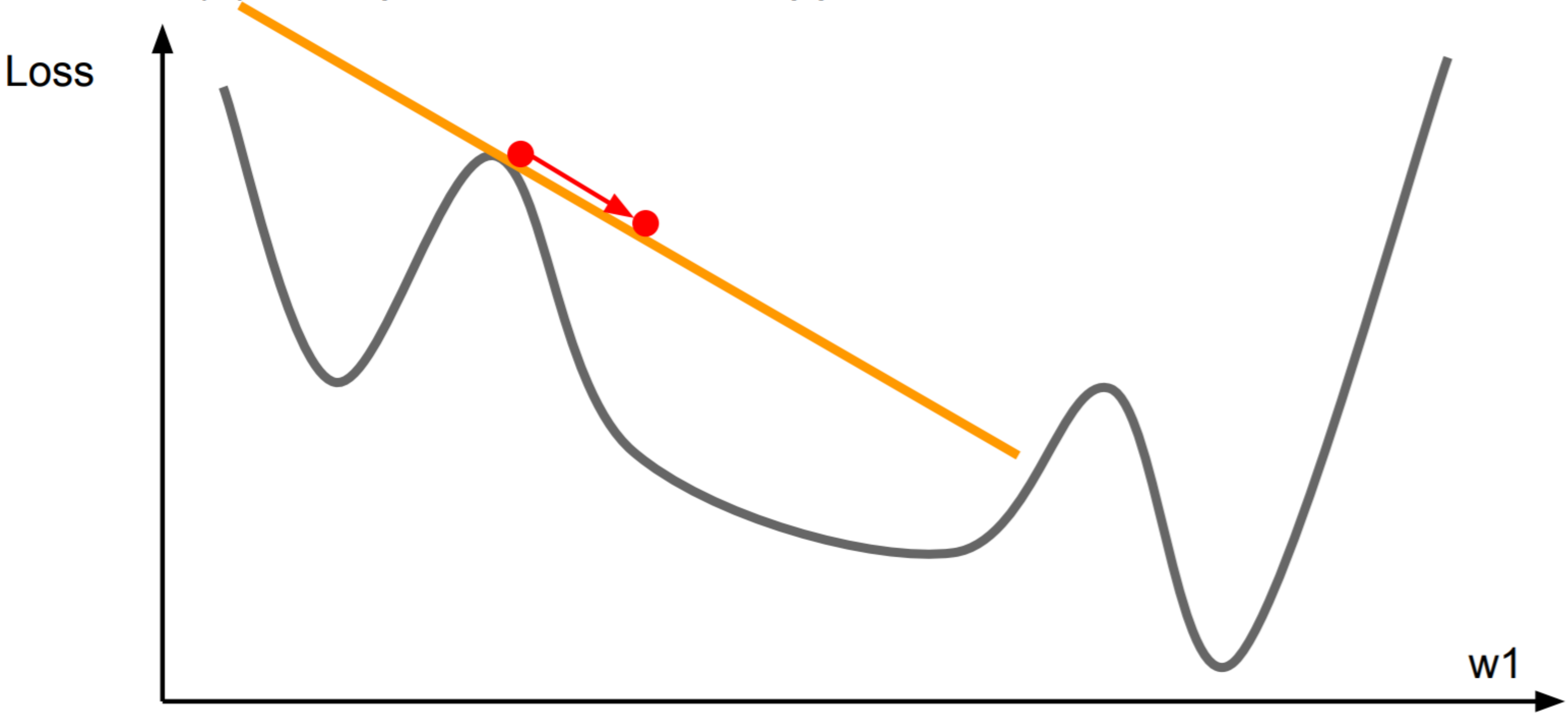
Training Neural Networks Optimization II

M. Soleymani
Sharif University of Technology
Spring 2024

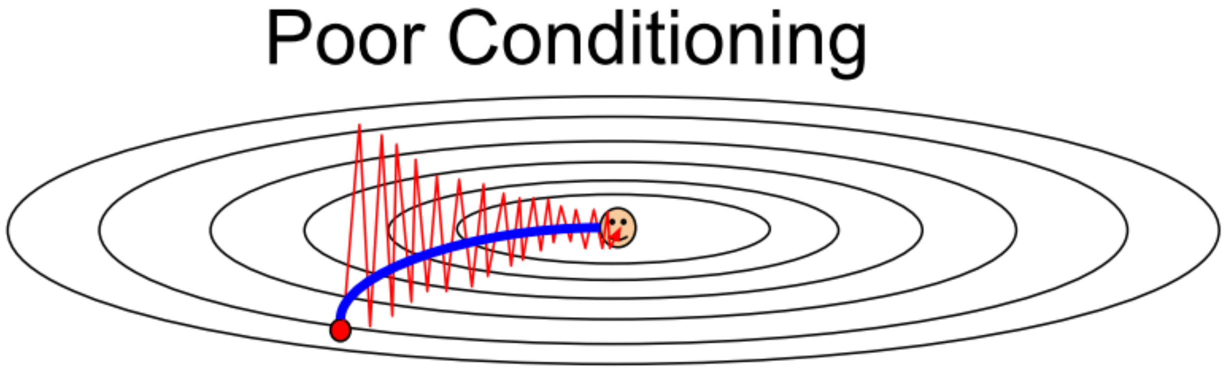
Most slides have been adapted from Bhiksha Raj, 11-785, CMU
and some from Fei Fei Li et. al, cs231n, Stanford

Recap: First-order optimization

(1) Use gradient form linear approximation
(2) Step to minimize the approximation

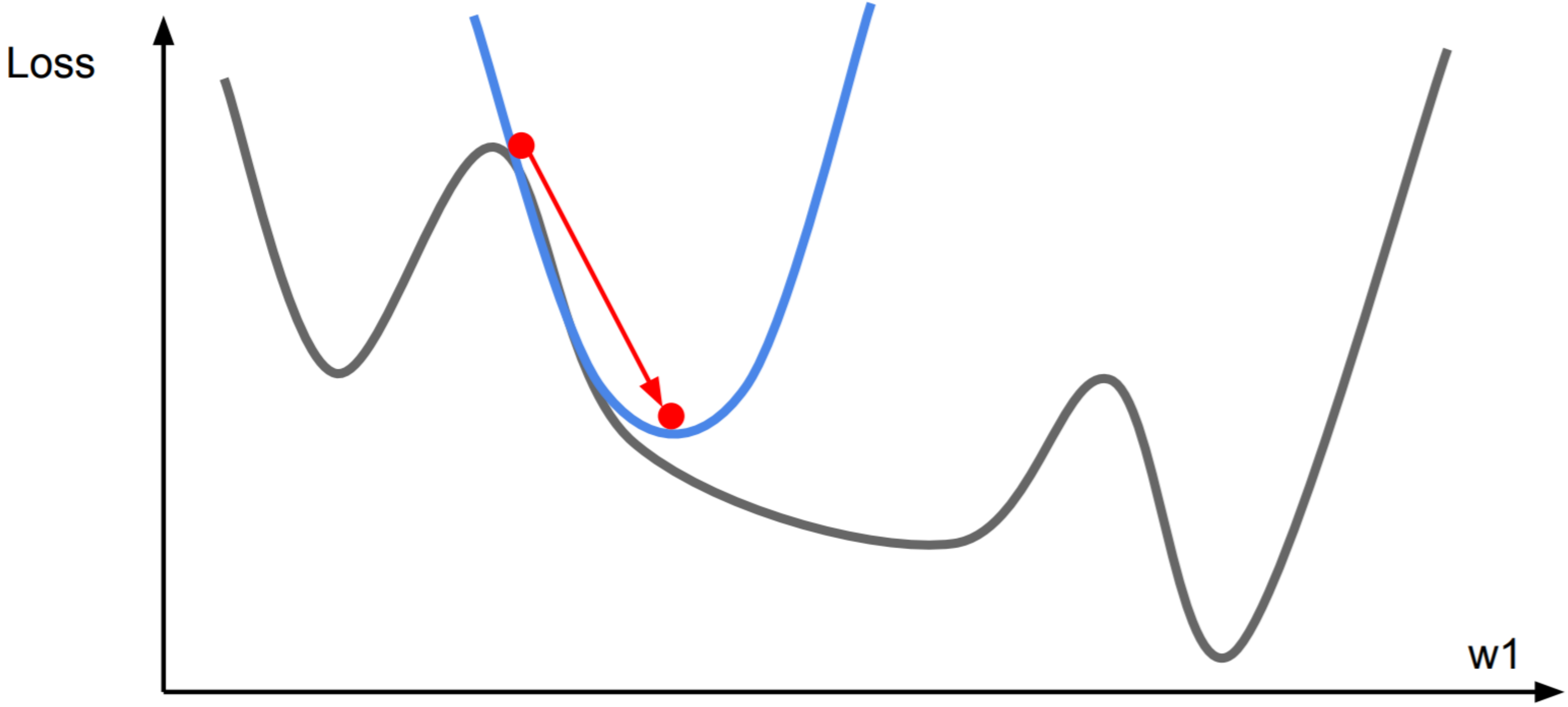


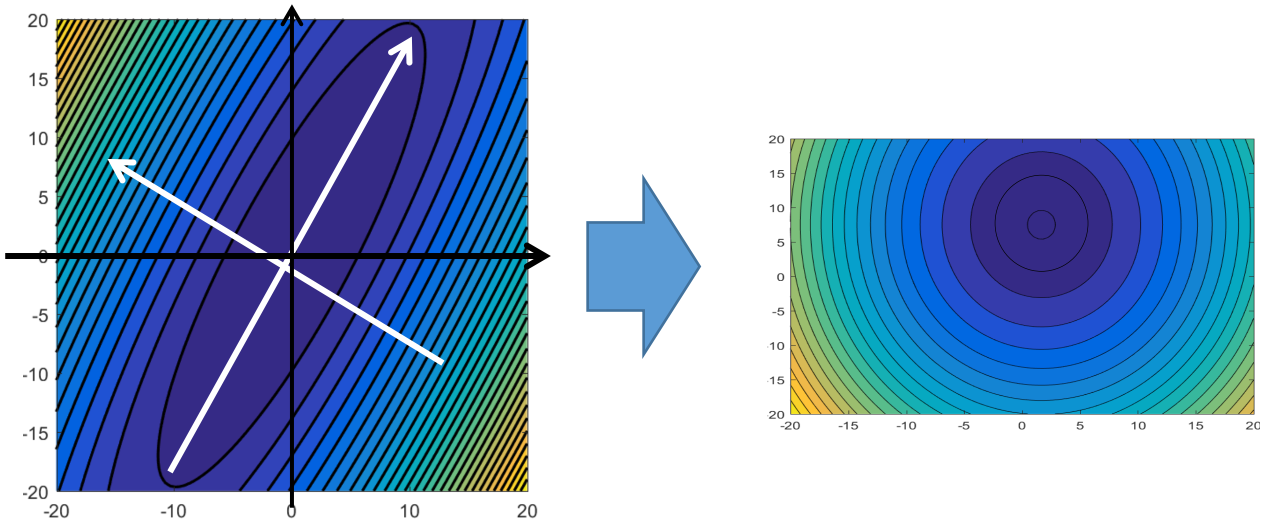
Recap: Poor conditioning problem



Recap: Second-order optimization

(1) Use gradient **and Hessian** to form **quadratic** approximation
(2) Step to the **minima** of the approximation





$$\mathbf{w}^{(k+1)} = \mathbf{w}^{(k)} - \eta \mathbf{H}^{-1} \nabla_{\mathbf{w}} E\left(\mathbf{w}^{(k)}\right)$$

$$\eta = 1$$

Recap: Second-order optimization

- Taylor expansion (second-order):

$$\boldsymbol{E}(\mathbf{w})$$

$$\approx \boldsymbol{E}\left(\mathbf{w}^{(k)}\right) + \nabla_{\mathbf{w}}\boldsymbol{E}\left(\mathbf{w}^{(k)}\right)^{T}\left(\mathbf{w} - \boldsymbol{w}^{(k)}\right) + \frac{1}{2}\left(\mathbf{w} - \boldsymbol{w}^{(k)}\right)^{T}\boldsymbol{H}_{E}\left(\boldsymbol{w}^{(k)}\right)\left(\mathbf{w} - \boldsymbol{w}^{(k)}\right) + \cdots$$

$$\mathbf{w}^{(k+1)} = \mathbf{w}^{(k)} - H_E\left(\mathbf{w}^{(k)}\right)^{-1}\nabla_{\mathbf{w}}E\left(\mathbf{w}^{(k)}\right)$$

No hyperparameters!
No learning rate!

Why is this bad for deep learning?

× Hessian has $O(n^2)$ elements
× Inverting takes $O(n^3)$
n = (Tens or Hundreds or … of) Millions

Recap: Second-order optimization: L-BFGS

- Quasi-Newton methods (BFGS most popular):
  - instead of inverting the Hessian (requiring $O(n^3)$), 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.
  - usually works very well in full batch, deterministic mode
    - i.e. work very well when you have a single, deterministic cost function
  - But 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.

Recap: Adaptive learning rates

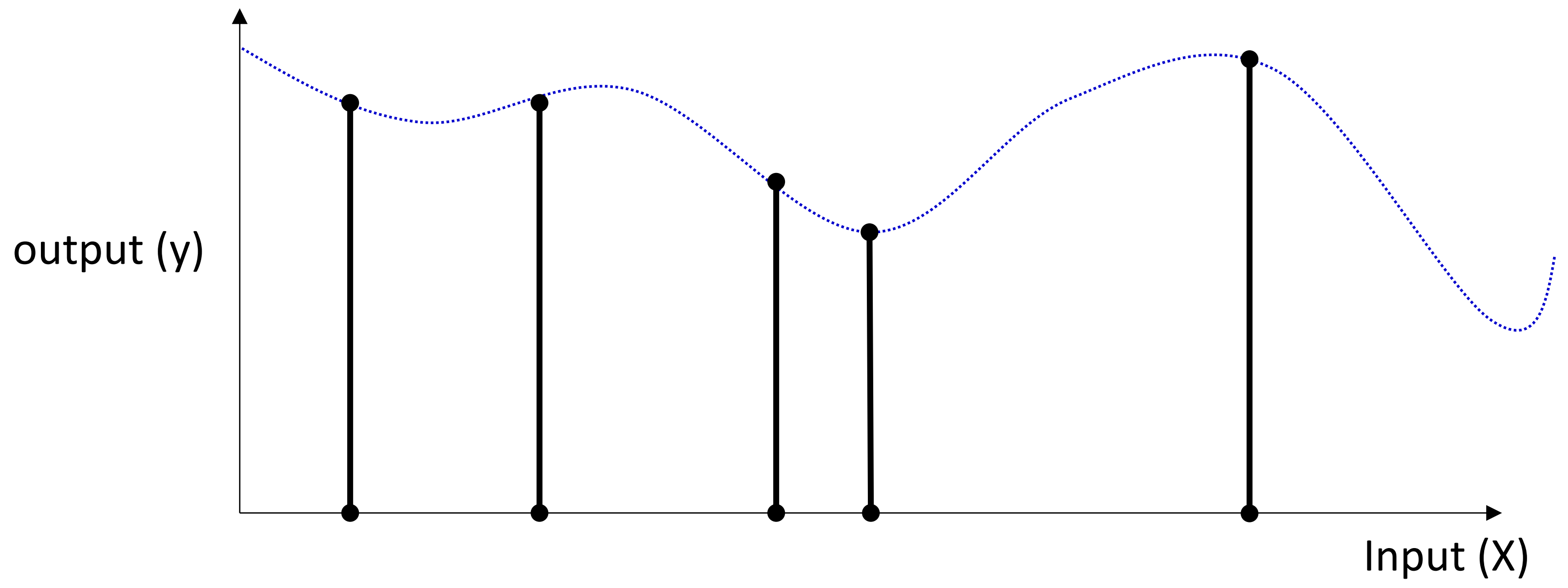
- ***Advanced methods***: Adaptive updates, where the learning rate of each parameter is itself adjusted as part of the estimation
  - RMS Prop, ADAM, …

In practice

- **Adam** is a good default choice in most cases
- **SGD** or **SGD+Momentum** can outperform Adam but may require more tuning of learning rate and decay schedule
- For some applications, the solutions found by adaptive methods **generalize worse** (often significantly worse) than SGD

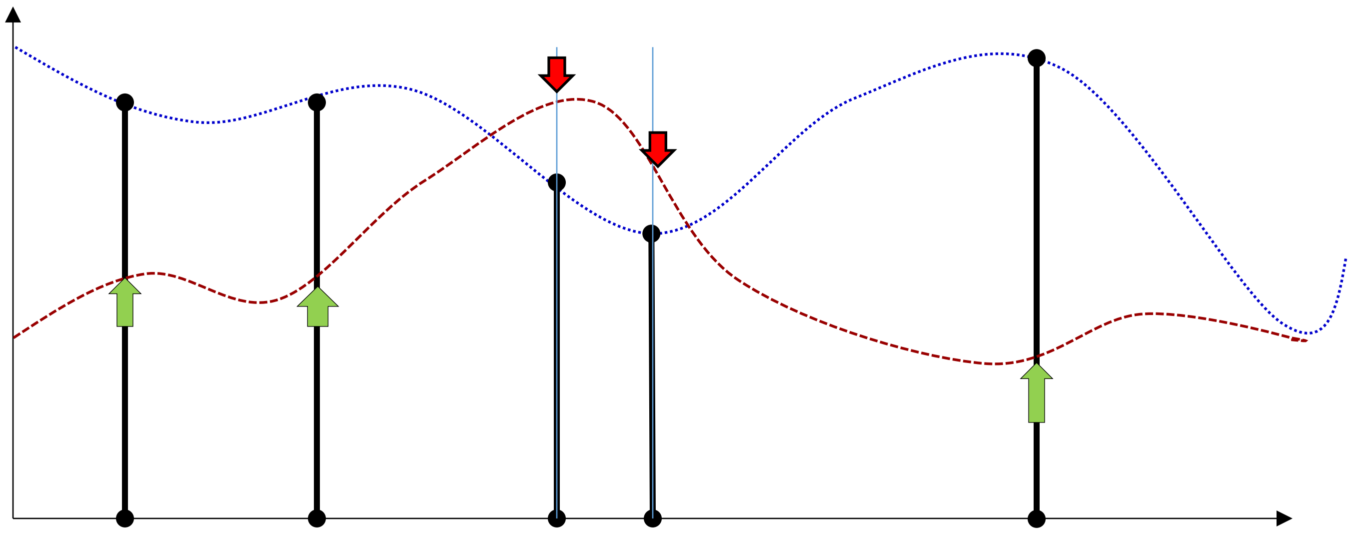
Winson et al., NeurIPS 2017

The training formulation



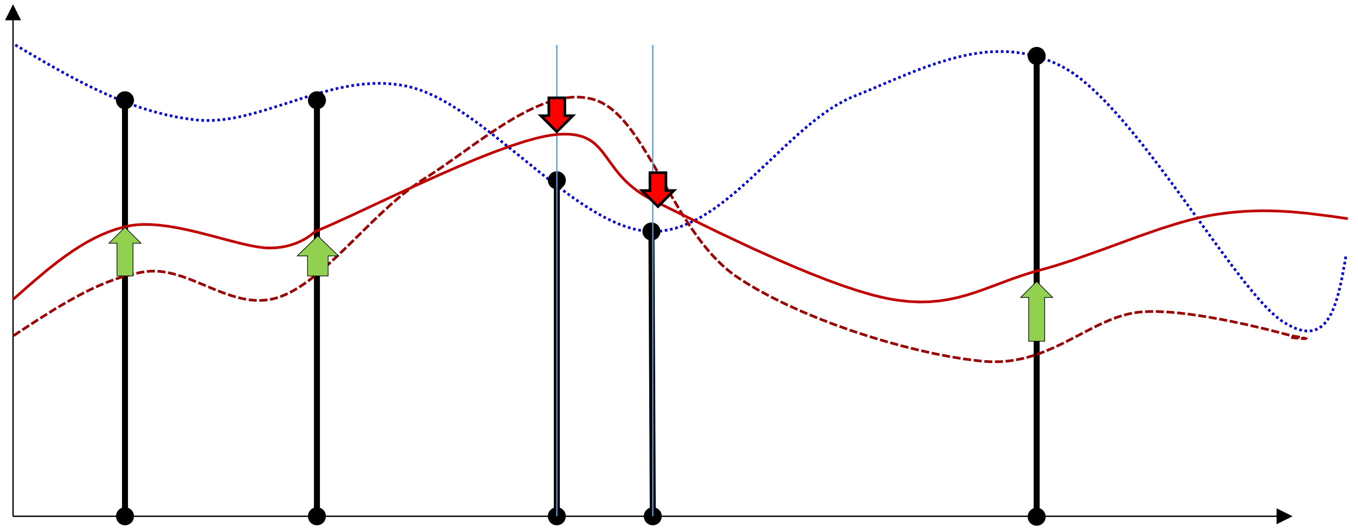
- Given input output pairs at a number of locations, estimate the entire function

Gradient descent



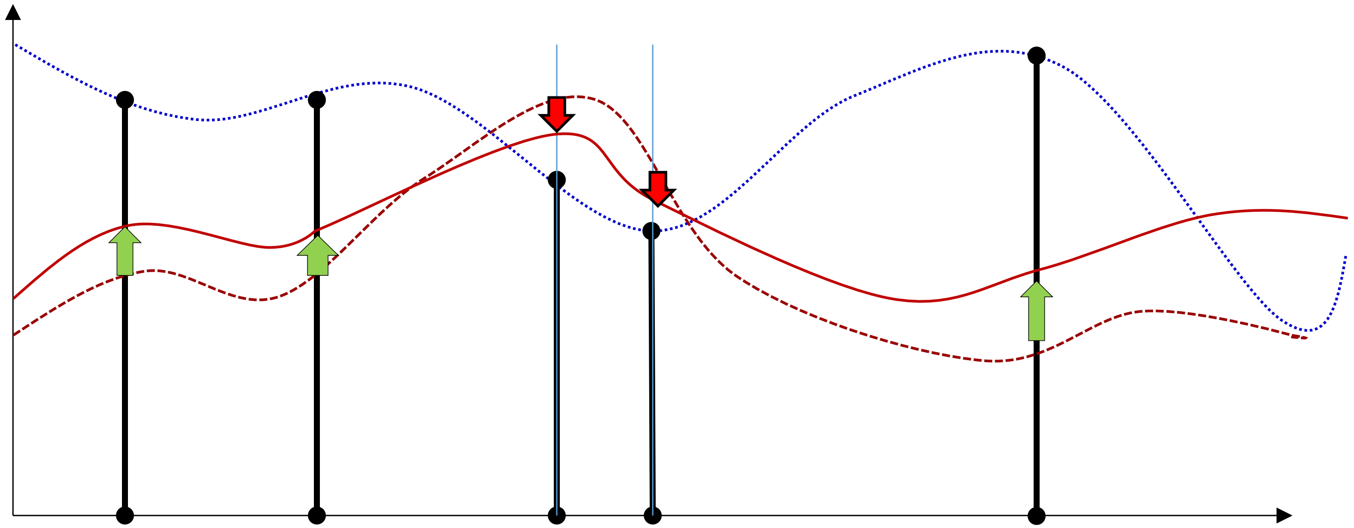
- Start with an initial function
- Adjust its value at *all* points to make the outputs closer to the required value
  - Gradient descent adjusts parameters to adjust the function value at *all* points
  - Repeat this iteratively until we get arbitrarily close to the target function at the training points

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

Effect of number of samples

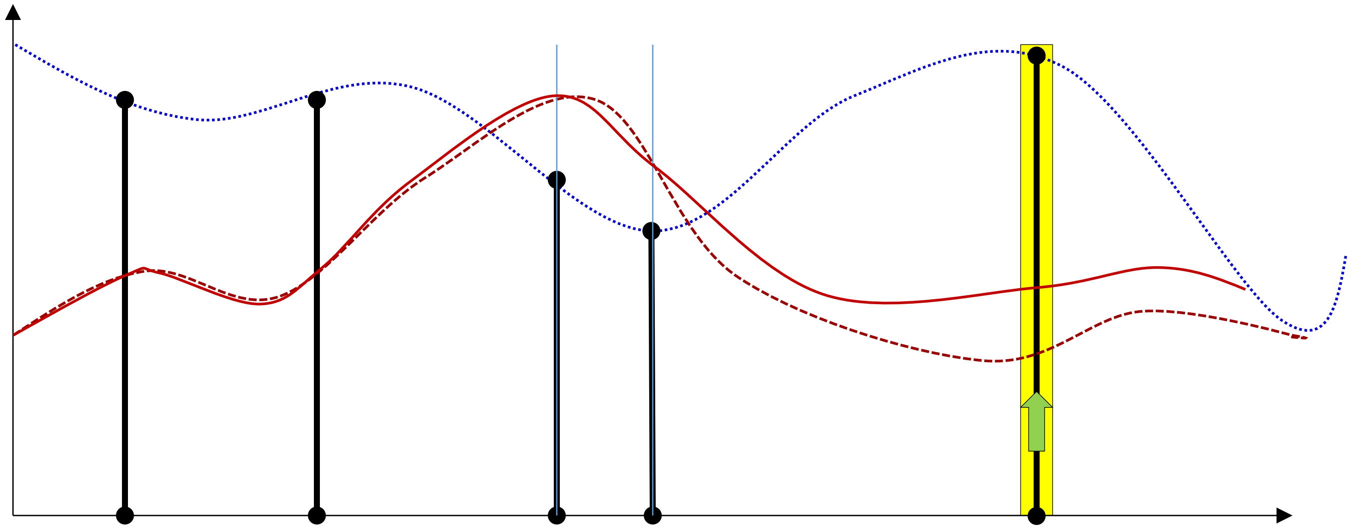


- Problem with conventional gradient descent: we try to simultaneously adjust the function at *all* training points
  - We must process *all* training points before making a single adjustment
  - "Batch" update

Incremental Update: Stochastic Gradient Descent

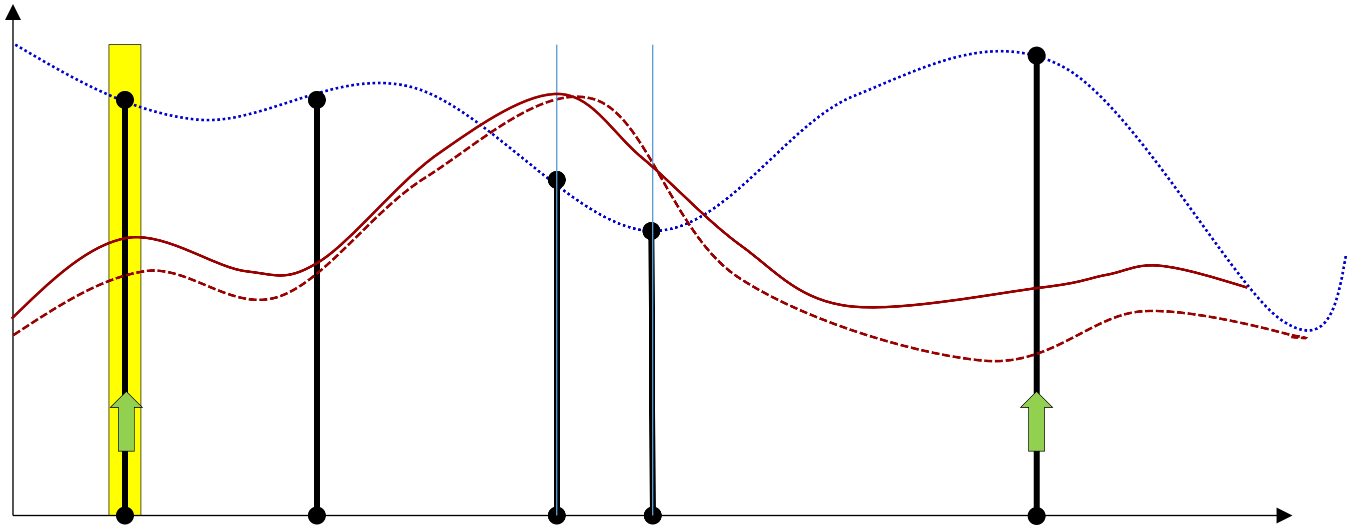
- Given $\left(\boldsymbol{x}^{(1)}, \boldsymbol{y}^{(1)}\right), \left(\boldsymbol{x}^{(2)}, \boldsymbol{y}^{(2)}\right), \ldots, \left(\boldsymbol{x}^{(N)}, \boldsymbol{y}^{(N)}\right)$
- Initialize all weights
- Do:
  - Randomly permute data
  - For all $n = 1: N$
    - Update. $W = W - \eta \nabla_W loss(\boldsymbol{o}^{(\boldsymbol{n})}, \boldsymbol{y}^{(\boldsymbol{n})})$
- Until convergence

Alternative: Incremental update



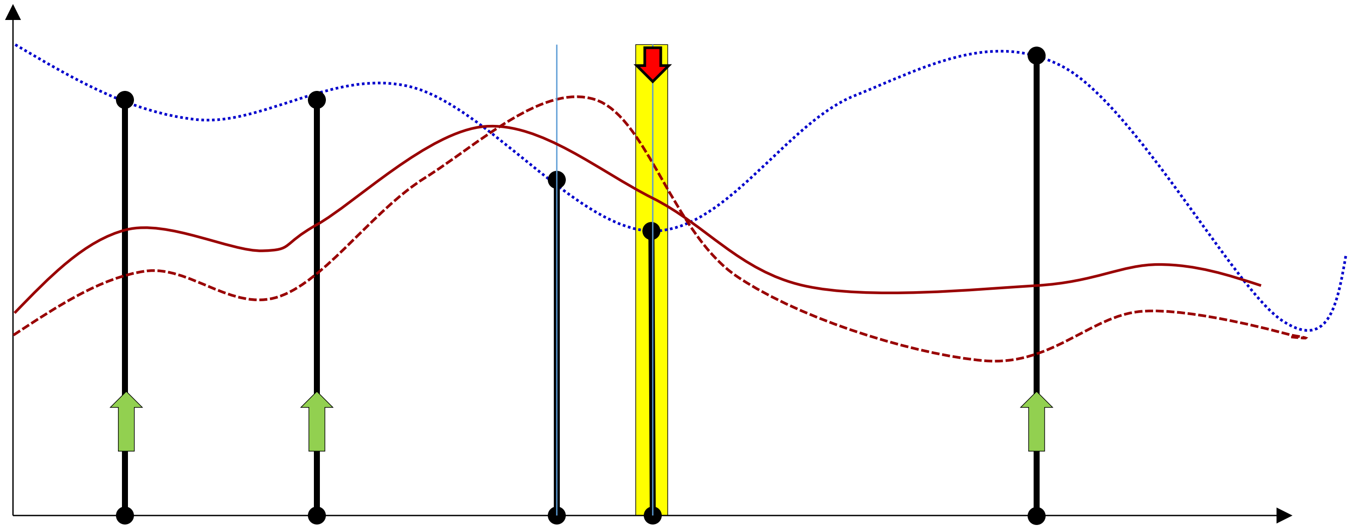
- Alternative: adjust the function at one training point at a time
  - Keep adjustments small

Alternative: Incremental update



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

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

Story so far

- In any gradient descent optimization, presenting training instances incrementally can be more effective than presenting them all at once
  - Provided training instances must be presented in random order
  - "**Stochastic Gradient Descent**"

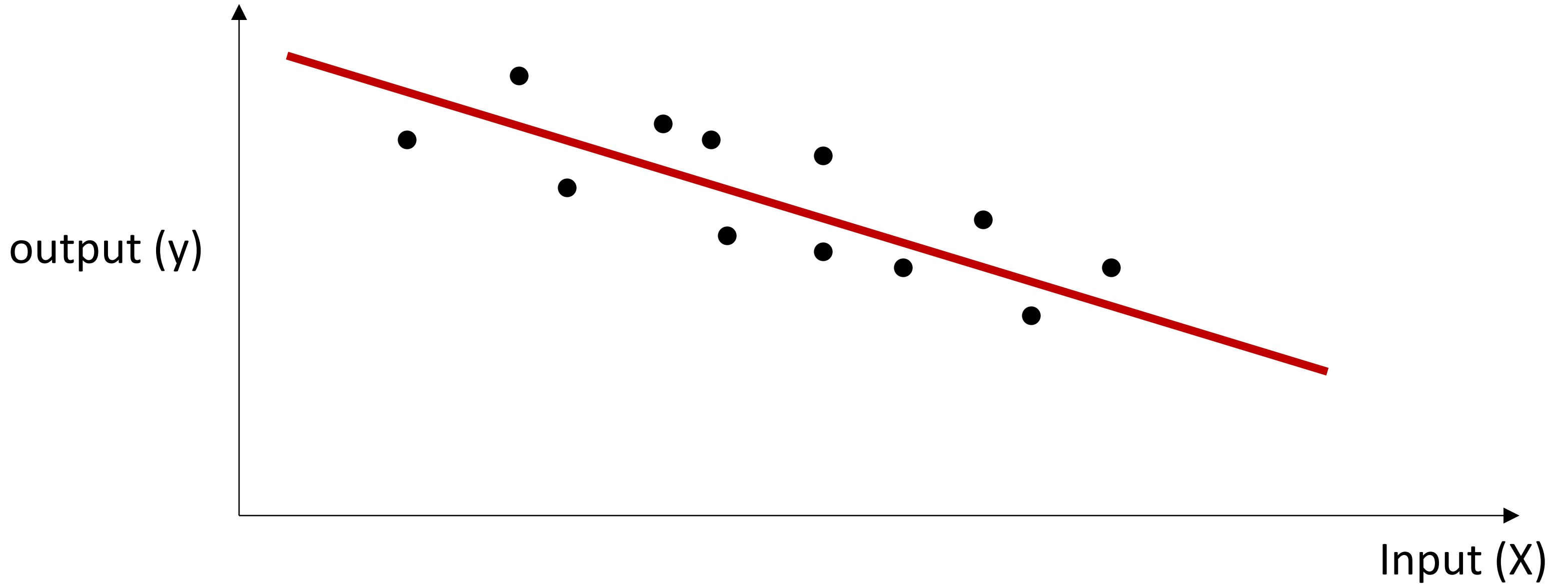
- This also holds for training neural networks

Explanations and restrictions

- So why does this process of incremental updates work?

- Under what conditions?

Caveats: learning rate



- Except in the case of a perfect fit, even an optimal overall fit will look incorrect to *individual* instances
  - Correcting the function for individual instances will lead to never-ending, non-convergent updates
  - We must *shrink* the learning rate with iterations to prevent this

Incremental Update: Stochastic Gradient Descent

- Given $\left(\boldsymbol{x}^{(1)}, \boldsymbol{y}^{(1)}\right), \left(\boldsymbol{x}^{(2)}, \boldsymbol{y}^{(2)}\right), \ldots, \left(\boldsymbol{x}^{(N)}, \boldsymbol{y}^{(N)}\right)$
- Initialize all weights
- $j = 0$
- Do:
  - Randomly permute data

    Randomize input order
  - For all $n = 1:N$

    An epoch

    Learning rate reduces with j
    - $j = j + 1$
    - Update. $W = W - \eta_j \nabla_W \boldsymbol{loss}(\boldsymbol{o}^{(n)}, \boldsymbol{y}^{(n)})$
- Until Err has converged

Stochastic Gradient Descent

- 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 N samples results in N updates of parameters

SGD convergence

- SGD converges “almost surely” to a global or local minimum for most functions
  - Sufficient condition: step sizes follow the following conditions

$$\sum_k \eta_k = \infty$$

    - Eventually the entire parameter space can be searched

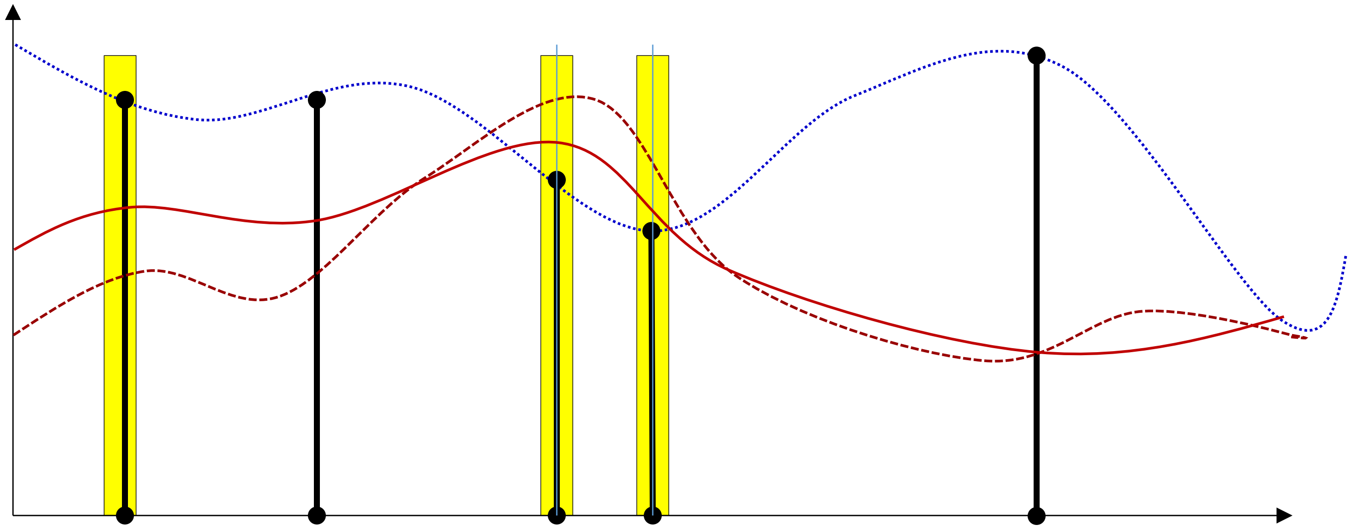
$$\sum_k \eta_k^2 < \infty$$

    - 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 stationary point

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
  - Shuffle first and reshuffle between epochs

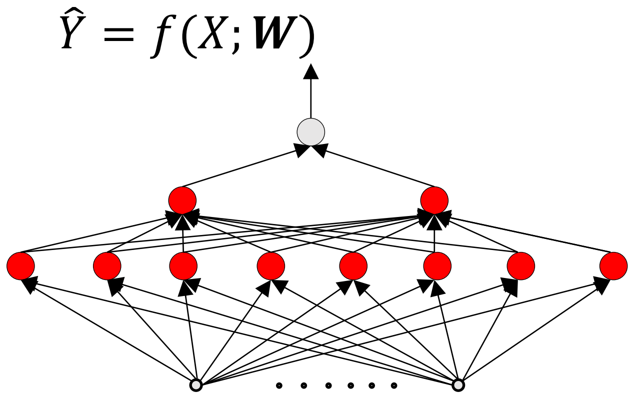
Mini-batch Gradient Descent

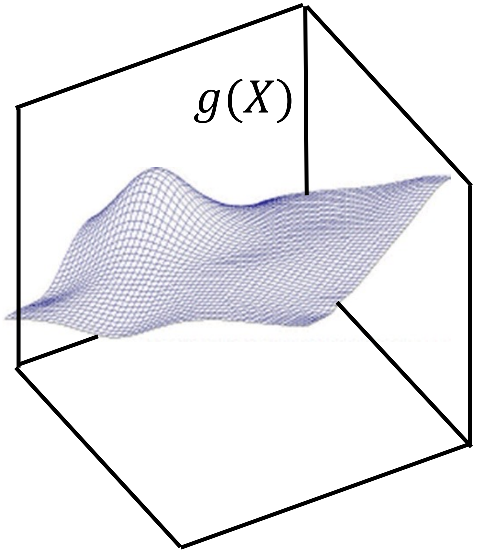
- Given $\left(\boldsymbol{x}^{(1)}, \boldsymbol{y}^{(1)}\right), \left(\boldsymbol{x}^{(2)}, \boldsymbol{y}^{(2)}\right), \ldots, \left(\boldsymbol{x}^{(N)}, \boldsymbol{y}^{(N)}\right)$
- Initialize all weights
- $j = 0$
- Do:
  - Randomly permute data
  - while $n < N$
    - $j = j + 1$
    - Update. $W = W - \eta_j \nabla_W \sum_{i=n}^{n+B-1} loss\left(\boldsymbol{o}^{(i)}, \boldsymbol{y}^{(i)}\right)$
    - $n = n + B$
- Until Err has converged

Randomize input order

An epoch

Recall: Modelling a function





- To learn a network $f(X; \boldsymbol{W})$ to model a function $g(X)$, it is ideal to minimize the *expected loss*

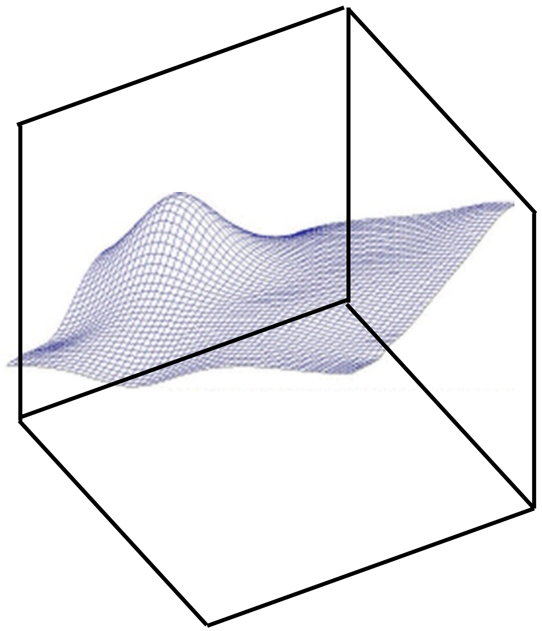
$$E\left[loss\big(f(X; W), g(X)\big)\right]$$

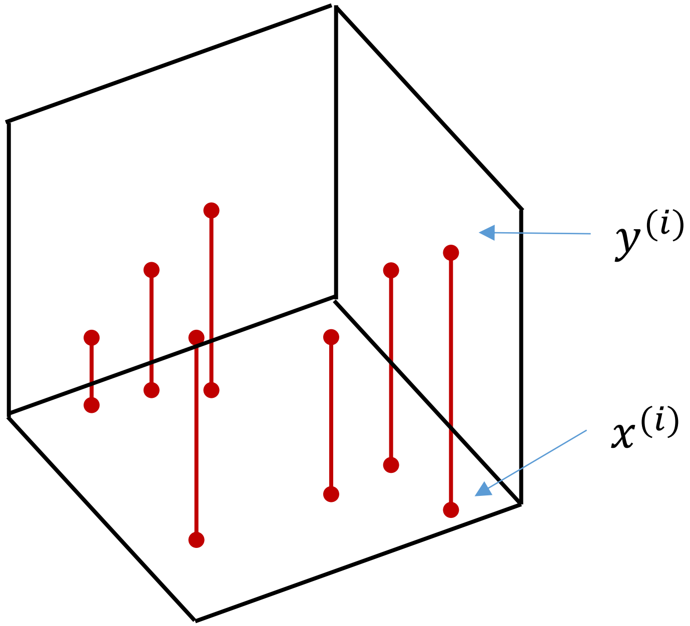
$$= \int_X loss\big(f(X; W), g(X)\big) P(X) dX$$

$$\boldsymbol{W}^* = \underset{W}{\operatorname{argmin}}\, E\left[loss\big(f(X; W), g(X)\big)\right]$$

minimizes the expected error

Recall: The *Empirical* risk





- In practice, we minimize the *empirical error*

$$Err(W) = \frac{1}{N}\sum_{i=1}^{N} loss\left(f\left(x^{(i)};W\right), y^{(i)}\right)$$

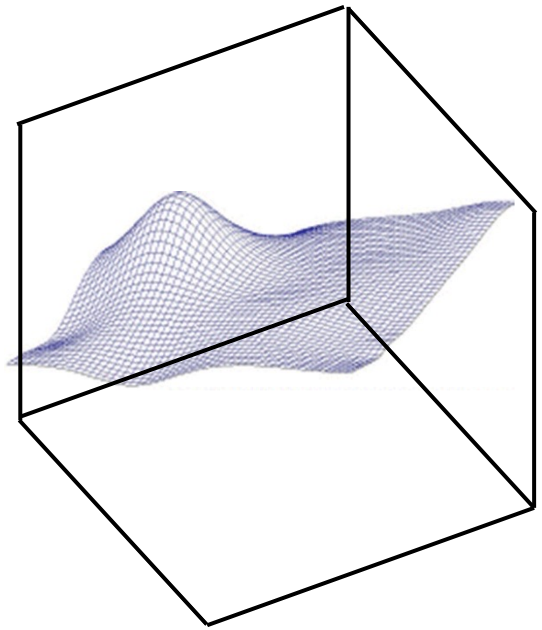
$$\widehat{\boldsymbol{W}} = \underset{W}{\operatorname{argmin}}\, Err\big(f(X;W), g(X)\big)$$

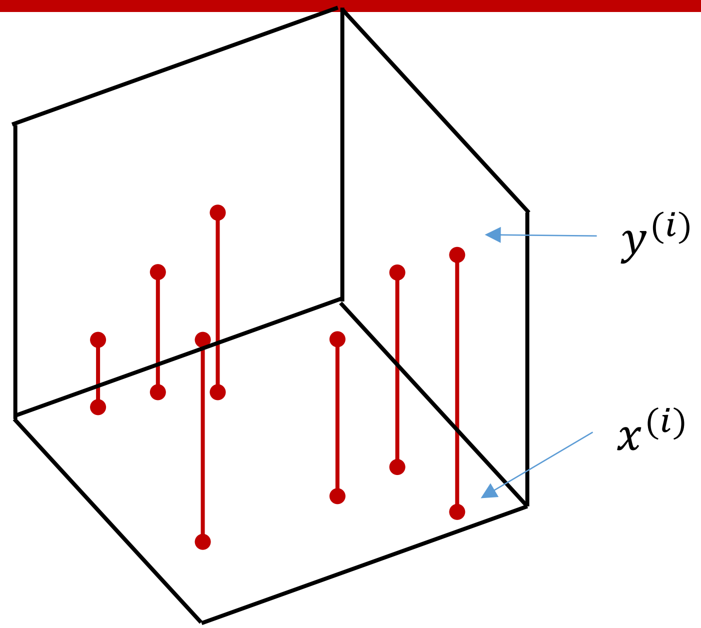
minimizes the empirical error

- The *expected value* of the *empirical error* is actually the *expected loss*

$$E[Err(W)] = E\left[loss\big(f(X;W), g(X)\big)\right]$$

The *Empirical* risk





The empirical error is an *unbiased* estimate of the expected error

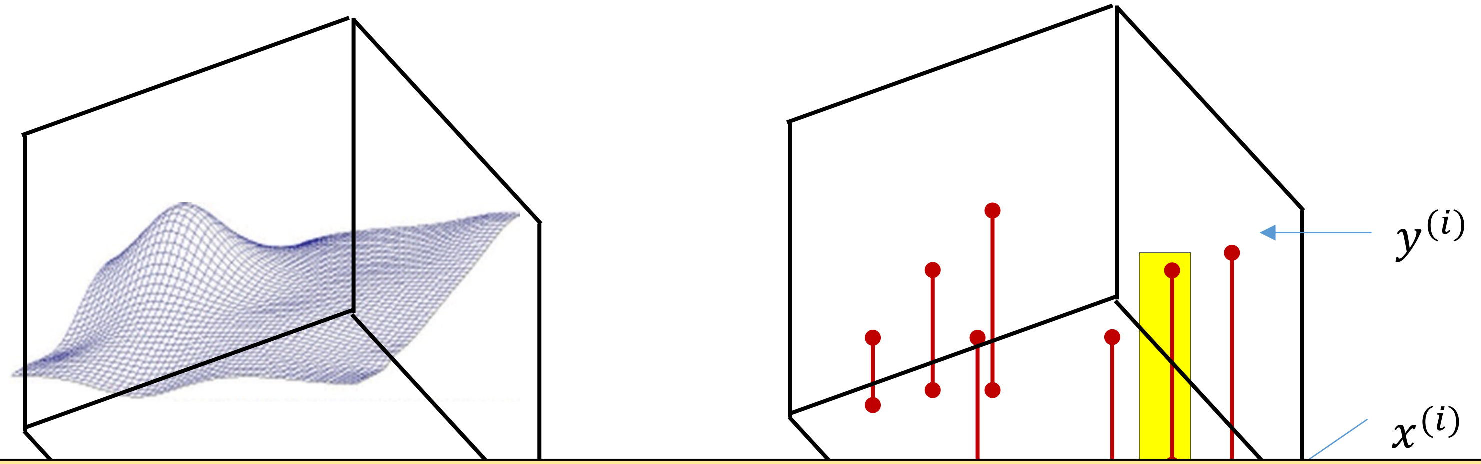
$$Err\big(f(X;W), g(X)\big) = \frac{1}{N}\sum_{i=1}^{N} loss\big(f(x^{(i)};W), y^{(i)}\big)$$

$$\widehat{\boldsymbol{W}} = \underset{W}{\operatorname{argmin}}\, Err\big(f(X;W), g(X)\big)$$

- The *expected value* of the *empirical error* is actually the *expected loss*

$$E[Err(W)] = E\big[loss\big(f(X;W), g(X)\big)\big]$$

SGD



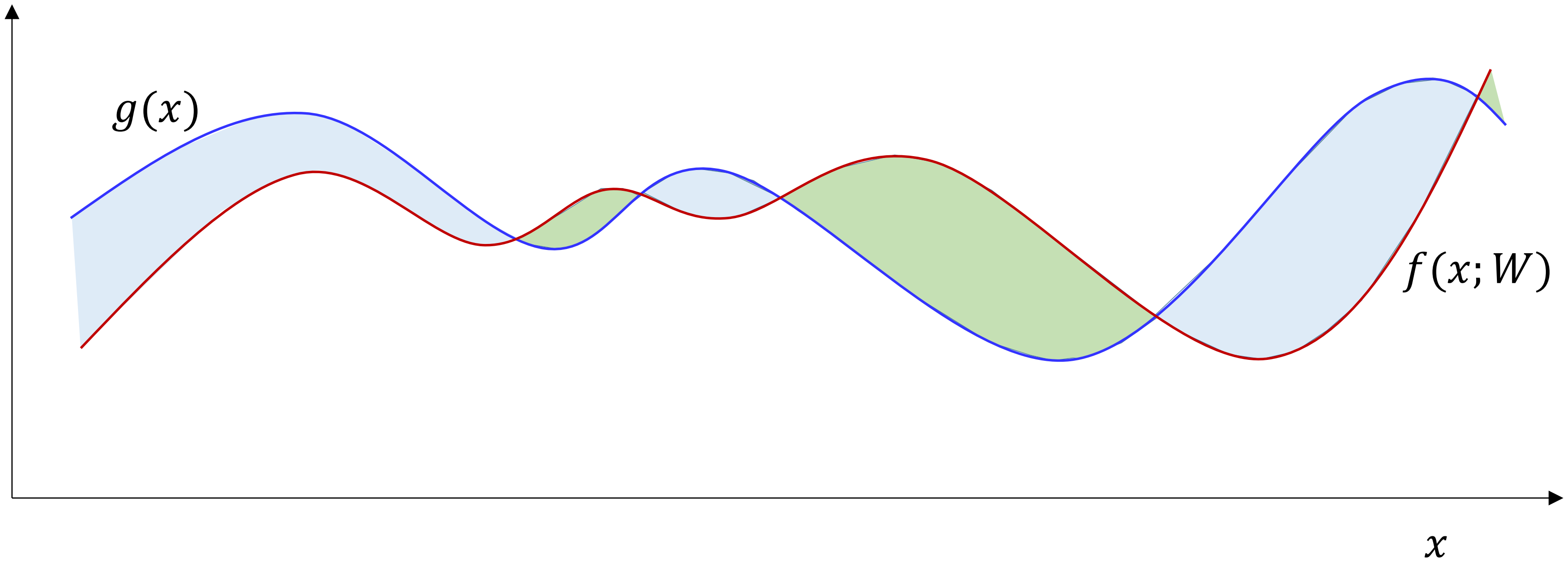
The variance of the sample error is the variance of the loss itself: $var(Err) = var(loss)$

This is N times the variance of the empirical average minimized by batch update

The sample error is also an *unbiased* estimate of the expected error

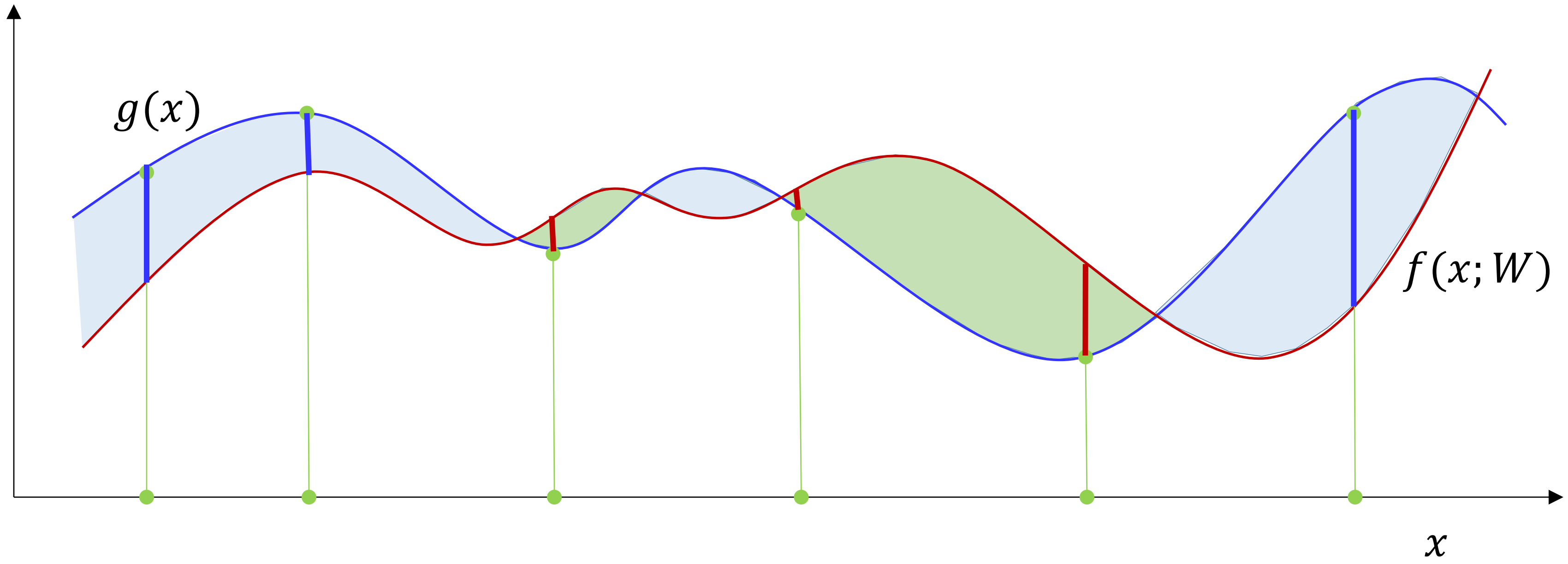
- At each iteration, **SGD** focuses on the loss of a ***single*** sample $loss\left(f\left(x^{(i)}; W\right), y^{(i)}\right)$
- The *expected value* of the *sample error* is ***still*** the *expected loss* $E[loss(f(x; W), y)]$

Explaining the variance



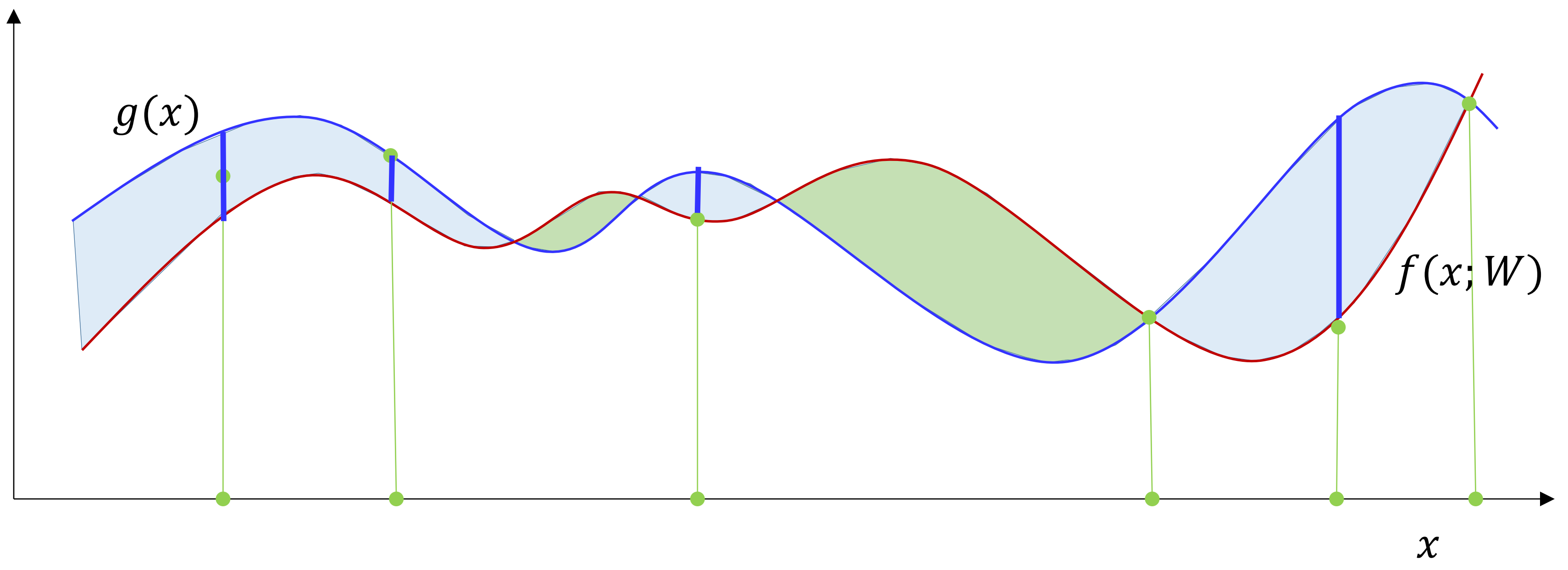
- The blue curve is the function being approximated
- The red curve is the approximation by the model at a given W
- The heights of the shaded regions represent the point-by-point error
  - We want to find the W that minimizes the average loss

Explaining the variance



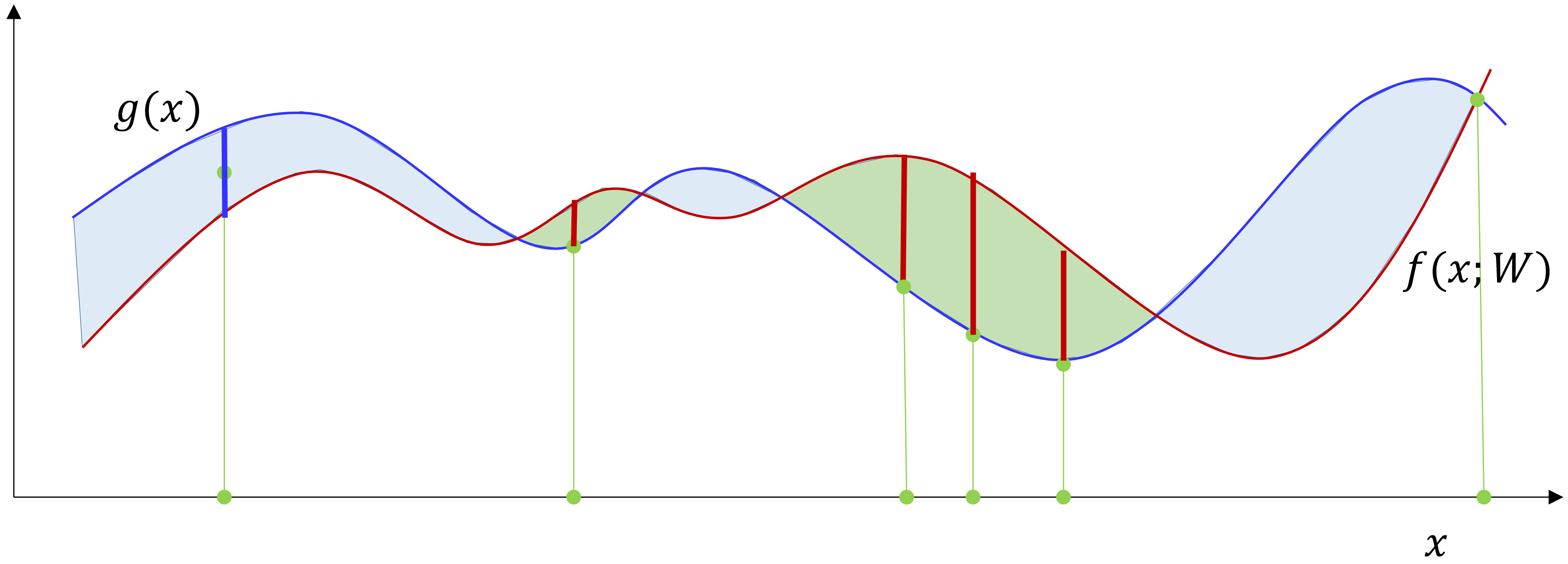
- Sample estimate approximates the shaded area with the average length of the lines

Explaining the variance



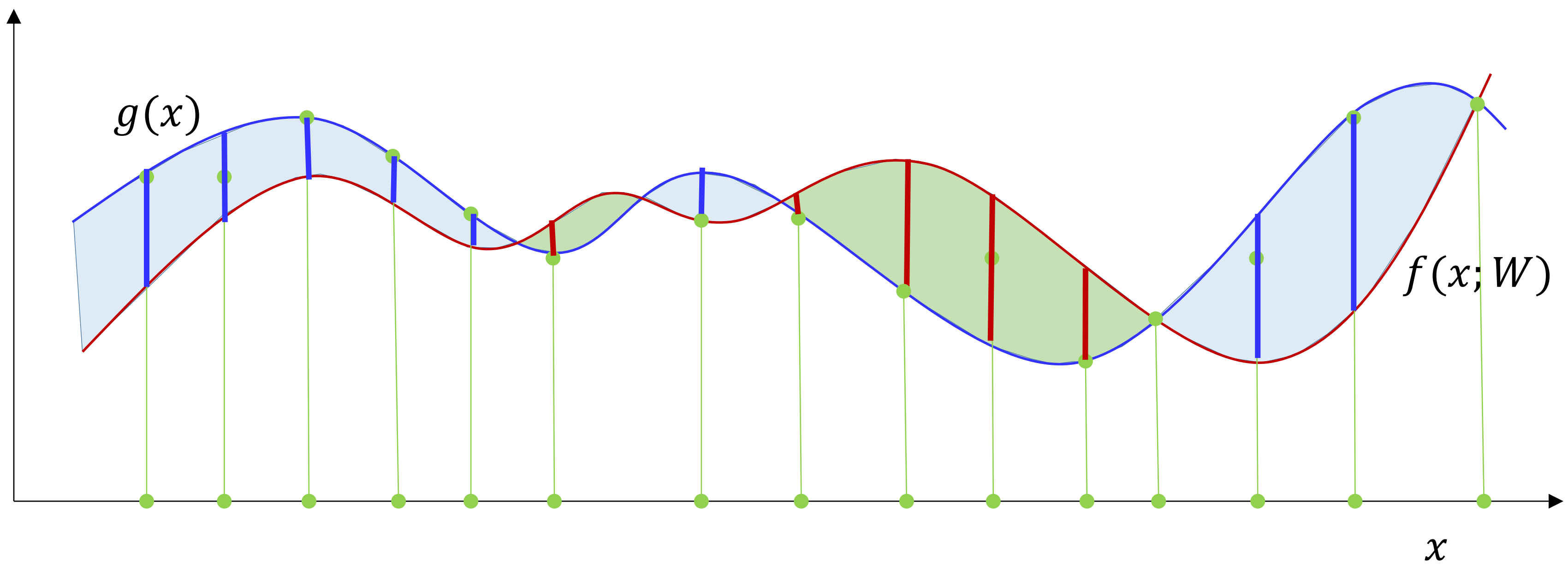
- Sample estimate approximates the shaded area with the average length of the lines
- This average length will change with position of the samples

Explaining the variance



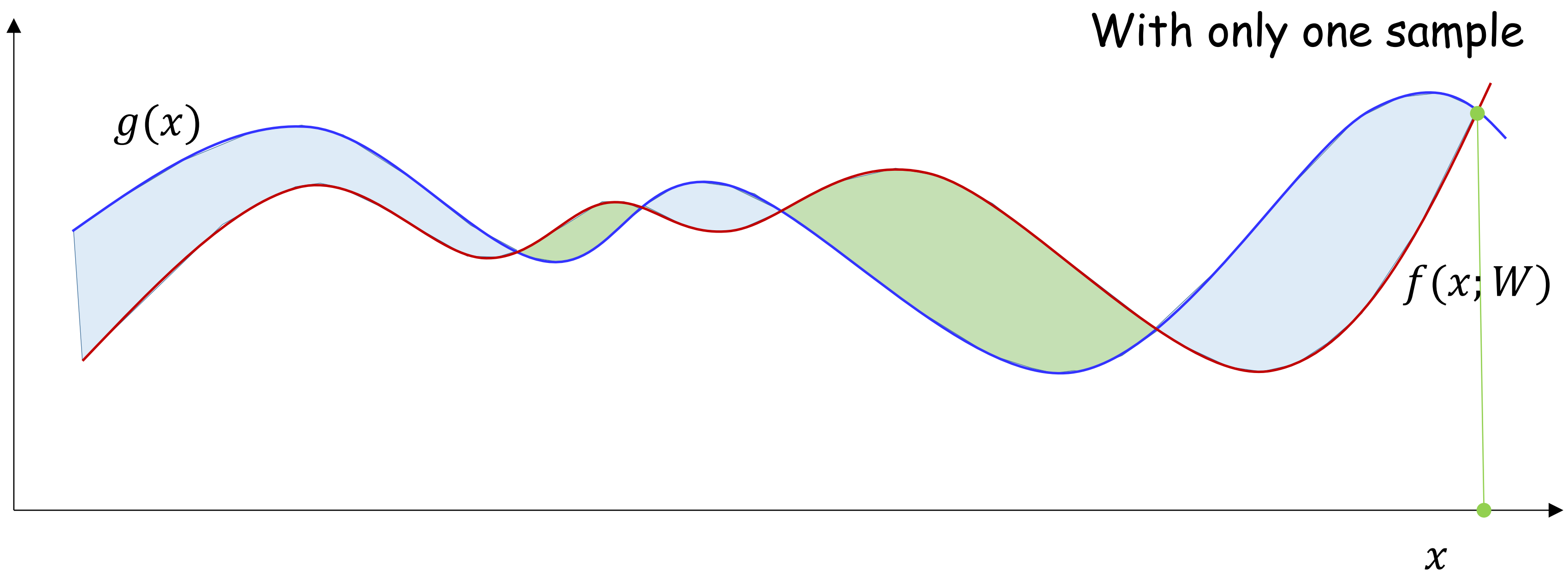
- Sample estimate approximates the shaded area with the average length of the lines
- This average length will change with position of the samples

Explaining the variance



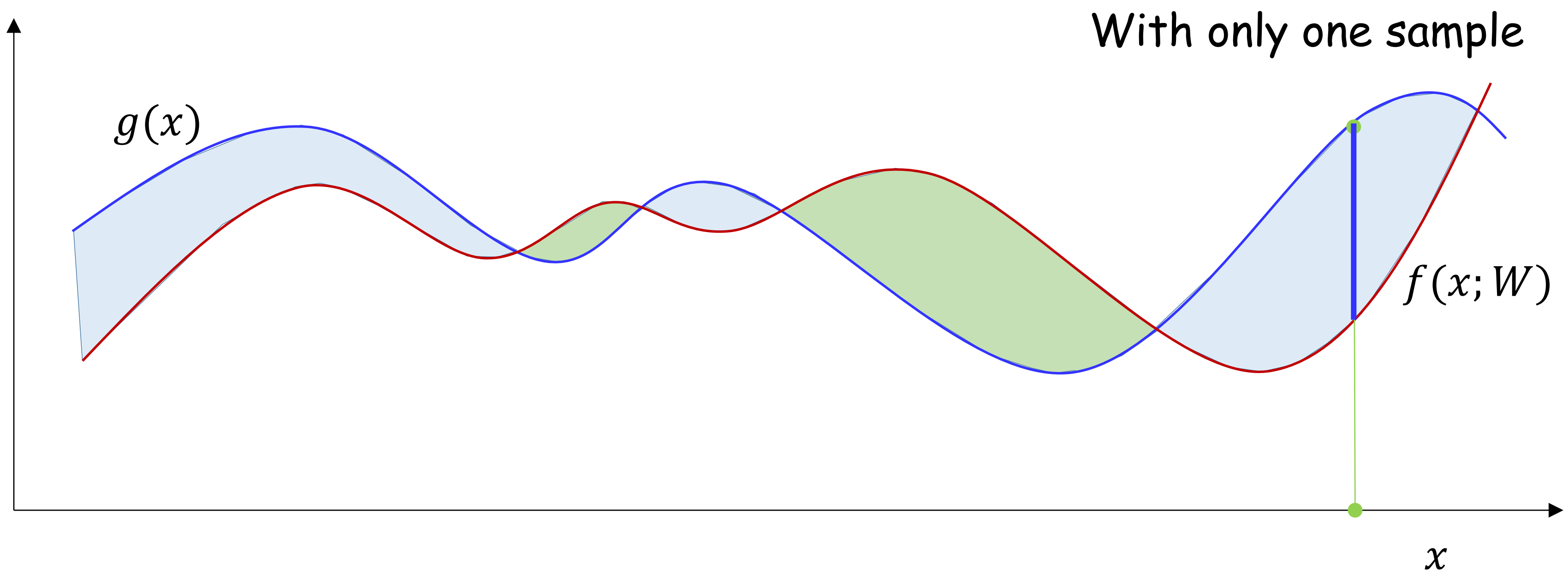
- Having more samples makes the estimate more robust to changes in the position of samples
  - The variance of the estimate is smaller

Explaining the variance



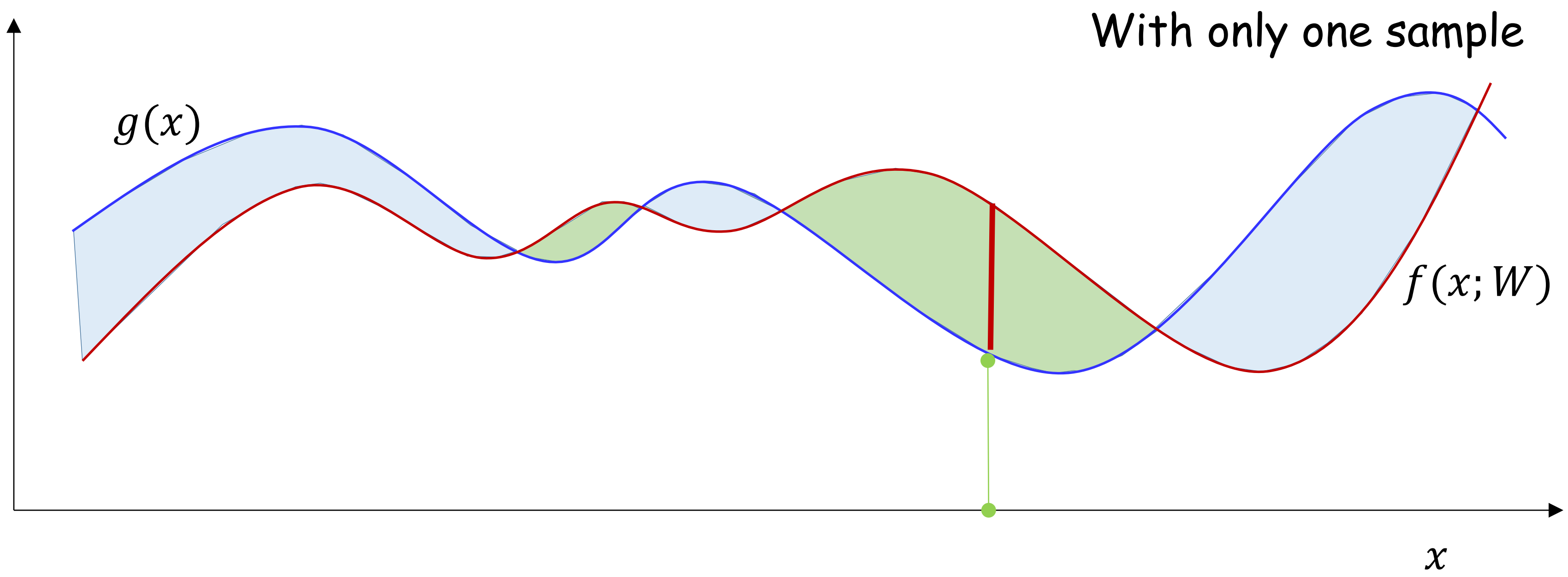
- Having very few samples makes the estimate swing wildly with the sample position
  - Since our estimator learns the W to minimize this estimate, the learned W too can swing wildly

Explaining the variance



- Having very few samples makes the estimate swing wildly with the sample position
  - Since our estimator learns the W to minimize this estimate, the learned W too can swing wildly

Explaining the variance

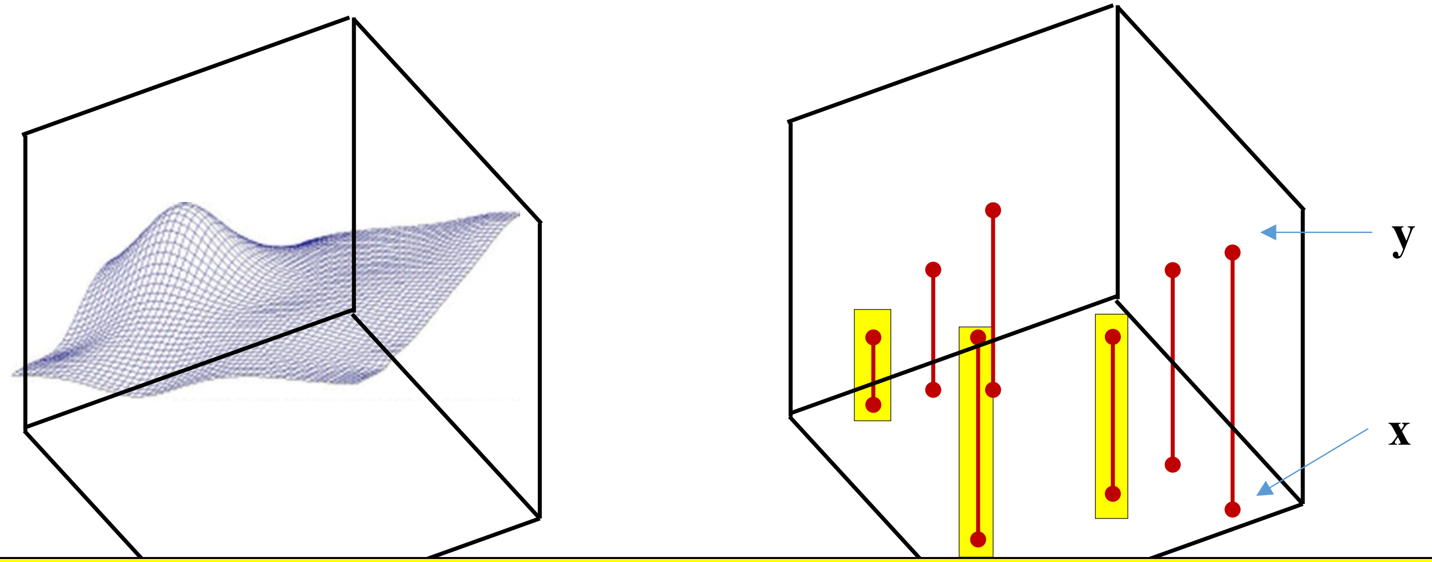


- Having very few samples makes the estimate swing wildly with the sample position
  - Since our estimator learns the W to minimize this estimate, the learned W too can swing wildly

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?

Mini Batches



The batch error is also an unbiased estimate of the expected error

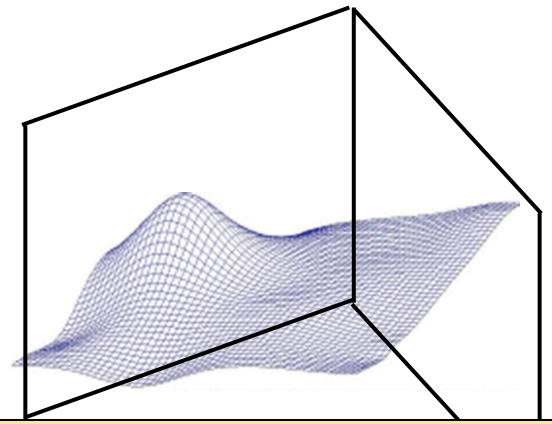
- Mini-batch updates compute and minimize a *batch error*

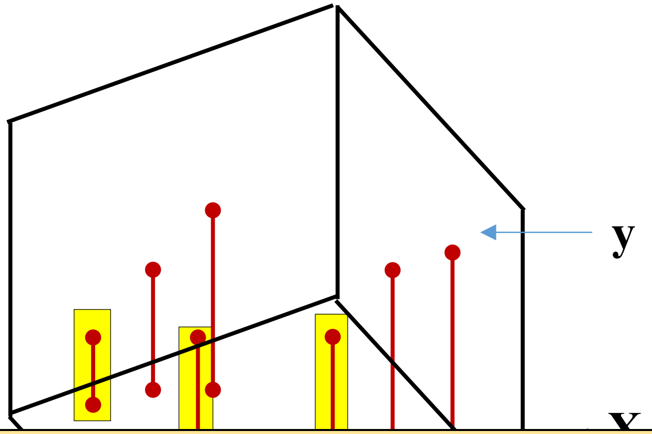
$$BatchErr\big(f(X;W),g(X)\big) = \frac{1}{b}\sum_{i=1}^{b} loss\big(f\big(x^{(i)};W\big),y^{(i)}\big)$$

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

$$E\big[BatchErr\big(f(X;W),g(X)\big)\big] = E\big[loss\big(f(X;W),g(X)\big)\big]$$

Mini Batches





The variance of the batch error: $var(E) = 1/b\ var(loss)$
This will be much smaller than the variance of the sample error in SGD

The batch error is also an unbiased estimate of the expected error

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

$$BatchErr\big(f(X;W), g(X)\big) = \frac{1}{b}\sum_{i=1}^{b} loss\left(f\left(x^{(i)};W\right), y^{(i)}\right)$$

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

$$E\left[BatchErr\big(f(X;W), g(X)\big)\right] = E\left[loss\big(f(X;W), g(X)\big)\right]$$

Mini-batch gradient descent

- Large datasets
  - Divide dataset into smaller batches containing one subset of the main training set
  - Weights are updated after seeing training data in each of these batches
- Vectorization provides efficiency

Gradient descent methods

Stochastic gradient

Stochastic mini-batch gradient

Batch gradient

Batch size=1

e.g., Batch size= 32, 64, 128, 256

Batch size=n
(the size of training set)

n: whole no of training data
bs: the size of batches
$m = \left\lceil \frac{n}{b} \right\rceil$: the number of batches

Batch 1 $X^{\{1\}}, Y^{\{1\}}$	Batch 2 $X^{\{2\}}, Y^{\{2\}}$								Batch m $X^{\{m\}}, Y^{\{m\}}$

Mini-batch gradient descent

For epoch=1,…,k

shuffle training data

For t=1,…,m

Forward propagation on $X^{\{t\}}$

$J^{\{t\}} = \frac{1}{m}\sum_{n \in Batch_t} L\left(\hat{Y}_n^{\{t\}}, Y_n^{\{t\}}\right) + \lambda R(W)$

Backpropagation on $J^{\{t\}}$ to compute gradients dW

For $l = 1, \dots, L$

$W^{[l]} = W^{[l]} - \alpha dW^{[l]}$

1 epoch: Single pass over all training samples

$A^{[0]} = X^{\{t\}}$
For $l = 1, \dots, L$
$Z^{[l]} = W^{[l]} A^{[l-1]}$
$A^{[l]} = f^{[l]}(Z^{[l]})$
$\hat{Y}_n^{\{t\}} = A_n^{[L]}$

Vectorized computaion

Batch 1 $X^{\{1\}}, Y^{\{1\}}$	Batch 2 $X^{\{2\}}, Y^{\{2\}}$								Batch m $X^{\{m\}}, Y^{\{m\}}$

Gradient descent methods

Stochastic gradient descent

Batch size=1

- Does not use vectorized form and thus not computationally efficient

Stochastic mini-batch gradient

e.g., Batch size= 32, 64, 128, 256

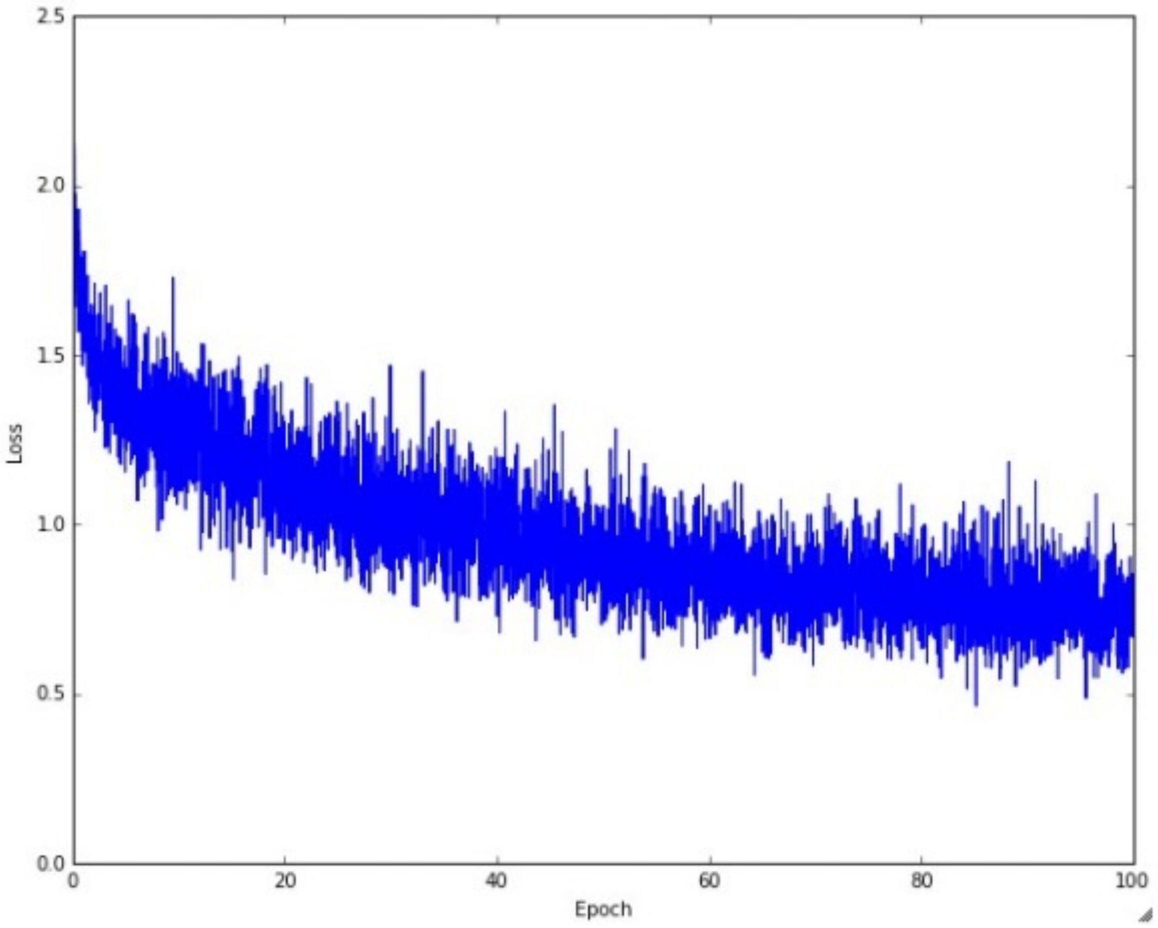
- Vectorization
- Fastest learning
  (for proper batch size)

Batch gradient descent

Batch size=n
(the size of training set)

- Need to process whole training set for weight update

Mini-batch gradient descent: loss-#epoch curve



Measuring error

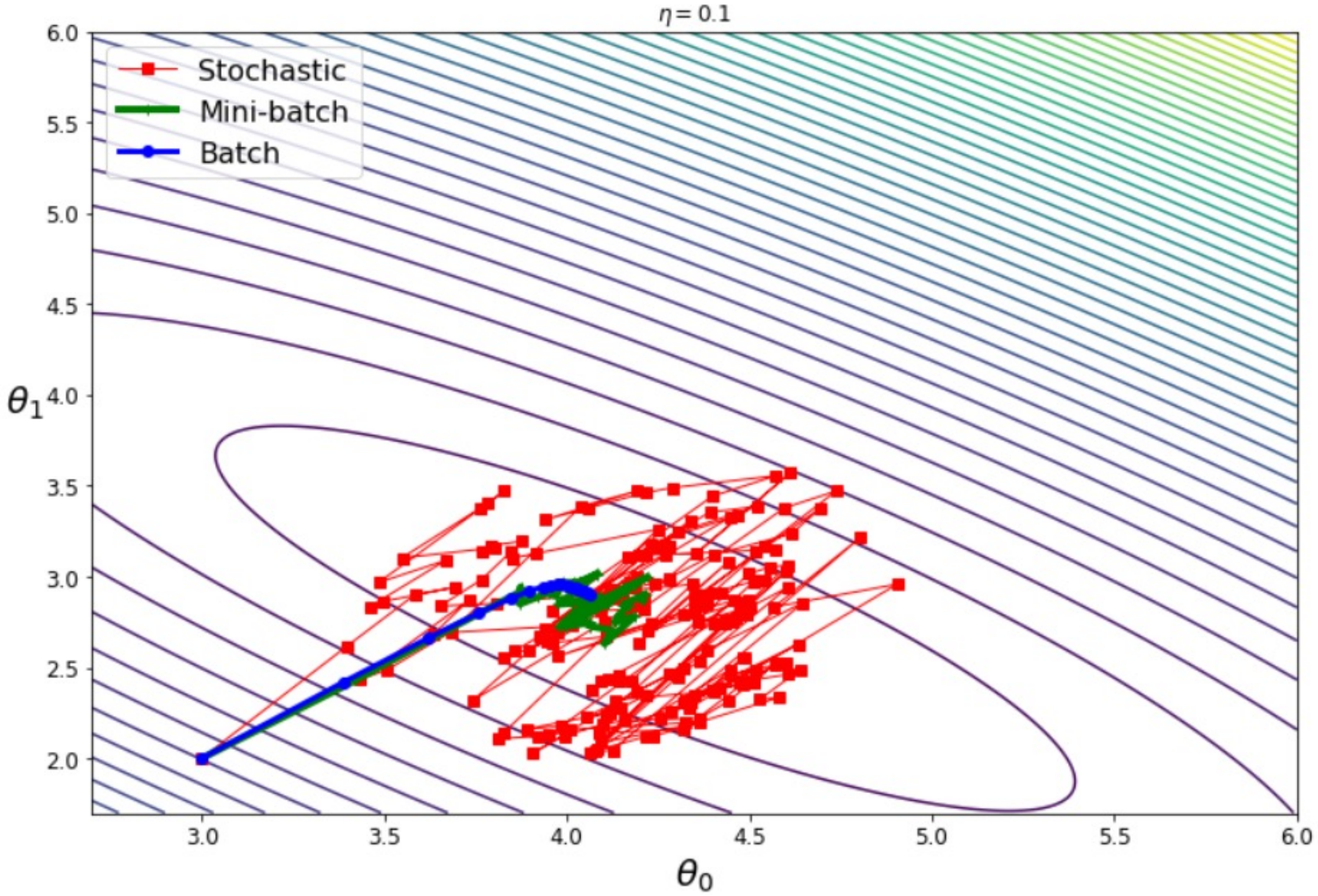
- Convergence is generally defined in terms of the *overall training* error
  - Not sample or batch error

- Infeasible to actually measure the overall training error after each iteration

- More typically, we estimate is as
  - Average sample/batch error over the past N samples/batches

Batch size

- Full batch (batch size = N)
- SGB (batch size = 1)
- SGD (batch size = 10)



Choosing mini-batch size

- For small training sets (e.g., n<2000) you can use full-batch gradient descent
- Typical mini-batch sizes for larger training sets:
  - 64, 128, 256, 512, 1024
- Make sure one batch of training data and the corresponding forward, backward required to be cached can fit in GPU memory

Story so far

- Gradient descent can be sped up by incremental updates
  - Convergence is guaranteed under most conditions
    - Learning rate must shrink with time for convergence
  - Stochastic gradient descent: update after each observation. Can be much faster than batch learning
  - Mini-batch updates: update after batches. Can be more efficient than SGD

- Convergence can be improved using smoothed updates
  - RMSprop and Adam

Story so far

- SGD: Presenting training instances one-at-a-time can be more effective than full-batch training
  - Provided they are provided in random order
- For SGD to converge, the learning rate must shrink sufficiently rapidly with iterations
  - Otherwise the learning will continuously "chase" the latest sample
- SGD estimates have higher variance than batch estimates
- Minibatch updates operate on *batches* of instances at a time
  - Estimates have lower variance than SGD
  - Convergence rate is theoretically worse than SGD
  - But we compensate by being able to perform batch processing

Story so far: 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)

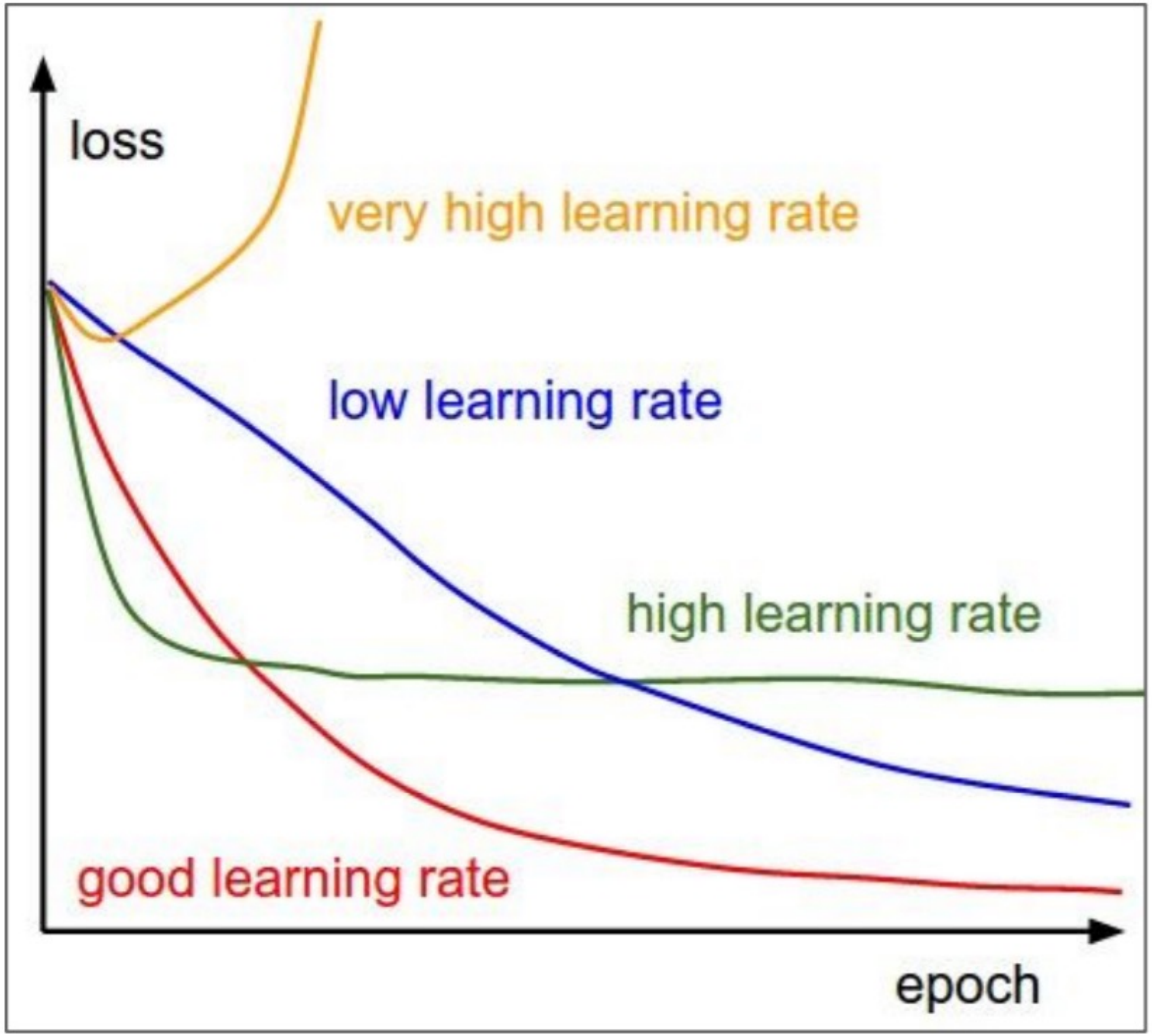
Some practical issues about learning rate

Learning rate

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

- Learning rate is an important hyperparameter that usually adjust first

Which one of these learning rates is best to use?



Learning rate

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

- loss not going down: learning rate too low

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

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

Loss barely changing: Learning rate is probably too low

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

Choosing learning rate parameter

- loss not going down: learning rate too low

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

- loss exploding: learning rate too high

```
Finished epoch 1 / 10: cost nan, train: 0.091000, val 0.087000, lr 1.000000e+06
Finished epoch 2 / 10: cost nan, train: 0.095000, val 0.087000, lr 1.000000e+06
Finished epoch 3 / 10: cost nan, train: 0.100000, val 0.087000, lr 1.000000e+06
```

cost: NaN almost always means high learning rate...

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

Learning rate

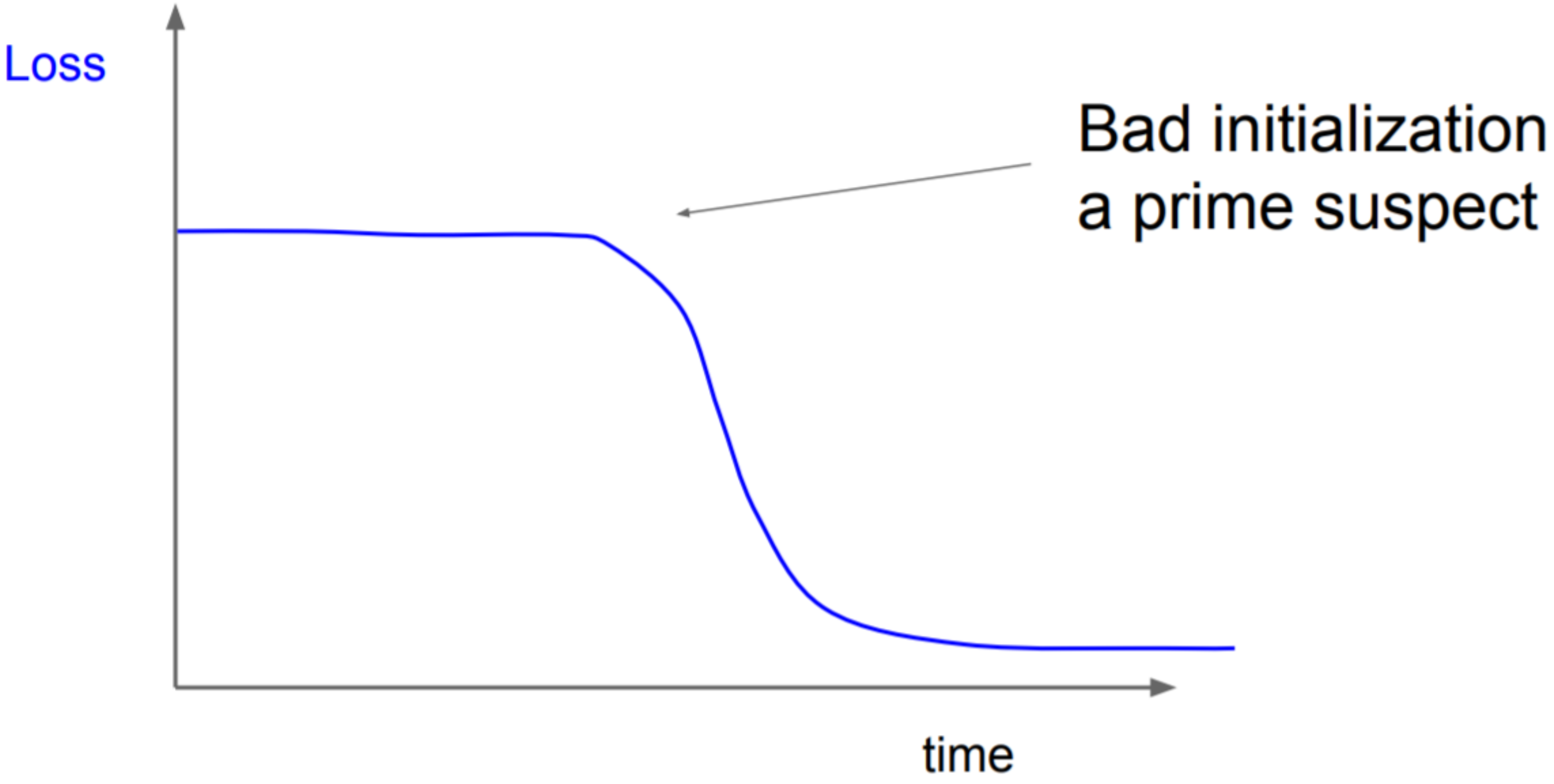
- Start with small regularization and find learning rate that makes the loss go down.
- **loss not going down: learning rate too low**
- **loss exploding: learning rate too high**

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

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

3e-3 is still too high.
Cost explodes....=> Rough range for learning rate we should be cross-validating is somewhere [1e-3 ... 1e-5]

Monitoring loss function during iterations



Learning rate decay

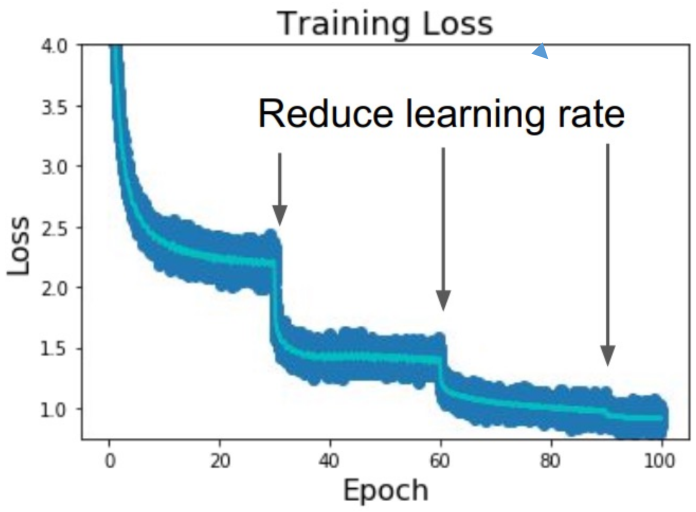
- Maybe during the initial steps of learning, you could afford to take much bigger steps

- But then as learning approaches converges, then having a slower learning rate allows you to take smaller steps

Learning rate scheduling

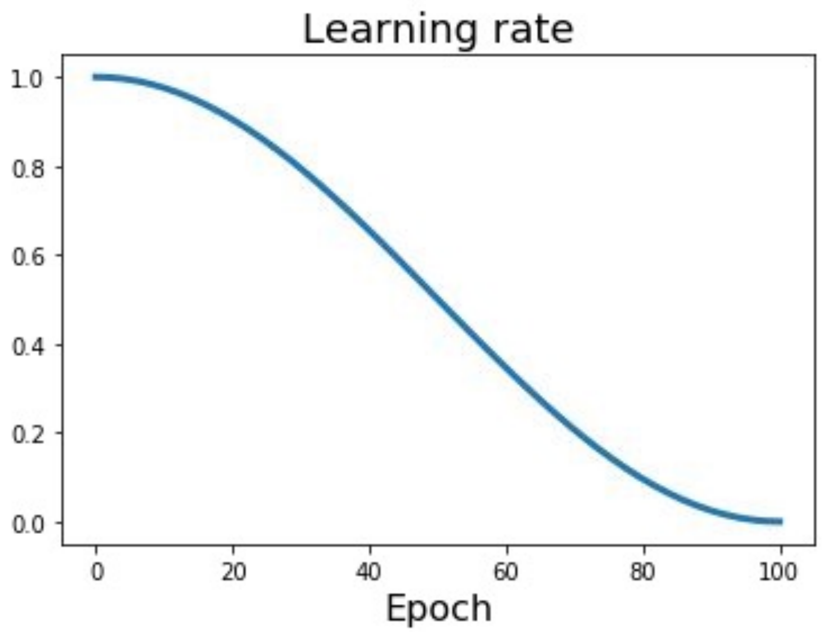
- Need for learning rate schedules
  - Benefits
    - Converge Faster
    - Higher accuracy
- Top Basic Learning Rate Schedules
  - Step-wise decay
  - Reduce on loss plateau decay
  - Cosine decay (Loshchilov & Hutter, 2017)
  - trapezoidal schedule (Xing et al., 2018)
  - …

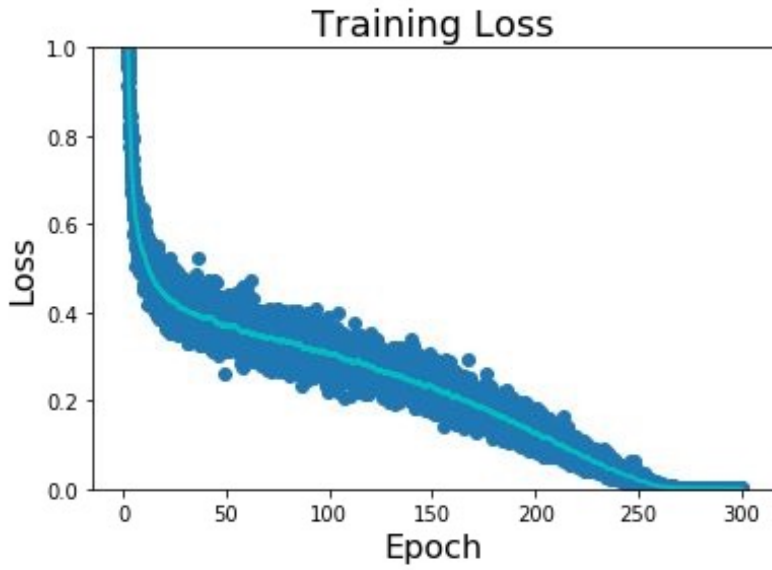
Step-wise: Reduce learning rate at a few fixed points. e.g. for ResNets, multiply LR by 0.1 after epochs 30, 60, and 90



Learning rate decay

- **Cosine:** $\alpha_t = \frac{1}{2}\alpha_0\left(1 + cos\left(\frac{t\pi}{T}\right)\right)$
  - α_0: Initial learning rate
  - α_t: Learning rate at epoch t
  - T: Total number of epochs

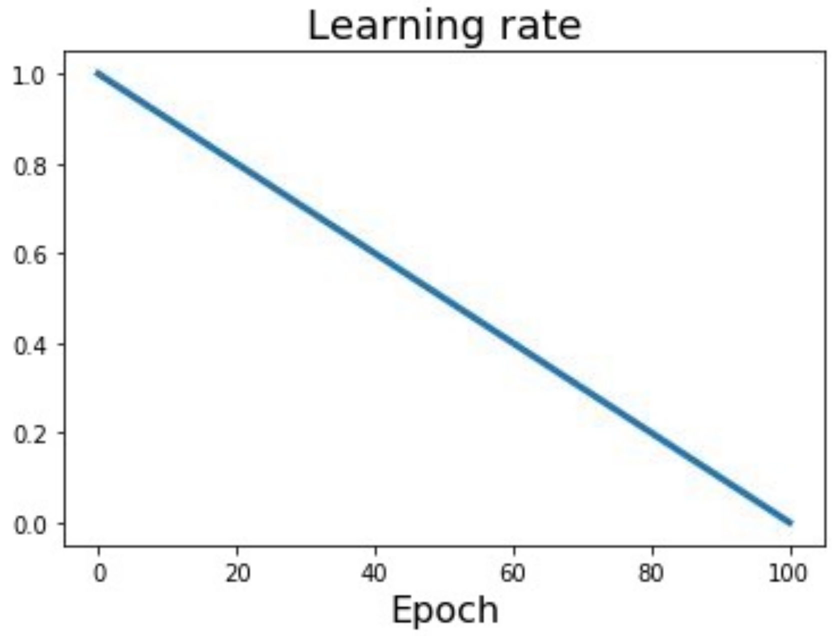




Loshchilov and Hutter, "SGDR: Stochastic Gradient Descent with Warm Restarts", ICLR 2017
Radford et al, "Improving Language Understanding by Generative Pre-Training", 2018
Feichtenhofer et al, "SlowFast Networks for Video Recognition", arXiv 2018
Child at al, "Generating Long Sequences with Sparse Transformers", arXiv 2019

Learning rate decay

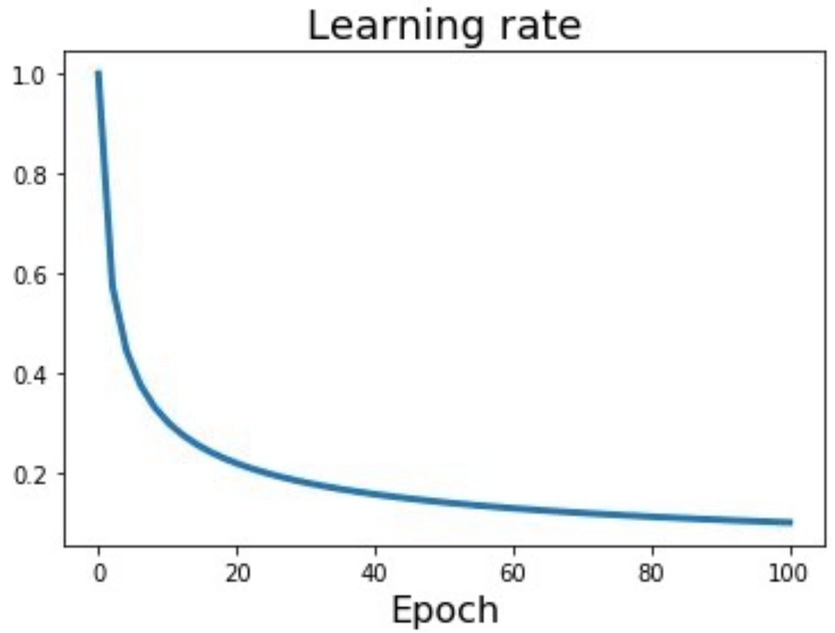
- **Linear:** $\alpha_t = \alpha_0 \left(1 - \frac{t}{T}\right)$
  - α_0: Initial learning rate
  - α_t: Learning rate at epoch t
  - T: Total number of epochs



Devlin et al, "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding", 2018

Learning Rate Decay

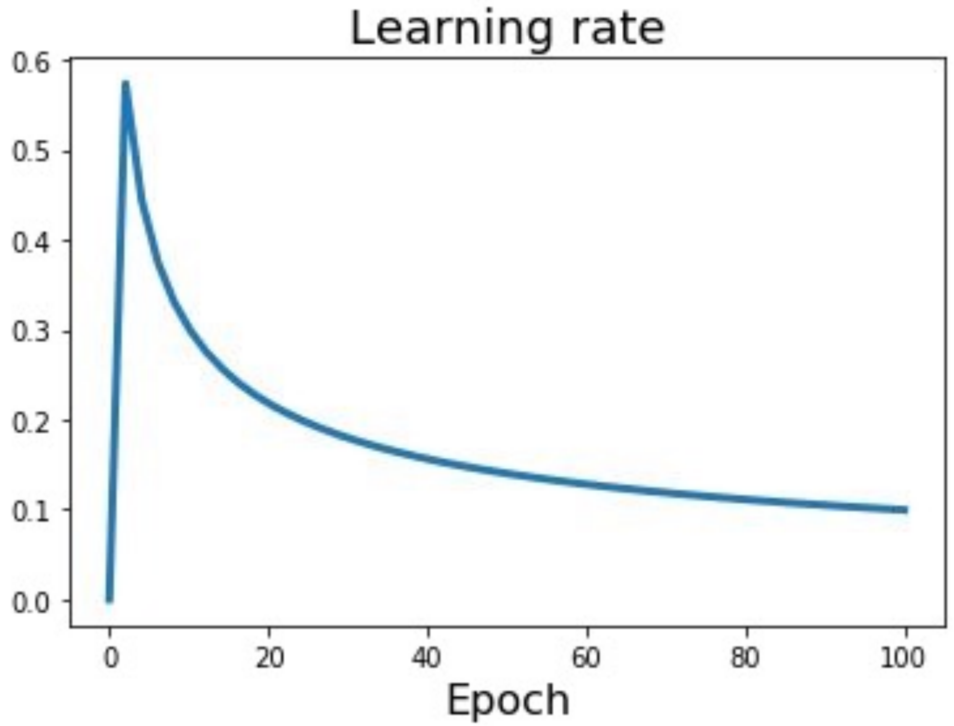
- **Inverse sqrt:** $\alpha_t = \frac{\alpha_0}{\sqrt{t}}$
  - α_0: Initial learning rate
  - α_t: Learning rate at epoch t
  - T: Total number of epochs



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 ~5,000 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

Summary

- Stochastic Gradient Descent (SGD)
- Mini-batch update
- Adjusting learning rate