

# CSC 561: Neural Networks and Deep Learning

## Optimization (part 2)

Marco Alvarez

Department of Computer Science and Statistics  
University of Rhode Island

Spring 2025



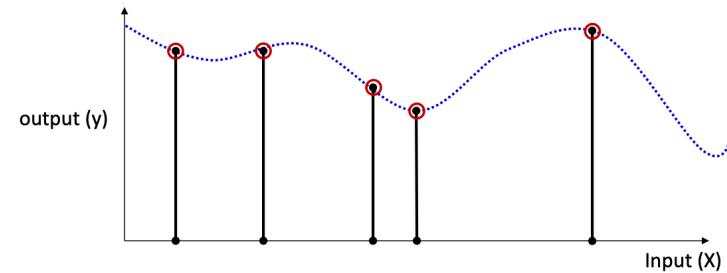
Credit: the following slides are taken from one of the “11-785 Introduction to Deep Learning - CMU” lectures

## Training Neural Networks: Optimization

Intro to Deep Learning, Spring 2025

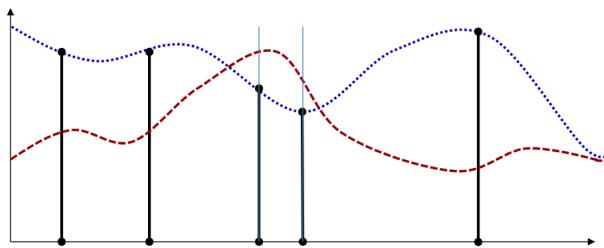


### The training formulation



- Given input output pairs at a number of locations, estimate the entire function

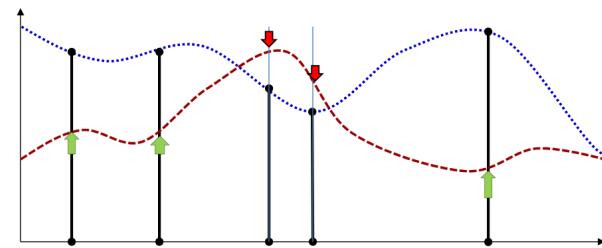
## Gradient descent



- Start with an initial function

7

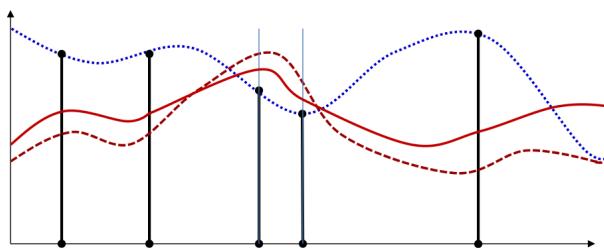
## Gradient descent



- Start with an initial function
- Adjust its value at *all* points to make the outputs closer to the required value
  - Gradient descent adjusts parameters to adjust the function value at *all* points
  - Repeat this iteratively until we get arbitrarily close to the target function at the training points

8

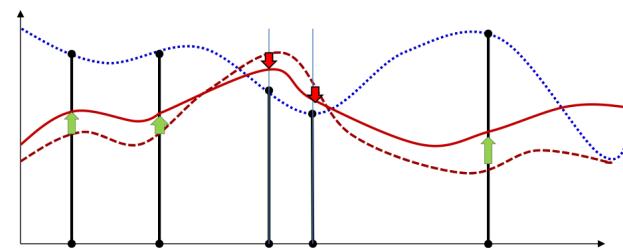
## Gradient descent



- Start with an initial function
- Adjust its value at *all* points to make the outputs closer to the required value
  - Gradient descent adjusts parameters to adjust the function value at *all* points
  - Repeat this iteratively until we get arbitrarily close to the target function at the training points

9

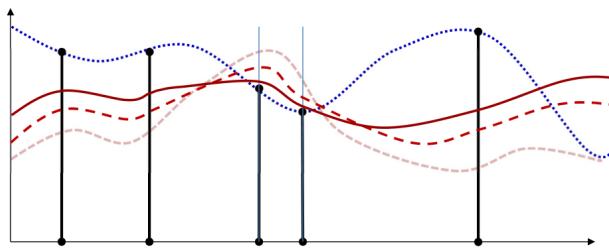
## Gradient descent



- Start with an initial function
- Adjust its value at *all* points to make the outputs closer to the required value
  - Gradient descent adjusts parameters to adjust the function value at *all* points
  - Repeat this iteratively until we get arbitrarily close to the target function at the training points

10

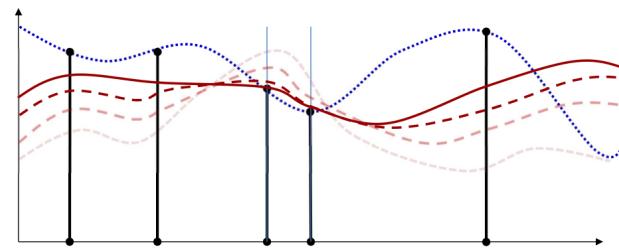
## Gradient descent



- Start with an initial function
- Adjust its value at *all* points to make the outputs closer to the required value
  - Gradient descent adjusts parameters to adjust the function value at *all* points
  - Repeat this iteratively until we get arbitrarily close to the target function at the training points

11

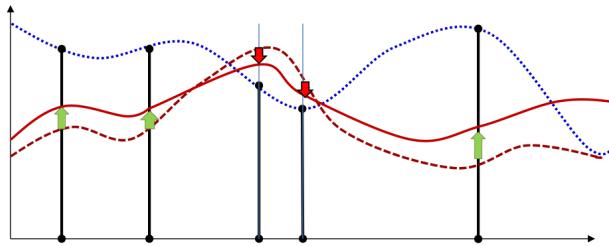
## Gradient descent



- Start with an initial function
- Adjust its value at *all* points to make the outputs closer to the required value
  - Gradient descent adjusts parameters to adjust the function value at *all* points
  - Repeat this iteratively until we get arbitrarily close to the target function at the training points

12

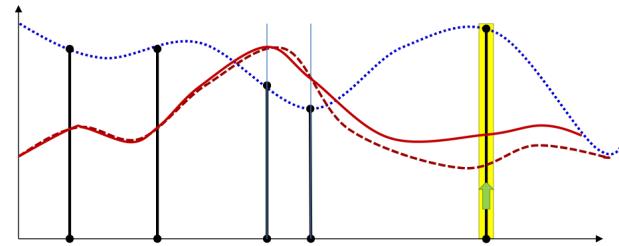
## Effect of number of samples



- Problem with conventional gradient descent: we try to simultaneously adjust the function at *all* training points
  - We must process *all* training points before making a single adjustment
  - “**Batch**” update

13

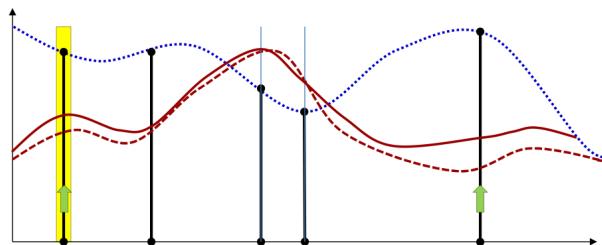
## Alternative: Incremental update



- Alternative: adjust the function at one training point at a time
  - Keep adjustments small

16

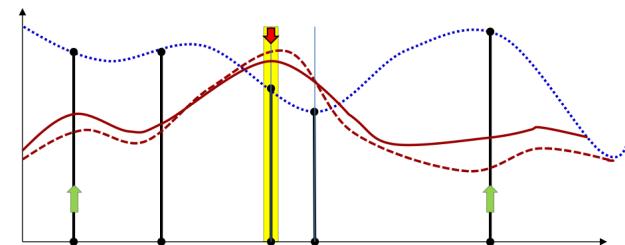
## Alternative: Incremental update



- Alternative: adjust the function at one training point at a time
  - Keep adjustments small

17

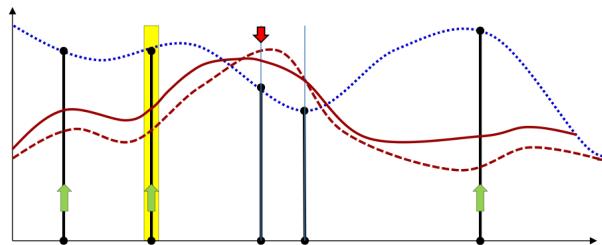
## Alternative: Incremental update



- Alternative: adjust the function at one training point at a time
  - Keep adjustments small

18

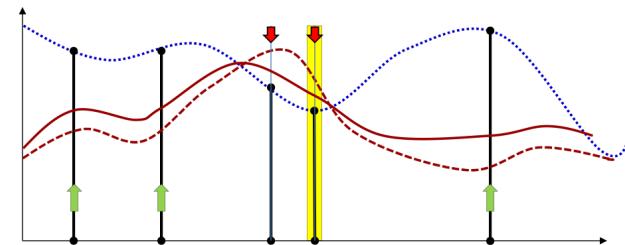
## Alternative: Incremental update



- Alternative: adjust the function at one training point at a time
  - Keep adjustments small

19

## Alternative: Incremental update



- Alternative: adjust the function at one training point at a time
  - Keep adjustments small
  - Eventually, when we have processed all the training points, we will have adjusted the entire function
    - With *greater* overall adjustment than we would if we made a single “Batch” update

20

## Incremental Updates

- The iterations can make multiple passes over the data
- A single pass through the entire training data is called an “epoch”
  - An epoch over a training set with  $T$  samples results in  $T$  updates of parameters

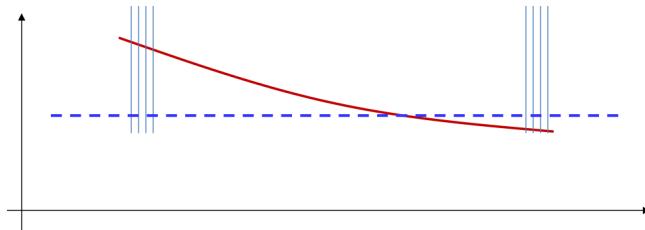
22

## Incremental Update

- Given  $(X_1, d_1), (X_2, d_2), \dots, (X_T, d_T)$
- Initialize all weights  $W_1, W_2, \dots, W_K$
- Do:
  - For all  $t = 1:T$ 
    - For every layer  $k$ :
      - Compute  $\nabla_{W_k} \text{Div}(Y_t, d_t)$
      - Update
    - $$W_k = W_k - \eta \nabla_{W_k} \text{Div}(Y_t, d_t)^T$$
 ← One update
- Until  $\text{Loss}$  has converged

23

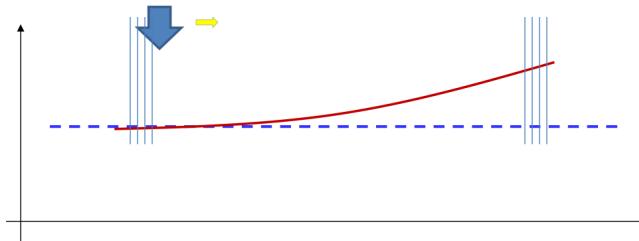
## Caveats: order of presentation



- If we loop through the samples in the same order, we may get *cyclic* behavior

24

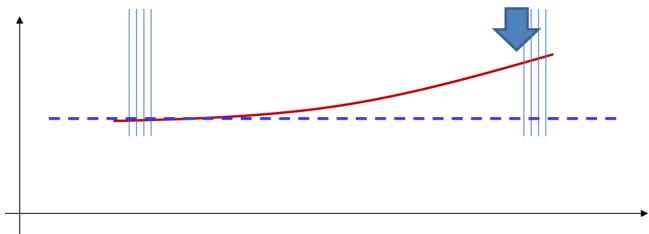
## Caveats: order of presentation



- If we loop through the samples in the same order, we may get *cyclic* behavior

25

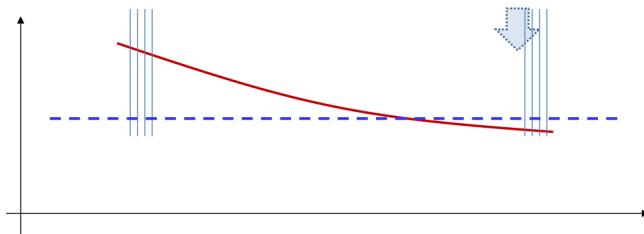
## Caveats: order of presentation



- If we loop through the samples in the same order, we may get *cyclic* behavior

26

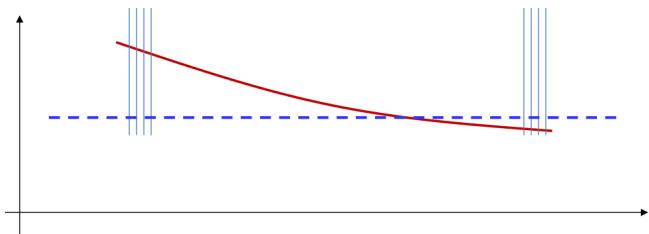
## Caveats: order of presentation



- If we loop through the samples in the same order, we may get *cyclic* behavior

27

## Caveats: order of presentation



- If we loop through the samples in the same order, we may get *cyclic* behavior
- We must go through them *randomly* to get more convergent behavior

28

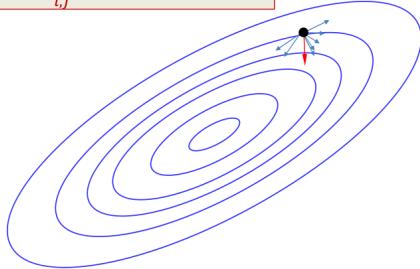
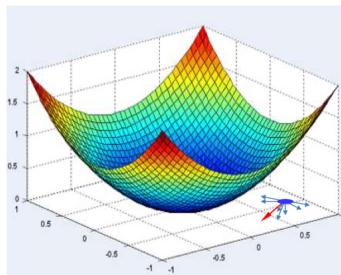
## Incremental Update: Stochastic Gradient Descent

- Given  $(X_1, d_1), (X_2, d_2), \dots, (X_T, d_T)$
- Initialize all weights  $W_1, W_2, \dots, W_K$
- Do:
  - Randomly permute  $(X_1, d_1), (X_2, d_2), \dots, (X_T, d_T)$
  - For all  $t = 1:T$ 
    - For every layer  $k$ :
      - Compute  $\nabla_{W_k} \text{Div}(Y_t, d_t)$
      - Update
$$W_k = W_k - \eta \nabla_{W_k} \text{Div}(Y_t, d_t)^T$$
- Until *Loss* has converged

33

## The expected behavior of the gradient

$$\frac{dE(\mathbf{W}^{(1)}, \mathbf{W}^{(2)}, \dots, \mathbf{W}^{(K)})}{d\mathbf{w}_{i,j}^{(K)}} = \frac{1}{T} \sum_t \frac{d\text{Div}(\mathbf{Y}(\mathbf{X}_t), \mathbf{d}_t; \mathbf{W}^{(1)}, \mathbf{W}^{(2)}, \dots, \mathbf{W}^{(K)})}{d\mathbf{w}_{i,j}^{(K)}}$$



- The individual training instances contribute different directions to the overall gradient
  - The final gradient points is the average of individual gradients
  - It points towards the *net* direction

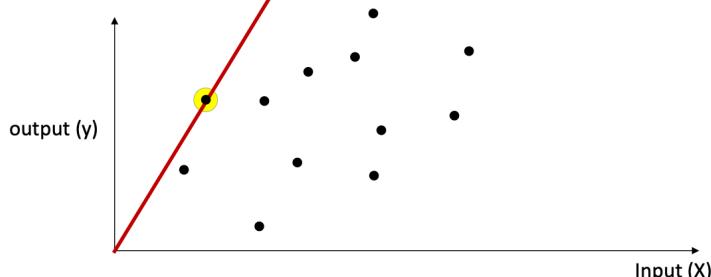
36

## When does it work

- What are the considerations?
- And how well does it work?

42

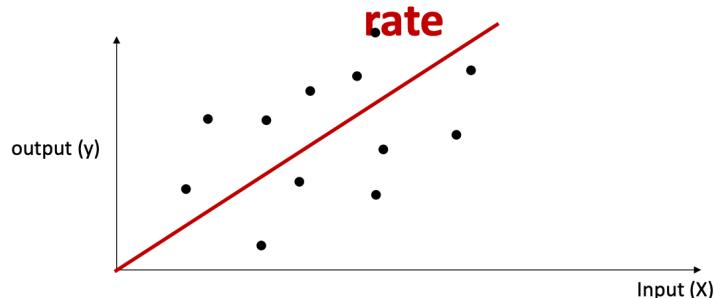
## Incremental learning runs the risk of always chasing the latest input



- Incremental learning:** Update the model to always minimize the error on the latest instance
  - It will never converge
  - Solution?**

48

## Incremental learning caveat: learning rate



- Incremental learning:** Update the model to always minimize the error on the latest instance
  - Caveat:** We must *shrink* the learning rate with iterations for convergence
    - Correction for individual instances with the eventual minuscule learning rates will not modify the function

55

## Incremental Update: Stochastic Gradient Descent

- Given  $(X_1, d_1), (X_2, d_2), \dots, (X_T, d_T)$
- Initialize all weights  $W_1, W_2, \dots, W_K; j = 0$
- Do:
  - Randomly permute  $(X_1, d_1), (X_2, d_2), \dots, (X_T, d_T)$
  - For all  $t = 1:T$ 
    - $j = j + 1$  Randomize input order
    - For every layer  $k$ :
      - Compute  $\nabla_{W_k} \text{Div}(Y_t, d_t)$
      - Update  $W_k = W_k - \eta_j \nabla_{W_k} \text{Div}(Y_t, d_t)^T$  Learning rate reduces with  $j$
- Until  $\text{Loss}$  has converged

57

## SGD convergence

- SGD converges “almost surely” to a global or local minimum for most functions
  - Sufficient condition: step sizes follow the following conditions (Robbins and Munro 1951)
 
$$\sum_k \eta_k = \infty$$
  - Eventually the entire parameter space can be searched
 
$$\sum_k \eta_k^2 < \infty$$
  - The steps shrink
  - The fastest converging series that satisfies both above requirements is
 
$$\eta_k \propto \frac{1}{k}$$
    - This is the optimal rate of shrinking the step size for strongly convex functions
    - More generally, the learning rates are heuristically determined
- If the loss is convex, SGD converges to the optimal solution
- For non-convex losses SGD converges to a local minimum

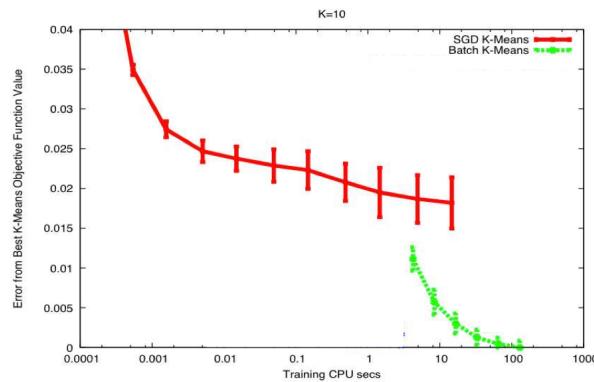
58

## Batch gradient convergence

- In contrast, using the batch update method, for *strongly convex* functions,
 
$$|f(W^{(k)}) - f(W^*)| < c^k |f(W^{(0)}) - f(W^*)|$$
  - Giving us the iterations to  $\epsilon$  convergence as  $O\left(\log\left(\frac{1}{\epsilon}\right)\right)$
- For generic convex functions, iterations to  $\epsilon$  convergence is  $O\left(\frac{1}{\epsilon}\right)$
- Batch gradients converge “**faster**”
  - But SGD performs  $T$  updates for every batch update

60

## SGD example



- A simpler problem: K-means
- Note: SGD converges faster
- But also has large variation between runs

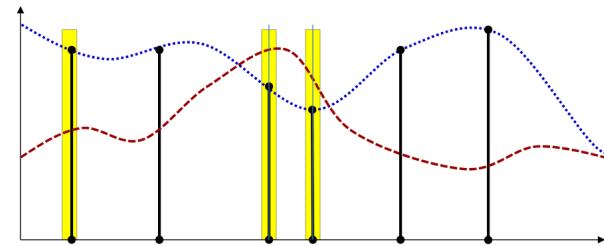
82

## SGD vs batch

- SGD uses the gradient from only one sample at a time, and is consequently high variance
- But also provides significantly quicker updates than batch
- Is there a good medium?

83

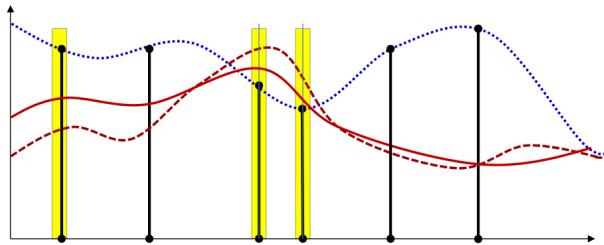
## Alternative: Mini-batch update



- Alternative: adjust the function at a small, randomly chosen subset of points
  - Keep adjustments small
  - If the subsets cover the training set, we will have adjusted the entire function
- As before, vary the subsets randomly in different passes through the training data

84

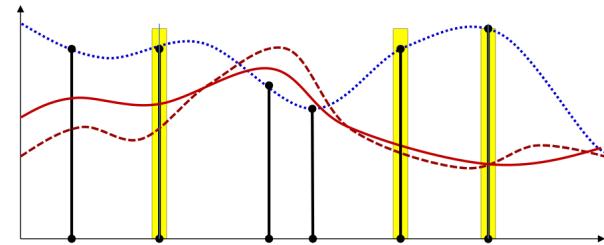
## Alternative: Mini-batch update



- Alternative: adjust the function at a small, randomly chosen subset of points
  - Keep adjustments small
  - If the subsets cover the training set, we will have adjusted the entire function
- As before, vary the subsets randomly in different passes through the training data

85

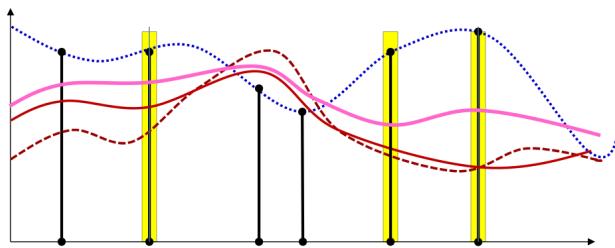
## Alternative: Mini-batch update



- Alternative: adjust the function at a small, randomly chosen subset of points
  - Keep adjustments small
  - If the subsets cover the training set, we will have adjusted the entire function
- As before, vary the subsets randomly in different passes through the training data

86

## Alternative: Mini-batch update



- Alternative: adjust the function at a small, randomly chosen subset of points
  - Keep adjustments small
  - If the subsets cover the training set, we will have adjusted the entire function
- As before, vary the subsets randomly in different passes through the training data

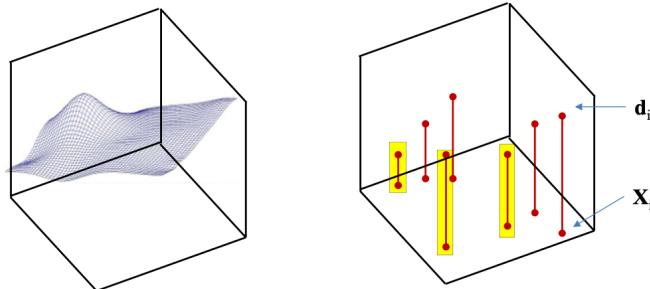
87

## Incremental Update: Mini-batch update

- Given  $(X_1, d_1), (X_2, d_2), \dots, (X_T, d_T)$
- Initialize all weights  $W_1, W_2, \dots, W_K; j = 0$
- Do:
  - Randomly permute  $(X_1, d_1), (X_2, d_2), \dots, (X_T, d_T)$
  - For  $t = 1:b:T$ 
    - $j = j + 1$
    - For every layer  $k$ :
      - $\Delta W_k = 0$
    - For  $t' = t : t+b-1$ 
      - For every layer  $k$ :
        - Compute  $\nabla_{W_k} \text{Div}(Y_t, d_t)$
        - $\Delta W_k = \Delta W_k + \frac{1}{b} \nabla_{W_k} \text{Div}(Y_t, d_t)^T$
    - Update
      - For every layer  $k$ :
 
$$W_k = W_k - \eta_j \Delta W_k$$
  - Until  $\text{Err}$  has converged

89

## Mini Batches

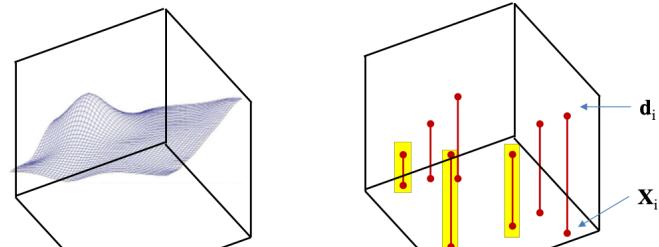


- Mini-batch updates compute and minimize a *batch loss*
- $$\text{MiniBatchLoss}(W) = \frac{1}{b} \sum_{i=1}^b \text{div}(f(X_i; W), d_i)$$
- The *expected value* of the batch loss is also the *expected divergence*

$$E[\text{MiniBatchLoss}(W)] = E[\text{div}(f(X; W), g(X))]$$

90

## Mini Batches



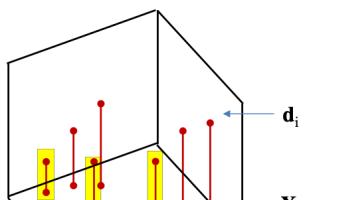
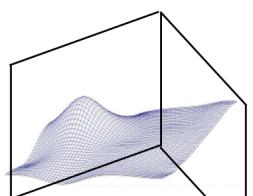
The minibatch loss is also an unbiased estimate of the expected loss

- Mini-batch updates compute and minimize a *batch loss*
- $$\text{MiniBatchLoss}(W) = \frac{1}{b} \sum_{i=1}^b \text{div}(f(X_i; W), d_i)$$
- The *expected value* of the batch loss is also the *expected divergence*

$$E[\text{MiniBatchLoss}(W)] = E[\text{div}(f(X; W), g(X))]$$

91

## Mini Batches



The variance of the minibatch loss:  $\text{var}(\text{BatchLoss}) = 1/b \text{ var}(\text{div})$   
This will be much smaller than the variance of the sample error in SGD

The minibatch loss is also an unbiased estimate of the expected error

- Mini-batch updates compute and minimize a *batch loss*

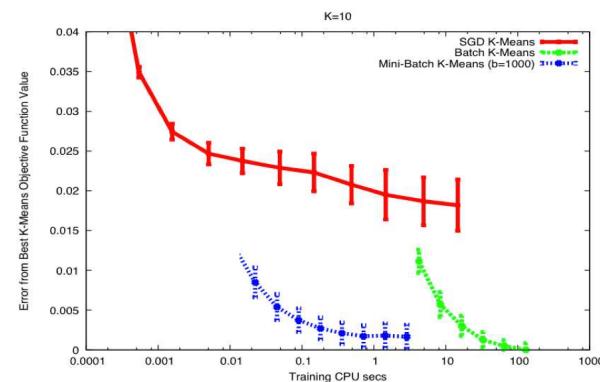
$$\text{MiniBatchLoss}(W) = \frac{1}{b} \sum_{i=1}^b \text{div}(f(X_i; W), d_i)$$

- The *expected value* of the batch loss is also the *expected divergence*

$$E[\text{MiniBatchLoss}(W)] = E[\text{div}(f(X; W), g(X))]$$

92

## SGD example



- Mini-batch performs comparably to batch training on this simple problem
  - But converges orders of magnitude faster

94

## Training and minibatches

- In practice, training is usually performed using mini-batches
  - The mini-batch size is generally set to the largest that your hardware will support (in memory) without compromising overall compute time
    - Larger minibatches = less variance
    - Larger minibatches = few updates per epoch
- Convergence depends on learning rate
  - Simple technique: fix learning rate until the error plateaus, then reduce learning rate by a fixed factor (e.g. 10)
  - **Advanced methods:** Adaptive updates, where the learning rate is itself determined as part of the estimation

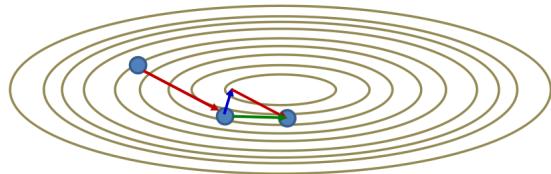
96

## Training and minibatches

- Convergence depends on learning rate
  - Simple technique: fix learning rate until the error plateaus, then reduce learning rate by a fixed factor (e.g. 10)
  - **Advanced methods:** Adaptive updates, where the learning rate is itself determined as part of the estimation

100

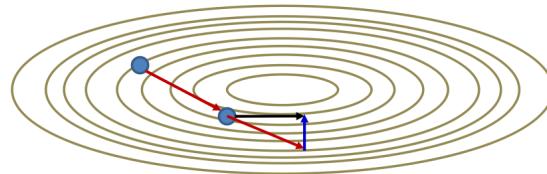
## Recall: Momentum Update



- The momentum method
$$\Delta W^{(k)} = \beta \Delta W^{(k-1)} - \eta \nabla_W Loss(W^{(k-1)})^T$$
- At any iteration, to compute the current step:
  - First compute the gradient step at the current location
  - Then add in the scaled *previous* step
    - Which is actually a running average
  - To get the final step

106

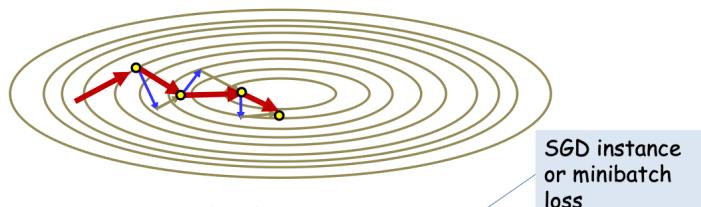
## Nestorov's Accelerated Gradient



- Change the order of operations
- At any iteration, to compute the current step:
  - First extend the previous step
  - Then compute the gradient step at the resultant position
  - Add the two to obtain the final step

111

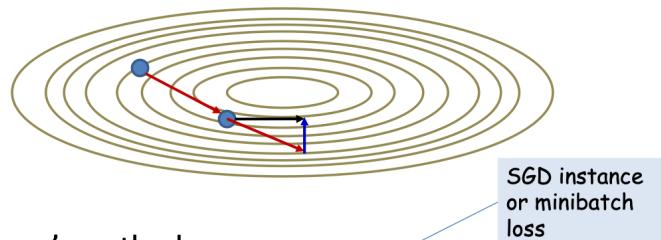
## Momentum and incremental updates



- The momentum method
$$\Delta W^{(k)} = \beta \Delta W^{(k-1)} - \eta \nabla_W Loss(W^{(k-1)})^T$$
- Incremental SGD and mini-batch gradients tend to have high variance
- Momentum smooths out the variations
  - Smoother and faster convergence

114

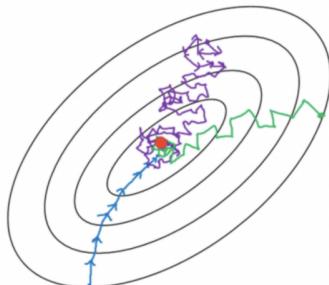
## Nestorov's Accelerated Gradient



- Nestorov's method
$$\Delta W^{(k)} = \beta \Delta W^{(k-1)} - \eta \nabla_W Loss(W^{(k-1)} + \beta \Delta W^{(k-1)})^T$$
$$W^{(k)} = W^{(k-1)} + \Delta W^{(k)}$$

117

# Momentum update



— Batch gradient descent  
— Mini-batch gradient Descent  
— Stochastic gradient descent

momentum provides faster convergence with sgd and mini-batch gradient descent

49

## The other term in the update

- Standard gradient descent rule

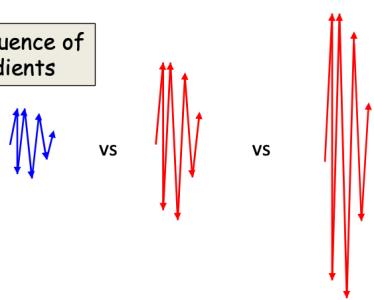
$$W \leftarrow W - \eta \nabla_W L(W)$$

- Gradient descent invokes two terms for updates
  - The derivative
  - and the learning rate
- Momentum methods fix this term to reduce unstable oscillation
- What about this term?

121

## Adjusting the learning rate

Sequence of gradients

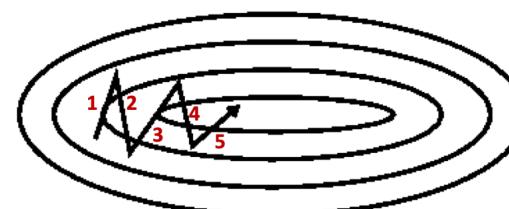


With separate learning rates in each direction, which should have the lowest learning rate in the vertical direction?

- Have separate learning rates for each component
- Directions in which the derivatives swing more should likely have lower learning rates
  - Is likely indicative of more wildly swinging behavior
- Directions of greater swing are indicated by total movement
  - Direction of greater movement should have lower learning rate

122

## Smoothing the trajectory



Step	X component	Y component
1	1	+2.5
2	1	-3
3	2	+2.5
4	1	-2
5	1.5	1.5

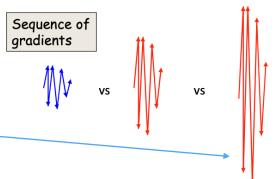
- Observation: Steps in “oscillatory” directions show large total movement
  - In the example, total motion in the vertical direction is much greater than in the horizontal direction
- Solution: Lower learning rate in the vertical direction than in the horizontal direction
  - Based on total motion
  - As quantified by RMS value

123

# Adagrad

- Idea: per-parameter learning rates

- ✓ steep directions
  - large “total movement”
  - want lower learning rates
- ✓ flat directions
  - want larger learning rates
- ✓ keep a historical sum of squares in each dimension
- ✓ squares capture total motion on both sides of oscillation



```
dw_squared = 0
for i in range(n_iter):      step size decays to zero over time
    dw = gradient(w)          (denominator grows)
    dw_squared += dw * dw
    w = w - lr * dw / sqrt(dw_squared + eps)
```

# RMS Prop

- Notation:

- Formulae are *by parameter*
- Derivative of loss w.r.t any individual parameter  $w$  is shown as  $\partial_w D$ 
  - Batch or minibatch loss, or individual divergence for batch/minibatch/SGD
- The *squared* derivative is  $\partial_w^2 D = (\partial_w D)^2$ 
  - Short-hand notation represents the squared derivative, not the second derivative
- The *mean squared* derivative is a running estimate of the average squared derivative. We will show this as  $E[\partial_w^2 D]$

- Modified update rule: We want to

- scale down learning rates for terms with large mean squared derivatives
- scale up learning rates for terms with small mean squared derivatives

124

# RMS Prop

- This is a variant on the *basic* mini-batch SGD algorithm
- Procedure:**
  - Maintain a running estimate of the mean squared value of derivatives for each parameter
  - Scale learning rate of the parameter by the *inverse of the root mean squared* derivative

$$E[\partial_w^2 D]_k = \gamma E[\partial_w^2 D]_{k-1} + (1 - \gamma)(\partial_w^2 D)_k$$

$$w_{k+1} = w_k - \frac{\eta}{\sqrt{E[\partial_w^2 D]_k + \epsilon}} \partial_w D$$

125

# From Adagrad to RMSProp

## Adagrad

```
dw_squared = 0
for i in range(n_iter):      step size decays to zero over time
    dw = gradient(w)
    dw_squared += dw * dw
    w = w - lr * dw / sqrt(dw_squared + eps)
```

## RMSProp

```
dw_squared = 0
for i in range(n_iter):
    dw = gradient(w)
    dw_squared = dr * dw_squared + (1-dr) * dw * dw
    w = w - lr * dw / sqrt(dw_squared + eps)
```

56

## All the terms in gradient descent

- Standard gradient descent rule

$$W \leftarrow W - \eta \nabla_W L(W)$$

- RMSprop only adapts the learning rate
  - by total movement
- Momentum only smooths the gradient
- How about combining both?

129

## Adam

- Removing bias

✓ as moments are initialized to zero, first update may be very large (small second\_moment)

```
first_moment = 0
second_moment = 0
for i in range(n_iter):
    dw = gradient(w)
    first_moment = beta1 * first_moment + (1-beta1) * dw
    second_moment = beta2 * second_moment + (1-beta2) * dw * dw

    first = first_moment / (1 - beta1** (i+1))      denominators approach 1
    second = second_moment / (1 - beta2** (i+1))      as i grows

    w = w - lr * first / (sqrt(second) + eps)
```

## Adam

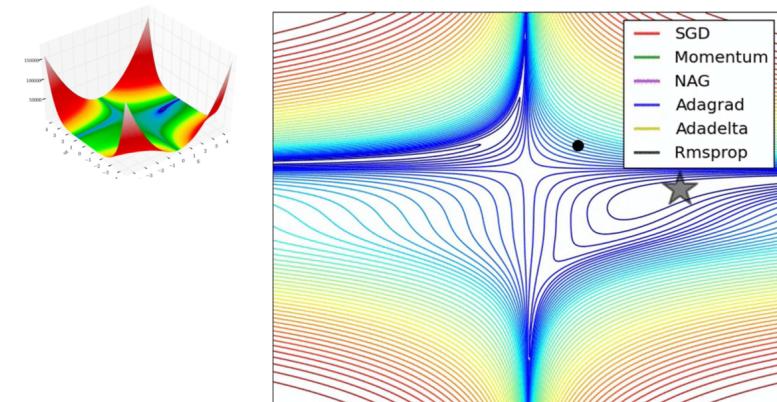
- Previous methods

✓ RMSProp adapts the learning rate per-parameter  
✓ Momentum smooths the gradient

- Adam (adaptive moments) combines both
  - ✓ moving averages for both the gradients (momentum) and the squared gradients (RMSProp)

```
first_moment = 0
second_moment = 0
for i in range(n_iter):
    dw = gradient(w)
    first_moment = beta1 * first_moment + (1-beta1) * dw
    second_moment = beta2 * second_moment + (1-beta2) * dw * dw
    w = w - lr * first_moment / (sqrt(second_moment) + eps)
```

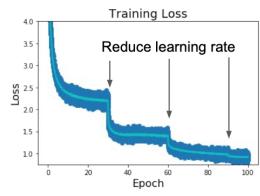
## Visualizing the optimizers: Beale's Function



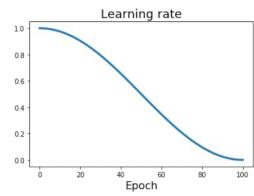
• <http://www.denizyuret.com/2015/03/alec-radfords-animations-for.html>

136

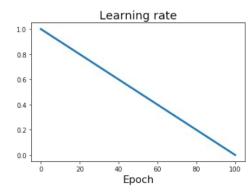
## LR schedules (decaying over time)



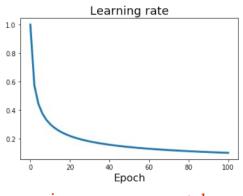
reduce LR at fixed intervals



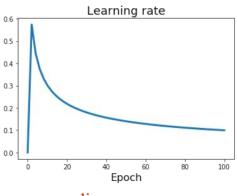
use cosine annealing



use linear decay



use inverse square root decay



use linear warmup