Deep Learning (for Computer Vision)

Arjun Jain

Last Week

- Wrapped up robust methods: RANSAC complexity, Lmeds
- Data driven paradigm, CIFAR-10 dataset, choosing hyper-parameters
- KNN, choice of loss function, cross-validation

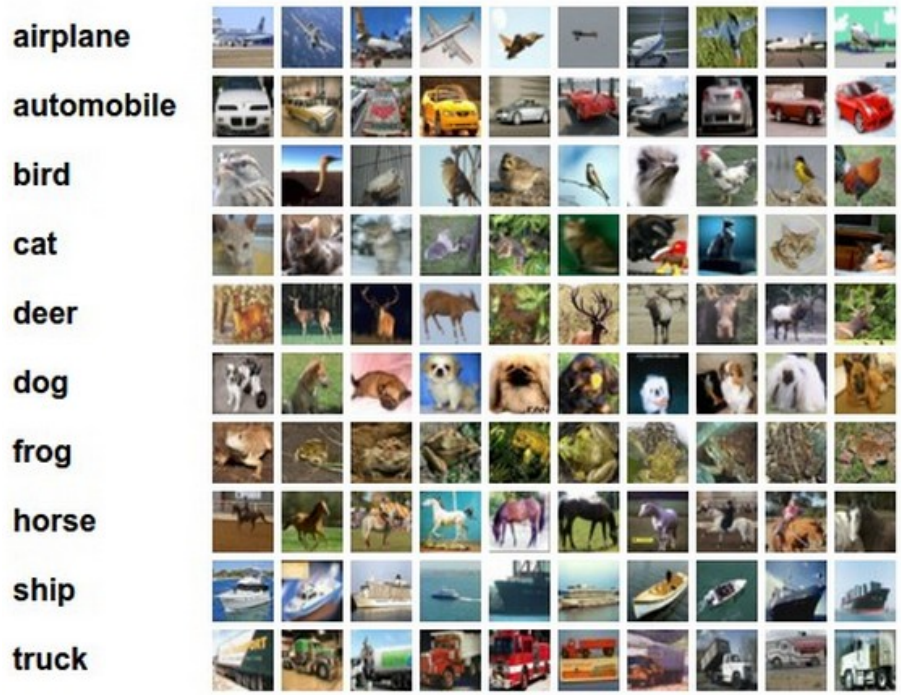
Sources

A lot of the material has been shamelessly and gratefully collected from:

- **http://cs231n.stanford.edu/**
- https://devblogs.nvidia.com/parallelforall/deep-learning-nutshell-history-training/
- https://adeshpande3.github.io/adeshpande3.github.io/The-9-Deep-Learning-Papers-You-Need-To-Know-About.html
- https://research.fb.com/learning-to-segment/
- https://research.fb.com/deep-learning-tutorial-at-cvpr-2014/
- https://www.cs.ox.ac.uk/people/nando.defreitas/machinelearning/practicals/practical4.pdf
- http://torch.ch/docs/developer-docs.html
- https://github.com/torch/nn/blob/31d7d2bc86a914e2a9e6b3874c497c60517dc853/doc/module.md
- https://web.stanford.edu/group/pdplab/pdphandbook/handbookch6.html
- http://neuralnetworksanddeeplearning.com/chap2.html

Parametric Approach: CIFAR-10

10 labels
50,000 training images
10,000 test images
each image is an array of size **32 x 32 x 3 = 3072** numbers total



Parametric Approach

[32x32x3]
array of numbers 0...1
(3072 numbers total)

Image

$f(x, W)$

10 numbers, indicating class scores

parameters, or "weights"

Parametric Approach: Linear Classifier

[32x32x3]
array of numbers 0...1
(3072 numbers total)

10x1 3072x1

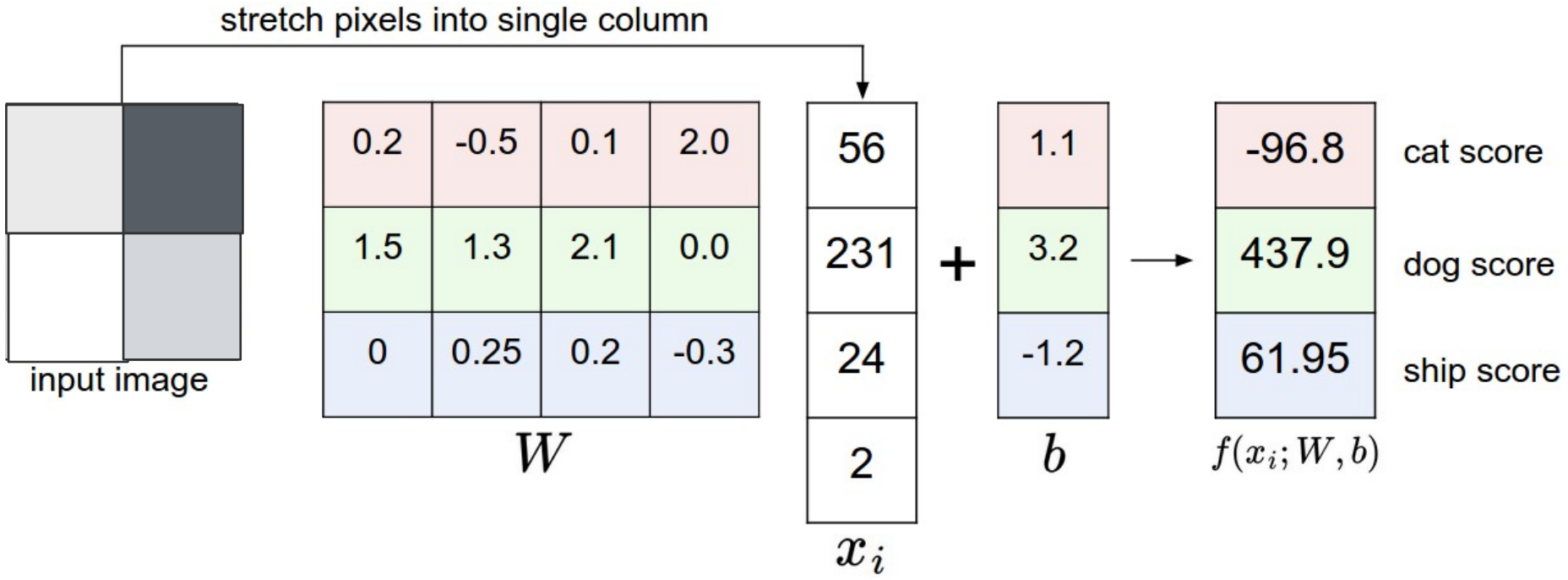
$f(x, W) = Wx$ (+b) 10x1

10x3072

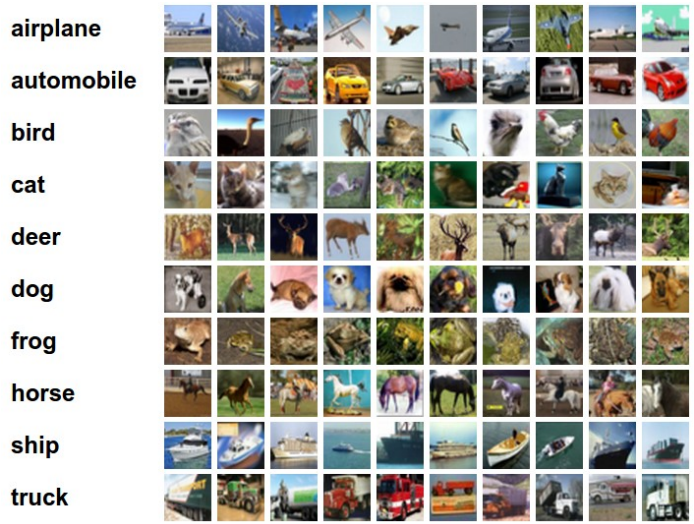
10 numbers, indicating class scores

parameters, or "weights"

Example with an Image with 4 Pixels, and 3 Classes (**cat**/**dog**/**ship**)

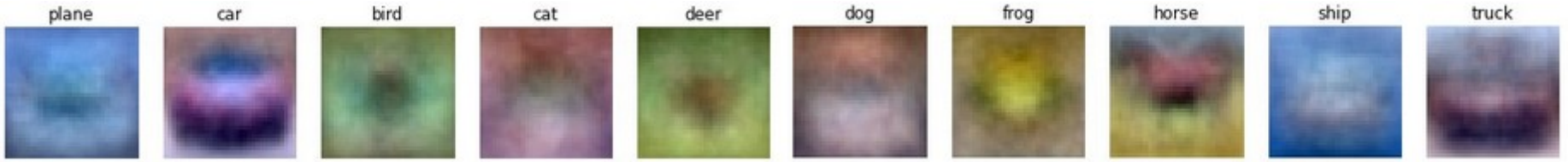


Interpreting a Linear Classifier

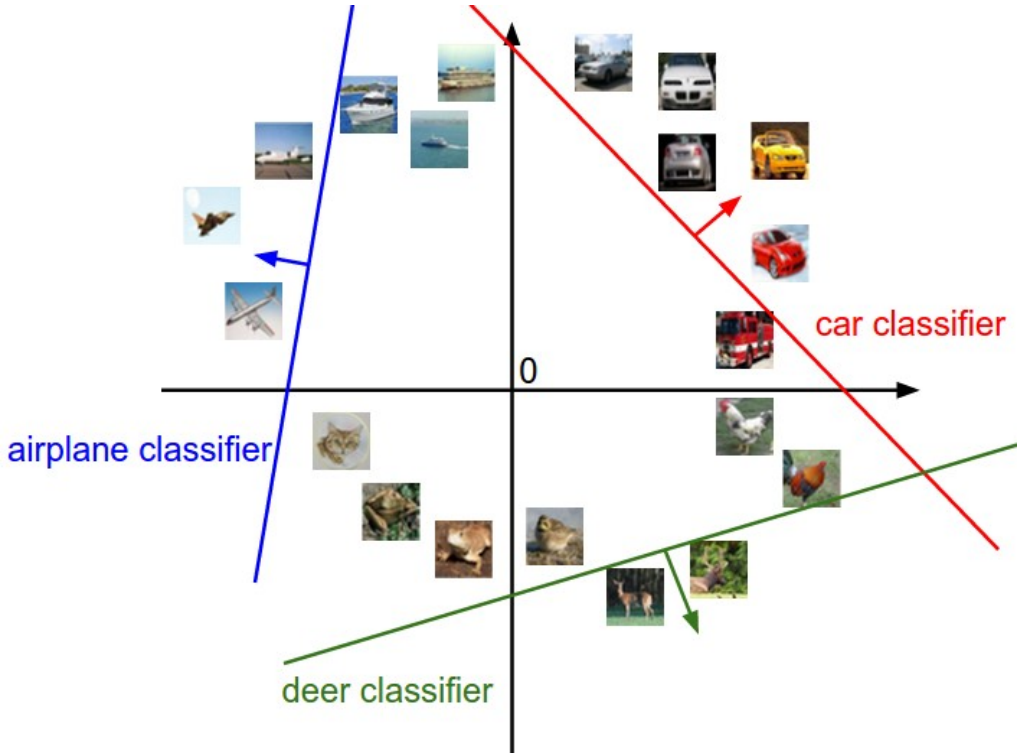


$$f(x_i, W, b) = Wx_i + b$$

Example trained weights of a linear classifier trained on CIFAR-10:



Interpreting a Linear Classifier



$$f(x_i, W, b) = Wx_i + b$$

[32x32x3]
array of numbers 0...1
(3072 numbers total)

So Far: We Defined a (Linear) Scoring Function: $f(x_i, W, b) = Wx_i + b$

Example class scores for 3 images, with a random W:



airplane	-3.45	-0.51	3.42
automobile	-8.87	**6.04**	4.64
bird	0.09	5.31	2.65
cat	**2.9**	-4.22	5.1
deer	4.48	-4.19	2.64
dog	8.02	3.58	5.55
frog	3.78	4.49	**-4.34**
horse	1.06	-4.37	-1.5
ship	-0.36	-2.09	-4.79
truck	-0.72	-2.93	6.14

Going forward: Loss function / Optimzation

Example class scores for 3 images, with a random W:



airplane	-3.45	-0.51	3.42
automobile	-8.87	**6.04**	4.64
bird	0.09	5.31	2.65
cat	**2.9**	-4.22	5.1
deer	4.48	-4.19	2.64
dog	8.02	3.58	5.55
frog	3.78	4.49	**-4.34**
horse	1.06	-4.37	-1.5
ship	-0.36	-2.09	-4.79
truck	-0.72	-2.93	6.14

1. Define a loss function that quantifies our unhappiness with the scores across the training data.

2. Come up with a way of efficiently finding the parameters (**W**, **b**) that minimize the loss function. (optimization)

Suppose: 3 training examples, 3 classes.
For some W the scores of $f(x, W) = Wx$ are:



cat	**3.2**	1.3	2.2
car	5.1	**4.9**	2.5
frog	-1.7	2.0	**-3.1**

Softmax Classifier (Multinomial Logistic Regression)

scores = unnormalized log probabilities of the classes.

cat	**3.2**
car	5.1
frog	-1.7

scores (s)

Softmax Classifier (Multinomial Logistic Regression)

scores = unnormalized log probabilities of the classes.

$$P(Y = k|X = x_i) = \frac{e^{s_k}}{\sum_j e^{s_j}}$$ where $$s = f(x_i; W)$$

Softmax function

cat	**3.2**
car	5.1
frog	-1.7

scores (s)

Softmax Classifier (Multinomial Logistic Regression)

cat	**3.2**
car	5.1
frog	-1.7
	scores (s)

scores = unnormalized log probabilities of the classes.

$$P(Y = k|X = x_i) = \frac{e^{s_k}}{\sum_j e^{s_j}}$$ where $$s = f(x_i; W)$$

$y = [1, 0, 0]^T$ (y_i is 1 (and 0 otherwise) if and only if sample belongs to class i)

Want to maximize the log likelihood of the correct class, or (for a loss function) to minimize the negative log likelihood of the correct class:

$$L_i = -\log P(Y = y_i|X = x_i)$$

---

in summary: $$L_i = -y_i \cdot log\left(\frac{e^{s_i}}{\sum_j e^{s_j}}\right) \quad L = \sum_i L_i$$

Softmax Classifier (Multinomial Logistic Regression)

$$s = f(x_i; W)$$

$$L_i = -y_i \cdot log\left(\frac{e^{s_i}}{\sum_j e^{s_j}}\right), y = [1,0,0]^T$$

cat	**3.2**
car	5.1
frog	-1.7

unnormalized log probabilities

Softmax Classifier (Multinomial Logistic Regression)

$$s = f(x_i; W)$$

$$L_i = -y_i \cdot log\left(\frac{e^{s_i}}{\sum_j e^{s_j}}\right), y = [1,0,0]^T$$

unnormalized probabilities

cat	**3.2**	exp →	**24.5**
car	5.1		164.0
frog	-1.7		0.18

unnormalized log probabilities

Softmax Classifier (Multinomial Logistic Regression)

$$s = f(x_i; W)$$

$$L_i = -y_i \cdot log\left(\frac{e^{s_i}}{\sum_j e^{s_j}}\right), y = [1,0,0]^T$$

unnormalized probabilities

	unnormalized log probabilities	exp	unnormalized probabilities	normalize	probabilities
cat	**3.2**	→	**24.5**	→	**0.13**
car	5.1	→	164.0	→	0.87
frog	-1.7	→	0.18	→	0.00

Softmax Classifier (Multinomial Logistic Regression)

$$s = f(x_i; W)$$

$$L_i = -y_i \cdot log\left(\frac{e^{s_i}}{\sum_j e^{s_j}}\right), y = [1,0,0]^T$$

unnormalized probabilities

	unnormalized log probabilities	exp	unnormalized probabilities	normalize	probabilities
cat	**3.2**	→	**24.5**	→	**0.13**
car	5.1	→	164.0	→	0.87
frog	-1.7	→	0.18	→	0.00

L_i = -log(0.13)
= **0.89**

Softmax Classifier (Multinomial Logistic Regression)

$$s = f(x_i; W)$$

$$L_i = -y_i \cdot log\left(\frac{e^{s_i}}{\sum_j e^{s_j}}\right), y = [1, 0, 0]^T$$

Q: What is the min/max possible loss L_i?

unnormalized probabilities

	unnormalized log probabilities	exp	unnormalized probabilities	normalize	probabilities
cat	**3.2**	→	**24.5**	→	**0.13**
car	5.1	→	164.0	→	0.87
frog	-1.7	→	0.18	→	0.00

L_i = -log(0.13)
= **0.89**

Softmax Classifier (Multinomial Logistic Regression)

$$s = f(x_i; W)$$

$$L_i = -y_i \cdot log\left(\frac{e^{s_i}}{\sum_j e^{s_j}}\right), y = [1,0,0]^T$$

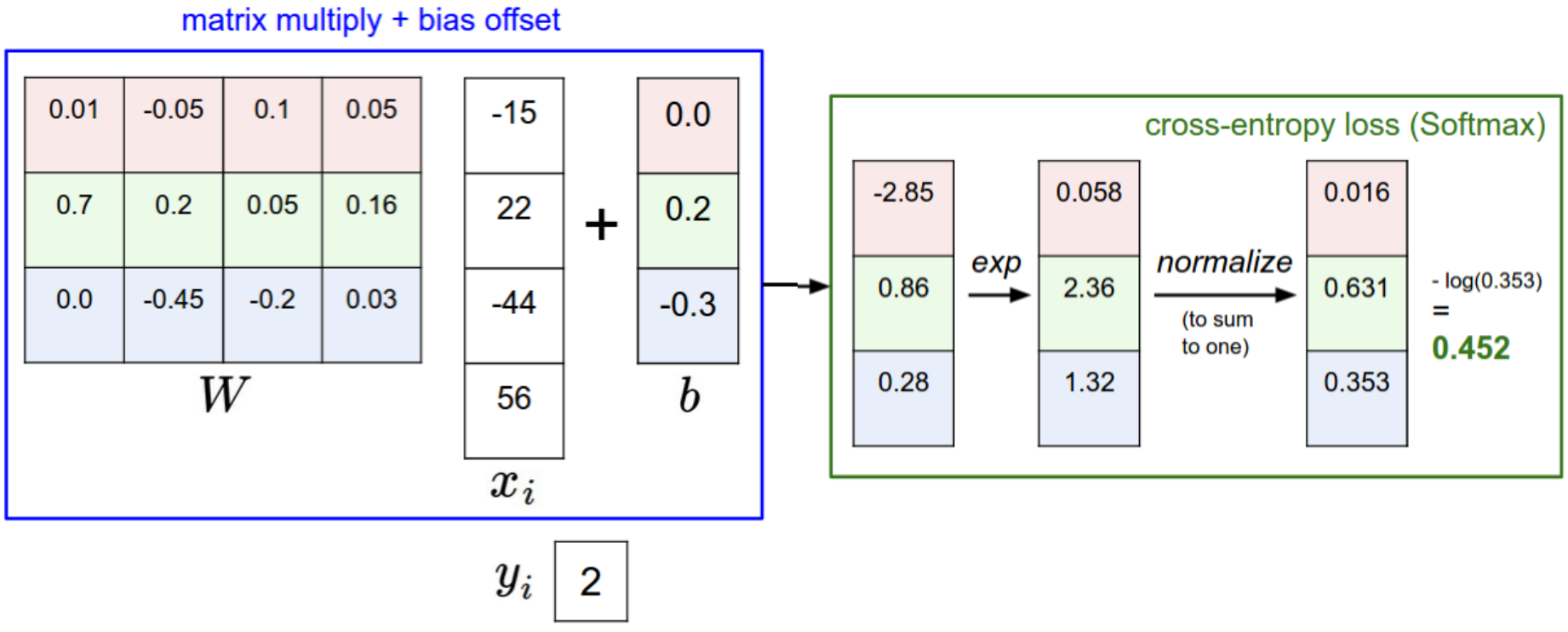
unnormalized probabilities

	unnormalized log probabilities	exp	unnormalized probabilities	normalize	probabilities
cat	**3.2**	→	**24.5**	→	**0.13**
car	5.1	→	164.0	→	0.87
frog	-1.7	→	0.18	→	0.00

L_i = -log(0.13)
= **0.89**

In torch7:
nn.CrossEntropyCriterion()
In pyTorch:
nn.CrossEntropyLoss()

Softmax Classifier (Multinomial Logistic Regression)



Weight Regularization

$$s = f(x, W) = Wx$$

$$L_i = -y_i \cdot log\left(\frac{e^{s_i}}{\sum_j e^{s_j}}\right)$$

$$L = \sum_i L_i + \lambda R(W)$$

(y is a one−hot vector where y_i is 1 (and 0 otherwise) if and only if sample belongs to class i

Weight Regularization

$$s = f(x, W) = Wx$$

$$L_i = -y_i \cdot log\left(\frac{e^{s_i}}{\sum_j e^{s_j}}\right)$$

$$L = \sum_i L_i + \lambda R(W)$$

(y is a one−hot vector where y_i is 1 (and 0 otherwise) if and only if sample belongs to class i

Weight Regularization

$$L = \sum_i L_i + \lambda R(W)$$

Regularization strength

(another hyperparameter)

In common use:

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

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

Elastic net (L1 + L2) $R(W) = \sum_k \sum_l \beta W_{k,l}^2 + |W_{k,l}|$

Dropout, BN (will see next week)

Weight Regularization: L2 - motivation

$$x = [1, 1, 1, 1]$$

$$w_1 = [1, 0, 0, 0]$$

$$w_2 = [0.25, 0.25, 0.25, 0.25]$$

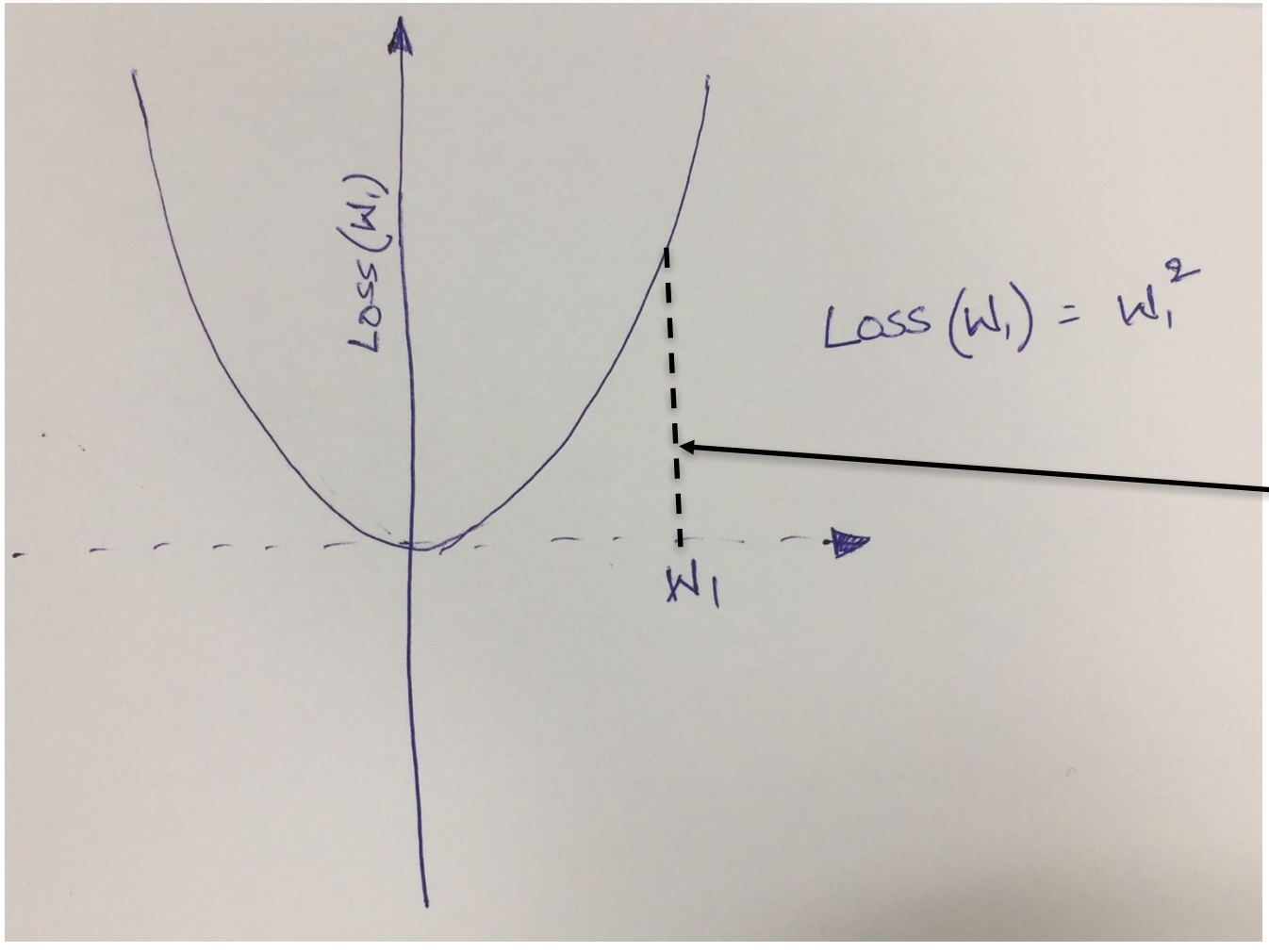
$$w_1^T x = w_2^T x = 1$$

Which $\boldsymbol{w}$ would L2 regularization choose?

Optimization: How to get good values for **W**?

Brute force search?

Optimization: How to get good values for **W**?



In 1-dimension, the derivative of a function:

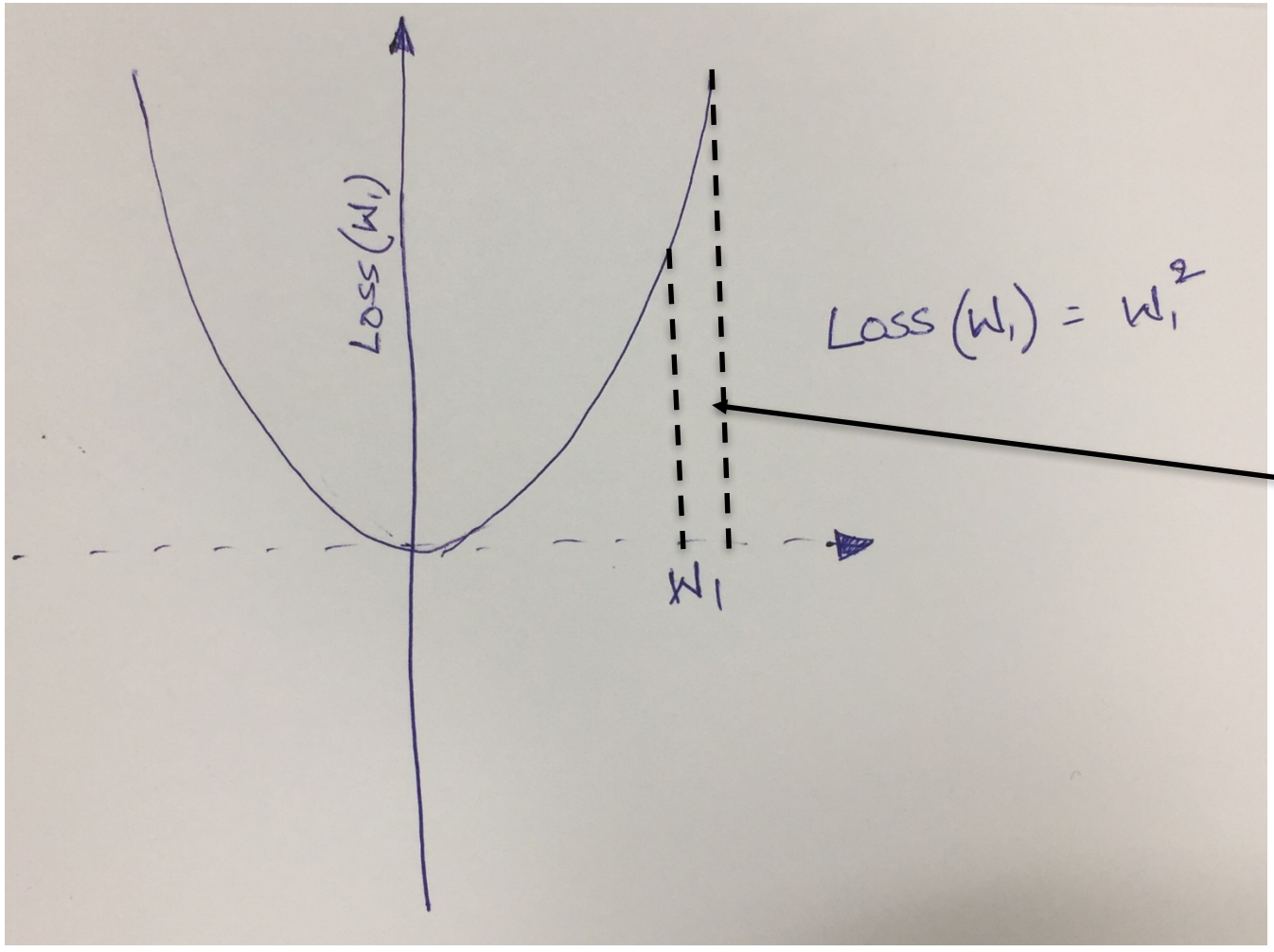
$$\frac{df(x)}{dx} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

$W_1 = 10,$

$Loss(W_1) = 100$

Suppose we have a 1D weight and the loss only depends on the weight

Optimization: How to get good values for **W**?



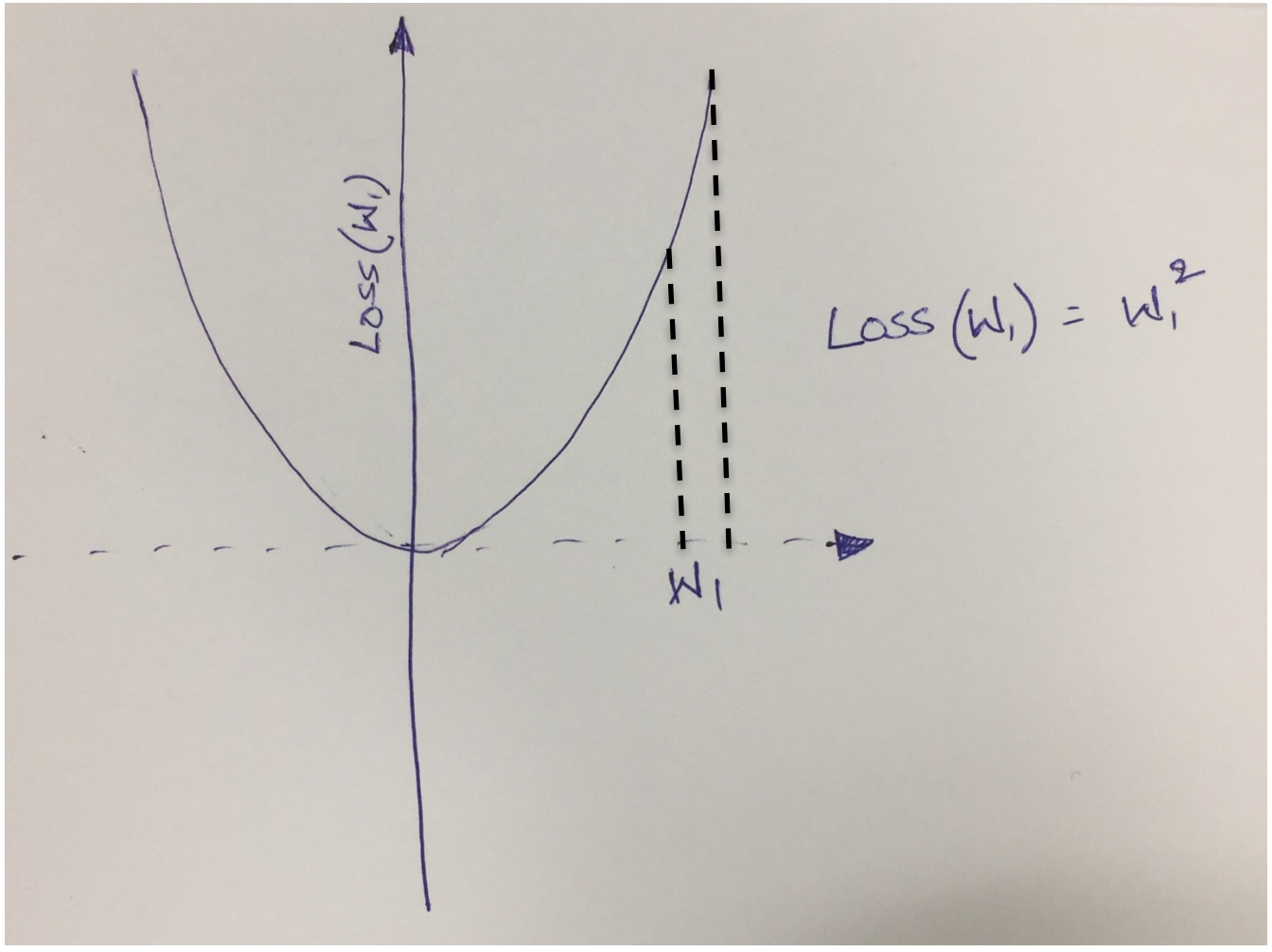
In 1-dimension, the derivative of a function:

$$\frac{df(x)}{dx} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

$W_1 = 10.01,$

$Loss(W_1) = 100.2001$

Optimization: How to get good values for **W**?



In 1-dimension, the derivative of a function:

$$\frac{df(x)}{dx} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

$$\frac{dLoss(W_1)}{dW_1} = (100.2001 - 100)/0.01$$

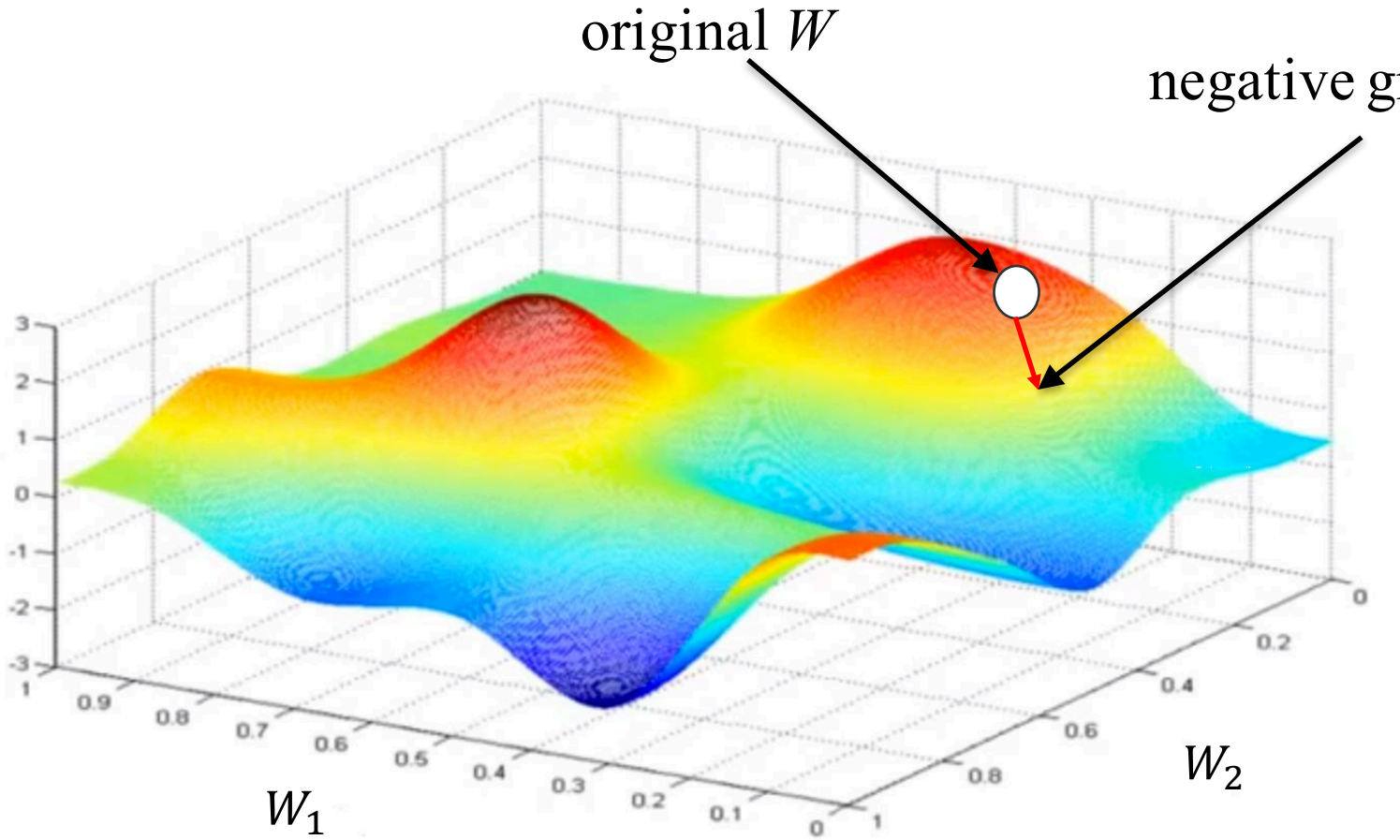
$$= 20.01$$

(assume λ=0.1)

$$W_1 = W_1 - \lambda \frac{d\ Loss(W_1)}{dW_1}$$

Move in the direction of negative slope

Optimization: How to get good values for **W**'s?



$$\frac{\partial Loss(W)}{\partial W} = \left[\frac{\partial Loss(W)}{\partial W_1}, \frac{\partial Loss(W)}{\partial W_2}\right]$$

$$W_i = W_i - \lambda \frac{\partial Loss(W_i)}{\partial W_i}$$

In multiple dimensions, the derivative of the loss with respect to the weight vector is called the gradient, and is nothing but a row vector, each component being component wise partial derivatives (more soon)

Optimization: How to get good values for **W**'s?

In [1]:
```
require 'nn';
model = nn.Linear(1,2)
x = torch.rand(1)
y = torch.Tensor({1})
```

In [2]:
```
op = model:forward(x)
print(x)
print(op)
print(y)
print(model.weight)
```

Out[2]:
```
 0.1854
[torch.DoubleTensor of size 1]

 0.1313
-0.4951
[torch.DoubleTensor of size 2]

 1
[torch.DoubleTensor of size 1]

-0.8399
-0.6943
[torch.DoubleTensor of size 2x1]
```

Optimization: How to get good values for **W**'s?

```
In [3]: criterion = nn.CrossEntropyCriterion()
        e1 = criterion:forward(op, y)
        print(e1)

Out[3]: 0.78581595010034
```

$f(x)$

In 1-dimension, the derivative of a function:

$$\frac{df(x)}{dx} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

In multiple dimensions, the **gradient** is the vector of partial derivatives.

Optimization: How to get good values for **W**'s?

```
In [4]: model.weight[1][1] = model.weight[1][1] + 0.00001
        e2 = criterion:forward(model:forward(x), y)
        print((e2 - e1)/0.00001)

Out[4]: -0.064589666493031
```

$f(x+h)$

```
In [5]: model.weight[1][1] = model.weight[1][1] - 0.00001
        model.weight[2][1] = model.weight[2][1] + 0.00001
        e3 = criterion:forward(model:forward(x), y)
        print((e3 - e1)/0.00001)

Out[5]: 0.06458974454171
```

$f(x+h)$

In summary: How to get good values for **W**'s?

Loss is just a function of W:

$$L = \frac{1}{N}\sum_{i=1}^{N} L_i + \sum_k W_k^2$$

$$L_i = -y_i \cdot log\left(\frac{e^{s_i}}{\sum_j e^{s_j}}\right)$$

$$s = f(x_i; W) = Wx$$

want $\nabla_W L$

- Numerical gradient: approximate, slow, easy to write
- Analytic gradient: exact, fast, error-prone

In practice: Always use analytic gradient, but check implementation with numerical gradient. This is called a **gradient check.**

y is a one−hot vector where y_i is 1 (and 0 otherwise) if and only if sample belongs to class i

Optimization: How to get good values for **W**'s and **b**'s?

Gradient descent:

```
while true -- for new x,y pairs
  df_do = criterion:backward(model:forward(x),y)
  model:backward(x,  df_do)
  model.weight  = model.weight  - learning_rate  * model.gradWeight
end
```

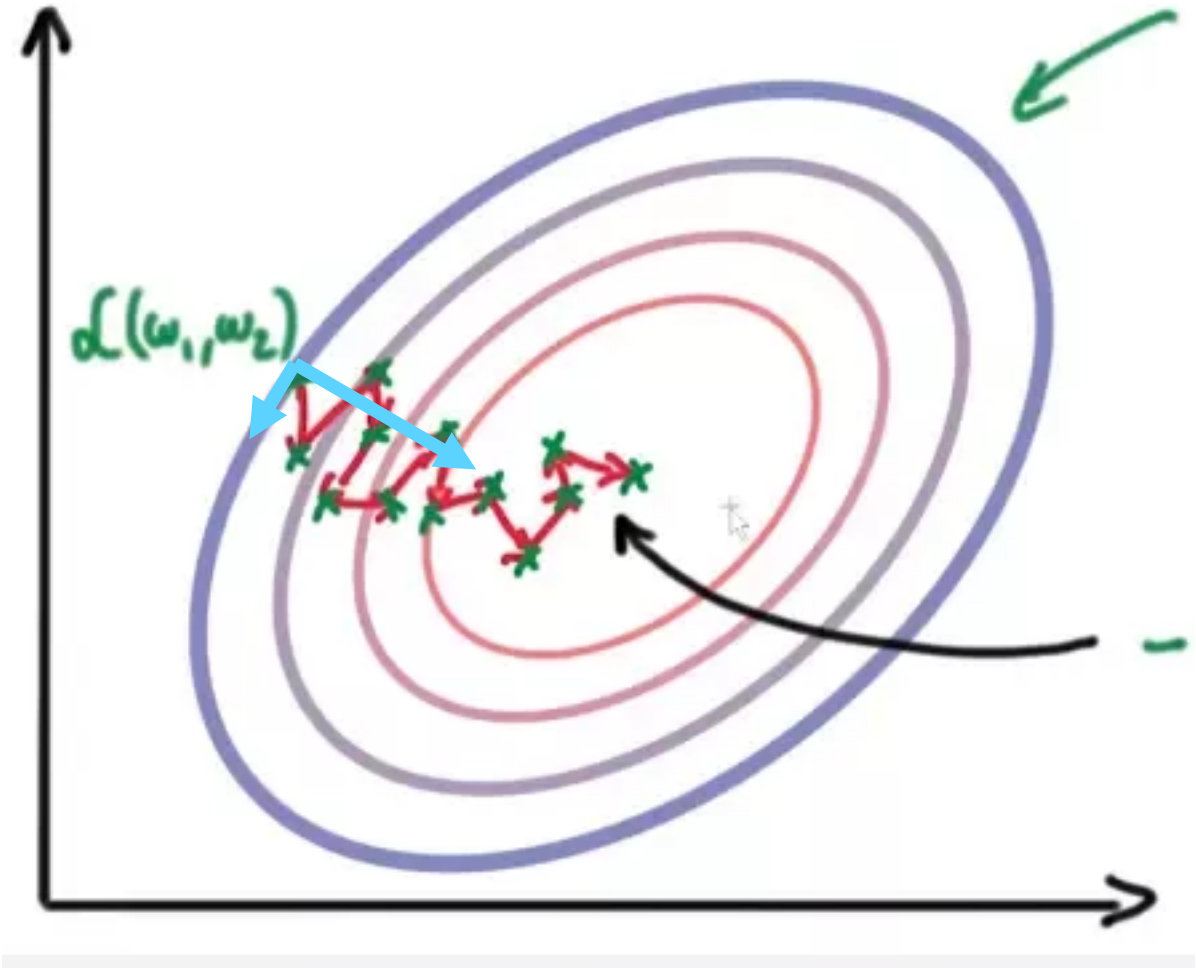
Optimization: How to get good values for **W**'s and **b**'s?

Gradient descent (vanilla mini-batch):

```
while true -- for new x,y pairs
  df_do_all = err:backward(model:forward(batch_x), batch_y)
  df_do = mean(df_do_all)
  model:backward(batch_x, df_do)
  model.weight = model.weight - learning_rate * model.gradWeight
end
```

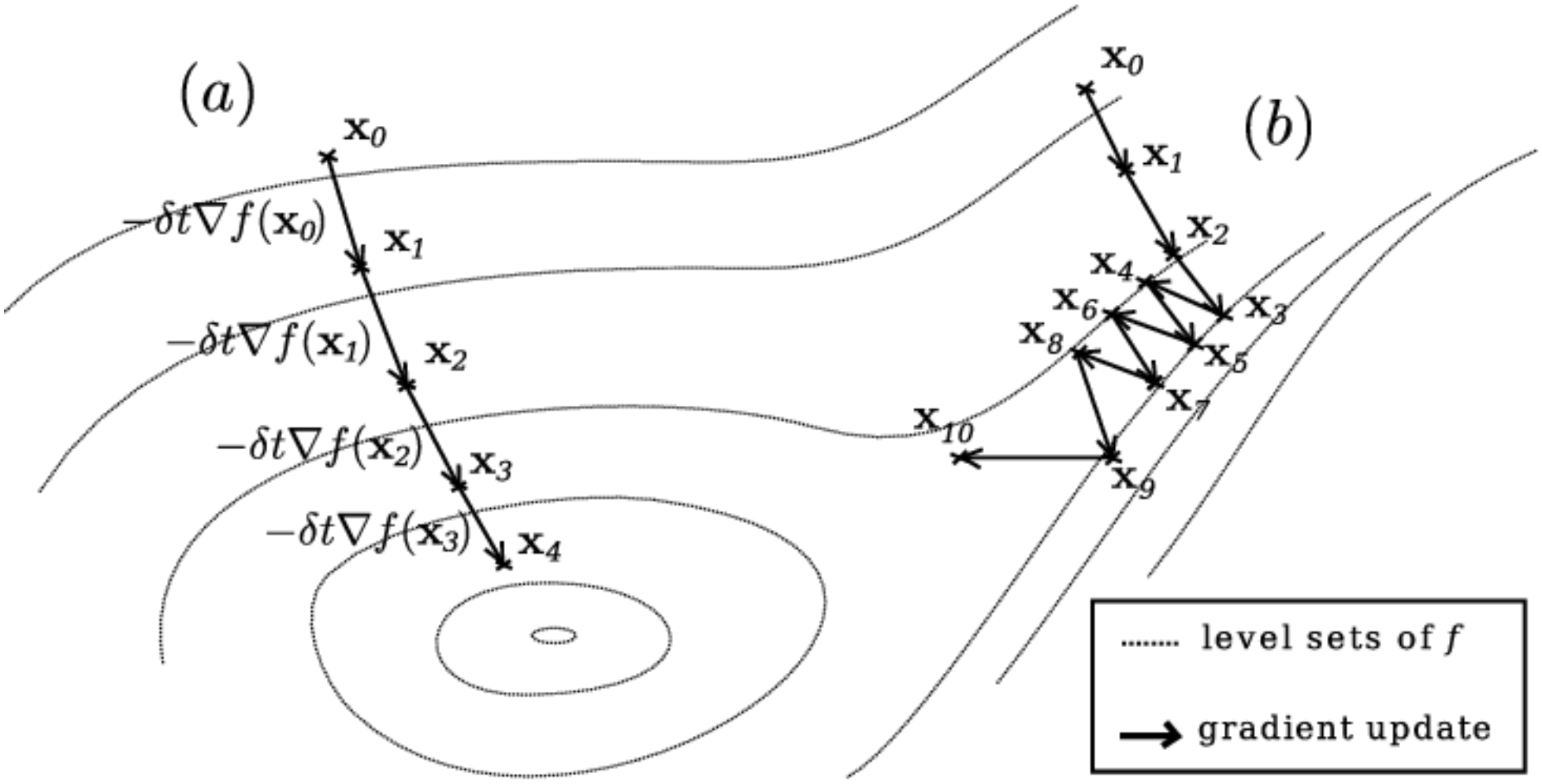
- only use a small portion of the training set to compute the gradient.

Gradient Descent without Momentum

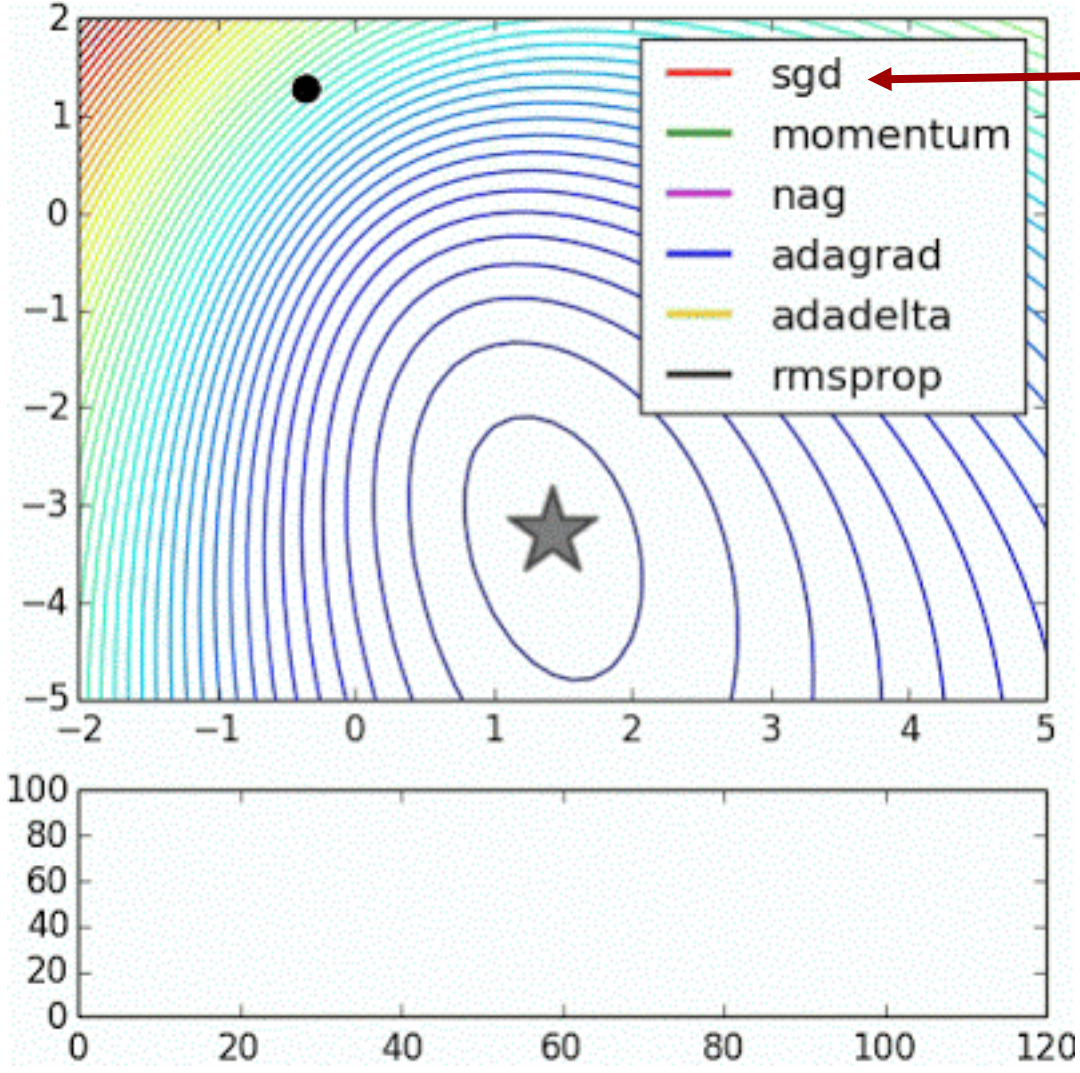


- http://sebastianruder.com/optimizing-gradient-descent/
- https://www.quora.com/What-are-differences-between-update-rules-like-AdaDelta-RMSProp-AdaGrad-and-AdaM

Gradient Descent without Momentum

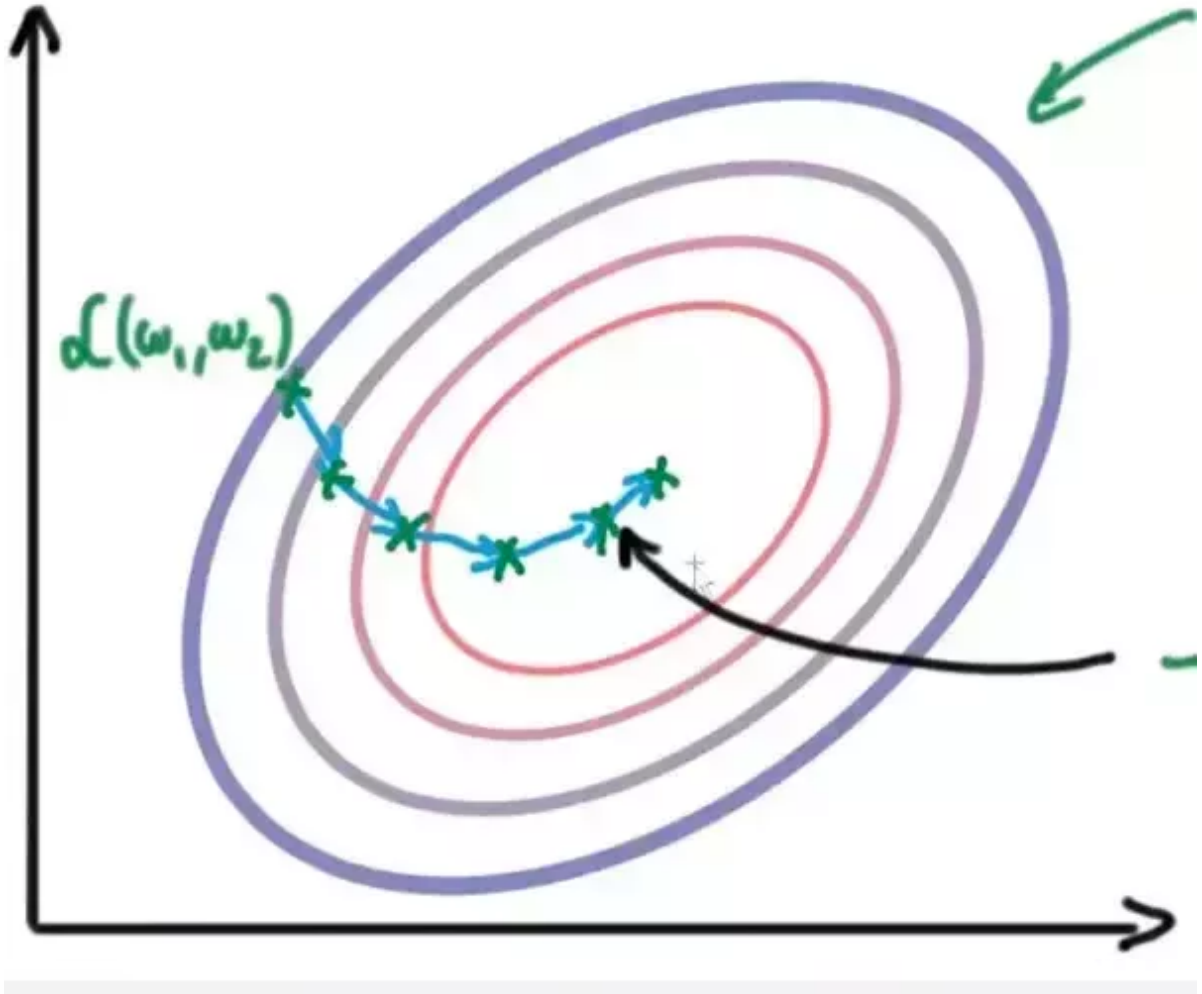


Gradient Descent without Momentum (Red)



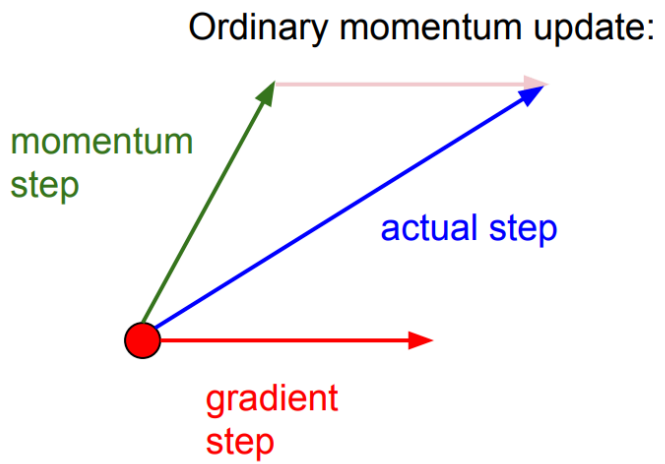
http://www.denizyuret.com/2015/03/alec-radfords-animations-for.html

SGD with Momentum

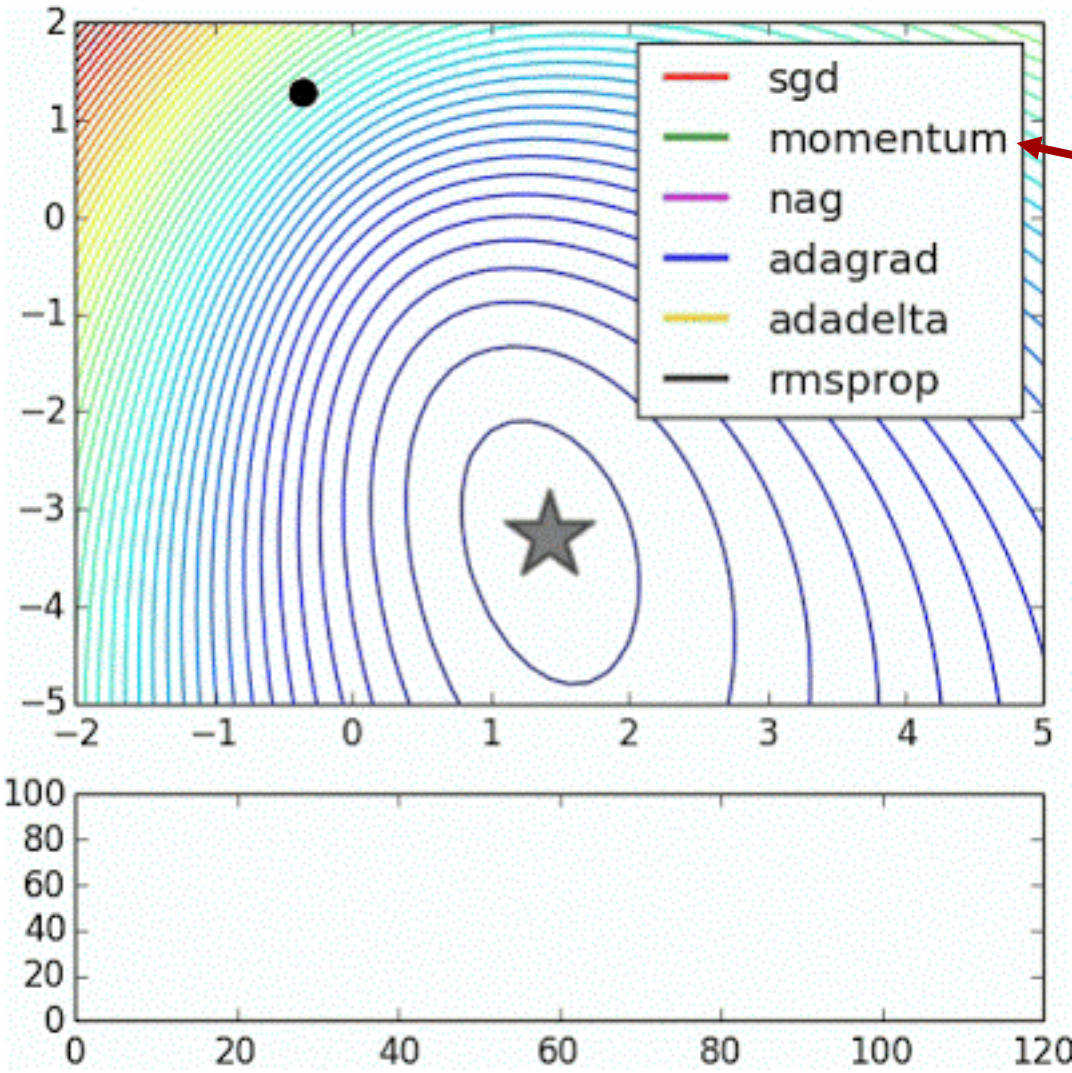


$$\boldsymbol{m_t} = \propto \boldsymbol{m_{t-1}} + \lambda \frac{\partial L}{\partial \boldsymbol{W}_t}$$

$$\boldsymbol{W_t} = \boldsymbol{W_t} - \boldsymbol{m_t}$$

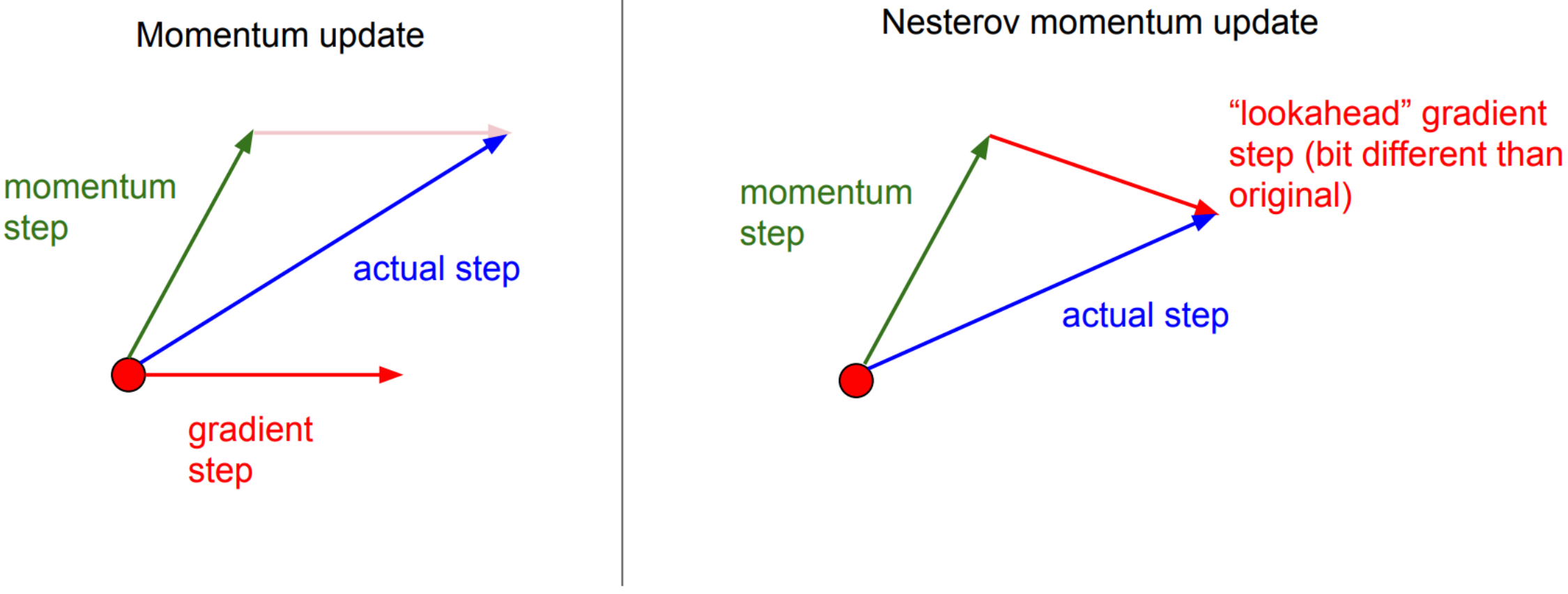


Gradient Descent with Momentum (Green)

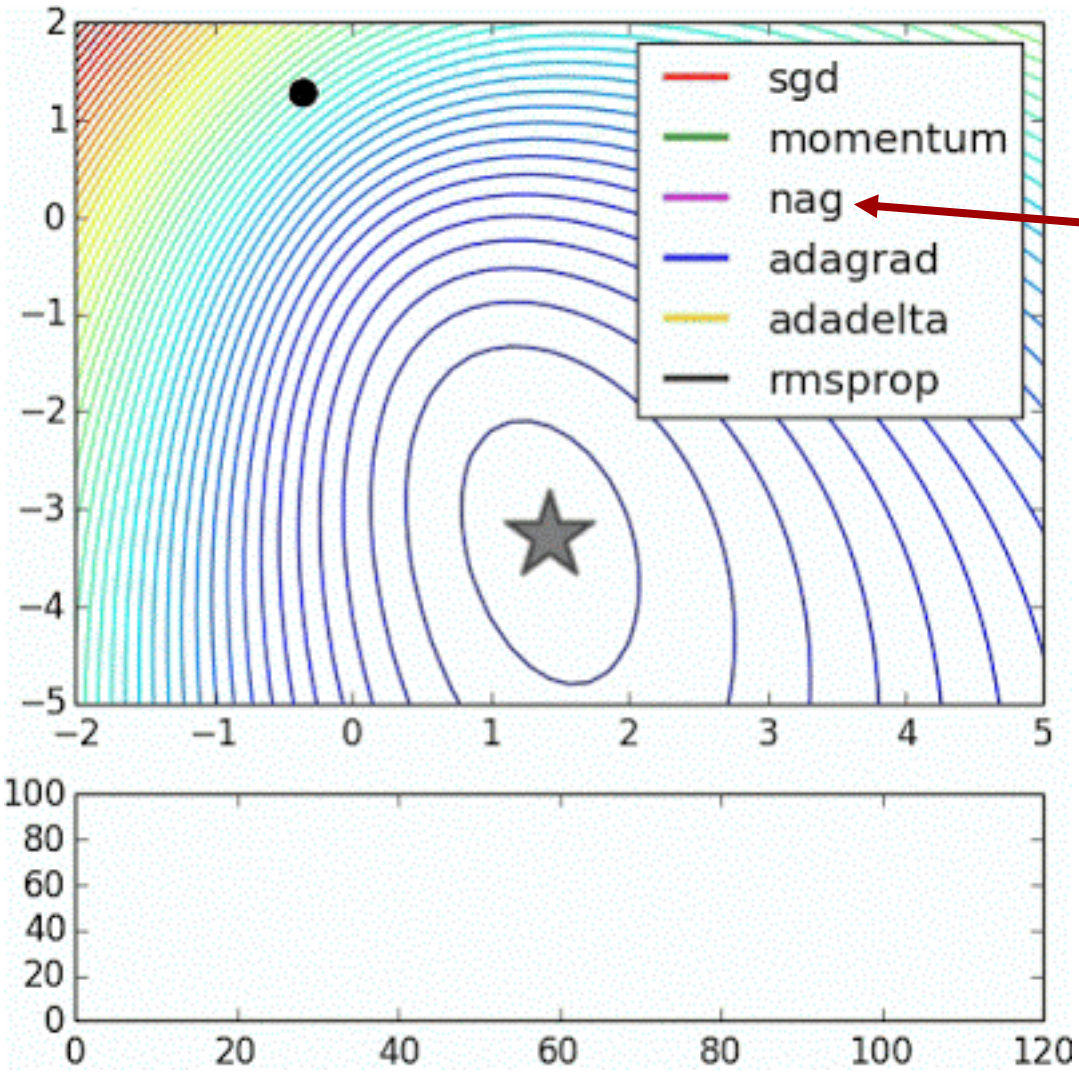


http://www.denizyuret.com/2015/03/alec-radfords-animations-for.html

Nesterov Momentum Update



Gradient Descent with Nesterov Momentum (Violet)



nag: Nesterov Accelerated Gradient

http://www.denizyuret.com/2015/03/alec-radfords-animations-for.html

AdaGrad Update

[Duchi et al., 2011]

$$\boldsymbol{W_t} = \boldsymbol{W_t} + \frac{-\lambda \frac{\partial L}{\partial \boldsymbol{W}_t}}{\sqrt{\boldsymbol{\kappa_t} + 10^{-7}}} \qquad \boldsymbol{\kappa_t} = \boldsymbol{\kappa_{t-1}} + \left(\frac{\partial L}{\partial \boldsymbol{W}_t}\right)^2$$

Elementwise

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

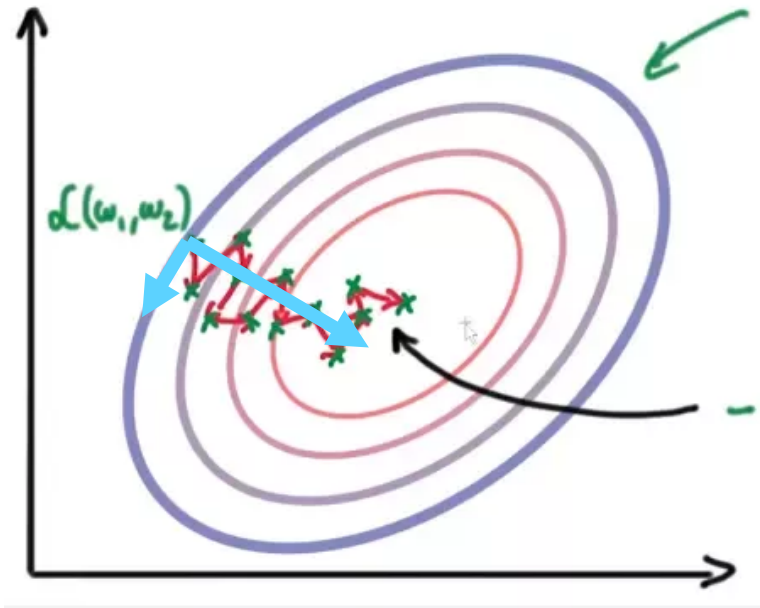
AdaGrad Update

[Duchi et al., 2011]

$$W_t = W_t + \frac{-\lambda \frac{\partial L}{\partial W_t}}{\sqrt{\kappa_t + 10^{-7}}}$$

$$\kappa_t = \kappa_{t-1} + \left(\frac{\partial L}{\partial W_t}\right)^2$$

Elementwise



Q: What happens with AdaGrad?

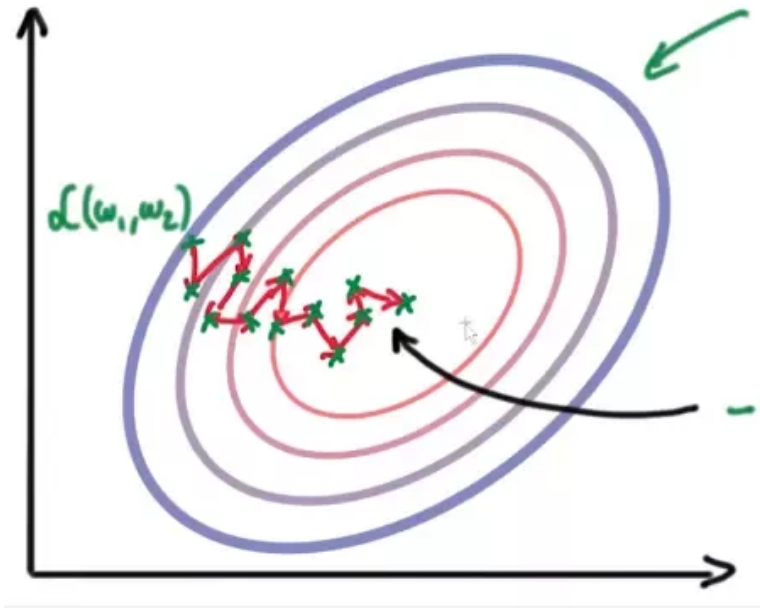
AdaGrad Update

[Duchi et al., 2011]

$$W_t = W_t + \frac{-\lambda \frac{\partial L}{\partial W_t}}{\sqrt{\kappa_t + 10^{-7}}}$$

$$\kappa_t = \kappa_{t-1} + \left(\frac{\partial L}{\partial W_t}\right)^2$$

Elementwise



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

RMSProp Update

[Tieleman and Hinton, 2012]

$$\boldsymbol{W_t = W_t + \frac{-\lambda \frac{\partial L}{\partial W_t}}{\sqrt{\kappa_t + 10^{-7}}}}$$

$$\boldsymbol{\kappa_t = \kappa_{t-1} + \left(\frac{\partial L}{\partial W_t}\right)^2}$$

AdaGrad

$$\boldsymbol{W_t = W_t + \frac{-\lambda \frac{\partial L}{\partial W_t}}{\sqrt{\kappa_t + 10^{-7}}}}$$

$$\boldsymbol{\kappa_t = \zeta \kappa_{t-1} + (1-\zeta)\left(\frac{\partial L}{\partial W_t}\right)^2}$$

RMSProp

~0.99 to make it leaky

RMSProp Update – Fun Fact

rmsprop: A mini-batch version of rprop

- rprop is equivalent to using the gradient but also dividing by the size of the gradient.
  - The problem with mini-batch rprop is that we divide by a different number for each mini-batch. So why not force the number we divide by to be very similar for adjacent mini-batches?
- rmsprop: Keep a moving average of the squared gradient for each weight

$$MeanSquare(w,t) = 0.9\, MeanSquare(w,\ t-1) + 0.1\left(\partial E / \partial w(t)\right)^2$$

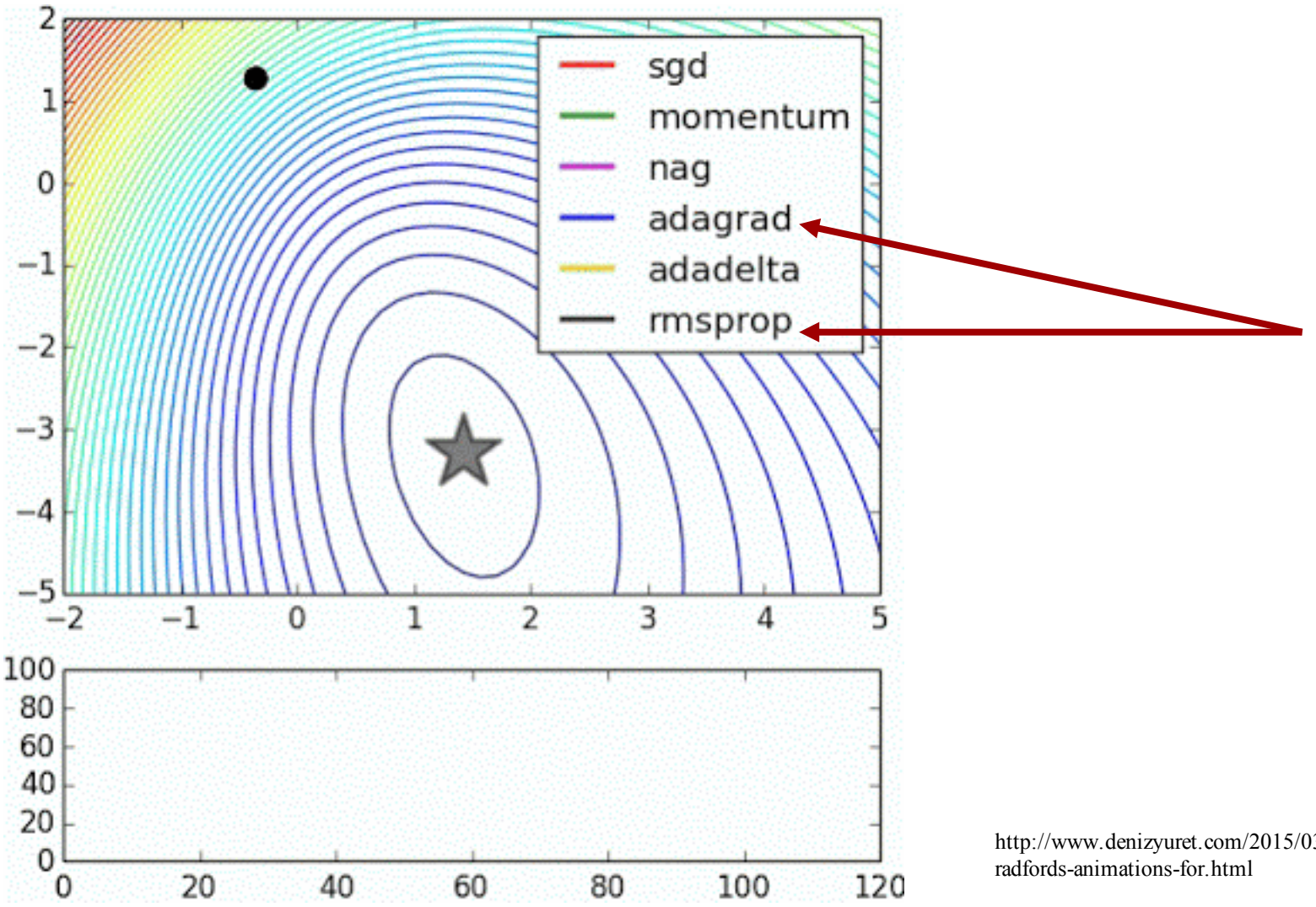
- Dividing the gradient by $\sqrt{MeanSquare(w,\ t)}$ makes the learning work much better (Tijmen Tieleman, unpublished).

Introduced in a slide in Geoff Hinton's Coursera class, lecture 6

Cited by several papers as:

[52] T. Tieleman and G. E. Hinton. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude., 2012.

Gradient Descent with AdaGrad and RMSProp



http://www.denizyuret.com/2015/03/alec-radfords-animations-for.html

Adam Update
(incomplete but close)

[Kingma and Ba, 2014]

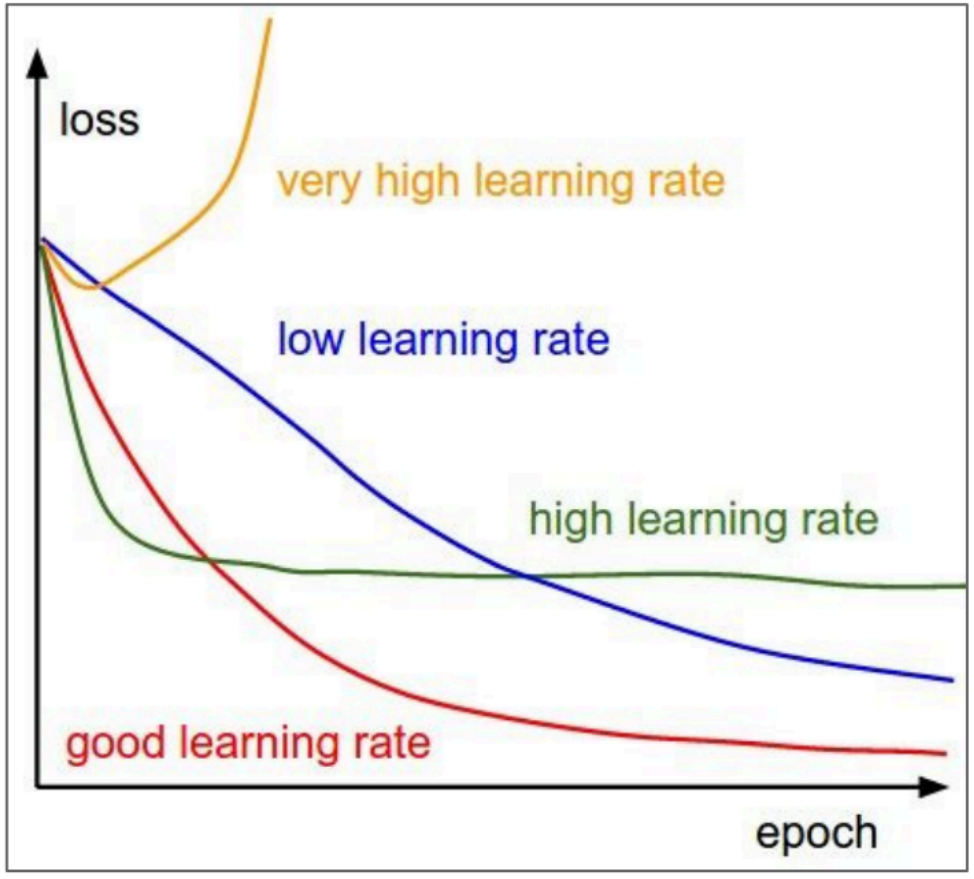
$$\boldsymbol{m_t} = \propto_1 \boldsymbol{m_{t-1}} + (1-\propto_1)\frac{\partial L}{\partial \boldsymbol{W}_t}$$ **Momentum like**

$$\boldsymbol{\kappa_t} = \zeta \boldsymbol{\kappa_{t-1}} + (1-\zeta)\left(\frac{\partial L}{\partial \boldsymbol{W}_t}\right)^2$$ **RMSProp like**

$$\boldsymbol{W_t} = \boldsymbol{W_t} + \frac{-\lambda \boldsymbol{m_t}}{\sqrt{\boldsymbol{\kappa_t} + 10^{-7}}}$$

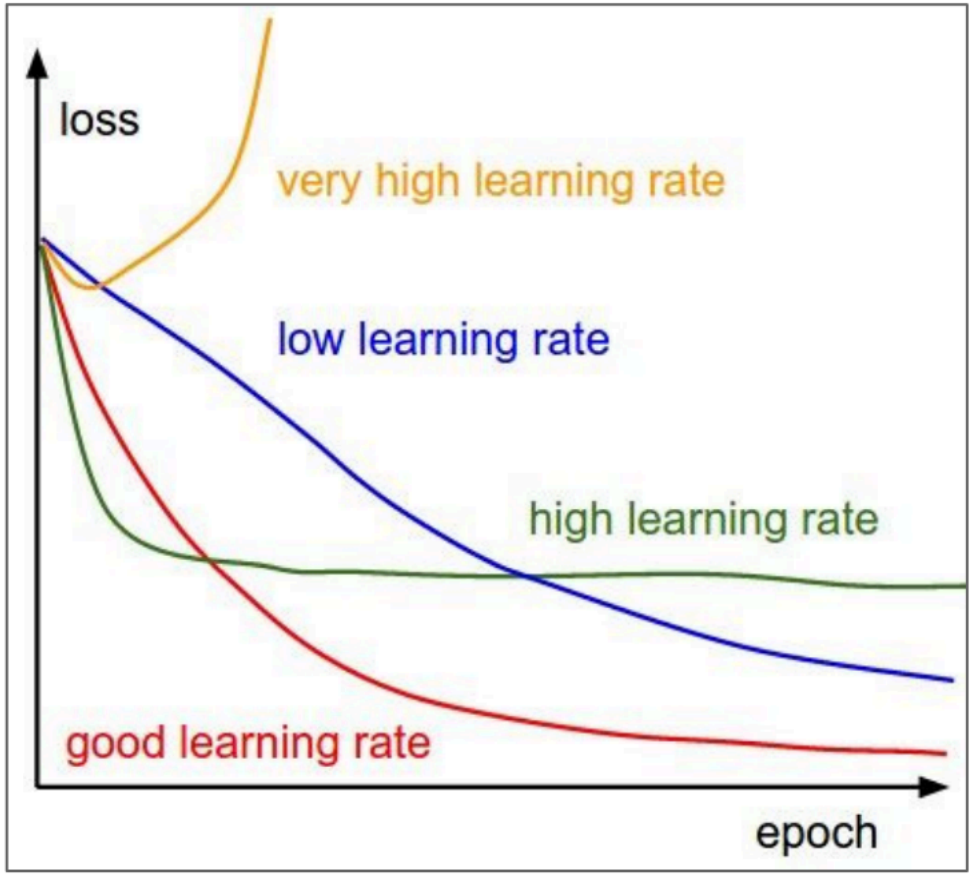
Looks a bit like RMSProp with momentum

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



Trick Q: Which one of these learning rates is best to use?

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:

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

Second order optimization methods

Second order Taylor expansion:

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

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

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

Q: what is nice about this update?

no hyperparameters! (e.g. learning rate), fast convegence

Second order optimization methods

Second order Taylor expansion:

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

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

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

Q: why is this impractical for training Deep Neural Nets?

If 100 million params, H = 100x100 million. Then we need to invert it!

Second order optimization methods

- Quasi-Newton methods (**BFGS** most popular):

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

instead of inverting the Hessian ($O(n^3)$), approximate inverse Hessian with rank 1 updates over time ($O(n^2)$ each).

- **L-BFGS** (Limited memory BFGS):

Does not form/store the full inverse Hessian

BFGS=Broyden–Fletcher–Goldfarb–Shanno

L-BFGS

- **Usually works very well in full batch, deterministic mode** i.e. if you have a single, deterministic f(x) then **L-BFGS** will probably work very nicely

- **Does not transfer very well to mini-batch setting.** Gives bad results. Adapting **L-BFGS** to large-scale, stochastic setting is an active area of research.

In practice:

- Start with SGD with vanilla momentum
- Try Adam, good choice in many cases
- If you can afford to do full batch updates then try out **L-BFGS** (and don't forget to disable all stochasticity)

Thank you!