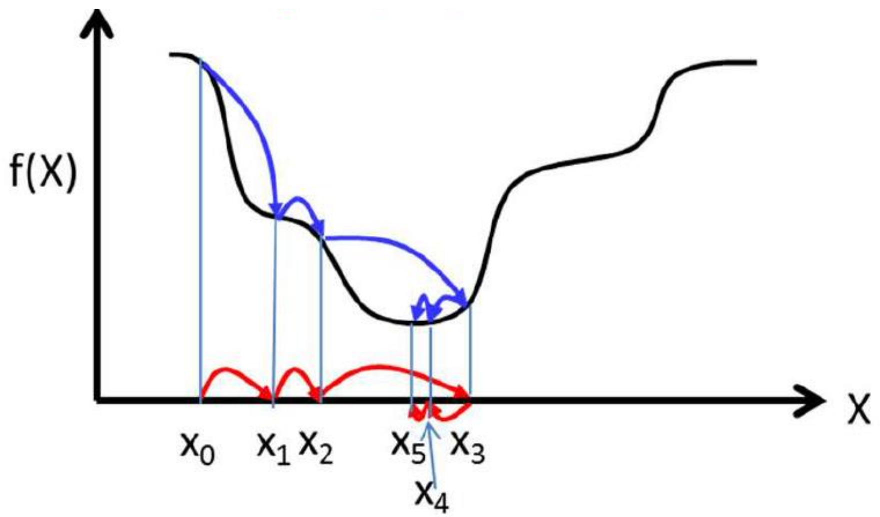
- Overview of CNN training
- CNN training as optimization
  - Data preprocessing
  - Weight initialization
  - Parameter update
  - Batch normalization (maybe next time)

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
- Avoid overfitting
  - Next time

Outline

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

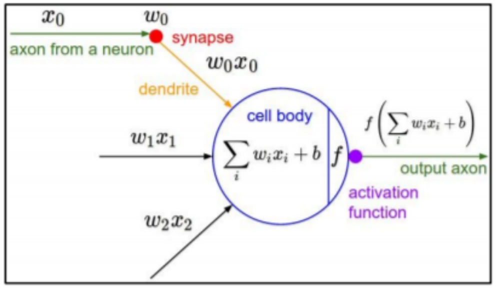
Acknowledgement: UofT, CMU & Feifei Li's cs231n notes

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!

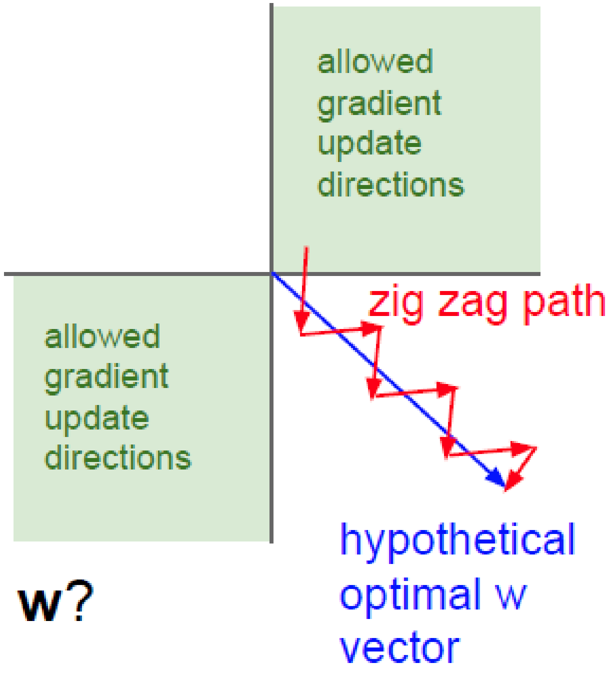
$$\boxed{\frac{\partial L}{\partial w}} = \boxed{\sigma(\textstyle\sum_i w_i x_i + b)(1-\sigma(\sum_i w_i x_i + b))}\boxed{x} \times \boxed{upstream\_gradient}$$

Data Preprocessing

- Motivation

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

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



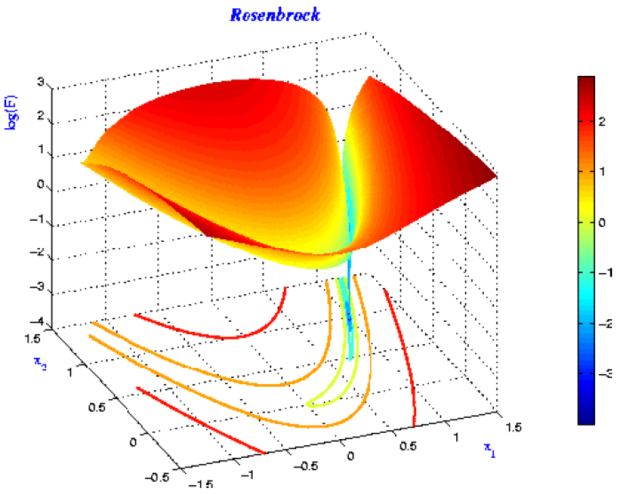
What can we say about the gradients on **w**?

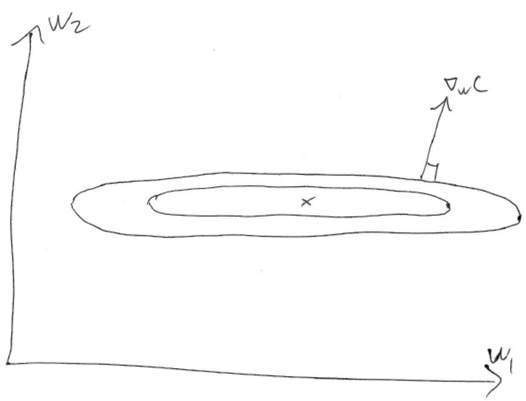
Always all positive or all negative :(

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

Data Preprocessing

- Motivation
  - Error surfaces with long, narrow ravines



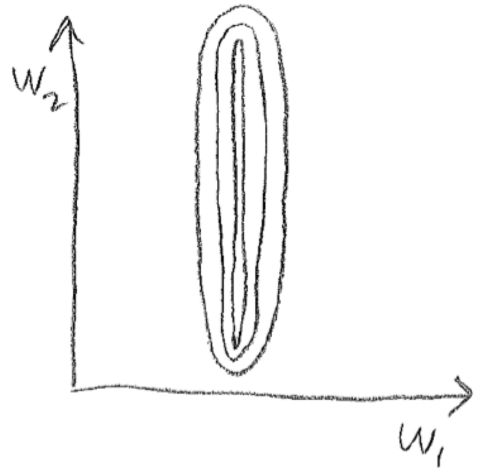


Data Preprocessing

- Motivation
  - Example of linear regression

x_1	x_2	t
114.8	0.00323	5.1
338.1	0.00183	3.2
98.8	0.00279	4.1
⋮	⋮	⋮

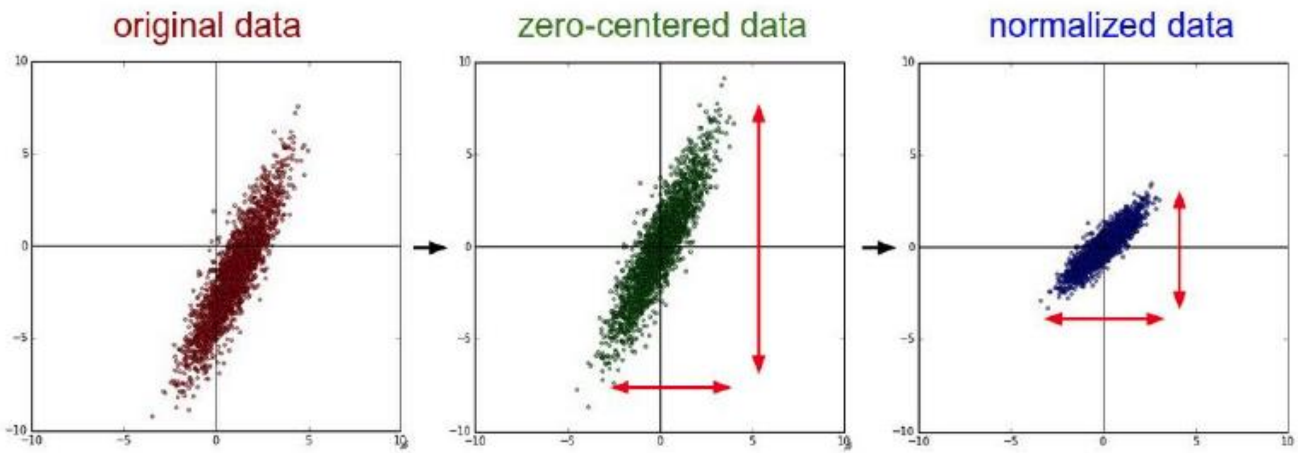
$$\overline{w_i} = \overline{y}\, x_i$$



  - Which direction of weights has a larger gradient updates?
  - Which one do you want to receive a larger update?

Data Preprocessing

- Data normalization
  - To avoid these problems, center your inputs to zero mean and unit variance




```
X -= np.mean(X, axis = 0)
```

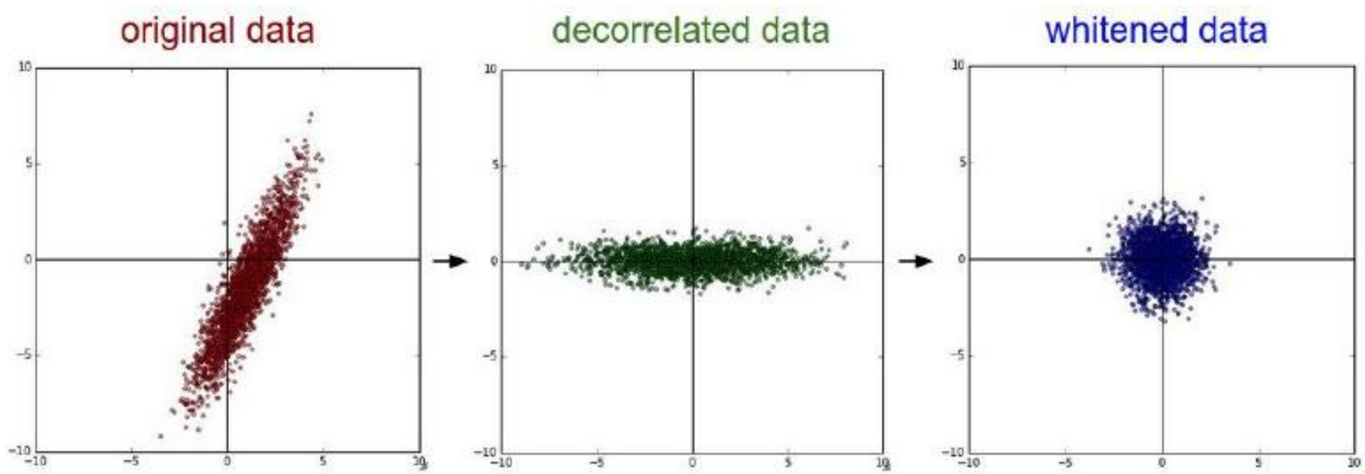
```
X /= np.std(X, axis = 0)
```

(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

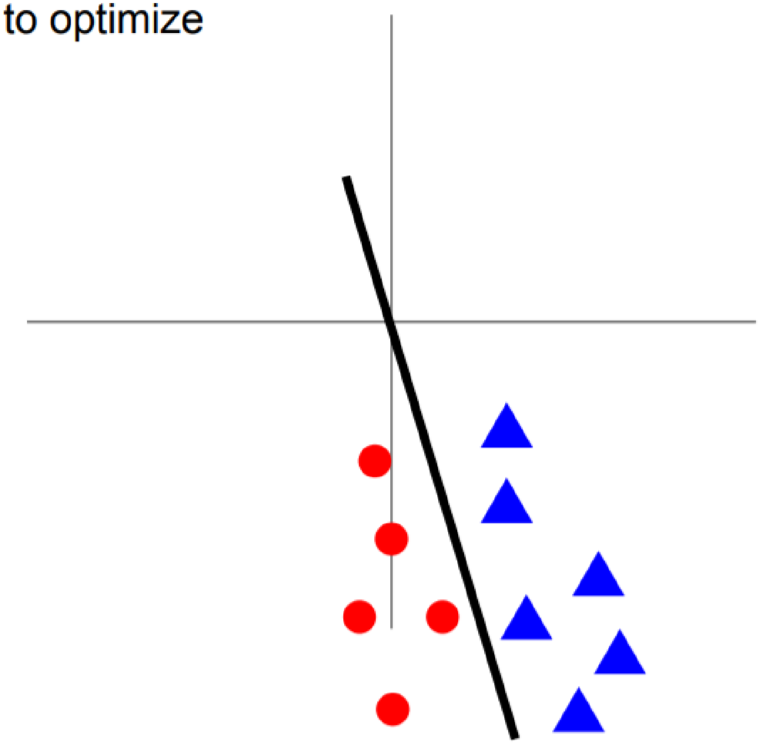


(data has diagonal covariance matrix)

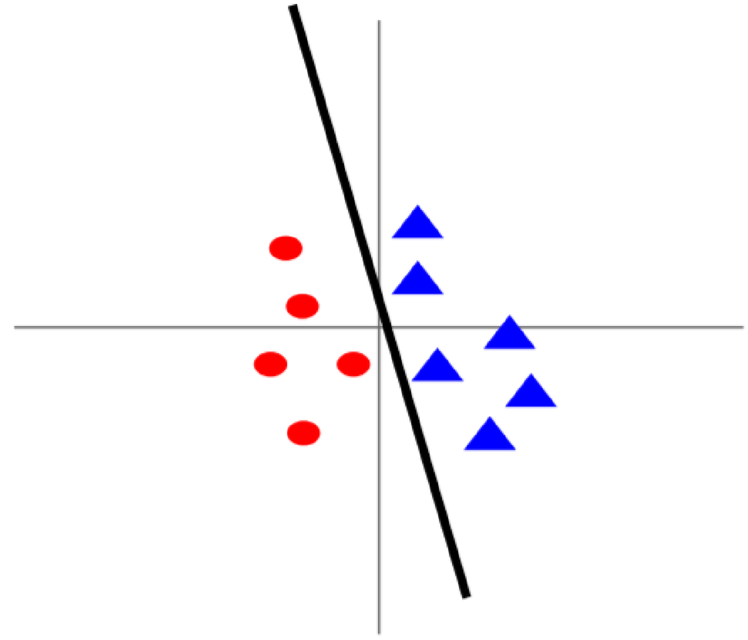
(covariance matrix is the identity matrix)

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)

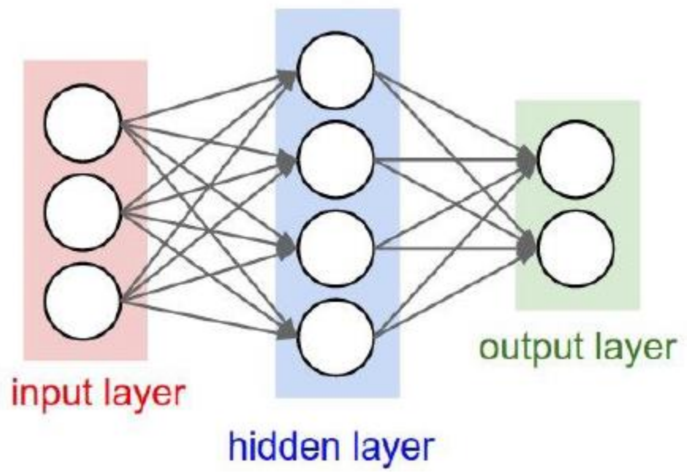
Outline

- 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

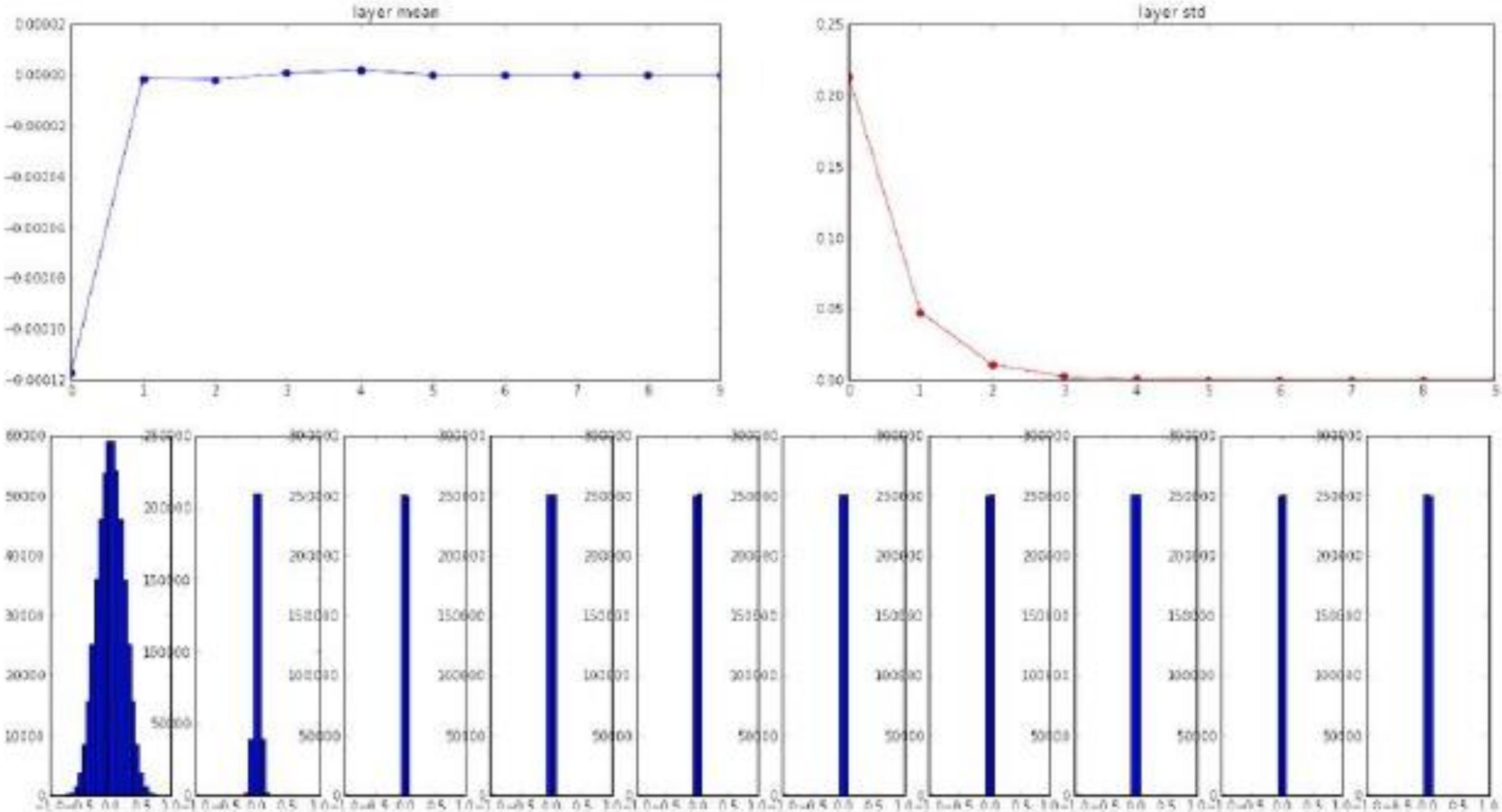
- Motivating example
  - Look at some activation statistics
  - E.g., 10-layer net with 500 neurons on each layer using tanh non-linearities.

```
# assume some unit gaussian 10-D input data
D = np.random.randn(1000, 500)
hidden_layer_sizes = [500]*10
nonlinearities = ['tanh']*len(hidden_layer_sizes)
```

```
act = {'relu':lambda x:np.maximum(0,x), 'tanh':lambda x:np.tanh(x)}
Hs = {}
for i in xrange(len(hidden_layer_sizes)):
    X = D if i == 0 else Hs[i-1] # input at this layer
    fan_in = X.shape[1]
    fan_out = hidden_layer_sizes[i]
    W = np.random.randn(fan_in, fan_out) * 0.01 # layer initialization

    H = np.dot(X, W) # matrix multiply
    H = act[nonlinearities[i]](H) # nonlinearity
    Hs[i] = H # cache result on this layer
```

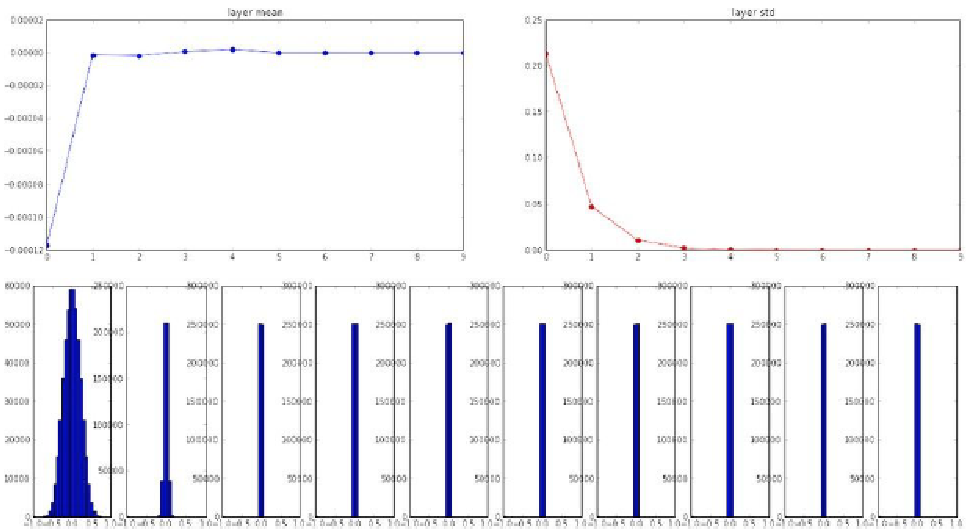
Weight Initialization

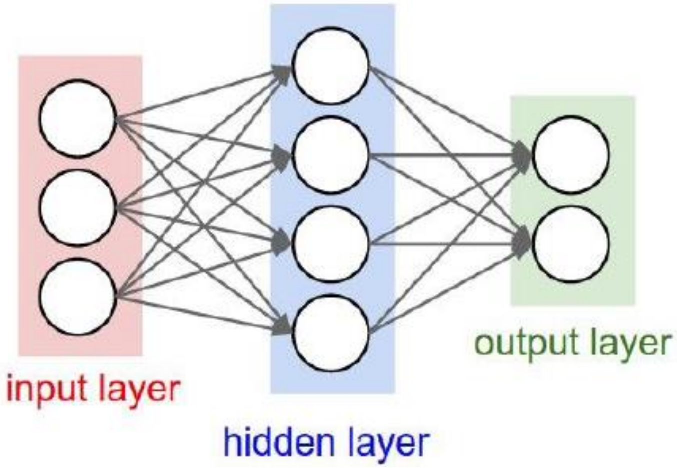
- Motivating example



Weight Initialization

- Motivating example
  - All activations tend to zero for deeper network layers
  - Q: What do the gradients dL/dW look like?





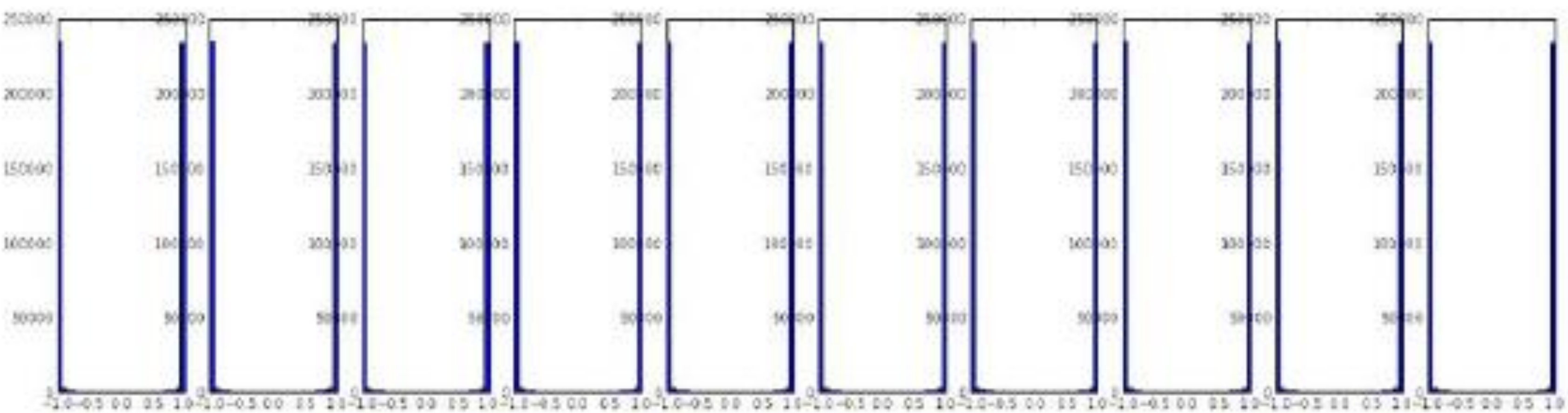
Weight Initialization

- Motivating example

```
W = np.random.randn(fan_in, fan_out) * 1.0 # layer initialization
```

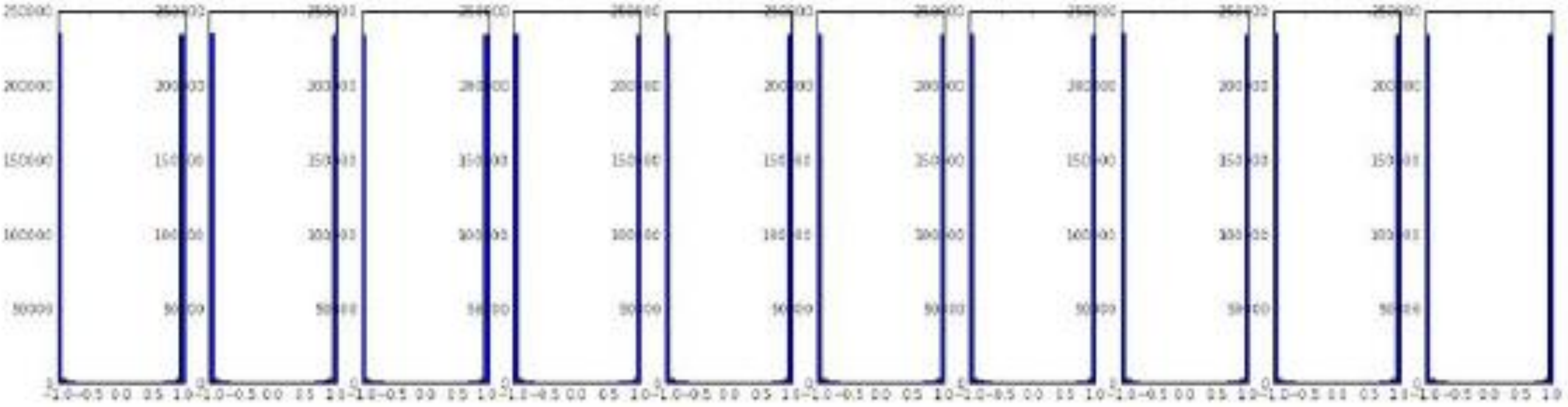
```
input layer had mean 0.001800 and std 1.001311
hidden layer 1 had mean -0.000430 and std 0.981879
hidden layer 2 had mean -0.000849 and std 0.981649
hidden layer 3 had mean 0.000566 and std 0.981601
hidden layer 4 had mean 0.000483 and std 0.981755
hidden layer 5 had mean -0.000682 and std 0.981614
hidden layer 6 had mean -0.000401 and std 0.981560
hidden layer 7 had mean -0.000237 and std 0.981520
hidden layer 8 had mean -0.000448 and std 0.981913
hidden layer 9 had mean -0.000899 and std 0.981728
hidden layer 10 had mean 0.000584 and std 0.981736
```

*1.0 instead of *0.01



Weight Initialization

- Motivating example
  - All activations saturate
  - Q: What do the gradients look like?
  - A: Local gradients all zero

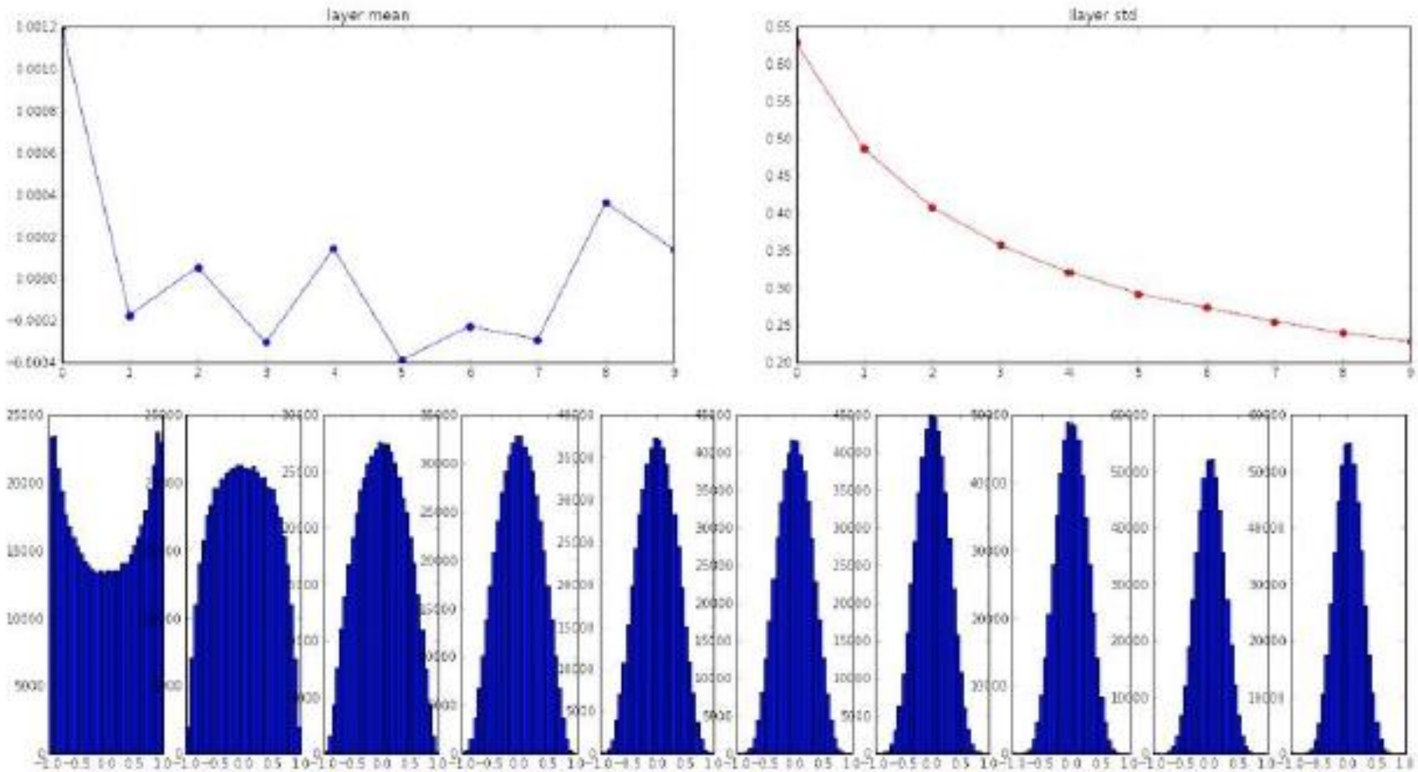


Weight Initialization

- Xavier initialization [Glorot and Bengio, AISTAT 2010]

```
W = np.random.randn(fan_in, fan_out) / np.sqrt(fan_in) # layer initialization
```

  - std = 1/sqrt(fan_in): activations are nicely scaled for all layers

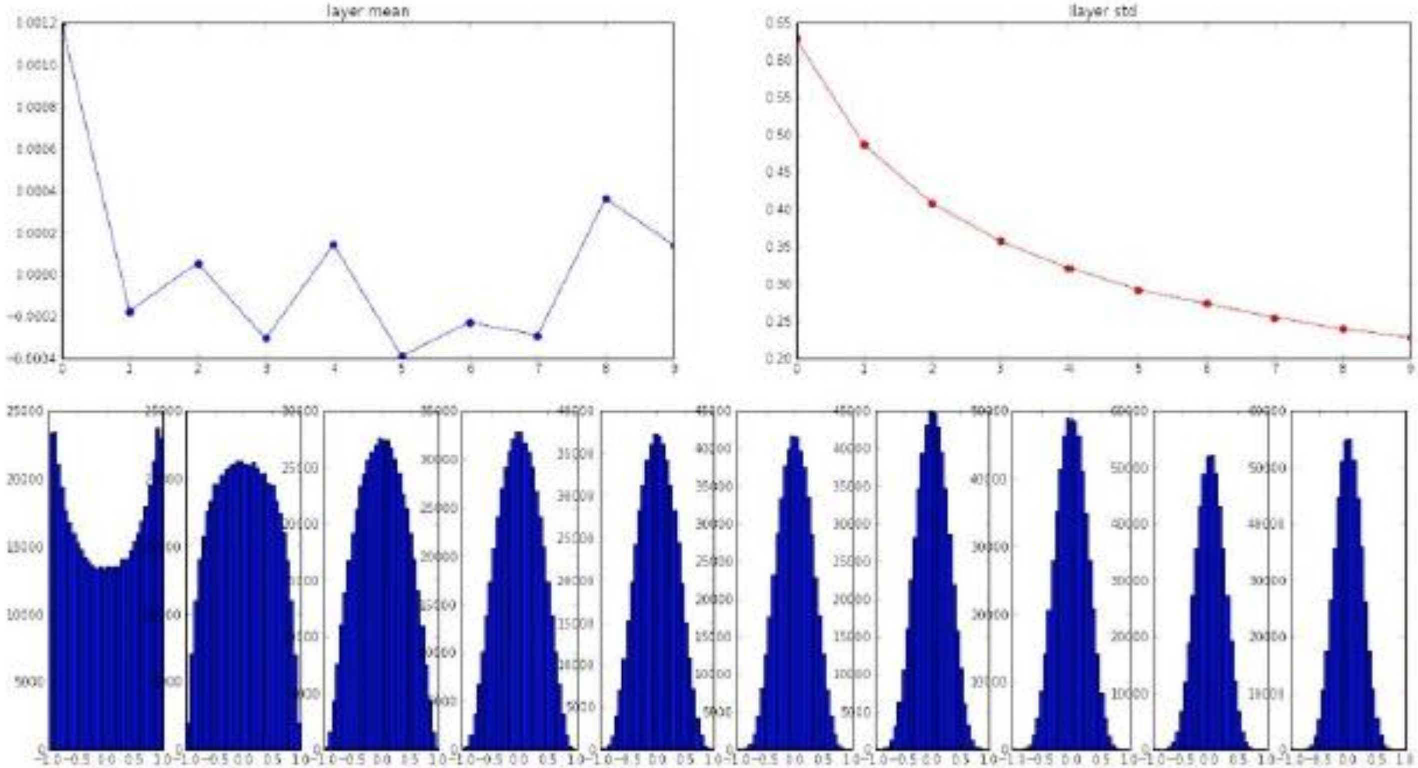


Weight Initialization

- Xavier initialization [Glorot and Bengio, AISTAT 2010]

```
W = np.random.randn(fan_in, fan_out) / np.sqrt(fan_in) # layer initialization
```

  - For conv layers, fan_in is filter_size^2 * input_channels



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

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

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).

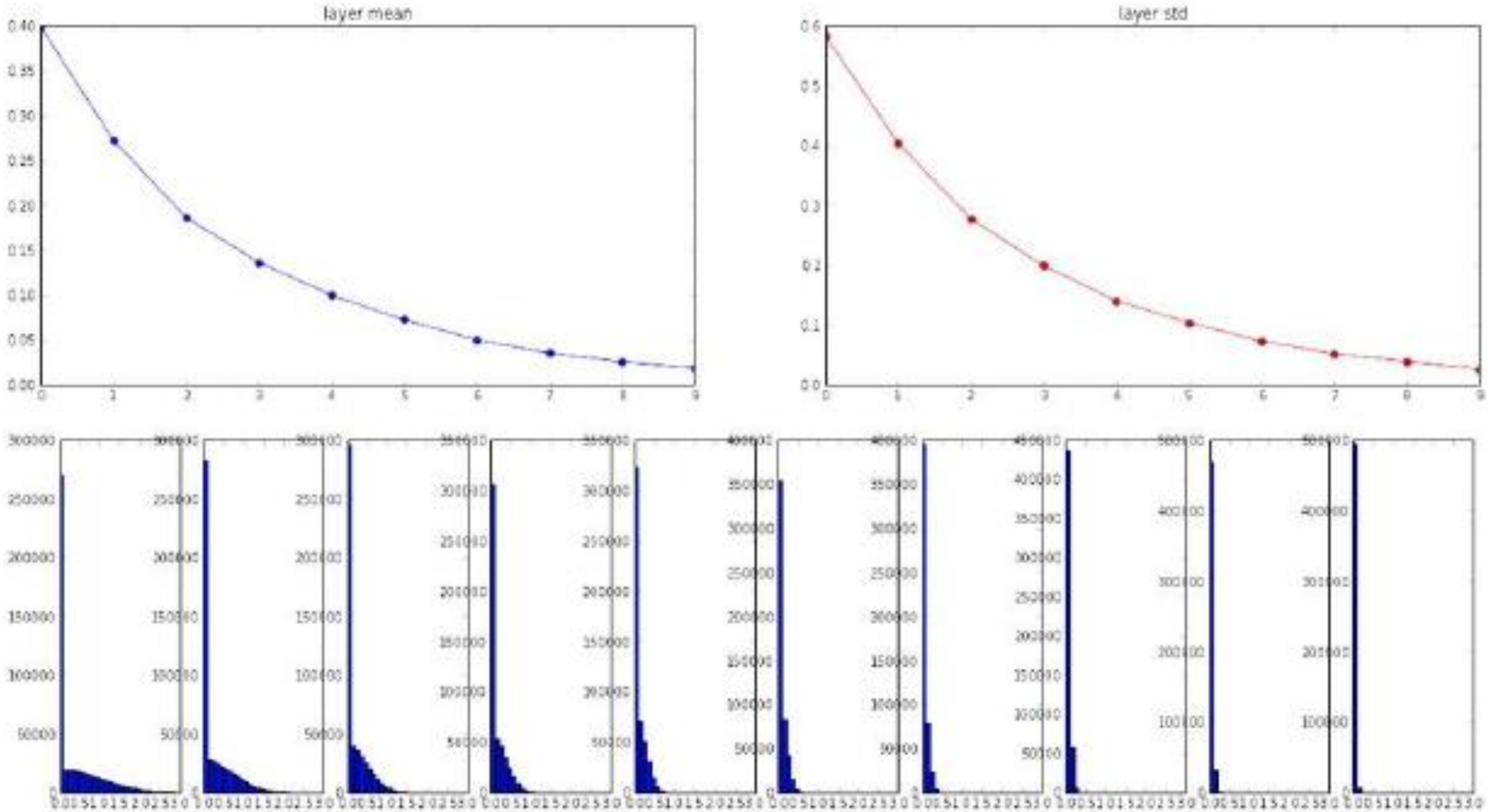
$$Var(Y) = Var(W_1X_1 + W_2X_2 + \ldots + W_nX_n) = nVar(W_i)Var(X_i)$$

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

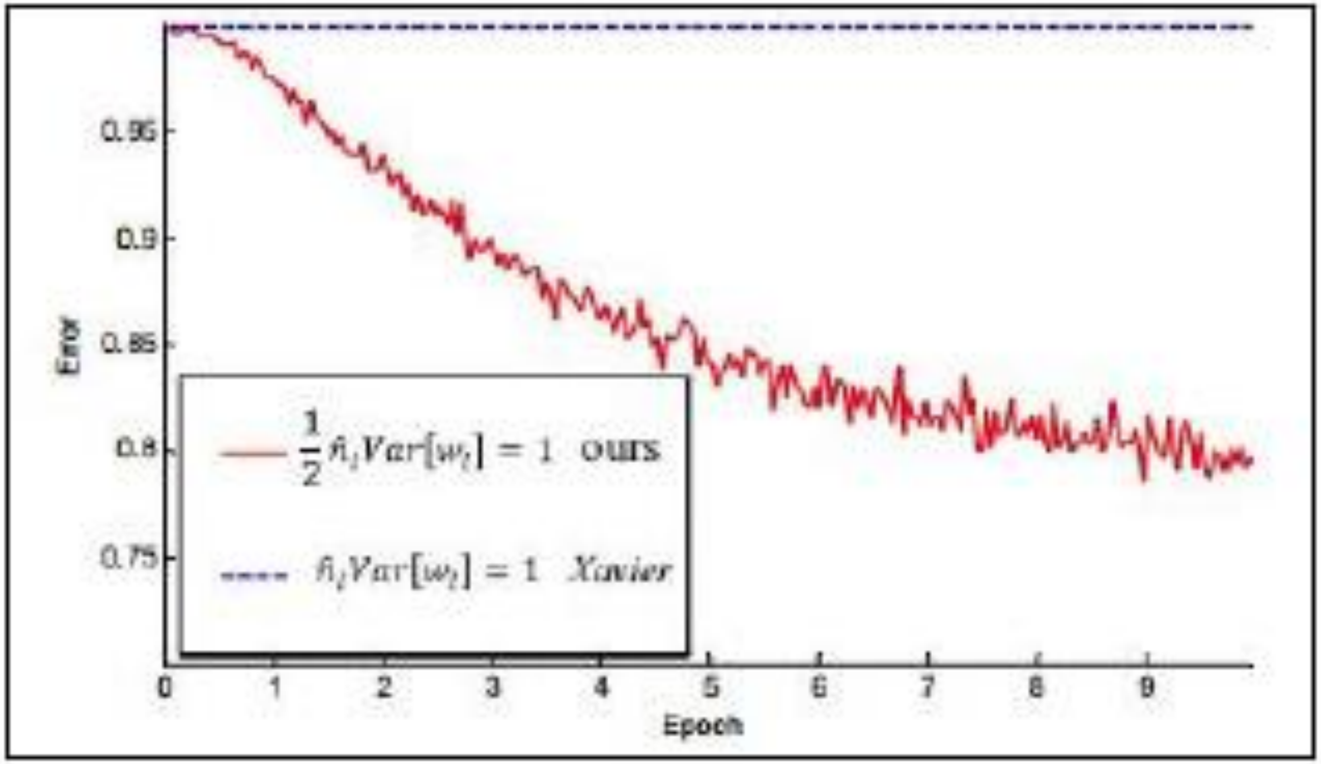
- Problems with ReLU activation
  - Xavier initialization assumes zero centered activation function, and hence breaks under ReLU



Weight Initialization

- Initialization for CNNs with ReLU [He et al., 2015]

```
W = np.random.randn(fan_in, fan_out) / np.sqrt(fan_in/2) # layer initialization
```



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

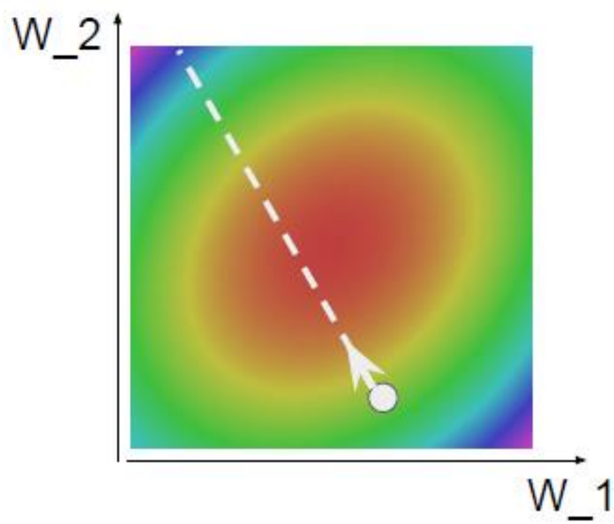
- Overview of CNN training
- CNN training as optimization
  - Data preprocessing
  - Weight initialization
  - Parameter update
  - Batch normalization

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

Optimization

- Stochastic Gradient Descent

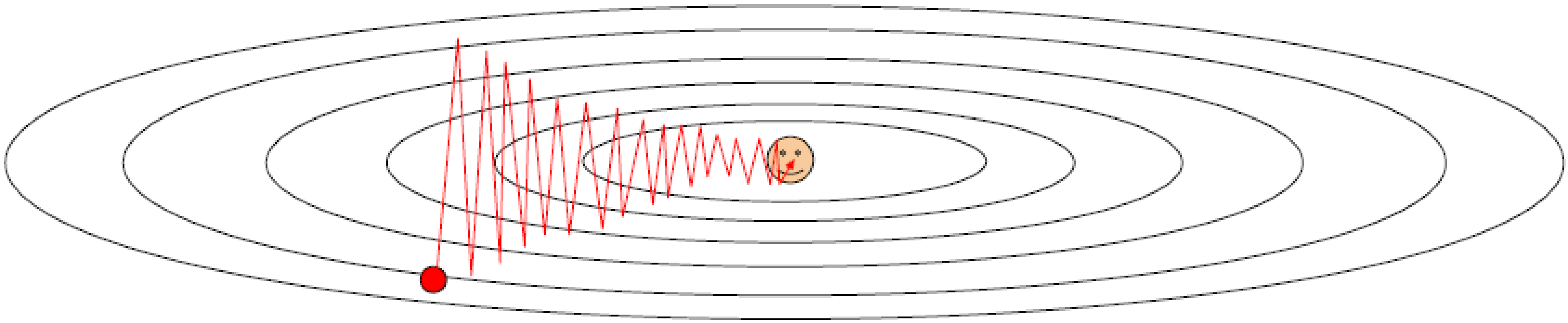
```
# Vanilla Gradient Descent

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



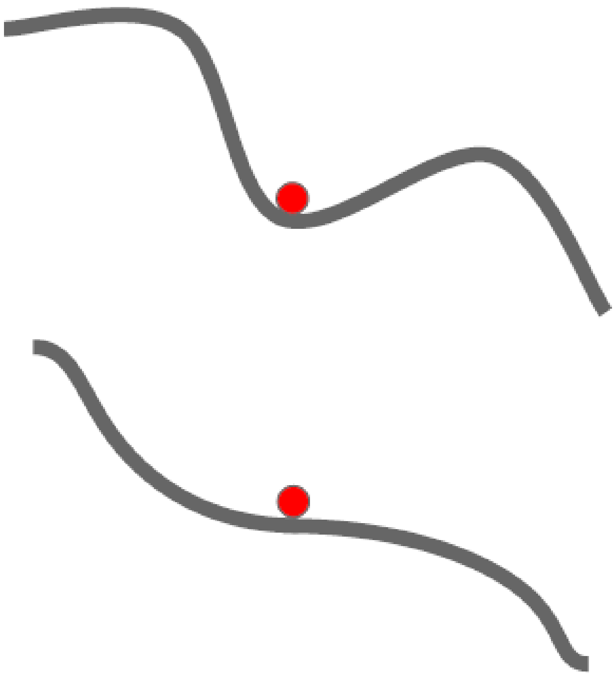
Loss function has high **condition number**: ratio of largest to smallest singular value of the Hessian matrix is large

Optimization

- Problems with SGD

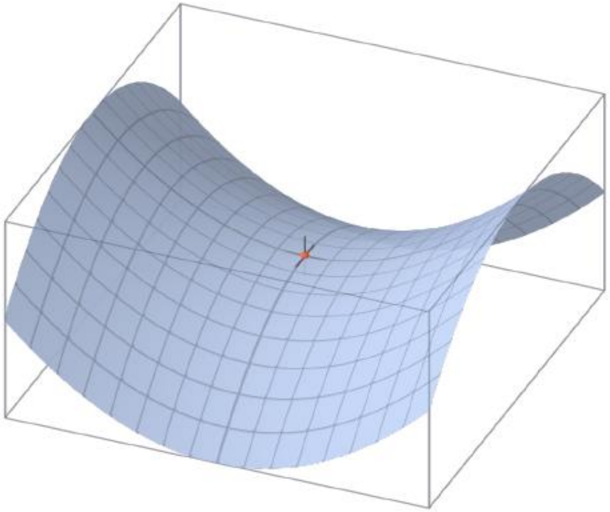
What if the loss function has a **local minima** or **saddle point**?

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.

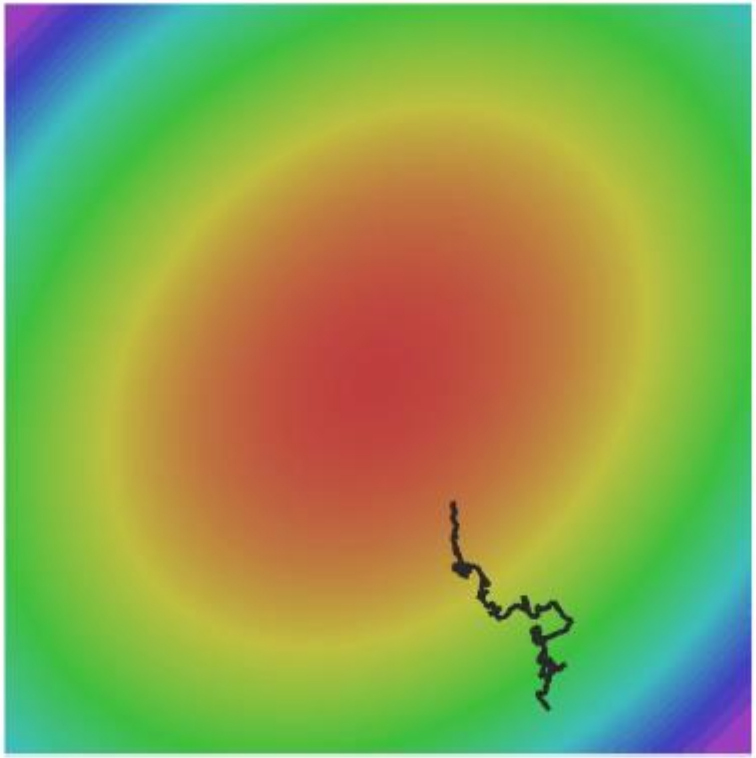
Optimization

- 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

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

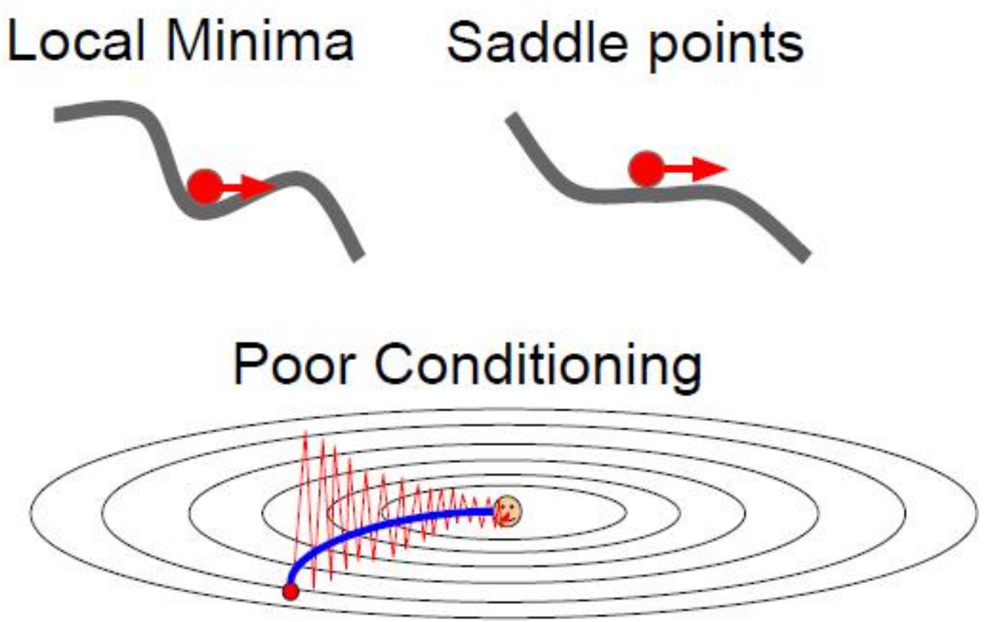
```
vx = 0
while True:
  dx = compute_gradient(x)
  vx = rho * vx + dx
  x += learning_rate * vx
```

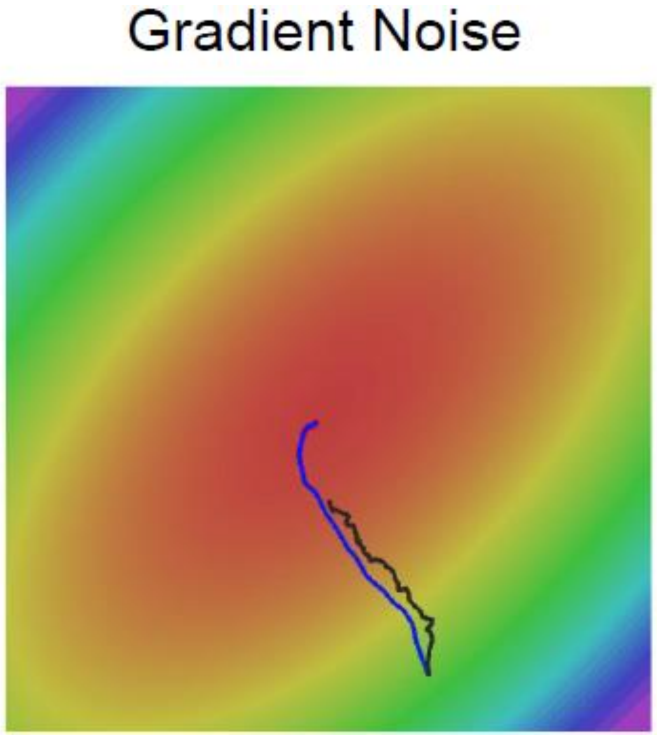
- Build up "velocity" as a running mean of gradients
- Rho gives "friction"; typically rho=0.9 or 0.99

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

Optimization

- SGD + Momentum
  - Momentum sometimes helps a lot, and almost never hurts





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
```

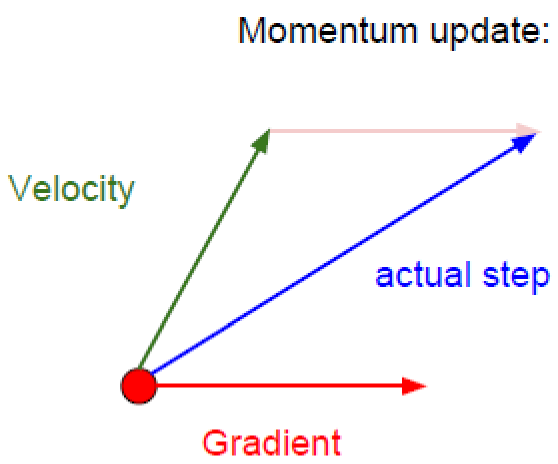
SGD+Momentum

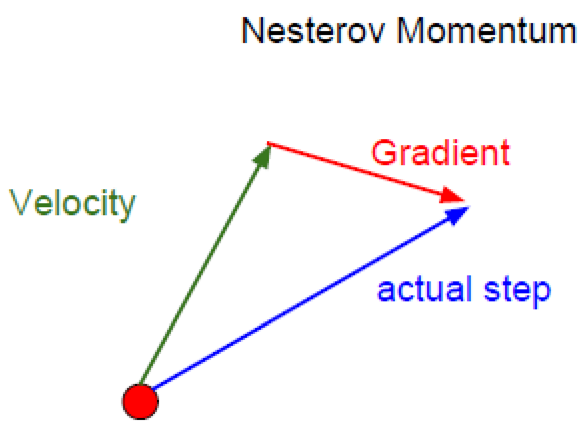
$$v_{t+1} = \rho v_t + \nabla f(x_t)$$
$$x_{t+1} = x_t - \alpha v_{t+1}$$

```
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





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

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 $x_t, \nabla f(x_t)$

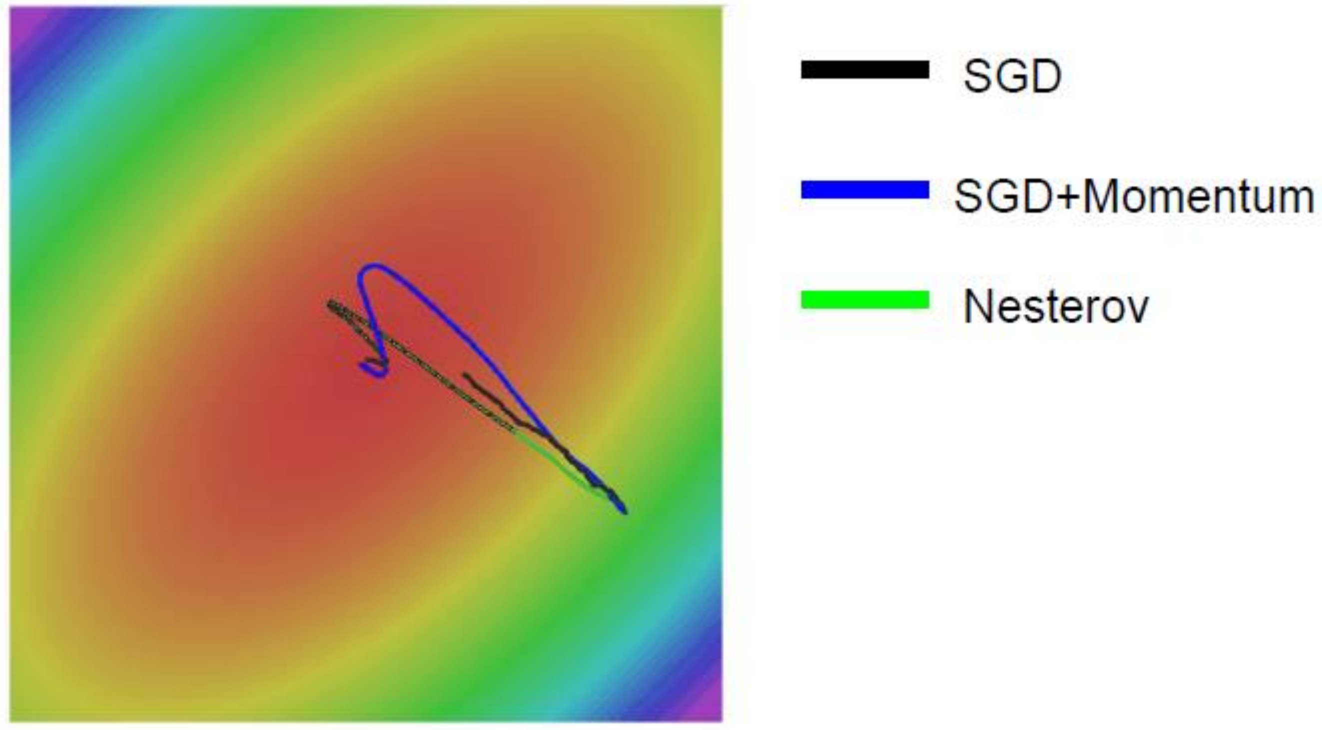
Change of variables $\tilde{x}_t = x_t + \rho v_t$ and rearrange:

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

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

Optimization

- Nesterov Momentum



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

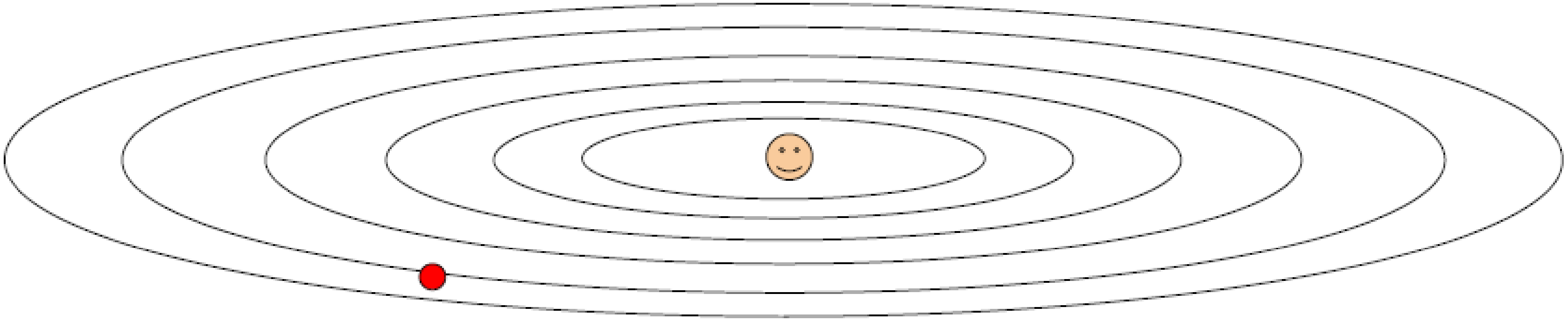
"Per-parameter learning rates" or "adaptive learning rates"

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)
```



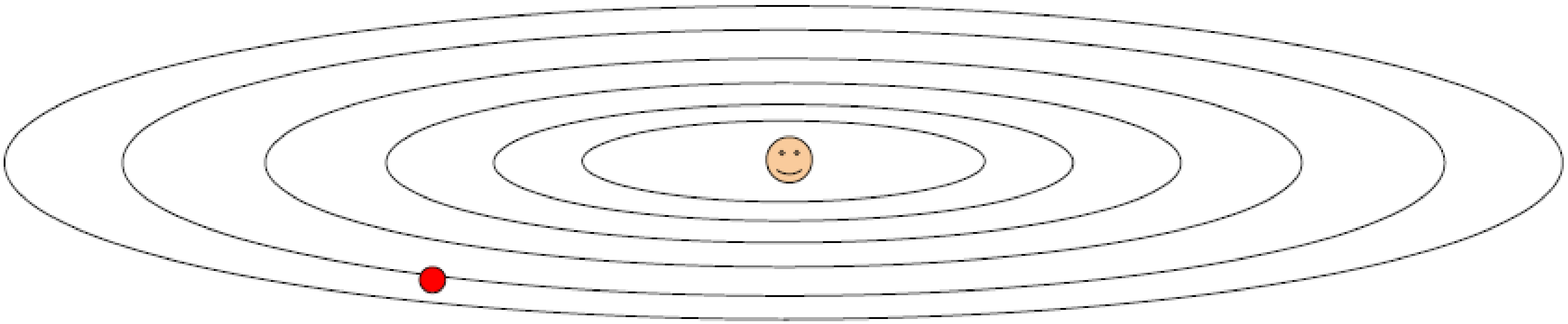
Q: What happens with AdaGrad?

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)
```

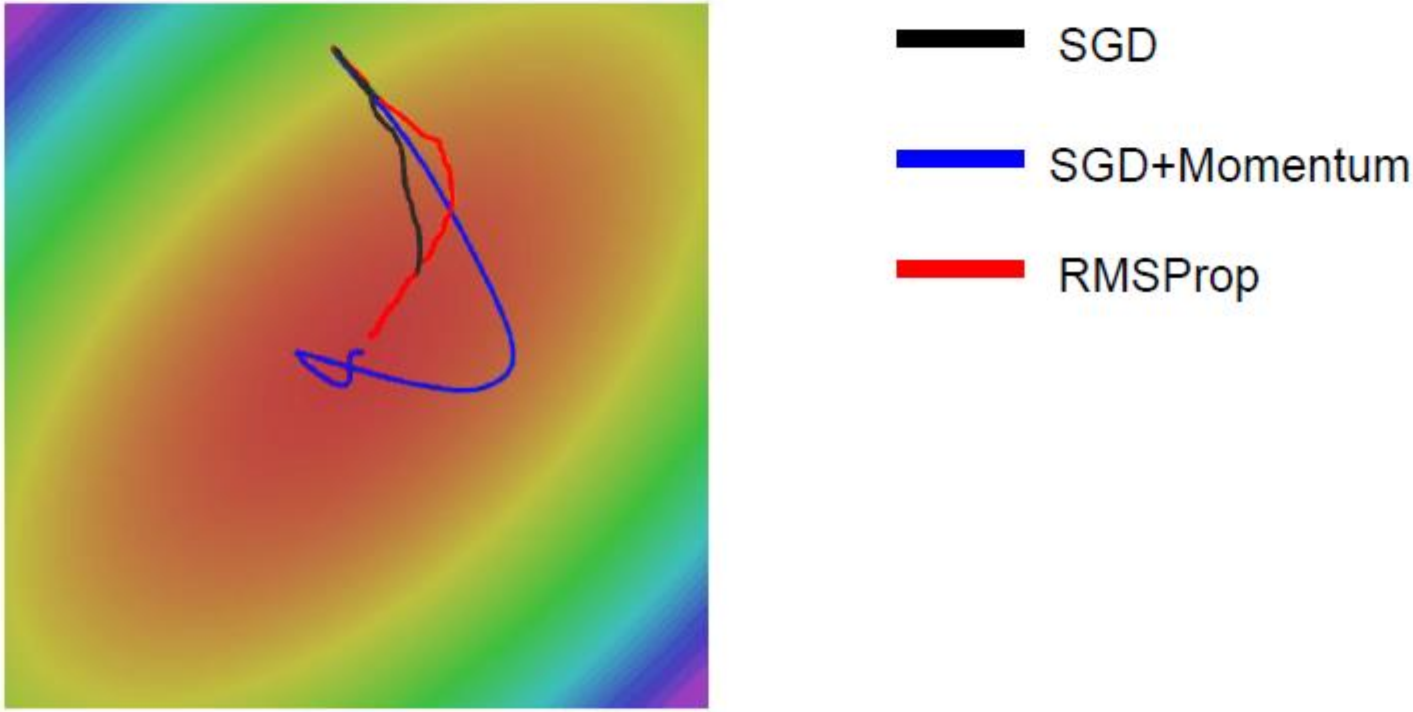
RMSProp

```
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

- RMSProp



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

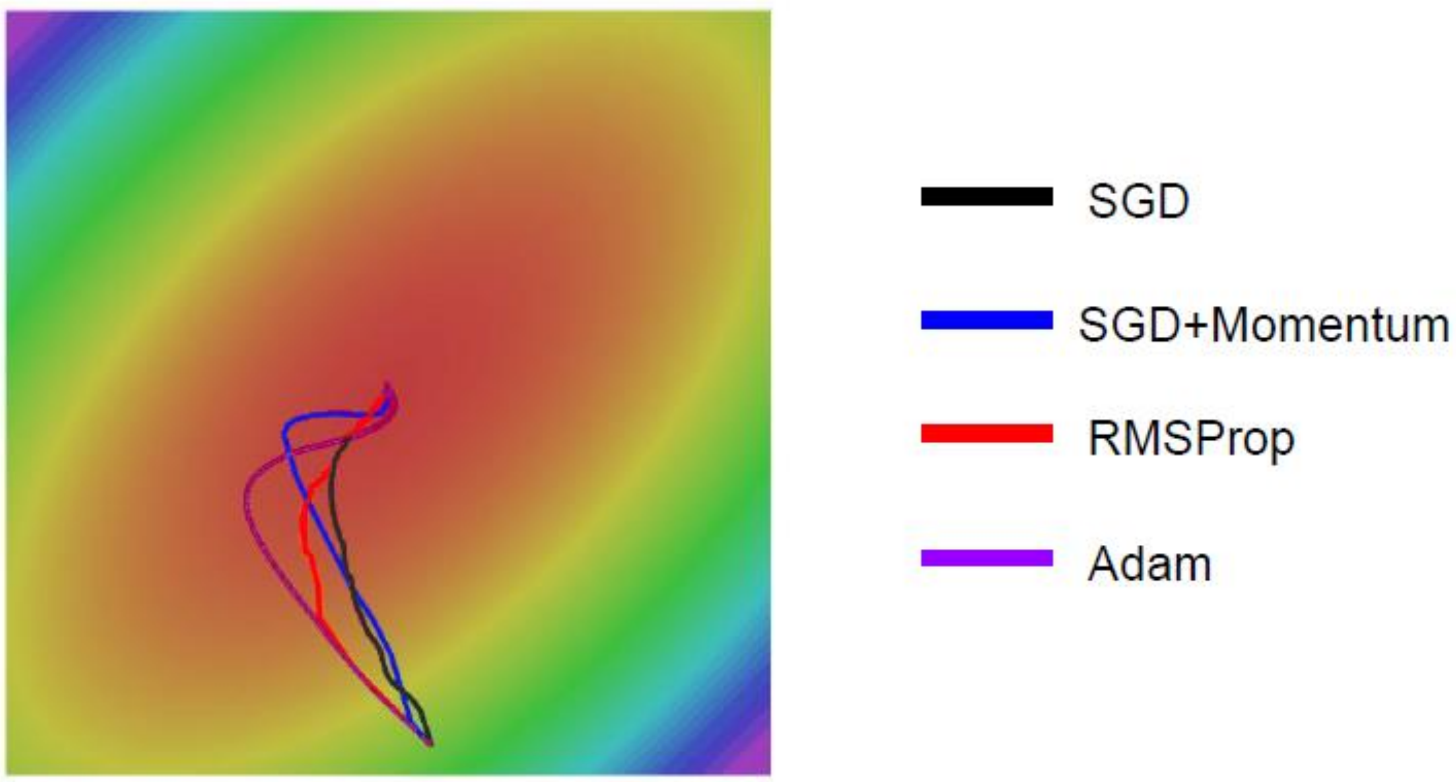
AdaGrad / RMSProp

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!

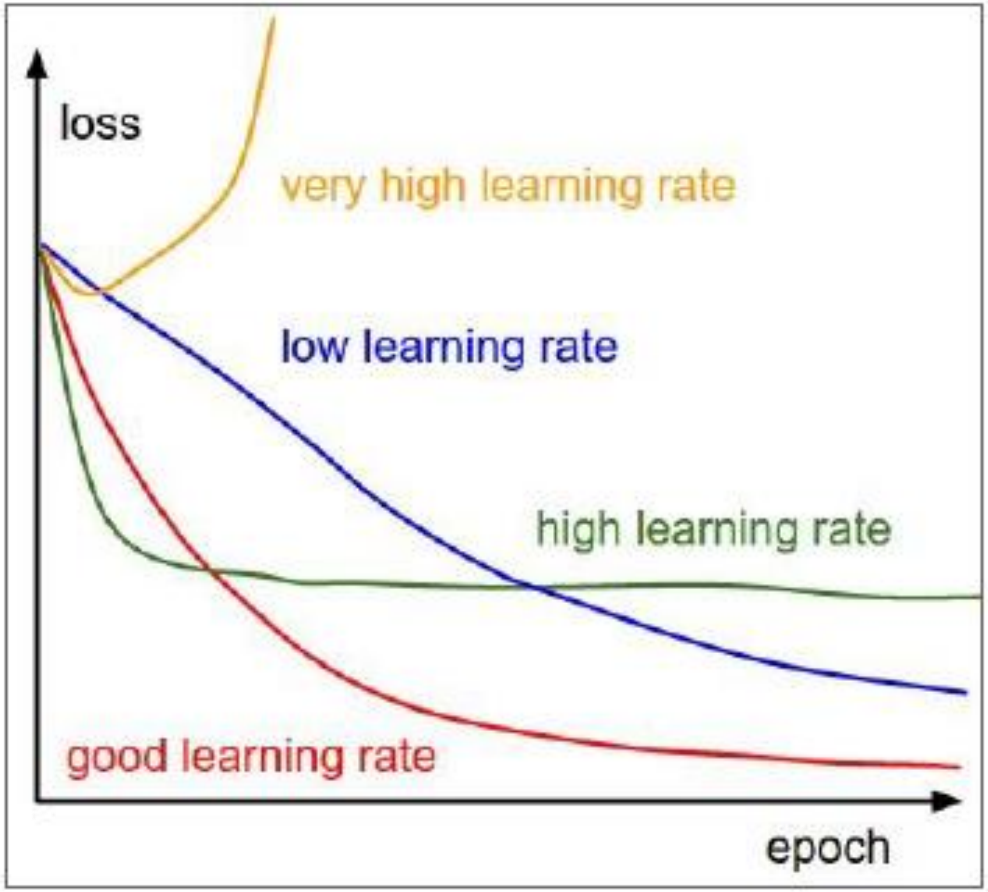
Optimization

- Adam (full form)



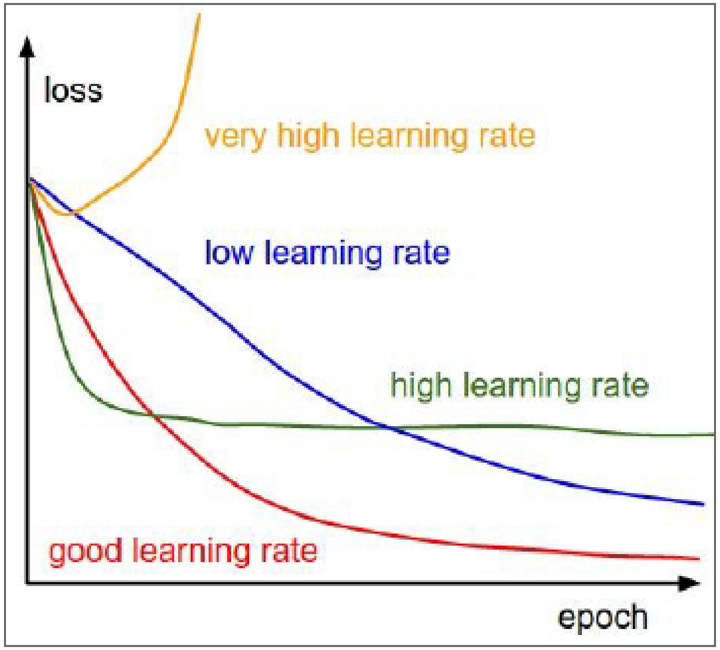
Learning rate

- 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:

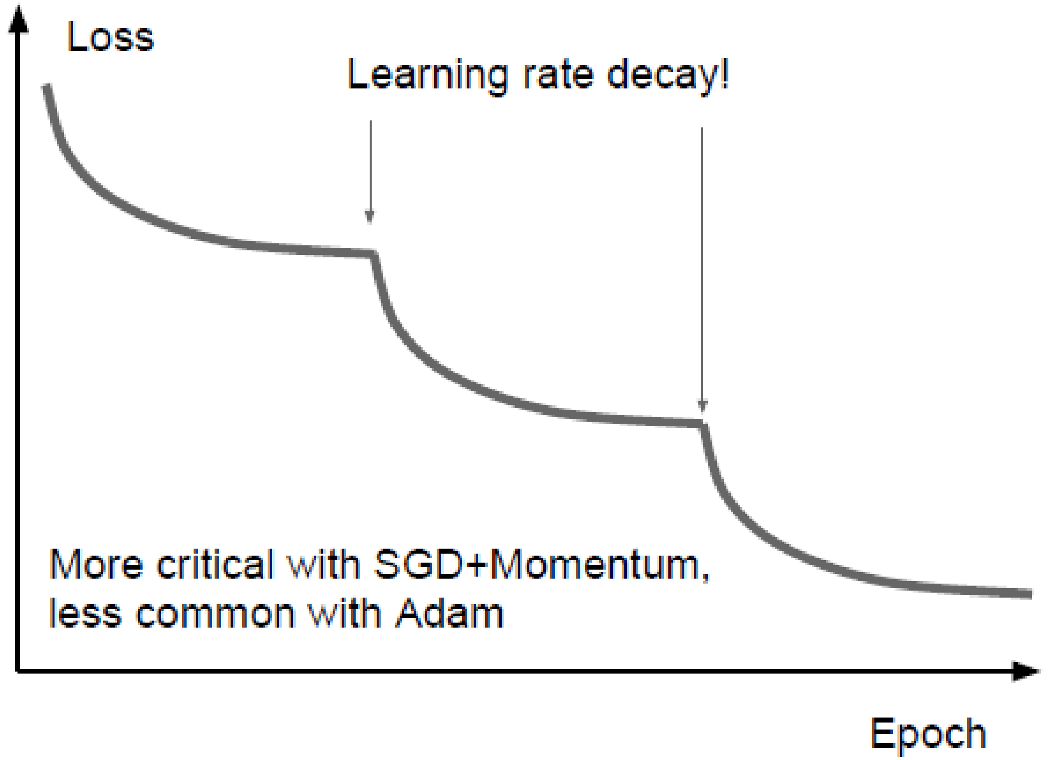
$$\alpha = \alpha_0 e^{-kt}$$

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

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

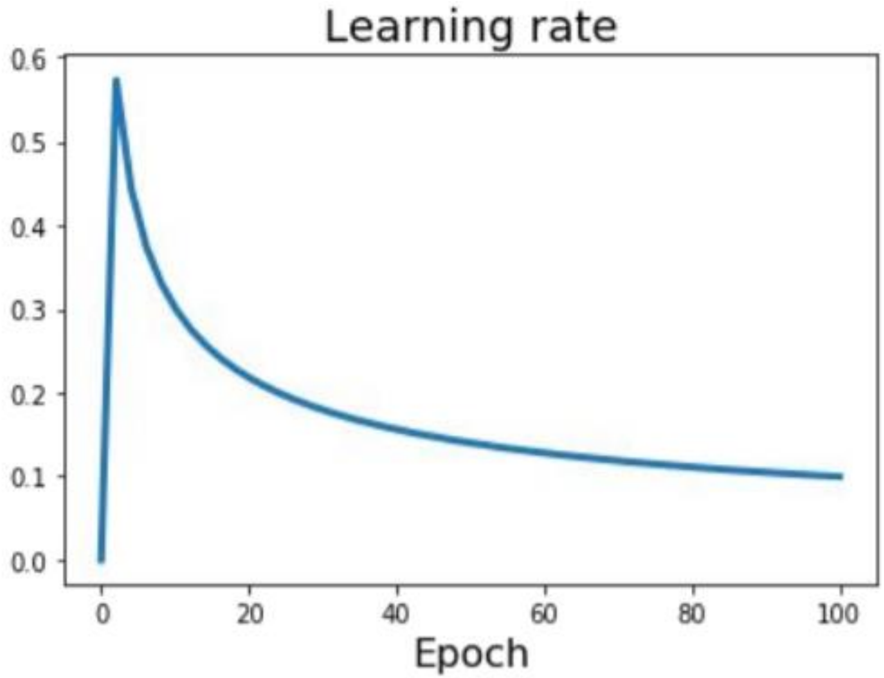
- Inverse sqrt

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

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

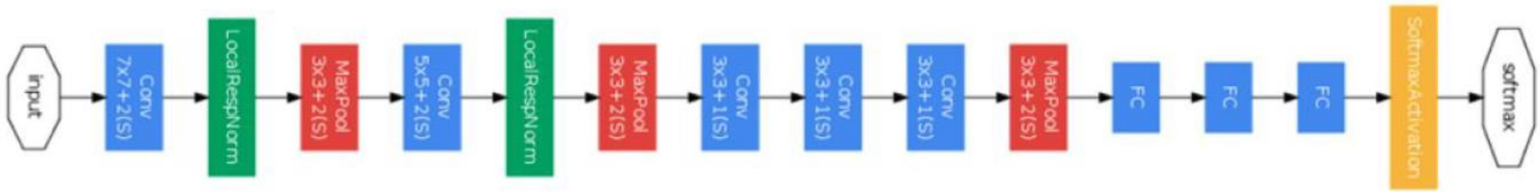
Outline

- 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:

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

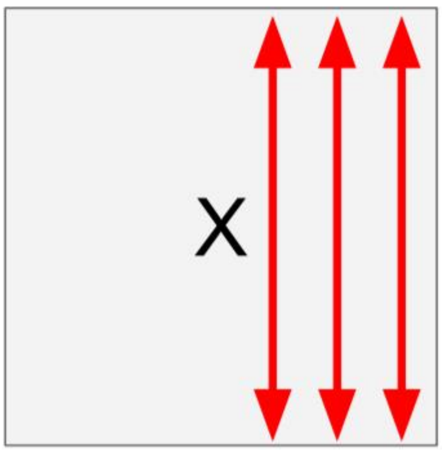
this is a vanilla differentiable function...

Batch Normalization

- Layer details

Input: $x : N \times D$

N



D

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

Per-channel mean, shape is D

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

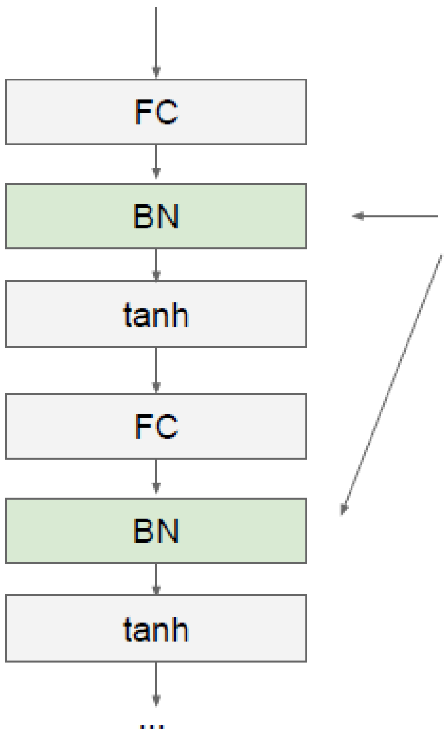
Per-channel var, shape is D

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

Normalized x, Shape is N x D

Batch Normalization

- 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

- 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

- Algorithm

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

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

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

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

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

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

- 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

- 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

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

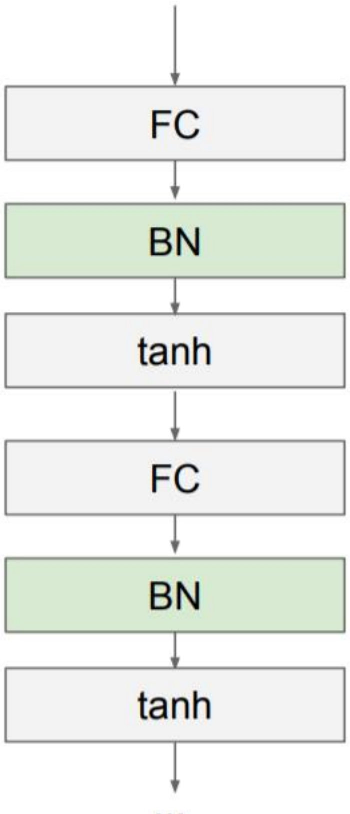
Normalized x, Shape is N x D

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

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!

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
- For Quiz-2 online: wangchy8@shanghaitech.edu.cn
- Next time …
  - Regularization to avoid overfitting