Deep Learning

Neural Networks
Regularization and Optimization

Slide Credit:

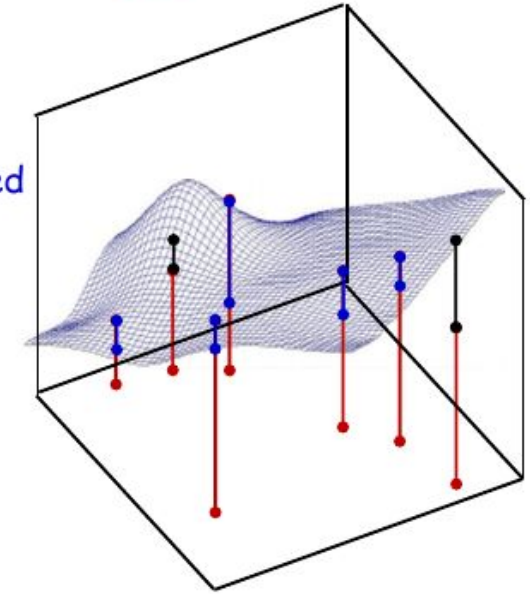
1. Bikash Raj, 11-785, Fall 2017
2. Fei-Fei Li, Justin Johnson, Serena Yeung, CS 231n

Topics for the session

- Regularization
  - Different views
  - L2 and L1 based smoothness
  - Dropout
  - Other heuristics
    - Early stopping
    - Gradient clipping
    - Data Augmentation

General approach to training

Blue lines: error when function is *below* desired output

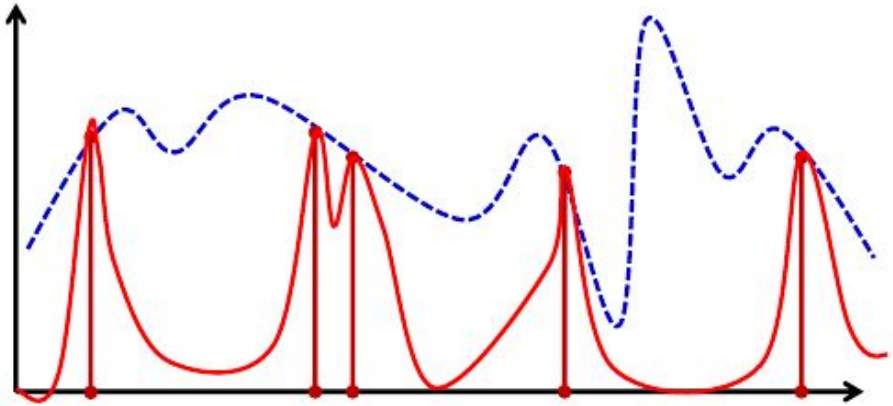


Black lines: error when function is *above* desired output

$$E = \sum_i (y_i - f(\mathbf{x}_i, \mathbf{W}))^2$$

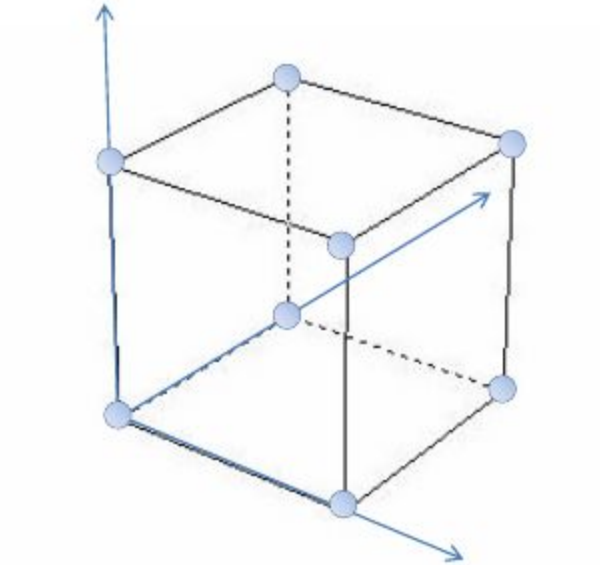
- Define an *error* between the ***actual*** network output for any parameter value and the *desired* output
  - Error typically defined as the *sum* of the squared error over individual training instances

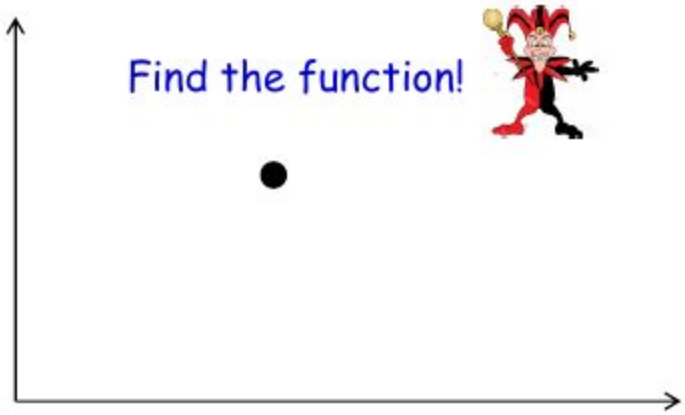
Overfitting



- Problem: Network may just learn the values at the inputs
  - Learn the red curve instead of the dotted blue one
    - Given only the red vertical bars as inputs

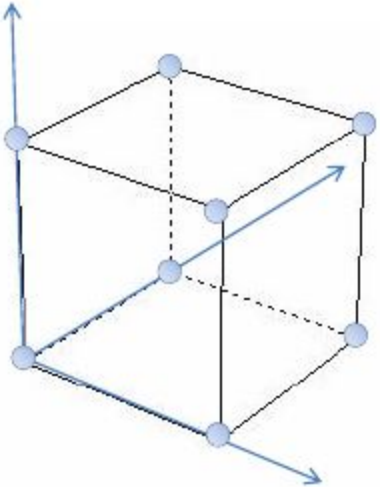
Data under-specification in learning

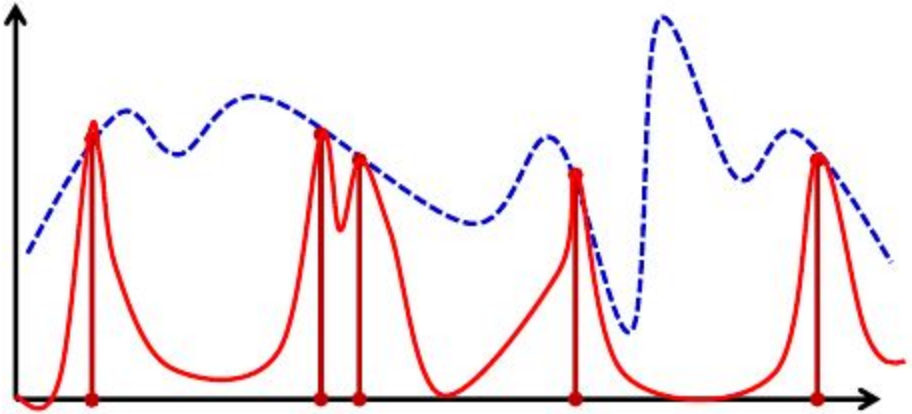




- Consider a binary 100-dimensional input
- There are $2^{100}=10^{30}$ possible inputs
- Complete specification of the function will require specification of 10^{30} output values
- A training set with only 10^{15} training instances will be off by a factor of 10^{15}

Need “smoothing” constraints



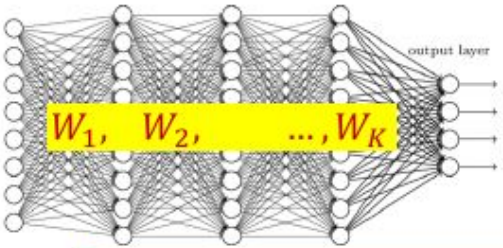


- Need additional constraints that will “fill in” the missing regions acceptably
  - Generalization

What is regularization?

- In general: any method to prevent overfitting or help the optimization
- Specifically: additional terms in the training optimization objective to prevent overfitting or help the optimization.

Objective function for neural networks



Y_t

Desired output of network: d_t

Error on i-th training input: $Div(Y_t, d_t; W_1, W_2, \ldots, W_K)$

Batch training error:

$$Err(W_1, W_2, \ldots, W_K) = \frac{1}{T}\sum_t Div(Y_t, d_t; W_1, W_2, \ldots, W_K)$$

- Conventional training: minimize the total error:

$$\widehat{W}_1, \widehat{W}_2, \ldots, \widehat{W}_K = \underset{W_1, W_2, \ldots, W_K}{\operatorname{argmin}} Err(W_1, W_2, \ldots, W_K)$$

Smoothness through weight constraints

- Regularized training: minimize the error while also minimizing the weights

$$L(W_1, W_2, \dots, W_K) = Err(W_1, W_2, \dots, W_K) + \frac{1}{2}\lambda \sum_k \|W_k\|_2^2$$

$$\widehat{W}_1, \widehat{W}_2, \dots, \widehat{W}_K = \underset{W_1, W_2, \dots, W_K}{\operatorname{argmin}} \; L(W_1, W_2, \dots, W_K)$$

- λ is the regularization parameter whose value depends on how important it is for us to want to minimize the weights

- Increasing l assigns greater importance to shrinking the weights
  - Make greater error on training data, to obtain a more acceptable network

Classical regularization

- Norm penalty
  - l_2 regularization
  - l_1 regularization

- Robustness to noise

l_2 regularization

$$\min_{\theta} \hat{L}_R(\theta) = \hat{L}(\theta) + \frac{\alpha}{2}||\theta||_2^2$$

- Effect on (stochastic) gradient descent
- Effect on the optimal solution

Effect on the optimal solution

- Consider a quadratic approximation around optimal θ^*

$$\hat{L}(\theta) \approx \hat{L}(\theta^*) + (\theta - \theta^*)^T \nabla \hat{L}(\theta^*) + \frac{1}{2}(\theta - \theta^*)^T H(\theta - \theta^*)$$

- Since θ^* is optimal, $\nabla \hat{L}(\theta^*) = 0$

$$\hat{L}(\theta) \approx \hat{L}(\theta^*) + \frac{1}{2}(\theta - \theta^*)^T H(\theta - \theta^*)$$

$$\nabla \hat{L}(\theta) \approx H(\theta - \theta^*)$$

Effect on the optimal solution

- Consider a quadratic approximation around θ^*

$$\hat{L}(\theta) \approx \hat{L}(\theta^*) + (\theta - \theta^*)^T \nabla \hat{L}(\theta^*) + \frac{1}{2}(\theta - \theta^*)^T H(\theta - \theta^*)$$

- Since θ^* is optimal, $\nabla \hat{L}(\theta^*) = 0$

$$\hat{L}(\theta) \approx \hat{L}(\theta^*) + \frac{1}{2}(\theta - \theta^*)^T H(\theta - \theta^*)$$

$$\nabla \hat{L}(\theta) \approx H(\theta - \theta^*)$$

Effect on the optimal solution

- Gradient of regularized objective

$$\nabla \hat{L}_R(\theta) \approx H(\theta - \theta^*) + \alpha\theta$$

- On the optimal θ_R^*

$$0 = \nabla \hat{L}_R(\theta_R^*) \approx H(\theta_R^* - \theta^*) + \alpha\theta_R^*$$

$$\theta_R^* \approx (H + \alpha I)^{-1} H\theta^*$$

Effect on the optimal solution

- The optimal

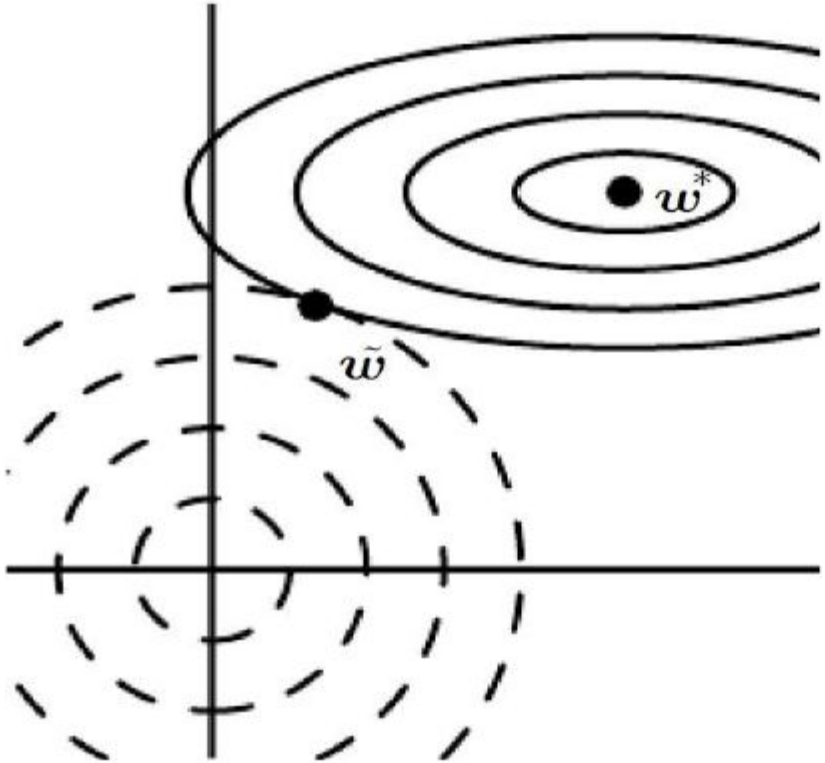
$$\theta_R^* \approx (H + \alpha I)^{-1} H\theta^*$$

- Suppose H has eigen-decomposition $H = Q\Lambda Q^T$

$$\theta_R^* \approx (H + \alpha I)^{-1} H\theta^* = Q(\Lambda + \alpha I)^{-1}\Lambda Q^T\theta^*$$

- Effect: rescale along eigenvectors of H

Effect on the optimal solution



Notations:

$\theta^* = w^*, \theta_R^* = \tilde{w}$

Figure from *Deep Learning*, Goodfellow, Bengio and Courville

l_1 regularization

$$\min_{\theta} \hat{L}_R(\theta) = \hat{L}(\theta) + \alpha||\theta||_1$$

- Effect on (stochastic) gradient descent
- Effect on the optimal solution

Effect on gradient descent

- Gradient of regularized objective

$$\nabla \hat{L}_R(\theta) = \nabla \hat{L}(\theta) + \alpha \, \text{sign}(\theta)$$

where sign applies to each element in θ

- Gradient descent update

$$\theta \leftarrow \theta - \eta \nabla \hat{L}_R(\theta) = \theta - \eta \, \nabla \hat{L}(\theta) - \eta\alpha \, \text{sign}(\theta)$$

Effect on the optimal solution

- Consider a quadratic approximation around θ^*

$$\hat{L}(\theta) \approx \hat{L}(\theta^*) + (\theta - \theta^*)^T \nabla \hat{L}(\theta^*) + \frac{1}{2}(\theta - \theta^*)^T H(\theta - \theta^*)$$

- Since θ^* is optimal, $\nabla \hat{L}(\theta^*) = 0$

$$\hat{L}(\theta) \approx \hat{L}(\theta^*) + \frac{1}{2}(\theta - \theta^*)^T H(\theta - \theta^*)$$

Effect on the optimal solution

- Further assume that H is diagonal and positive $(H_{ii} > 0, \forall i)$
  - not true in general but assume for getting some intuition
- The regularized objective is (ignoring constants)

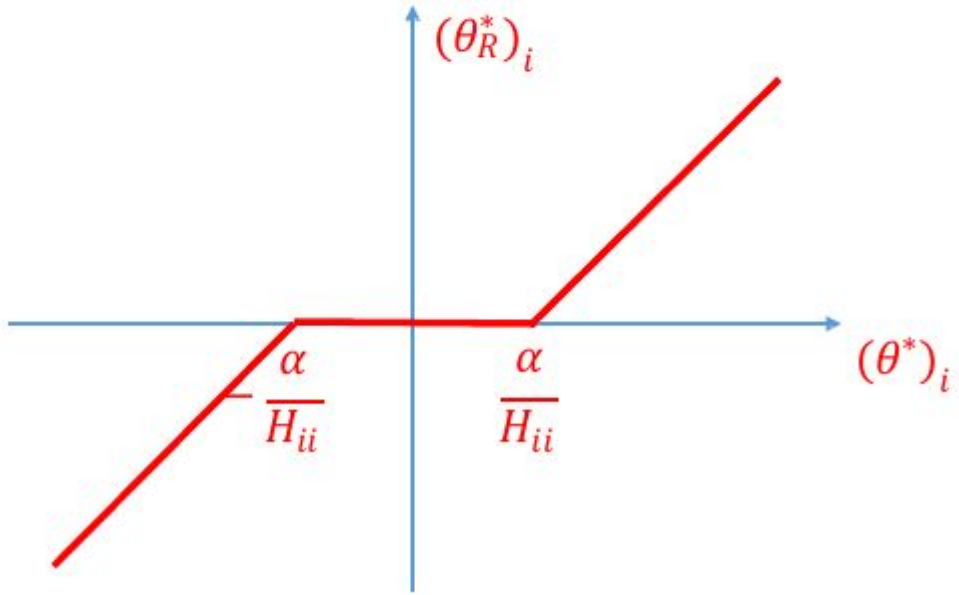
$$\hat{L}_R(\theta) \approx \sum_i \frac{1}{2} H_{ii}(\theta_i - \theta_i^*)^2 + \alpha |\theta_i|$$

- The optimal θ_R^*

$$(\theta_R^*)_i \approx \begin{cases} \max\left\{\theta_i^* - \frac{\alpha}{H_{ii}}, 0\right\} & \text{if } \theta_i^* \geq 0 \\ \min\left\{\theta_i^* + \frac{\alpha}{H_{ii}}, 0\right\} & \text{if } \theta_i^* < 0 \end{cases}$$

Effect on the optimal solution

- Effect: induce sparsity (*aka* feature selection)

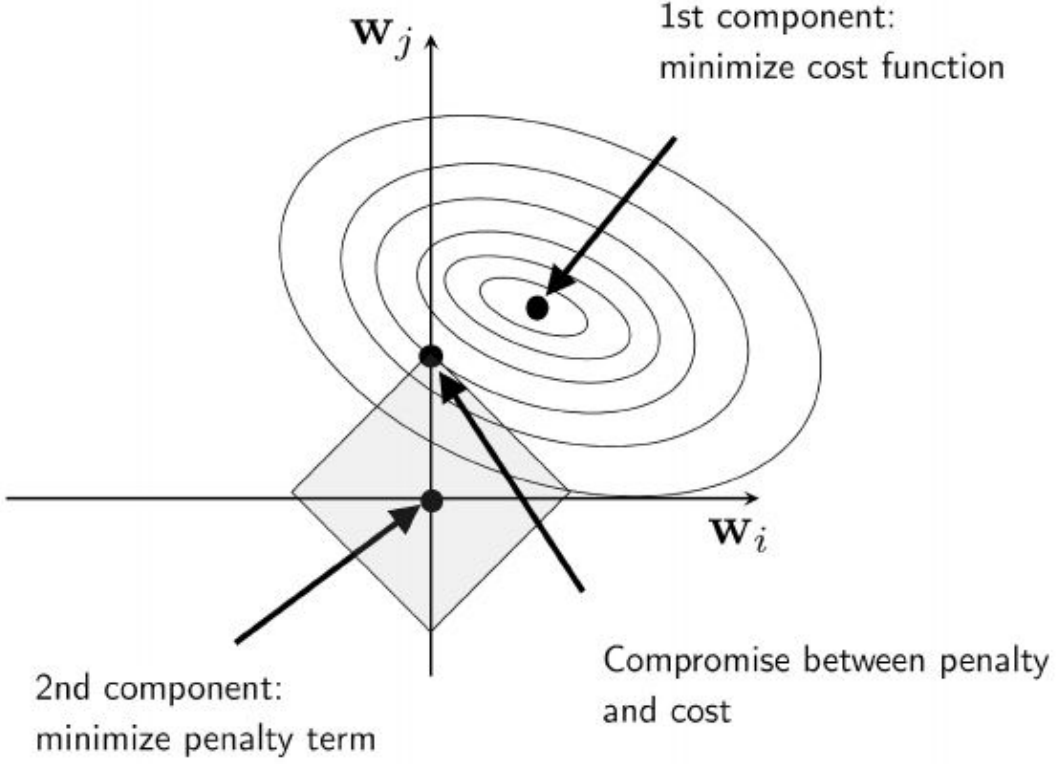


Effect on the optimal solution

- Further assume that H is diagonal
- Compact expression for the optimal θ_R^*

$$(\theta_R^*)_i \approx \text{sign}(\theta_i^*) \max\{|\theta_i^*| - \frac{\alpha}{H_{ii}}, 0\}$$

Geometric Interpretation of L1 Minimization



Regularization as hard constraint

- When Ω measured by some quantity R

$$\min_{\theta} \hat{L}(\theta) = \frac{1}{n}\sum_{i=1}^{n} l(\theta, x_i, y_i)$$

$$\text{subject to: } R(\theta) \leq r$$

- Example: l_2 regularization

$$\min_{\theta} \hat{L}(\theta) = \frac{1}{n}\sum_{i=1}^{n} l(\theta, x_i, y_i)$$

$$\text{subject to: } ||\theta||_2^2 \leq r^2$$

Regularization as soft constraint

- The hard-constraint optimization is equivalent to soft-constraint

$$\min_{\theta} \hat{L}_R(\theta) = \frac{1}{n}\sum_{i=1}^{n} l(\theta, x_i, y_i) + \lambda^* R(\theta)$$

  for some regularization parameter $\lambda^* > 0$

- Example: l_2 regularization

$$\min_{\theta} \hat{L}_R(\theta) = \frac{1}{n}\sum_{i=1}^{n} l(\theta, x_i, y_i) + \lambda^* ||\theta||_2^2$$

Regularizing the weights

$$L(W_1, W_2, \ldots, W_K) = \frac{1}{T}\sum_t Div(Y_t, d_t) + \frac{1}{2}\lambda \sum_k \|W_k\|_2^2$$

- Batch mode:

$$\Delta W_k = \frac{1}{T}\sum_t \nabla_{W_k} Div(Y_t, d_t)^T + \lambda W_k$$

- SGD:

$$\Delta W_k = \nabla_{W_k} Div(Y_t, d_t)^T + \lambda W_k$$

- Minibatch:

$$\Delta W_k = \frac{1}{b}\sum_{\tau=t}^{t+b-1} \nabla_{W_k} Div(Y_\tau, d_\tau)^T + \lambda W_k$$

- Update rule:

$$W_k \leftarrow W_k - \eta \Delta W_k$$

Summary: Regularization

λ = regularization strength (hyperparameter)

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

Data loss: Model predictions should match training data

Regularization: Prevent the model from doing *too* well on training data

Simpler examples

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

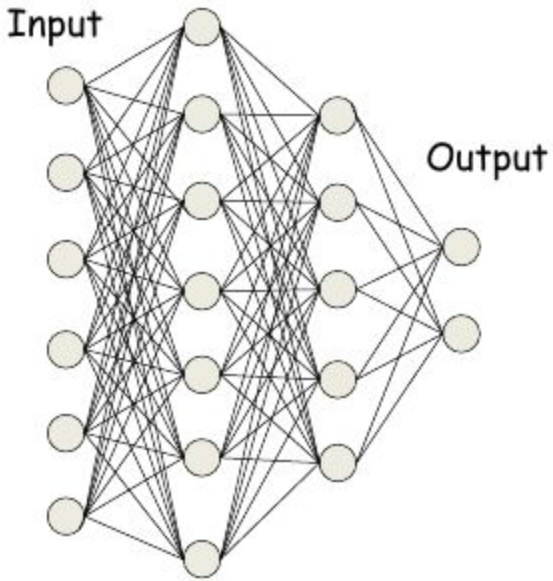
More complex:

Dropout

Batch normalization

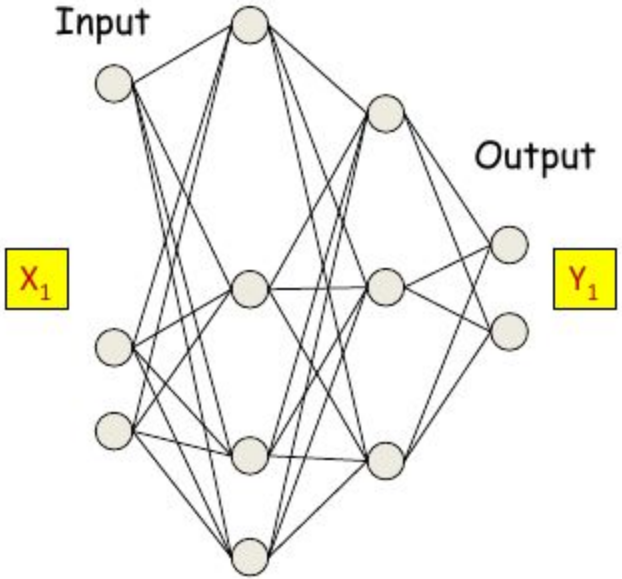
Stochastic depth, fractional pooling, etc

Dropout



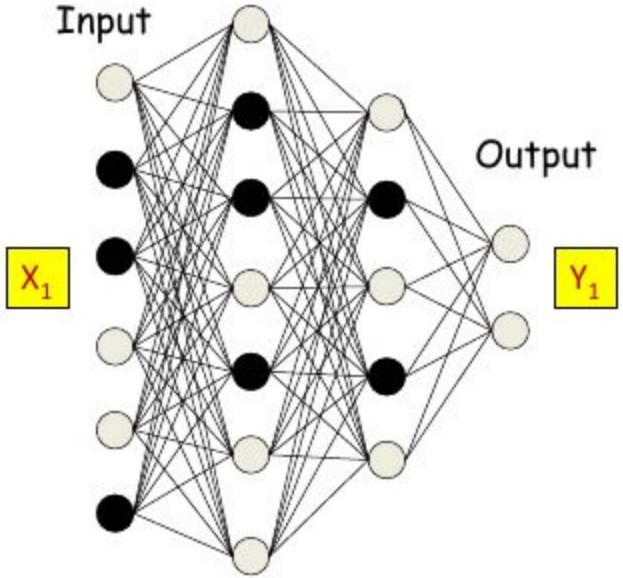
- **During training:** For each input, at each iteration, "turn off" each neuron with a probability 1-α

Dropout



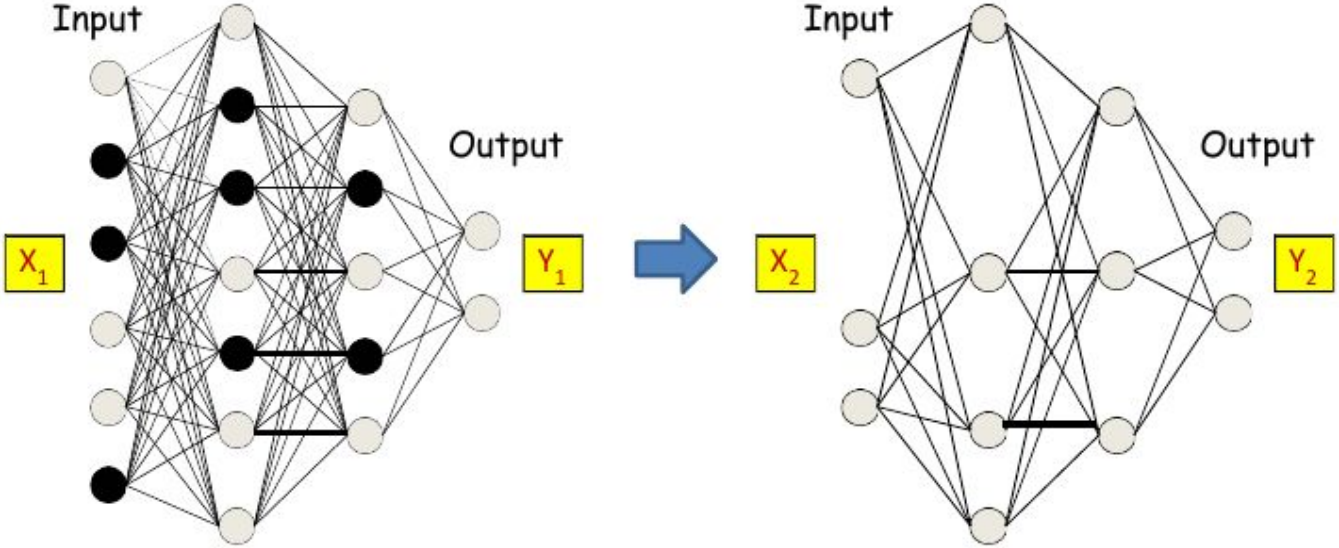
- **During training:** For each input, at each iteration, "turn off" each neuron with a probability 1-α
  - Also turn off inputs similarly

Dropout



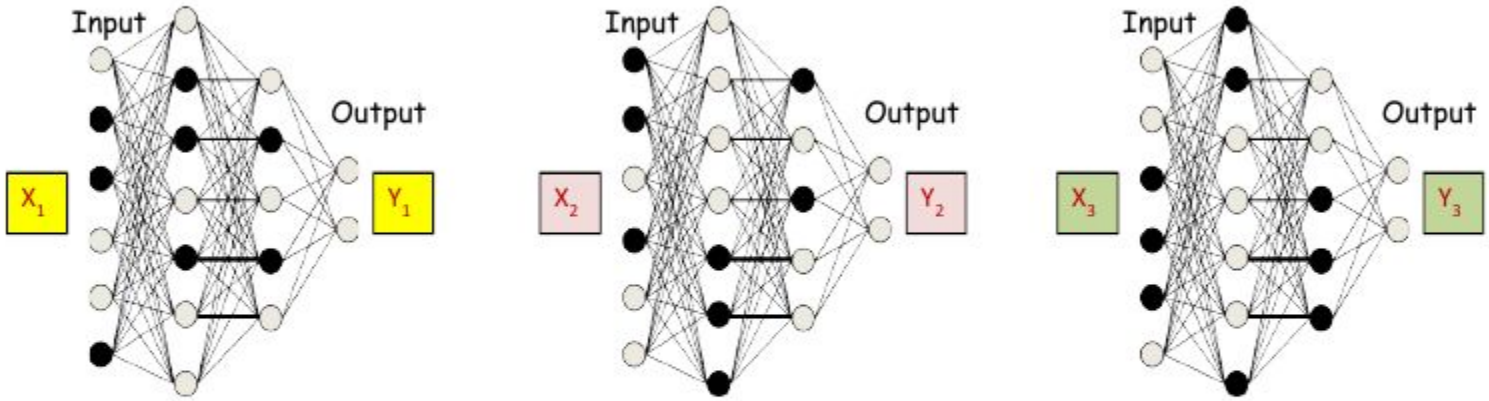
- **During training:** For each input, at each iteration, "turn off" each neuron (including inputs) with a probability 1-α
  - In practice, set them to 0 according to the success of a Bernoulli random number generator with success probability 1-α

Dropout



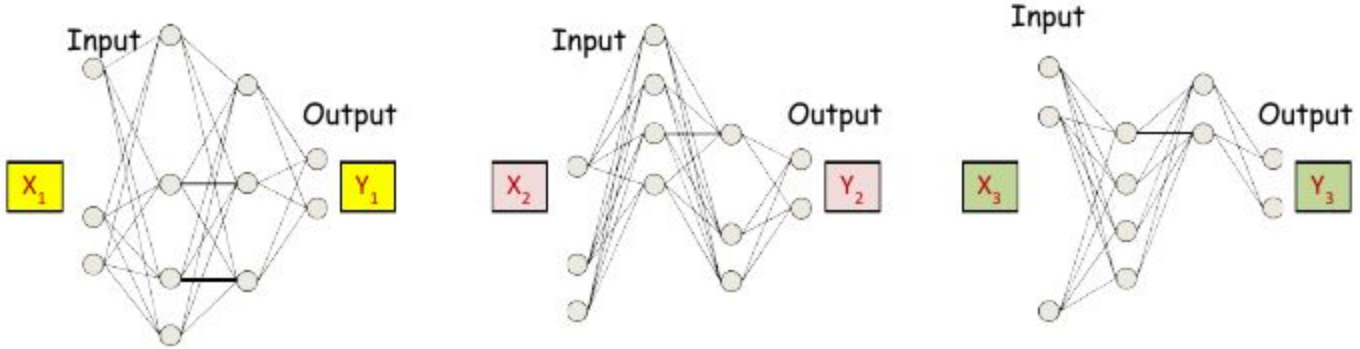
- **During training:** For each input, at each iteration, "turn off" each neuron (including inputs) with a probability 1-a
  - In practice, set them to 0 according to the success of a Bernoulli random number generator with success probability 1-a
  - Also turn off inputs similarly

Dropout



- The pattern of dropped nodes changes for *each input*, *i.e.*, in every pass through the net

Dropout



- **During training:** Backpropagation is effectively performed only over the remaining network
  - The effective network is different for different inputs
  - Gradients are obtained only for the weights and biases *from* "On" nodes *to* "On" nodes
    - For the remaining, the gradient is just 0

What each neuron computes

- Each neuron actually has the following activation:

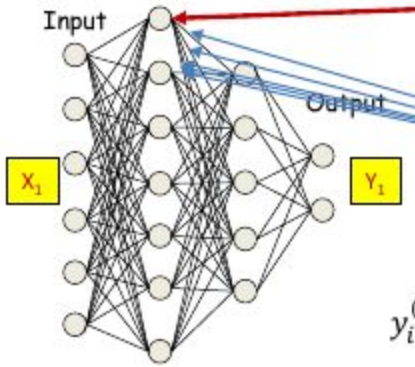
$$y_i^{(k)} = D\sigma\left(\sum_j w_{ji}^{(k)} y_j^{(k-1)} + b_i^{(k)}\right)$$

  - Where D is a Bernoulli variable that takes a value 1 with probability α
- D may be switched on or off for individual sub networks, but over the ensemble, the *expected output* of the neuron is

$$y_i^{(k)} = \alpha\sigma\left(\sum_j w_{ji}^{(k)} y_j^{(k-1)} + b_i^{(k)}\right)$$

- During *test* time, we will use the *expected* output of the neuron
  - Which corresponds to the bagged average output
  - Consists of simply scaling the output of each neuron by α

Dropout during test: implementation



apply α here (to the output of the neuron) OR..

Push the α to all outgoing weights

$$y_i^{(k)} = \alpha\sigma\left(\sum_j w_{ji}^{(k)} y_j^{(k-1)} + b_i^{(k)}\right)$$

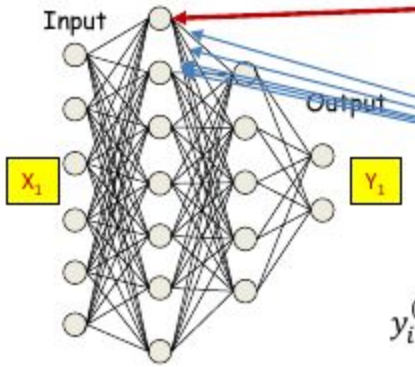
$$= \alpha\sigma\left(\sum_j w_{ji}^{(k)} \alpha\sigma\left(\sum_j w_{ji}^{(k-1)} y_j^{(k-2)} + b_i^{(k-1)}\right) + b_i^{(k)}\right)$$

$$= \alpha\sigma\left(\sum_j \left(\alpha w_{ji}^{(k)}\right)\sigma\left(\sum_j w_{ji}^{(k-1)} y_j^{(k-2)} + b_i^{(k-1)}\right) + b_i^{(k)}\right)$$

$$W_{test} = \alpha W_{trained}$$

- Instead of multiplying every output by α, multiply all weights by α

Dropout during test: implementation



apply α here (to the output of the neuron) OR..

Push the α to all outgoing weights

$$y_i^{(k)} = \alpha\sigma\left(\sum_j w_{ji}^{(k)} y_j^{(k-1)} + b_i^{(k)}\right)$$

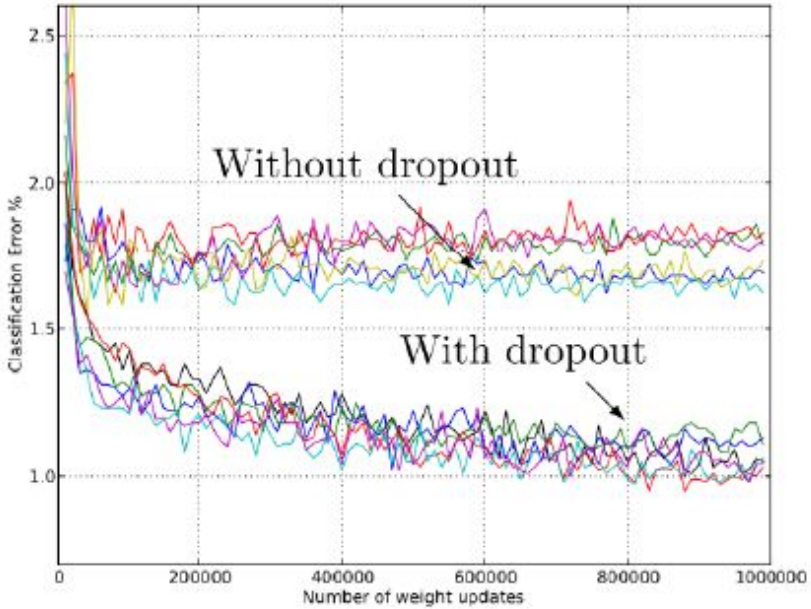
$$= \alpha\sigma\left(\sum_j w_{ji}^{(k)} \alpha\sigma\left(\sum_j w_{ji}^{(k-1)} y_j^{(k-2)} + b_i^{(k-1)}\right) + b_i^{(k)}\right)$$

$$= \alpha\sigma\left(\sum_j \left(\alpha w_{ji}^{(k)}\right)\sigma\left(\sum_j w_{ji}^{(k-1)} y_j^{(k-2)} + b_i^{(k-1)}\right) + b_i^{(k)}\right)$$

$$W_{test} = \alpha W_{trained}$$

- Instead of multiplying every output by α, multiply all weights by α

Dropout: Typical results



- From Srivastava et al., 2013. Test error for different architectures on MNIST with and without dropout
  - 2-4 hidden layers with 1024-2048 units

Variations on dropout

- **Zoneout: For RNNs**
  - Randomly chosen units remain unchanged across a time transition
- **Dropconnect**
  - Drop individual connections, instead of nodes
- **Shakeout**
  - Scale *up* the weights of randomly selected weights
    - $|w| \rightarrow a\,|w| + (1-a)\,c$
  - Fix remaining weights to a negative constant
    - $w \rightarrow -c$
- **Whiteout**
  - Add or multiply weight-dependent Gaussian noise to the signal on each connection

Early stopping

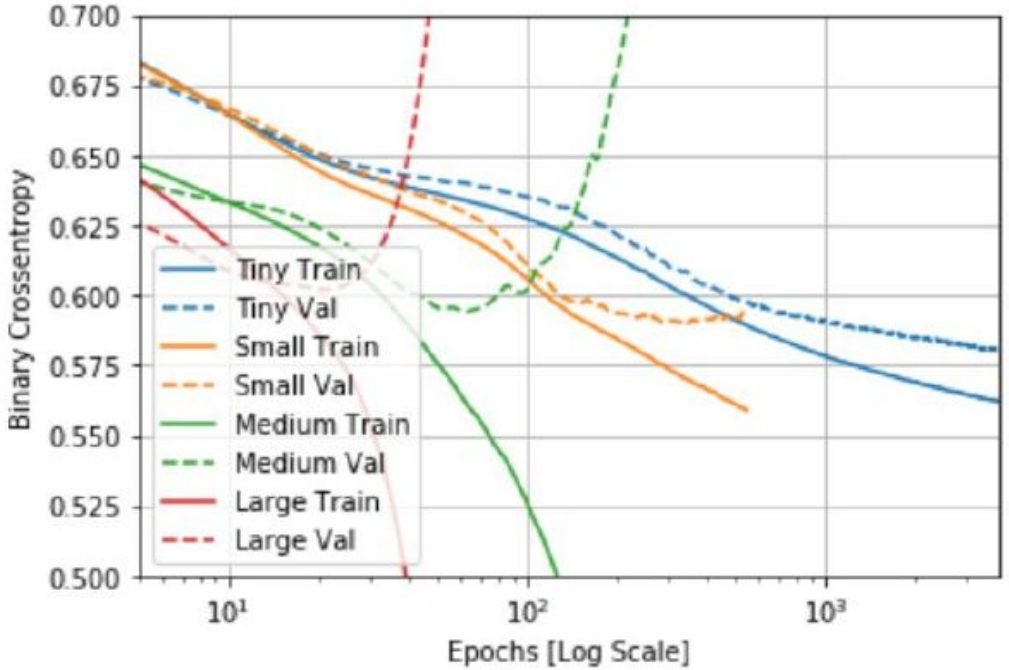
- When training, also output validation error
- Every time validation error improved, store a copy of the weights
- When validation error not improved for some time, stop
- Return the copy of the weights stored

Early stopping

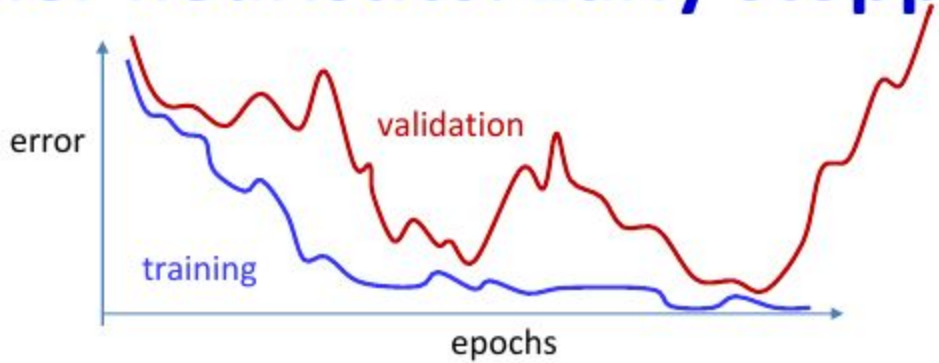
- hyperparameter selection: training step is the hyperparameter
- Advantage
  - Efficient: along with training; only store an extra copy of weights
  - Simple: no change to the model/algo
- Disadvantage: need validation data

Early Stopping

```
def get_callbacks(name):
  return [
    tfdocs.modeling.EpochDots(),
    tf.keras.callbacks.EarlyStopping(monitor='val_binary_crossentropy', patience=200),
    tf.keras.callbacks.TensorBoard(logdir/name),
  ]
```



Other heuristics: Early stopping



- Continued training can result in severe over fitting to training data
  - Track performance on a held-out validation set
  - Apply one of several early-stopping criterion to terminate training when performance on validation set degrades significantly

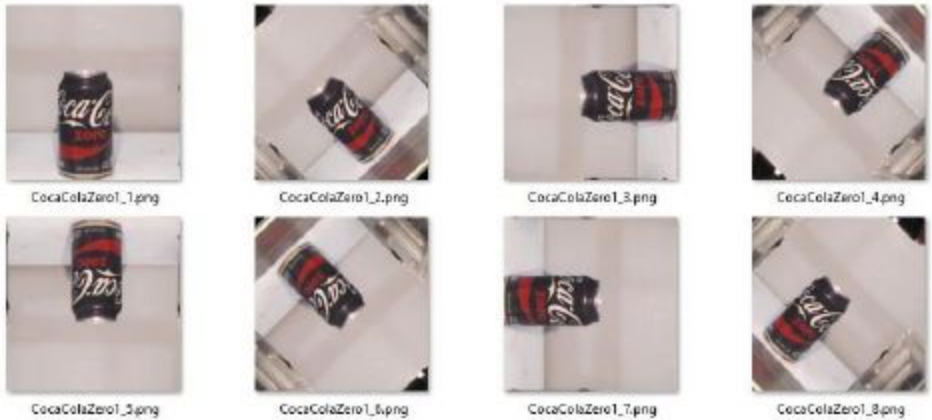
Additional heuristics: Gradient clipping

- Often the derivative will be too high
  - When the divergence has a steep slope
  - This can result in instability
- **Gradient clipping**: set a ceiling on derivative value

$$if\ \partial_w D > \theta\ then\ \partial_w D = \theta$$

  - Typical θ value is 5

Additional heuristics: Data Augmentation



- Available training data will often be small
- "Extend" it by distorting examples in a variety of ways to generate synthetic labelled examples
  - E.g. rotation, stretching, adding noise, other distortion

Data augmentation with Noise

- Adding noise to the input: a special kind of augmentation
- (Adding noise to the weights, outputs)

- Be careful about the transformation applied:
  - Example: classifying ‘b’ and ‘d’
  - Example: classifying ‘6’ and ‘9’

Other tricks

- Normalize the input:
  - Apply covariate shift to entire training data to make it 0 mean, unit variance
  - Equivalent of batch norm on input
- A variety of other tricks are applied
  - Initialization techniques
    - Typically initialized randomly
    - Key point: neurons with identical connections that are identically initialized will never diverge
  - Practice makes man perfect

Setting up a problem

- Obtain training data
  - Use appropriate representation for inputs and outputs
- Choose network architecture
  - More neurons need more data
  - Deep is better, but harder to train
- Choose the appropriate divergence function
  - Choose regularization
- Choose heuristics (batch norm, dropout, etc.)
- Choose optimization algorithm
  - E.g. Adagrad
- Perform a grid search for hyper parameters (learning rate, regularization parameter, …) on held-out data
- Train
  - Evaluate periodically on validation data, for early stopping if required

Experiment

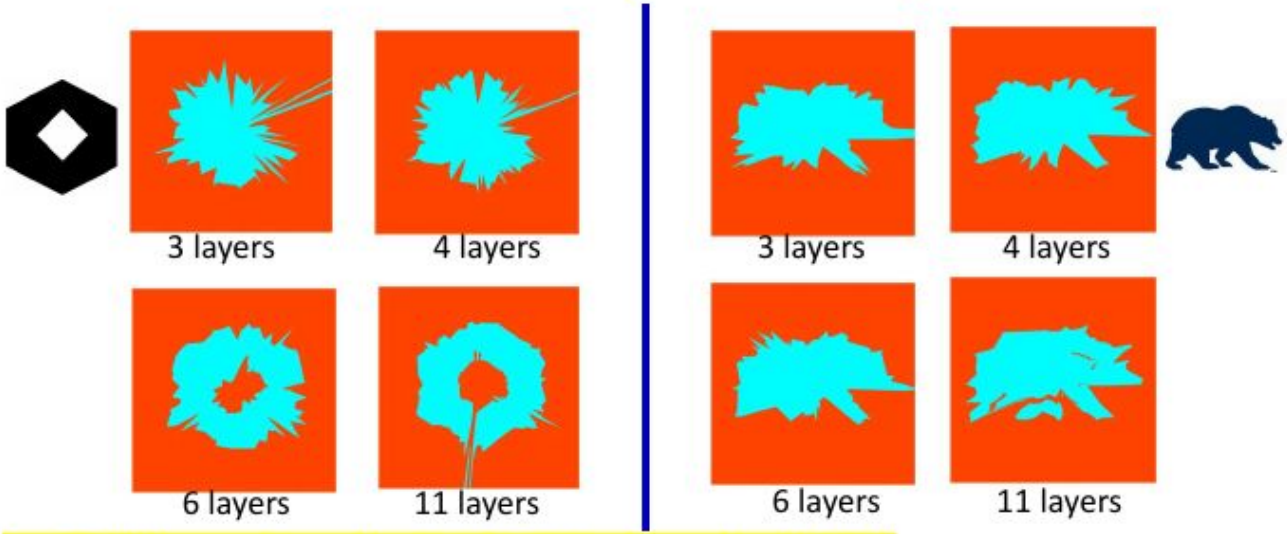
https://playground.tensorflow.org

- Multiple synthetic dataset available
- Observe the effect of varying hyper-parameters
  - # of hidden layers
  - # of hidden nodes in hidden layers
  - Regularization type, amount
  - Activation function
  - Learning rate
  - Feature transformation

Topics for the session

- Optimization
  - Characteristics of Loss Surface
  - Optimal Learning Rate
  - Derivative-Inspired weight updates
  - Momentum based weight updates
  - Batch Vs. Stochastic Gradient Vs. Mini-batch
  - Higher-order weight updates
  - Batch Normalization

Variance and Depth



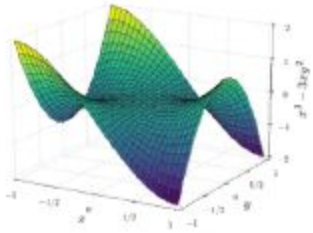
- Dark figures show desired decision boundary (2D)
  - 1000 training points, 660 hidden neurons
  - Network heavily overdesigned even for shallow nets
- **Anecdotal: Variance decreases with**
  - Depth
  - Data

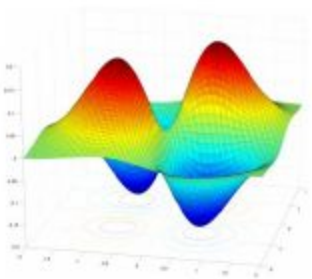
10000 training instances

33

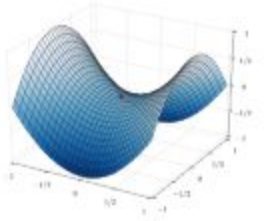
The Error Surface

- **Popular hypothesis**:
  - In large networks, saddle points are far more common than local minima
    - Frequency exponential in network size
  - Most local minima are equivalent
    - And close to global minimum
  - This is not true for small networks



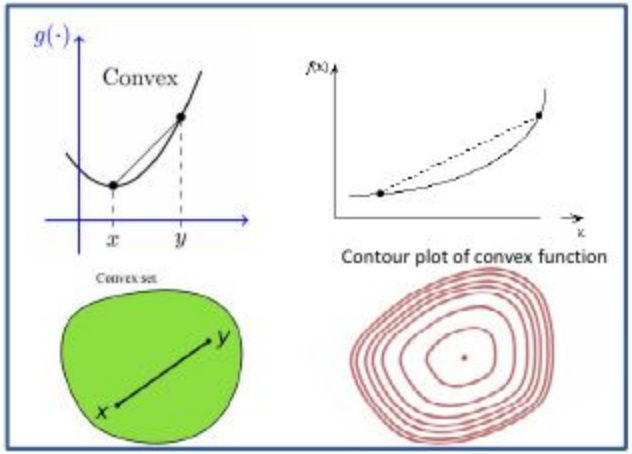


- **Saddle point:** A point where
  - The slope is zero
  - The surface increases in some directions, but decreases in others
    - Some of the Eigenvalues of the Hessian are positive; others are negative
  - Gradient descent algorithms like saddle points

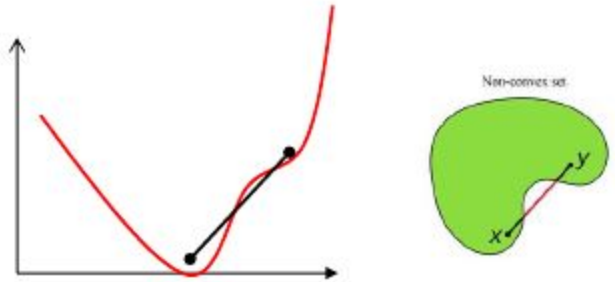


Convex Loss Functions

- A surface is "convex" if it is continuously curving upward
  - We can connect any two points above the surface without intersecting it
  - Many mathematical definitions that are equivalent

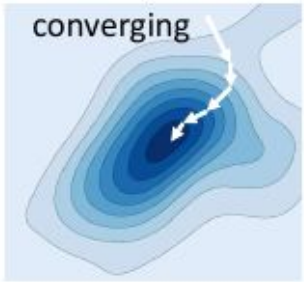


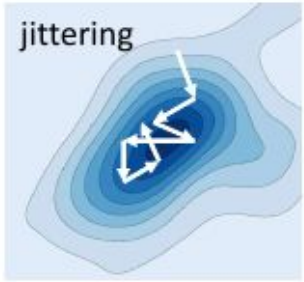
- Caveat: Neural network error surface is generally not convex
  - Streetlight effect

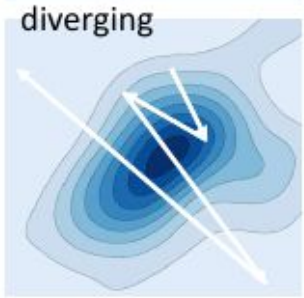


Convergence of gradient descent

- An iterative algorithm is said to *converge* to a solution if the value updates arrive at a fixed point
  - Where the gradient is 0 and further updates do not change the estimate
- The algorithm may not actually converge
  - It may jitter around the local minimum
  - It may even diverge
- Conditions for convergence?







Convergence and convergence rate

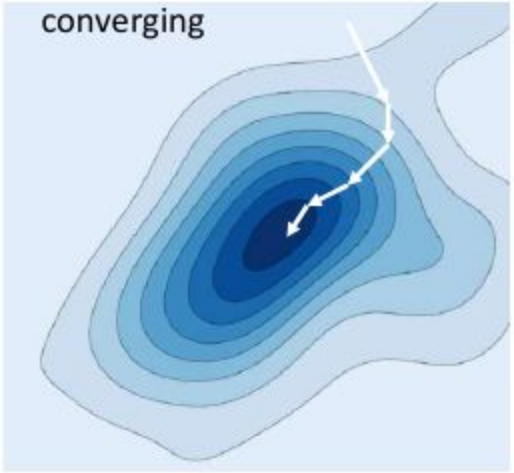
- Convergence rate: How fast the iterations arrive at the solution
- Generally quantified as

$$R = \frac{\left|f\left(x^{(k+1)}\right) - f(x^*)\right|}{\left|f(x^{(k)}) - f(x^*)\right|}$$

  - $x^{(k+1)}$ is the k-th iteration
  - x^* is the optimal value of x

- If R is a constant (or upper bounded), the convergence is *linear*
  - In reality, its arriving at the solution exponentially fast

$$\left|f\left(x^{(k)}\right) - f(x^*)\right| = c^k \left|f\left(x^{(0)}\right) - f(x^*)\right|$$

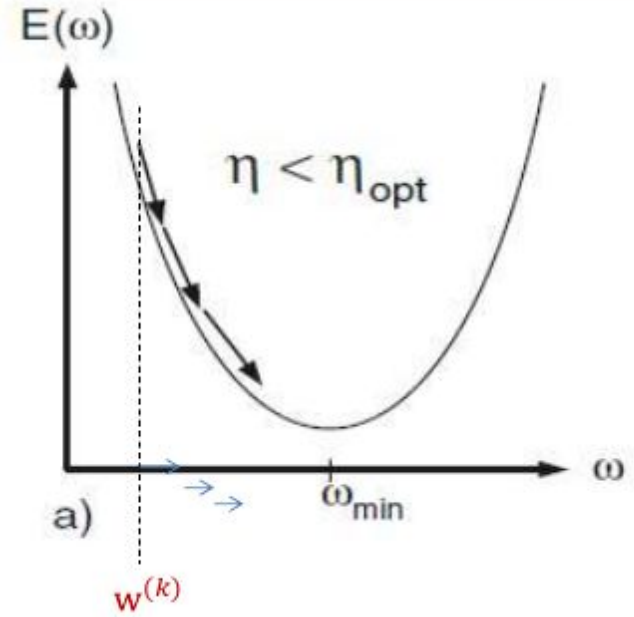


Convergence for quadratic surfaces

$$Minimize\ E = \frac{1}{2}aw^2 + bw + c$$

$$\mathrm{w}^{(k+1)} = \mathrm{w}^{(k)} - \eta \frac{dE\left(\mathrm{w}^{(k)}\right)}{d\mathrm{w}}$$

Gradient descent with fixed step size η to estimate *scalar* parameter w

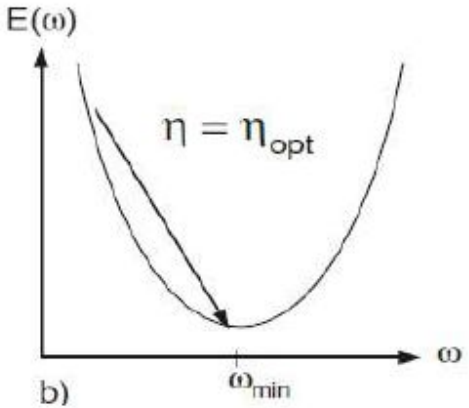


- Gradient descent to find the optimum of a quadratic, starting from $\mathrm{w}^{(k)}$
- Assuming fixed step size η
- What is the optimal step size η to get there fastest?

Convergence for quadratic surfaces

$$E = \frac{1}{2}aw^2 + bw + c$$

$$w^{(k+1)} = w^{(k)} - \eta \frac{dE\left(w^{(k)}\right)}{dw}$$



- Any quadratic objective can be written as

$$E = E\left(w^{(k)}\right) + E'\left(w^{(k)}\right)\left(w - w^{(k)}\right) + \frac{1}{2}E''\left(w^{(k)}\right)\left(w - w^{(k)}\right)^2$$

  - Taylor expansion

- Minimizing w.r.t w, we get (Newton's method)

$$w_{min} = w^{(k)} - E''\left(w^{(k)}\right)^{-1} E'\left(w^{(k)}\right)$$

- Note:

$$\frac{dE\left(w^{(k)}\right)}{dw} = E'\left(w^{(k)}\right)$$

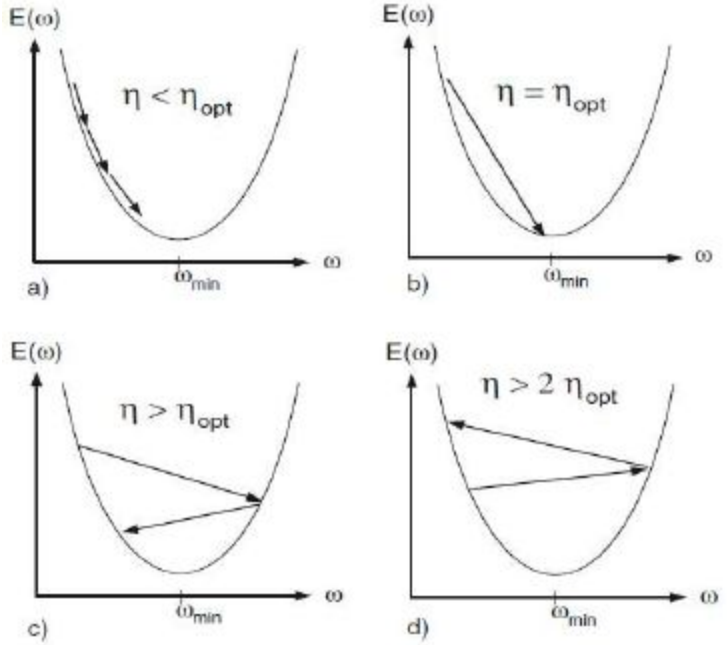
- Comparing to the gradient descent rule, we see that we can arrive at the optimum in a single step using the optimum step size

$$\eta_{opt} = E''\left(w^{(k)}\right)^{-1}$$

With non-optimal step size

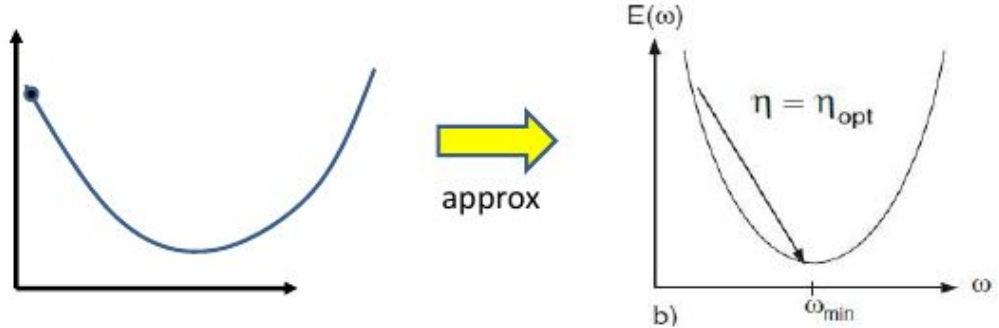
$$w^{(k+1)} = w^{(k)} - \eta \frac{dE\left(w^{(k)}\right)}{dw}$$

Gradient descent with fixed step size η to estimate scalar parameter w



- For $\eta < \eta_{opt}$ the algorithm will converge monotonically
- For $2\eta_{opt} > \eta > \eta_{opt}$ we have oscillating convergence
- For $\eta > 2\eta_{opt}$ we get divergence

For generic differentiable convex objectives



- Any differentiable convex objective $E(w)$ can be approximated as

$$E \approx E\left(\mathrm{w}^{(k)}\right) + \left(w - \mathrm{w}^{(k)}\right)\frac{dE\left(\mathrm{w}^{(k)}\right)}{dw} + \frac{1}{2}\left(w - \mathrm{w}^{(k)}\right)^2 \frac{d^2E\left(\mathrm{w}^{(k)}\right)}{dw^2} + \cdots$$

  - Taylor expansion
- Using the same logic as before, we get (Newton's method)

$$\eta_{opt} = \left(\frac{d^2E\left(\mathrm{w}^{(k)}\right)}{dw^2}\right)^{-1}$$

- We can get divergence if $\eta \geq 2\eta_{opt}$

For functions of *multivariate* inputs

$E = g(\mathbf{w})$, $\mathbf{w}$ is a vector $\mathbf{w} = [w_1, w_2, \ldots, w_N]$

- Consider a simple quadratic convex (paraboloid) function

$$E = \frac{1}{2}\mathbf{w}^T\mathbf{A}\mathbf{w} + \mathbf{w}^T\mathbf{b} + c$$

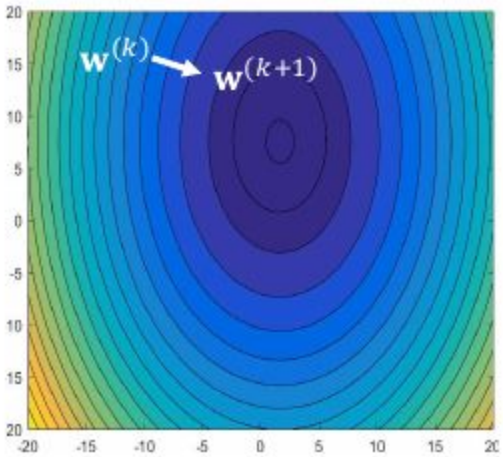
  - Since $E^T = E$ (E is scalar), $\mathbf{A}$ can always be made symmetric
    - For **convex** E, $\mathbf{A}$ is always positive definite, and has positive eigenvalues

- When $\mathbf{A}$ is diagonal:

$$E = \frac{1}{2}\sum_i \left(a_{ii}w_i^2 + b_i w_i\right) + c$$

  - The w_is are *uncoupled*
  - For *convex* (paraboloid) E, the a_{ii} values are all positive
  - Just an sum of N independent quadratic functions

Vector update rule



$$\mathbf{w}^{(k+1)} \leftarrow \mathbf{w}^{(k)} - \eta \nabla_{\mathbf{w}} E$$

$$w_i^{(k+1)} = w_i^{(k)} - \eta \frac{dE\left(w_i^{(k)}\right)}{d\mathrm{w}}$$

- Conventional vector update rules for gradient descent: update entire vector against direction of gradient
  - Note : Gradient is perpendicular to equal value contour
  - The same learning rate is applied to all components

Problem with vector update rule

$$\mathbf{w}^{(k+1)} \leftarrow \mathbf{w}^{(k)} - \eta \nabla_{\mathbf{w}} E^T$$

$$w_i^{(k+1)} = w_i^{(k)} - \eta \frac{dE\left(w_i^{(k)}\right)}{dw}$$

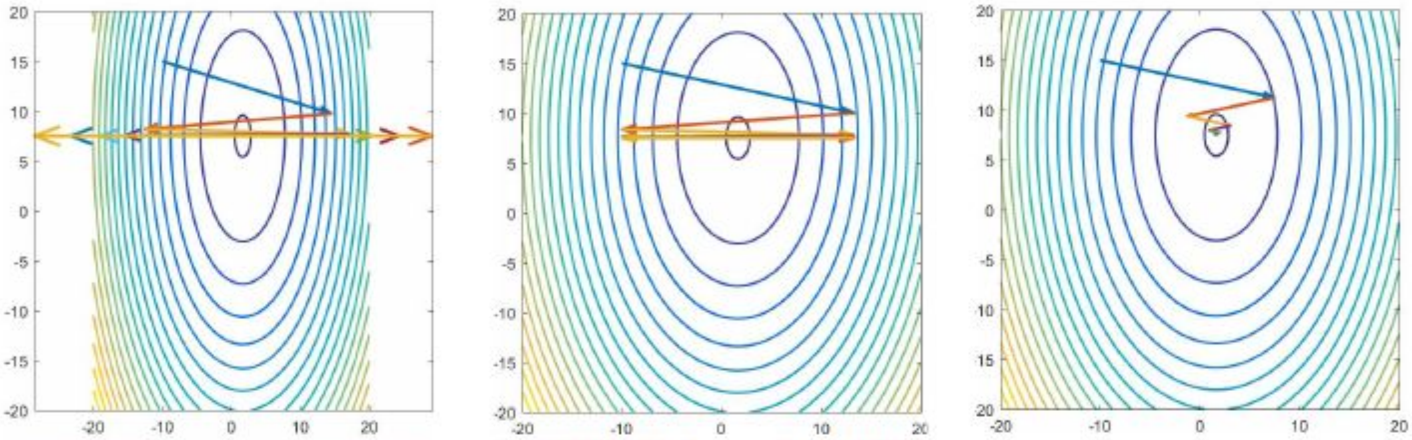
$$\eta_{i,opt} = \left(\frac{d^2 E\left(w_i^{(k)}\right)}{dw_i^2}\right)^{-1} = a_{ii}^{-1}$$

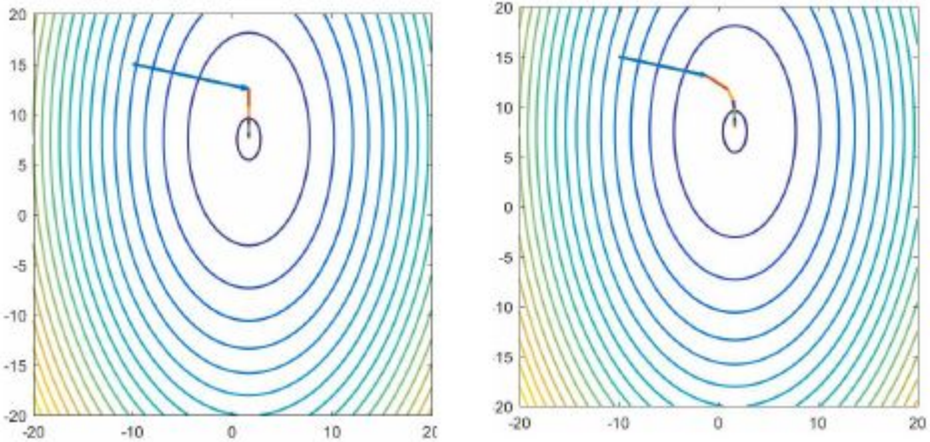
- The learning rate must be lower than twice the *smallest* optimal learning rate for any component

$$\eta < 2 \min_i \eta_{i,opt}$$

  - Otherwise the learning will diverge
- This, however, makes the learning very slow
  - And will oscillate in all directions where $\eta_{i,opt} \leq \eta < 2\eta_{i,opt}$

Dependence on learning rate





- $\eta_{1,opt} = 1;\ \eta_{2,opt} = 0.33$
- $\eta = 2.1\eta_{2,opt}$
- $\eta = 2\eta_{2,opt}$
- $\eta = 1.5\eta_{2,opt}$
- $\eta = \eta_{2,opt}$
- $\eta = 0.75\eta_{2,opt}$

Convergence

- Convergence behaviors become increasingly unpredictable as dimensions increase

- For the fastest convergence, ideally, the learning rate η must be close to both, the largest $\eta_{i,opt}$ and the smallest $\eta_{i,opt}$
  - To ensure convergence in every direction
  - Generally infeasible

- Convergence is particularly slow if $\frac{\max_i \eta_{i,opt}}{\min_i \eta_{i,opt}}$ is large
  - The "condition" number is small

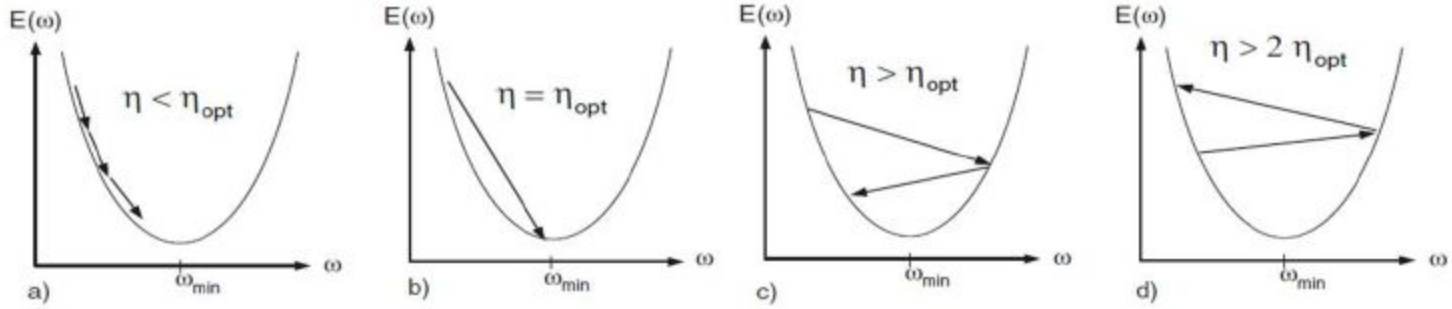
Issues: 1. The Hessian

- Normalized update rule

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

- For complex models such as neural networks, with a very large number of parameters, the Hessian $H_E\left(\boldsymbol{w}^{(k)}\right)$ is extremely difficult to compute
  - For a network with only 100,000 parameters, the Hessian will have 10^{10} cross-derivative terms
  - And its even harder to invert, since it will be enormous

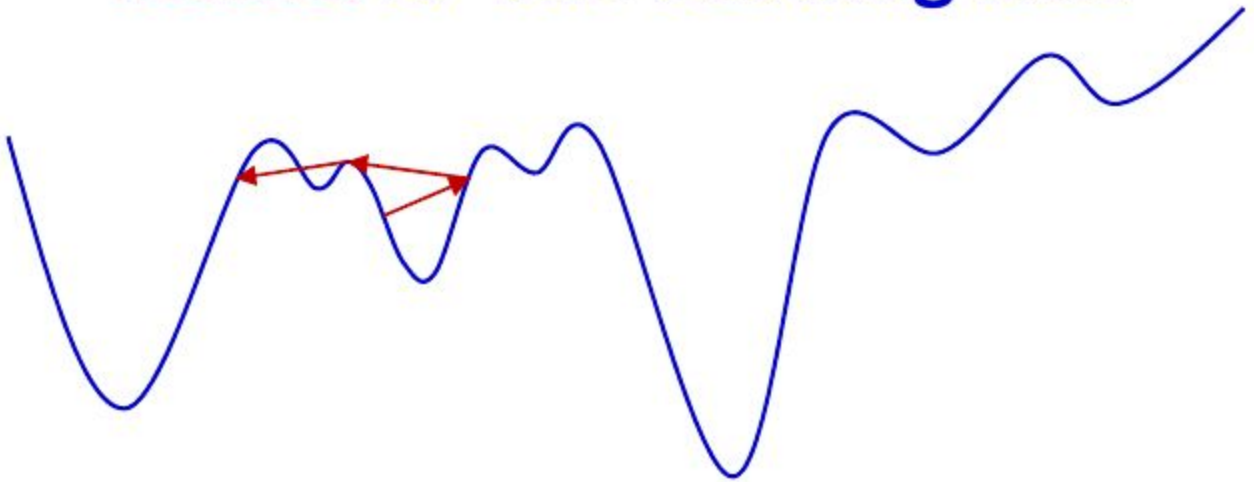
Issues: 2. The learning rate



- Much of the analysis we just saw was based on trying to ensure that the step size was not so large as to cause divergence within a convex region

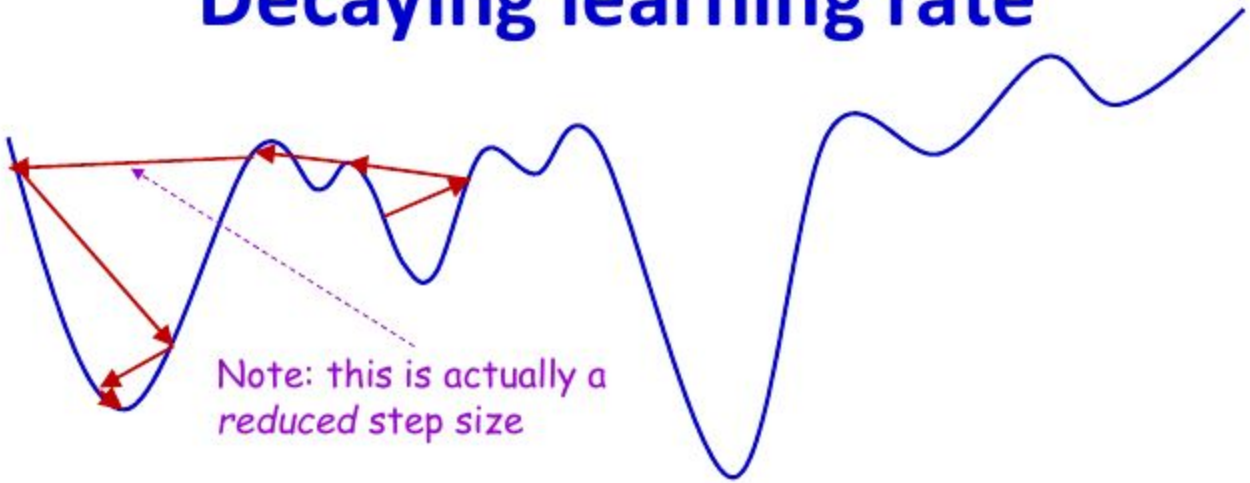
  $-\ \eta < 2\eta_{opt}$

Issues: 2. The learning rate



- For complex models such as neural networks the loss function is often not convex
  - Having $\eta > 2\eta_{opt}$ can actually help escape local optima
- However *always* having $\eta > 2\eta_{opt}$ will ensure that you never ever actually find a solution

Decaying learning rate



- Start with a large learning rate
  - Greater than 2 (assuming Hessian normalization)
  - Gradually reduce it with iterations

Decaying learning rate

- Typical decay schedules
  - Linear decay: $\eta_k = \frac{\eta_0}{k+1}$
  - Quadratic decay: $\eta_k = \frac{\eta_0}{(k+1)^2}$
  - Exponential decay: $\eta_k = \eta_0 e^{-\beta k}$, where $\beta > 0$

- A common approach (for nnets):
  1. Train with a fixed learning rate η until loss (or performance on a held-out data set) stagnates
  2. $\eta \leftarrow \alpha\eta$, where $\alpha < 1$ (typically 0.1)
  3. Return to step 1 and continue training from where we left off

Derivative-*inspired* algorithms

- Algorithms that use derivative information for trends, but do not follow them absolutely

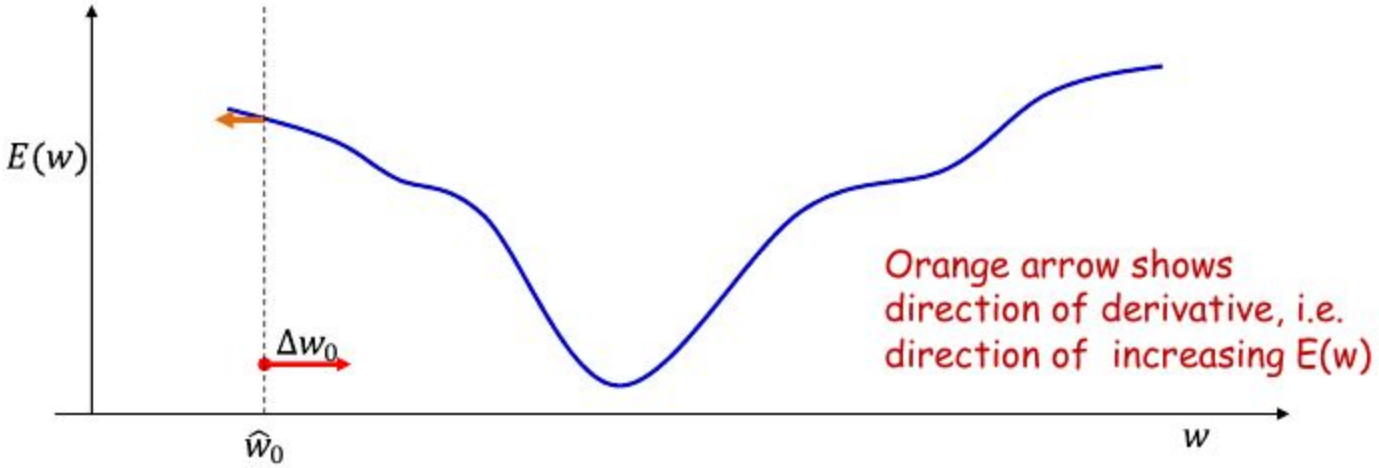
- Rprop
- Quick prop

RProp

- *Resilient* propagation
- Simple algorithm, to be followed *independently* for each component
  - I.e. steps in different directions are not coupled

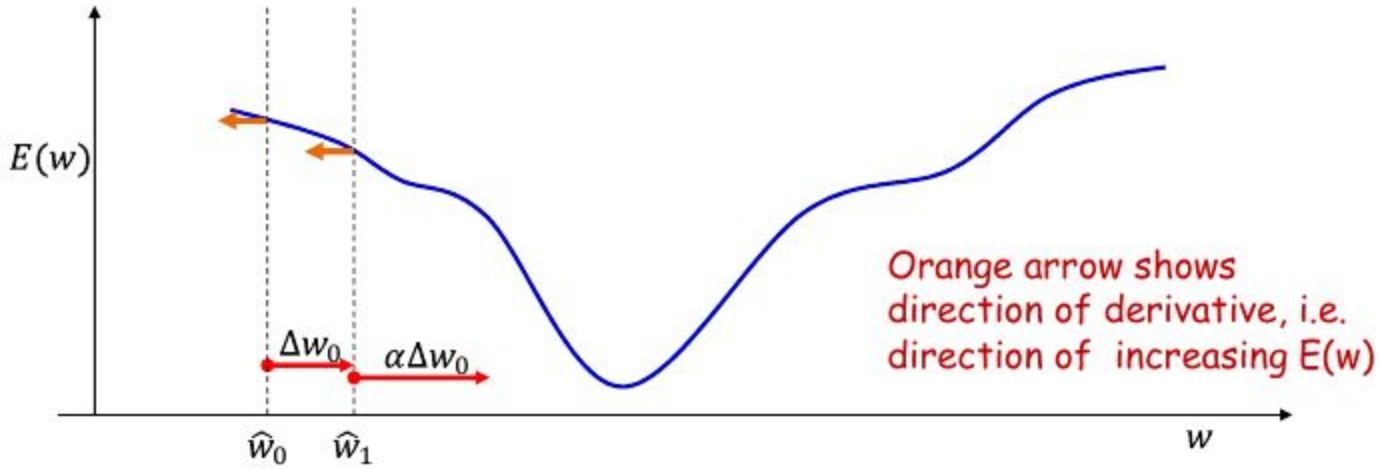
- At each time
  - If the derivative at the current location recommends continuing in the same direction as before (i.e. has not changed sign from earlier):
    - *increas* the step, and continue in the same direction
  - If the derivative has changed sign (i.e. we've overshot a minimum)
    - *reduce* the step and reverse direction

Rprop



- Select an initial value $\hat{w}$ and compute the derivative
  - Take an initial step Δw against the derivative
    - In the direction that reduces the function
      - $\Delta w = sign\left(\frac{dE(\hat{w})}{dw}\right)\Delta w$
      - $\hat{w} = \hat{w} - \Delta w$

Rprop

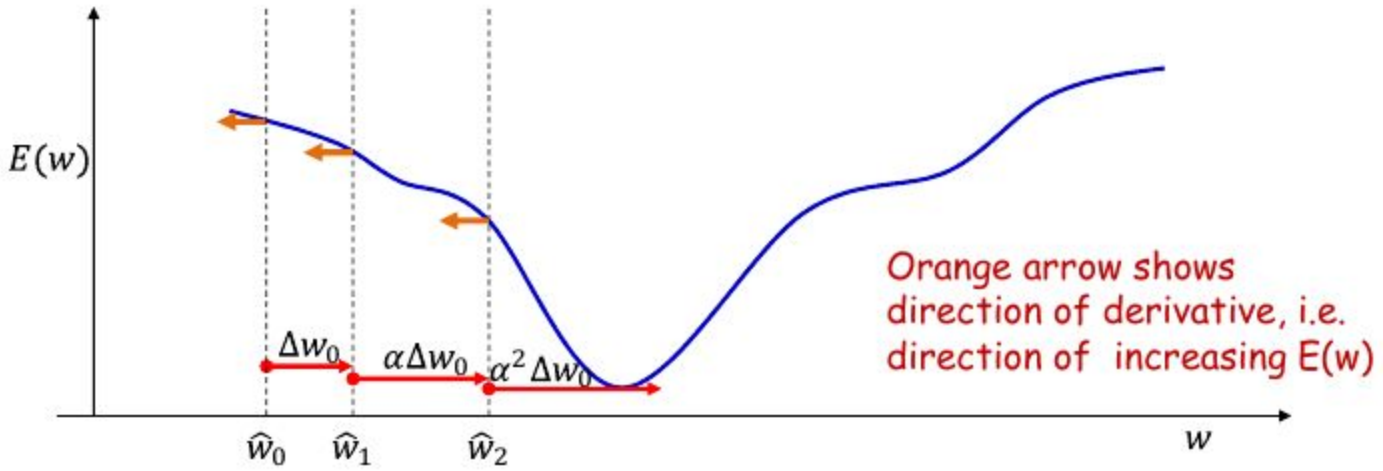


- Compute the derivative in the new location
  - If the derivative has not changed sign from the previous location, increase the step size and take a step

$\alpha > 1$

- $\Delta w = \alpha \Delta w$
- $\widehat{w} = \widehat{w} - \Delta w$

Rprop

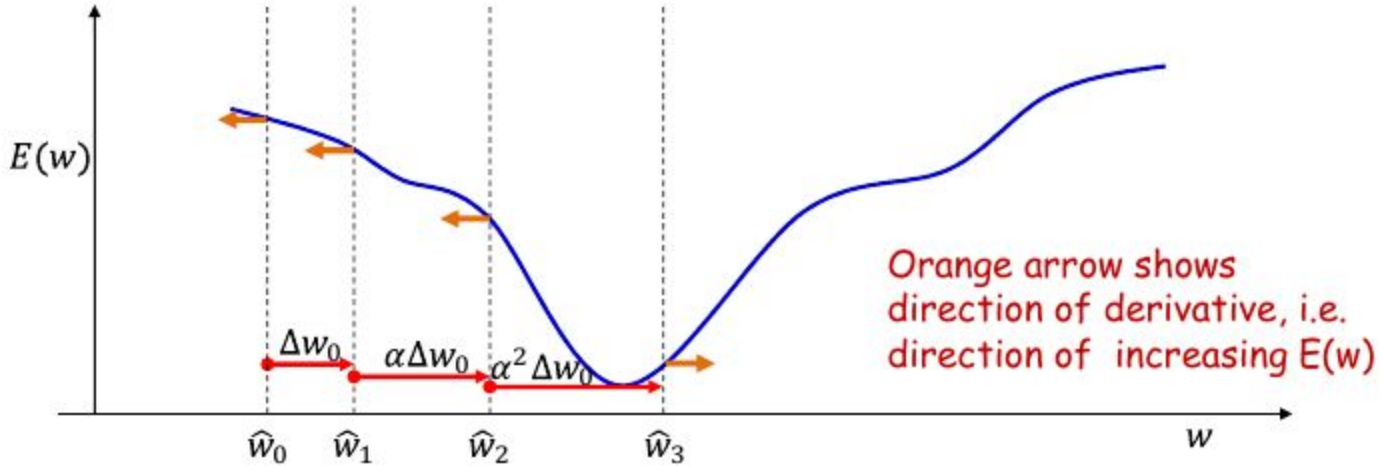


- Compute the derivative in the new location
  - If the derivative has not changed sign from the previous location, increase the step size and take a step

$\alpha > 1$

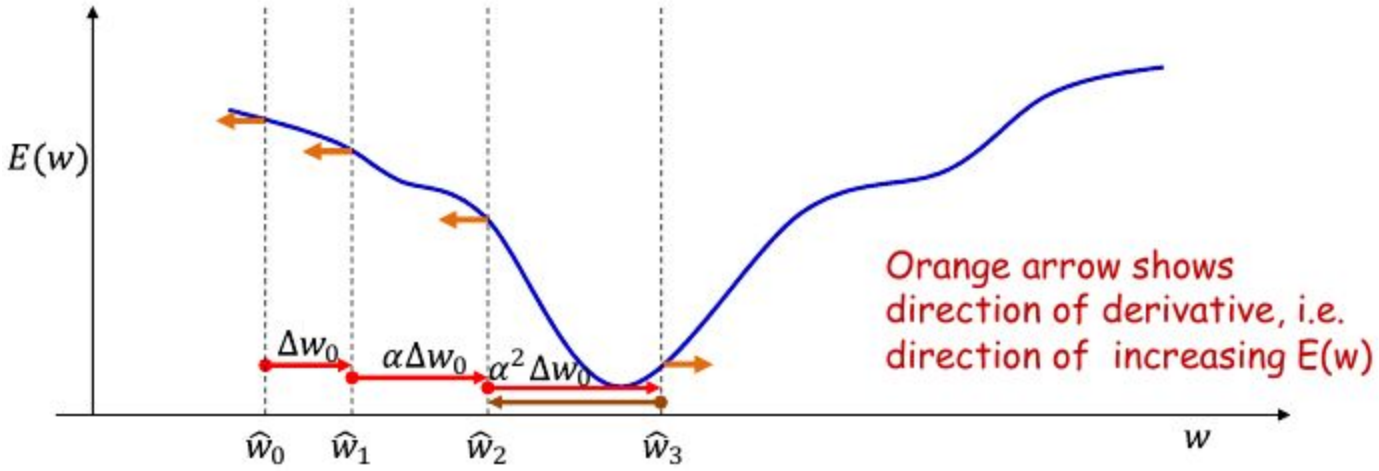
- $\Delta w = \alpha\Delta w$
- $\widehat{w} = \widehat{w} - \Delta w$

Rprop



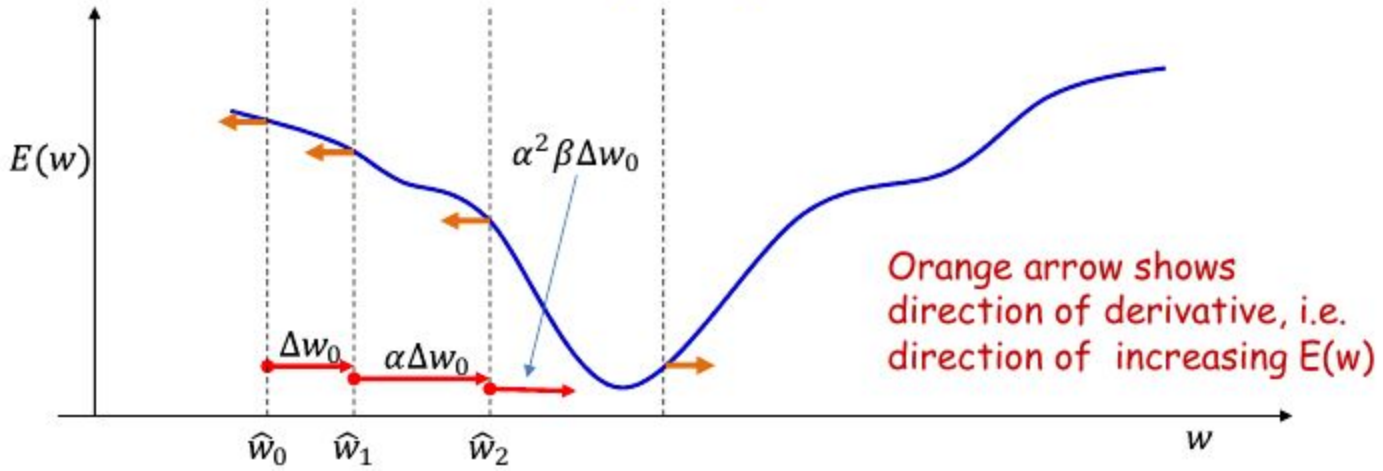
- Compute the derivative in the new location
  - If the derivative has changed sign

Rprop



- Compute the derivative in the new location
  - If the derivative has changed sign
  - Return to the previous location
    - $\hat{w} = \hat{w} + \Delta w$

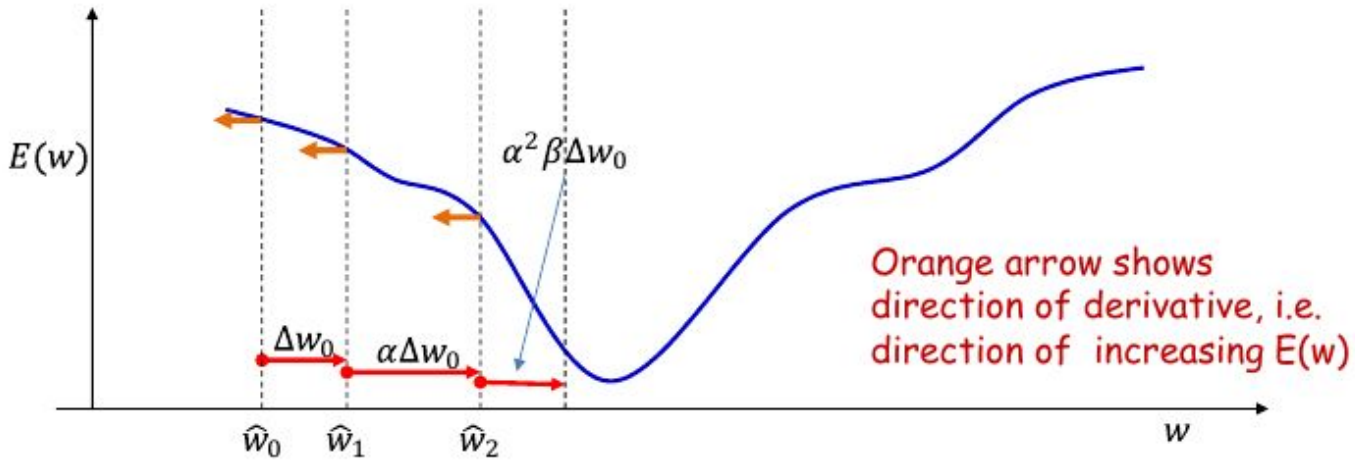
Rprop



- Compute the derivative in the new location
  - If the derivative has changed sign
  - Return to the previous location
    - $\hat{w} = \hat{w} + \Delta w$
  - Shrink the step
    - $\Delta w = \beta \Delta w$

$\beta < 1$

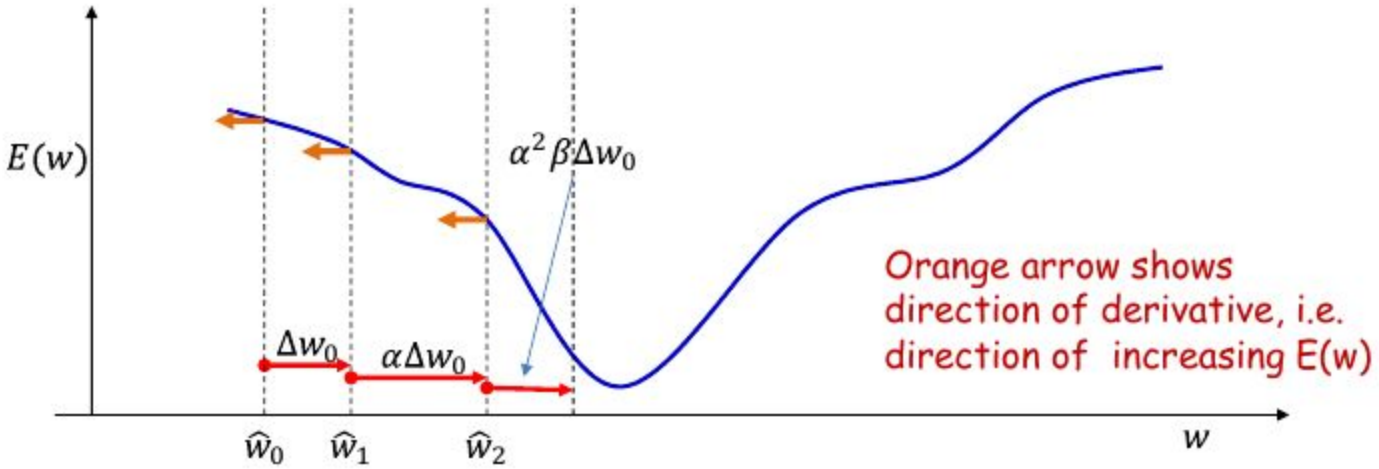
Rprop



- Compute the derivative in the new location
  - If the derivative has changed sign
  - Return to the previous location
    - $\hat{w} = \hat{w} + \Delta w$
  - Shrink the step
    - $\Delta w = \beta \Delta w$
  - Take the smaller step forward
    - $\hat{w} = \hat{w} - \Delta w$

$\beta < 1$

Rprop



- Compute the derivative in the new location
  - If the derivative has changed sign
  - Return to the previous location
    - $\hat{w} = \hat{w} + \Delta w$
  - Shrink the step
    - $\Delta w = \beta \Delta w$
  - Take the smaller step forward
    - $\hat{w} = \hat{w} - \Delta w$

$\beta < 1$

Rprop (simplified)

- Set $\alpha = 1.2, \beta = 0.5$
- For each layer l, for each i, j:
  - Initialize $w_{l,i,j}$, $\Delta w_{l,i,j} > 0$,
  - $prevD(l,i,j) = \frac{dErr(w_{l,i,j})}{dw_{l,i,j}}$
  - $\Delta w_{l,i,j} = \text{sign}\big(prevD(l,i,j)\big)\Delta w_{l,i,j}$
  - While not converged:
    - $w_{l,i,j} = w_{l,i,j} - \Delta w_{l,i,j}$
    - $D(l,i,j) = \frac{dErr(w_{l,i,j})}{dw_{l,i,j}}$
    - If $\text{sign}\big(prevD(l,i,j)\big) == \text{sign}\big(D(l,i,j)\big)$:
      - $\Delta w_{l,i,j} = \min(\alpha \Delta w_{l,i,j}, \Delta_{max})$
      - $prevD(l,i,j) = D(l,i,j)$
    - else:
      - $w_{l,i,j} = w_{l,i,j} + \Delta w_{l,i,j}$
      - $\Delta w_{l,i,j} = \max(\beta \Delta w_{l,i,j}, \Delta_{min})$

Ceiling and floor on step

Rprop (simplified)

- Set $\alpha = 1.2, \beta = 0.5$
- For each layer l, for each i, j:
  - Initialize $w_{l,i,j}$, $\Delta w_{l,i,j} > 0$,
  - $prevD(l,i,j) = \frac{dErr(w_{l,i,j})}{dw_{l,i,j}}$
  - $\Delta w_{l,i,j} = \text{sign}\big(prevD(l,i,j)\big)\Delta w_{l,i,j}$
  - While not converged:
    - $w_{l,i,j} = w_{l,i,j} - \Delta w_{l,i,j}$
    - $D(l,i,j) = \frac{dErr(w_{l,i,j})}{dw_{l,i,j}}$
    - If $\text{sign}\big(prevD(l,i,j)\big) == \text{sign}\big(D(l,i,j)\big)$:
      - $\Delta w_{l,i,j} = \alpha \Delta w_{l,i,j}$
      - $prevD(l,i,j) = D(l,i,j)$
    - else:
      - $w_{l,i,j} = w_{l,i,j} + \Delta w_{l,i,j}$
      - $\Delta w_{l,i,j} = \beta \Delta w_{l,i,j}$

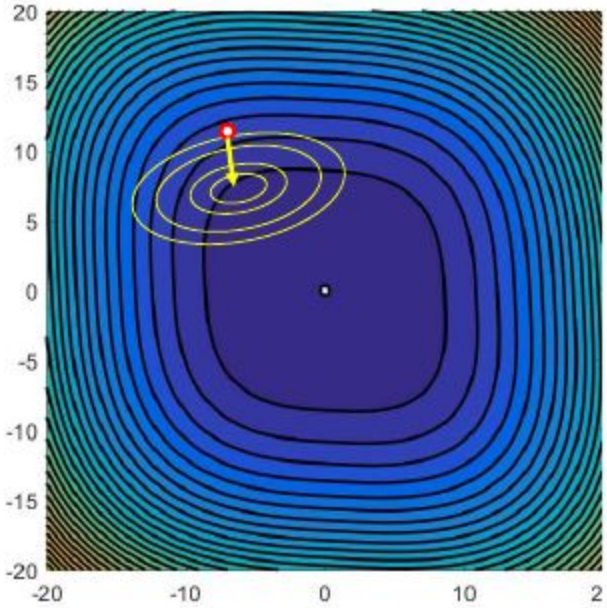
Obtained via backprop

Note: Different parameters updated independently

RProp

- A remarkably simple first-order algorithm, that is frequently much more efficient than gradient descent.
  - And can even be competitive against some of the more advanced second-order methods
- Only makes minimal assumptions about the loss function
  - No convexity assumption

QuickProp

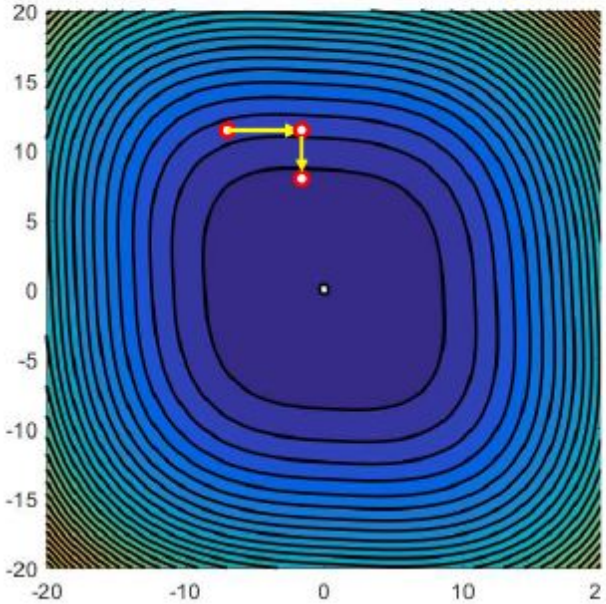


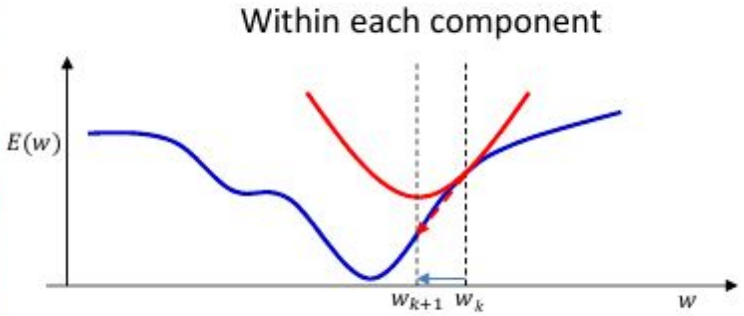
- Quickprop employs the Newton updates with two modifications

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

- But with two modifications

QuickProp: Modification 1



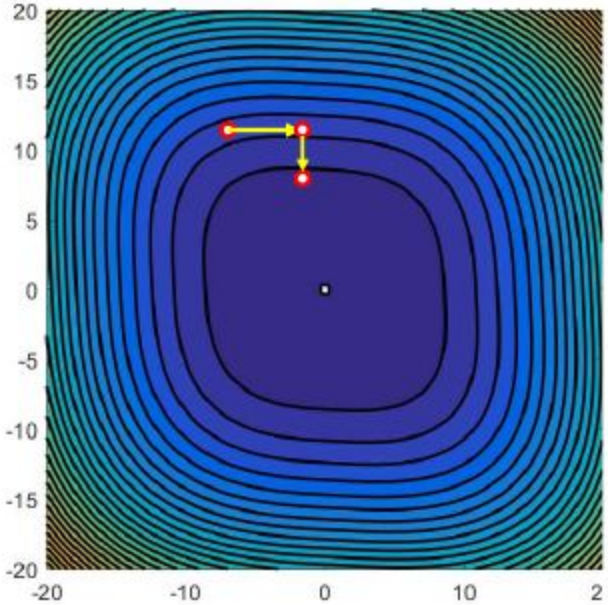


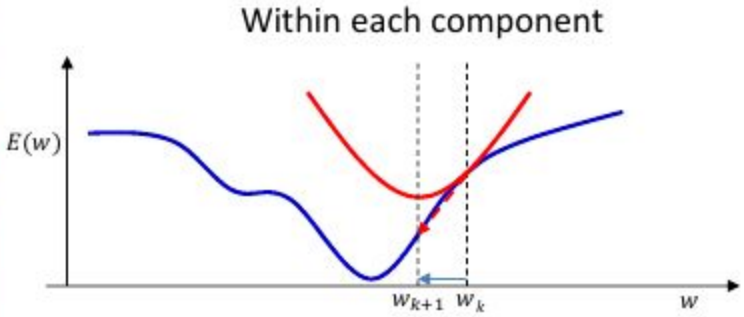
- It treats each dimension independently
- For $i = 1{:}N$

$$w_i^{k+1} = w_i^k - E''\left(w_i^k | w_j^k, j \neq i\right)^{-1} E'\left(w_i^k | w_j^k, j \neq i\right)$$

- This eliminates the need to compute and invert expensive Hessians

QuickProp: Modification 2





- **It approximates the second derivative through finite differences**
- For $i = 1{:}N$

$$w_i^{k+1} = w_i^k - D\left(w_i^k, w_i^{k-1}\right)^{-1} E'\left(w_i^k | w_j^k, j \neq i\right)$$

- This eliminates the need to compute expensive double derivatives

QuickProp

$$w^{(k+1)} = w^{(k)} - \left(\frac{E'\left(w^{(k)}\right) - E'(w^{(k-1)})}{\Delta w^{(k-1)}}\right)^{-1} E'(w^{(k)})$$

Finite-difference approximation to double derivative obtained assuming a quadratic $E()$

- Updates are independent for every parameter
- For every layer l, for every connection from node i in the $(l-1)^{\text{th}}$ layer to node j in the l^{th} layer:

$$\Delta w_{l,ij}^{(k)} = \frac{\Delta w_{l,ij}^{(k-1)}}{Err'\left(w_{l,ij}^{(k)}\right) - Err'\left(w_{l,ij}^{(k-1)}\right)} Err'\left(w_{l,ij}^{(k)}\right)$$

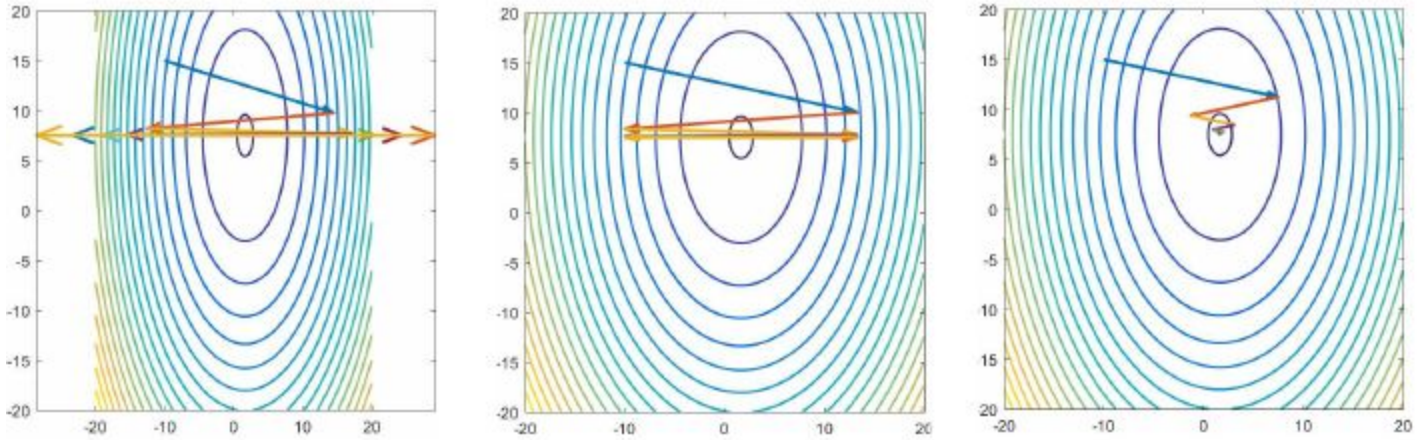
$$w_{l,ij}^{(k+1)} = w_{l,ij}^{(k)} - \Delta w_{l,ij}^{(k)}$$

Computed using backprop

Quickprop

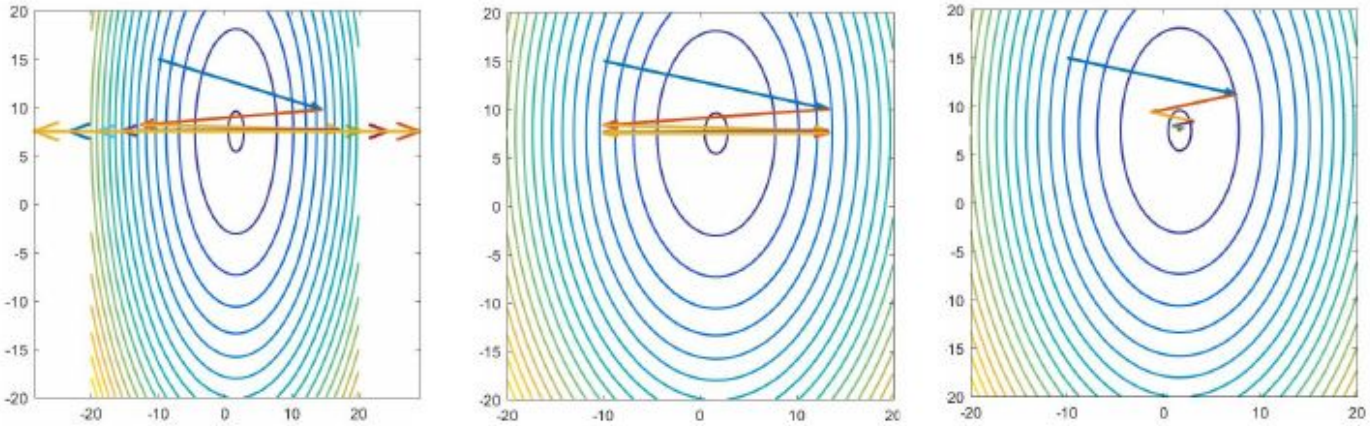
- Prone to some instability for non-convex objective functions
- But is still one of the fastest training algorithms for many problems

A closer look at the convergence problem



- With dimension-independent learning rates, the solution will converge smoothly in some directions, but oscillate or diverge in others

A closer look at the convergence problem

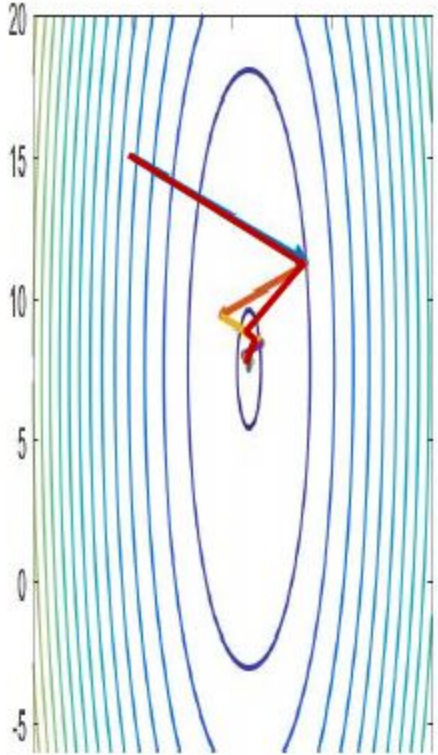


- With dimension-independent learning rates, the solution will converge smoothly in some directions, but oscillate or diverge in others

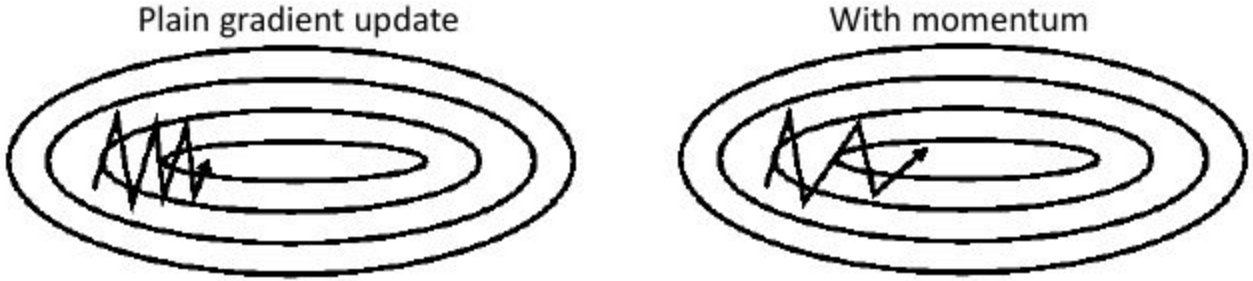
- **Proposal:**
  - Keep track of oscillations
  - Emphasize steps in directions that converge smoothly
  - Shrink steps in directions that bounce around..

The momentum methods

- Maintain a running average of all past steps
  - In directions in which the convergence is smooth, the average will have a large value
  - In directions in which the estimate swings, the positive and negative swings will cancel out in the average
- Update with the running average, rather than the current gradient



Momentum Update



- The momentum method maintains a running average of all gradients until the *current* step

$$\Delta W^{(k)} = \beta \Delta W^{(k-1)} - \eta \nabla_W Err\left(W^{(k-1)}\right)$$

$$W^{(k)} = W^{(k-1)} + \Delta W^{(k)}$$

  - Typical β value is 0.9
- The running average steps
  - Get longer in directions where gradient stays in the same sign
  - Become shorter in directions where the sign keeps flipping

Training by gradient descent

- Initialize all weights $\mathbf{W}_1, \mathbf{W}_2, \ldots, \mathbf{W}_K$
- Do:
  - For all i, j, k, initialize $\nabla_{W_k} Err = 0$
  - For all $t = 1:T$
    - For every layer k:
      - Compute $\nabla_{W_k} \boldsymbol{Div}(Y_t, d_t)$
      - Compute $\nabla_{W_k} Err \mathrel{+}= \frac{1}{T} \nabla_{W_k} \boldsymbol{Div}(Y_t, d_t)$
  - For every layer k:

$$W_k = W_k - \eta \nabla_{W_k} Err$$

- Until Err has converged

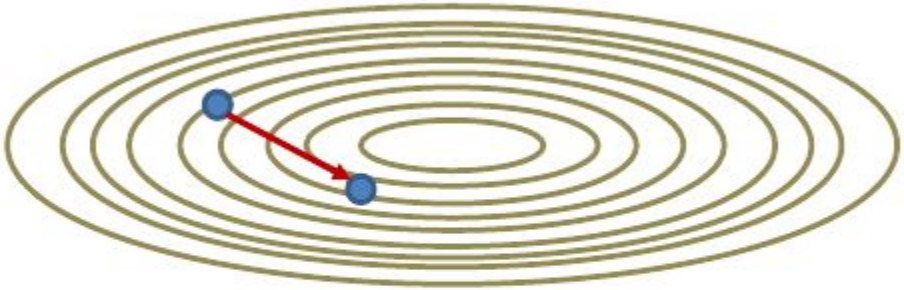
Training with momentum

- Initialize all weights $\mathbf{W}_1, \mathbf{W}_2, \ldots, \mathbf{W}_K$
- Do:
  - For all layers k, initialize $\nabla_{W_k} Err = 0$, $\Delta W_k = 0$
  - For all $t = 1:T$
    - For every layer k:
      - Compute gradient $\nabla_{W_k} \boldsymbol{Div}(Y_t, d_t)$
      - $\nabla_{W_k} Err \mathrel{+}= \frac{1}{T} \nabla_{W_k} \boldsymbol{Div}(Y_t, d_t)$
  - For every layer k

$$\Delta W_k = \beta \Delta W_k - \eta \nabla_{W_k} Err$$
$$W_k = W_k + \Delta W_k$$

- Until Err has converged

Momentum Update

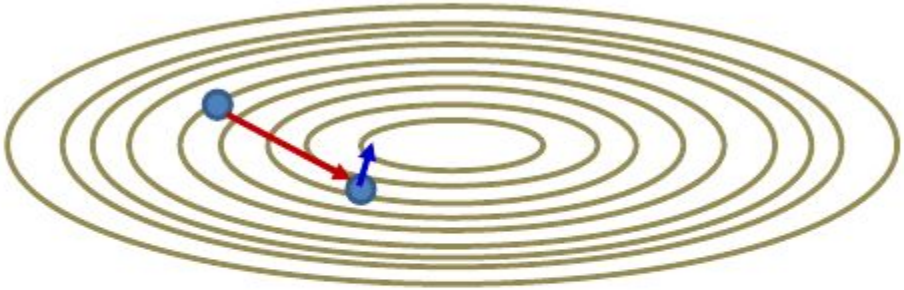


- The momentum method

$$\Delta W^{(k)} = \beta \Delta W^{(k-1)} - \eta \nabla_W Err\left(W^{(k-1)}\right)$$

- At any iteration, to compute the current step:

Momentum Update

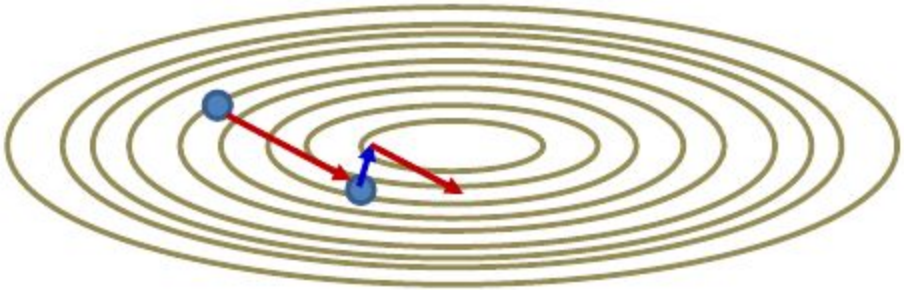


- The momentum method

$$\Delta W^{(k)} = \beta \Delta W^{(k-1)} - \eta \nabla_W Err\left(W^{(k-1)}\right)$$

- At any iteration, to compute the current step:
  - First computes the gradient step at the current location

Momentum Update

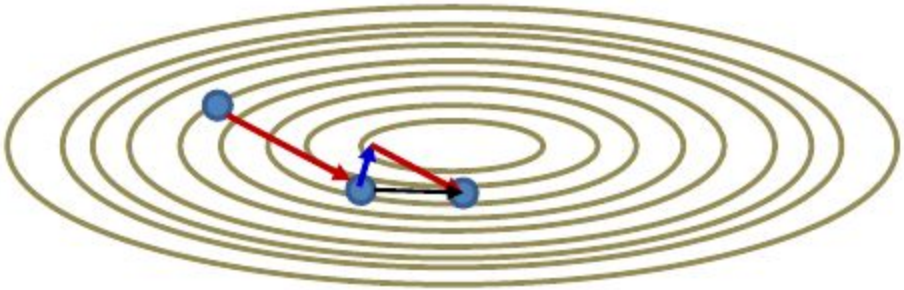


- The momentum method

$$\Delta W^{(k)} = \beta \Delta W^{(k-1)} - \eta \nabla_W Err\left(W^{(k-1)}\right)$$

- At any iteration, to compute the current step:
  - First computes the gradient step at the current location
  - Then adds in the scaled *previous* step
    - Which is actually a running average

Momentum Update



- The momentum method

$$\Delta W^{(k)} = \beta \Delta W^{(k-1)} - \eta \nabla_W Err\left(W^{(k-1)}\right)$$

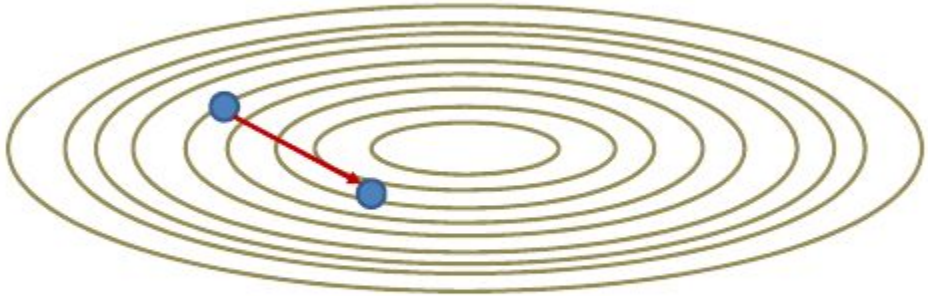
- At any iteration, to compute the current step:
  - First computes the gradient step at the current location
  - Then adds in the scaled *previous* step
    - Which is actually a running average
  - To get the final step

Momentum update

- Takes a step along the past running average *after* walking along the gradient

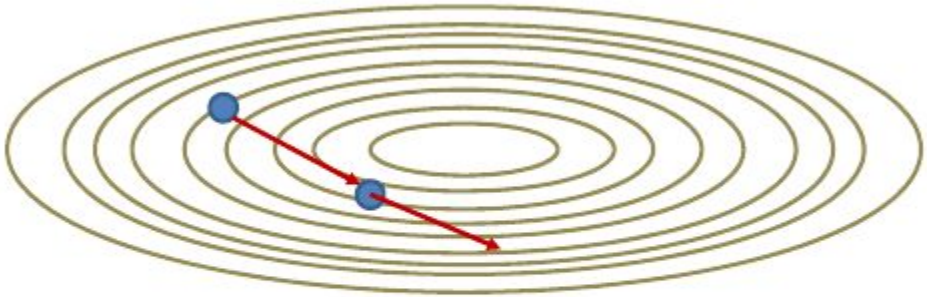
- The procedure can be made more optimal by reversing the order of operations..

Nestorov's Accelerated Gradient



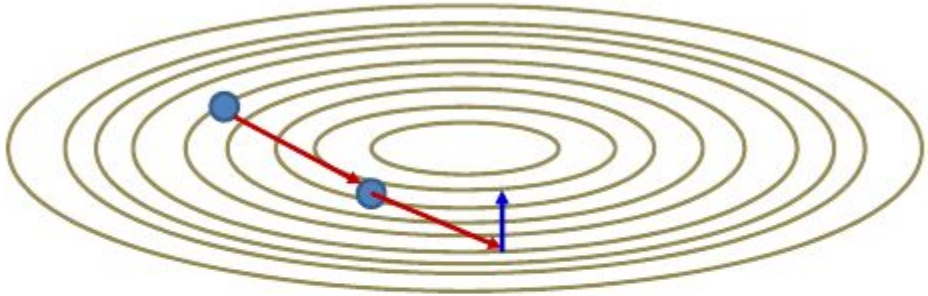
- Change the order of operations
- At any iteration, to compute the current step:

Nestorov's Accelerated Gradient



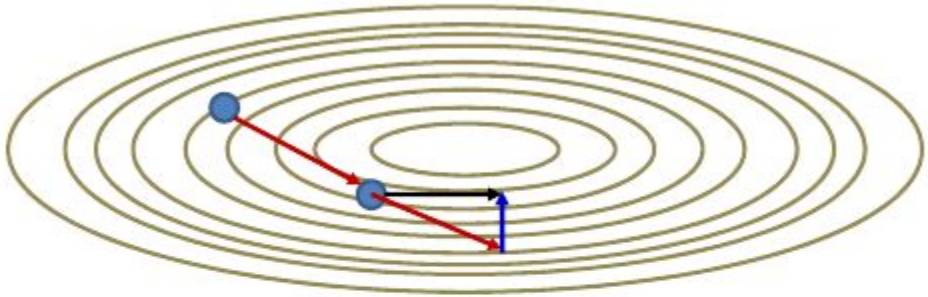
- Change the order of operations
- At any iteration, to compute the current step:
  - First extend the previous step

Nestorov's Accelerated Gradient



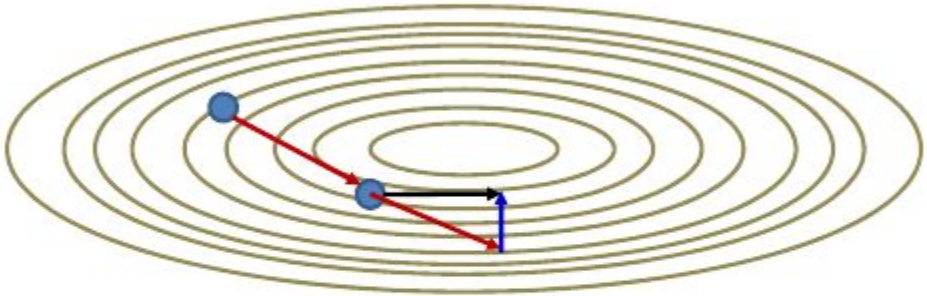
- Change the order of operations
- At any iteration, to compute the current step:
  - First extend the previous step
  - Then compute the gradient step at the resultant position

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

Nestorov's Accelerated Gradient

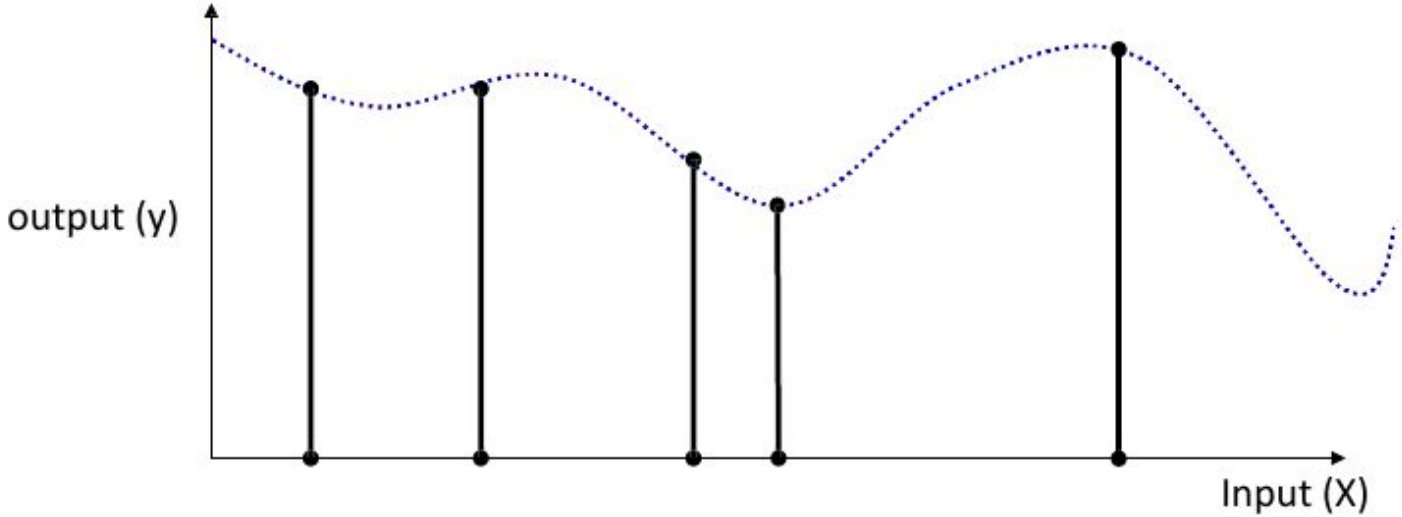


- Nestorov's method

$$\Delta W^{(k)} = \beta \Delta W^{(k-1)} - \eta \nabla_W Err\left(W^{(k)} + \beta \Delta W^{(k-1)}\right)$$

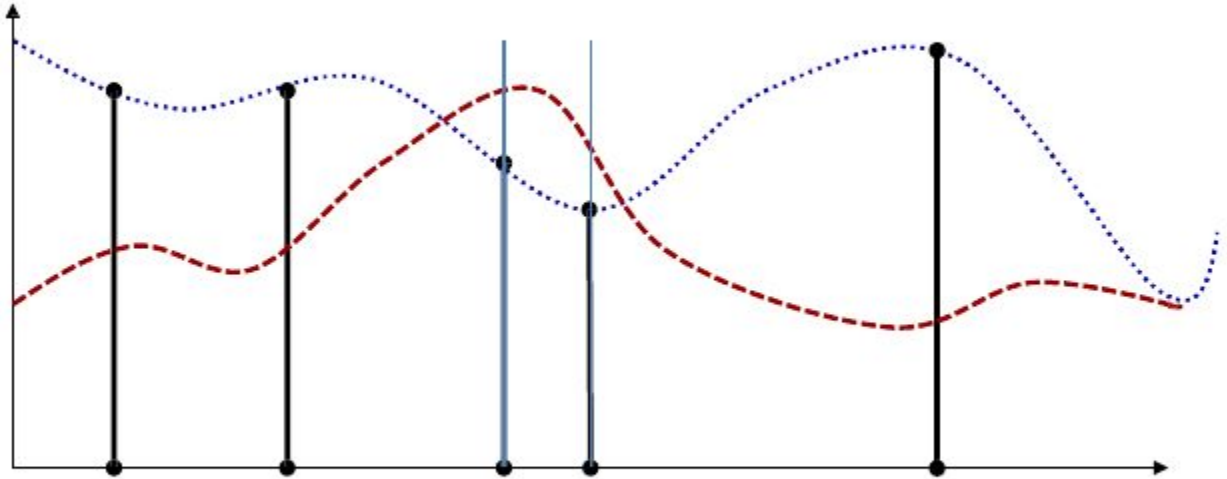
$$W^{(k)} = W^{(k-1)} + \Delta W^{(k)}$$

The training formulation



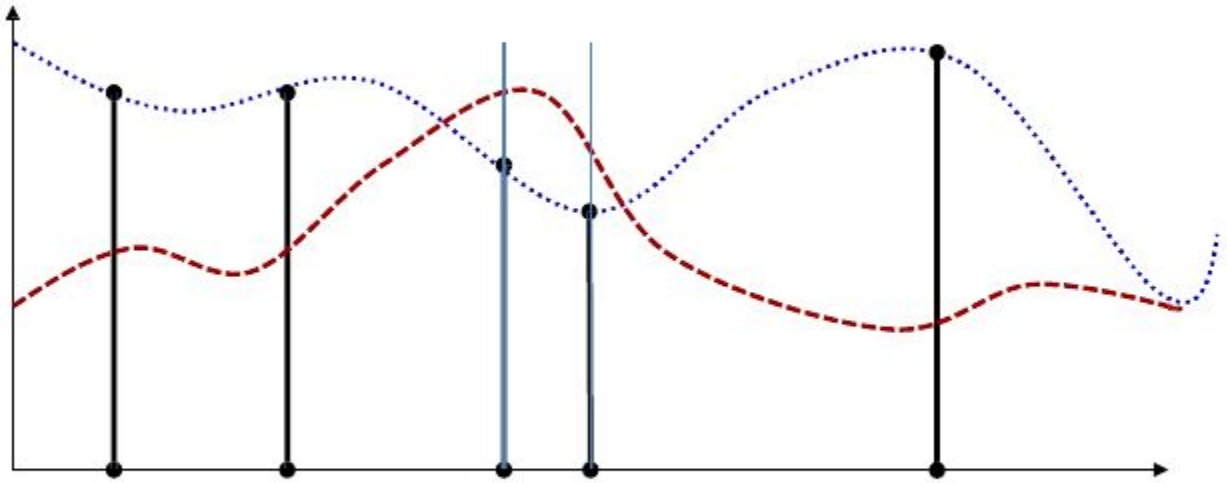
- Given input output pairs at a number of locations, estimate the entire function

Gradient descent



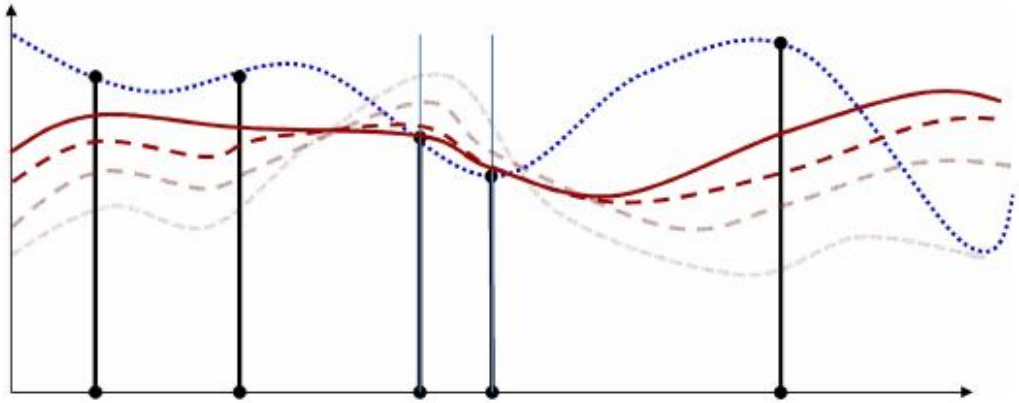
- Start with an initial function

Gradient descent



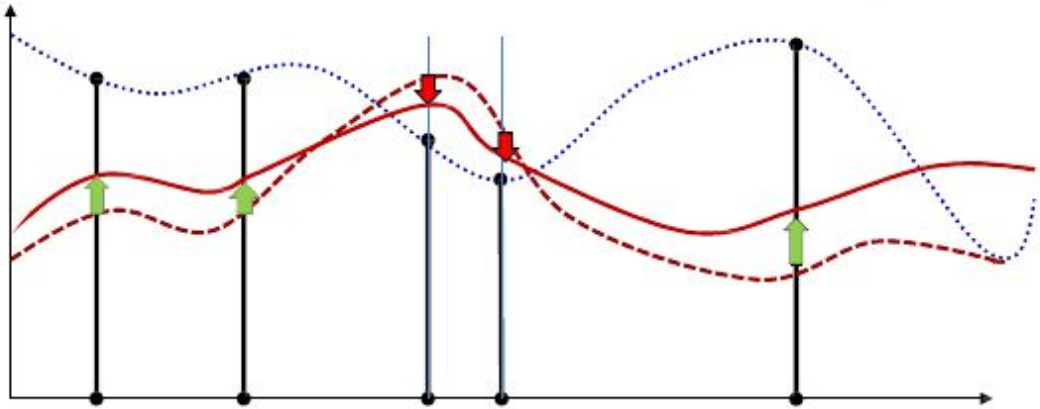
- Start with an initial function

Gradient descent



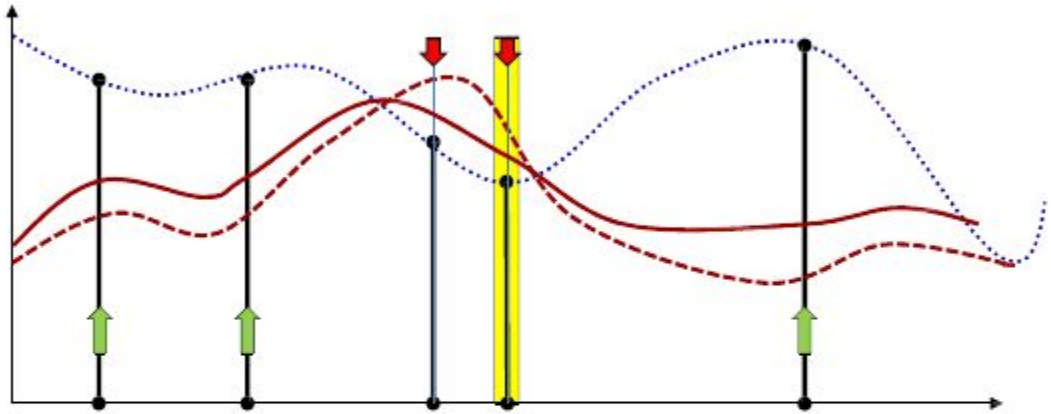
- Start with an initial function
- Adjust its value at *all* points to make the outputs closer to the required value
  - Gradient descent adjusts parameters to adjust the function value at *all* points
  - Repeat this iteratively until we get arbitrarily close to the target function at the training points

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

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

Incremental Update: Stochastic Gradient Descent

- Given $(X_1, d_1), (X_2, d_2), \ldots, (X_T, d_T)$
- Initialize all weights $W_1, W_2, \ldots, W_K$; $j = 0$
- Do:
  - Randomly permute $(X_1, d_1), (X_2, d_2), \ldots, (X_T, d_T)$
  - For all $t = 1:T$
    - $j = j + 1$
    - For every layer k:
      - Compute $\nabla_{W_k} \boldsymbol{Div}(\boldsymbol{Y_t}, \boldsymbol{d_t})$
      - Update

$$W_k = W_k - \eta_j \nabla_{W_k} \boldsymbol{Div}(\boldsymbol{Y_t}, \boldsymbol{d_t})$$

Randomize input order

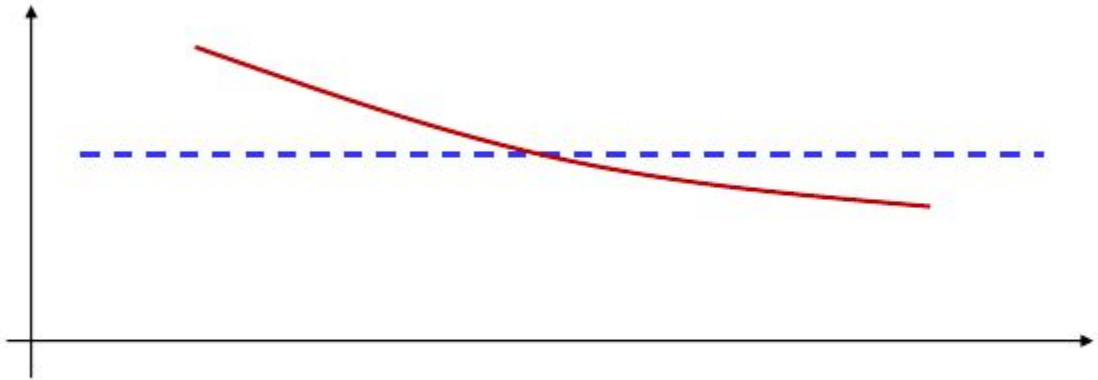
Learning rate reduces with j

- Until Err has converged

Stochastic Gradient Descent

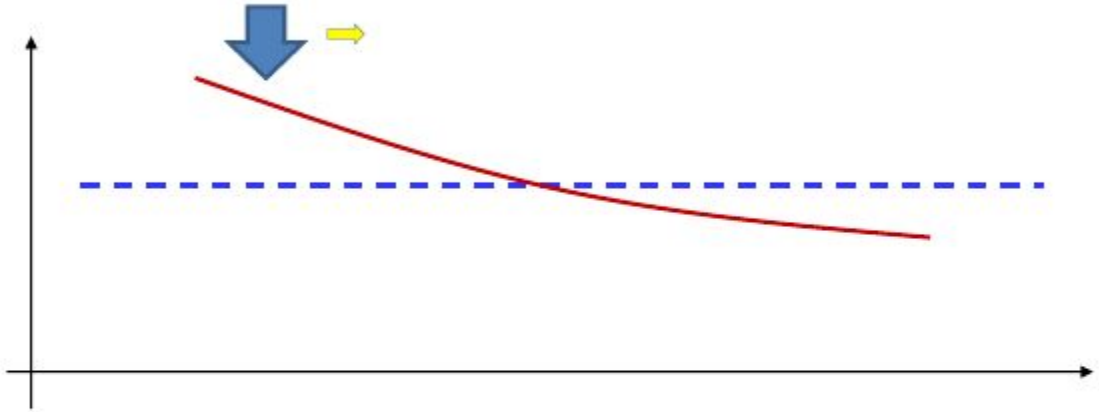
- The iterations can make multiple passes over the data
- A single pass through the entire training data is called an "epoch"
  - An epoch over a training set with T samples results in T updates of parameters

Caveats: order of presentation



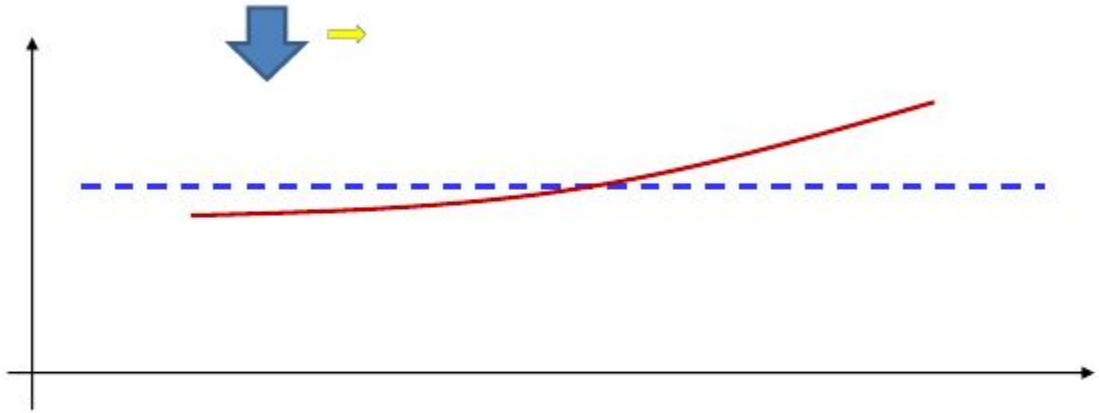
- If we loop through the samples in the same order, we may get *cyclic* behavior

Caveats: order of presentation



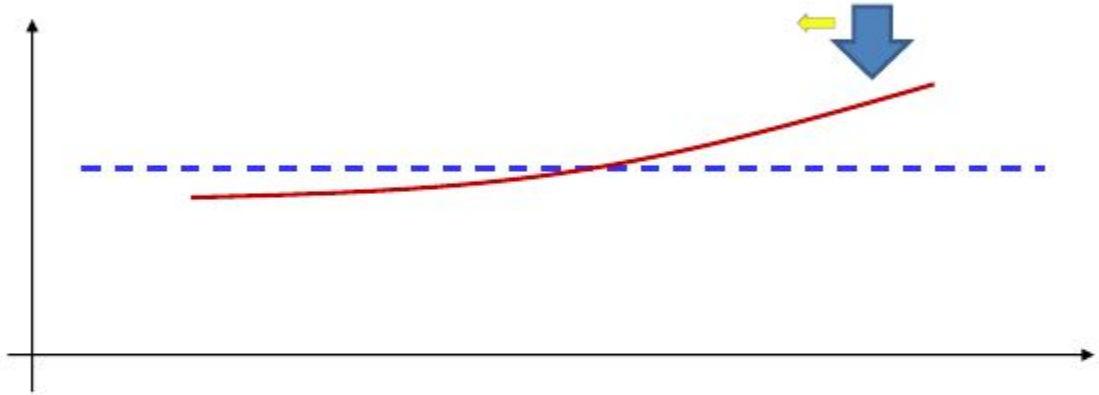
- If we loop through the samples in the same order, we may get *cyclic* behavior
- We must go through them *randomly*

Caveats: order of presentation



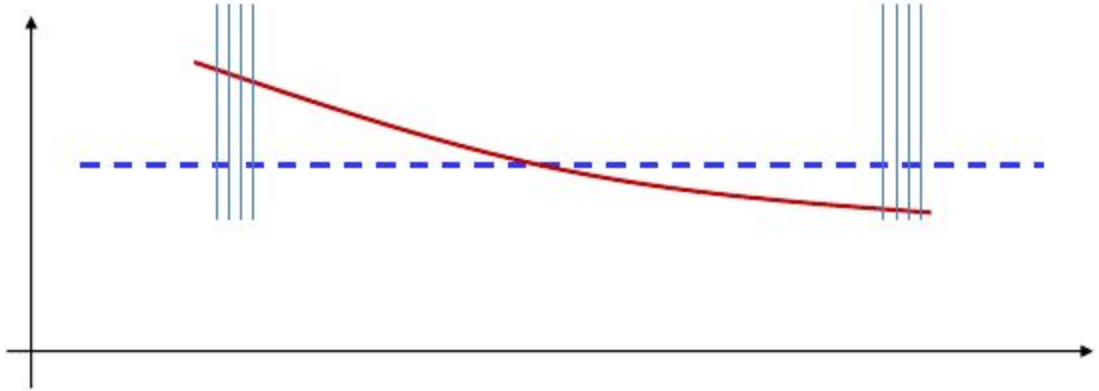
- If we loop through the samples in the same order, we may get *cyclic* behavior

Caveats: order of presentation



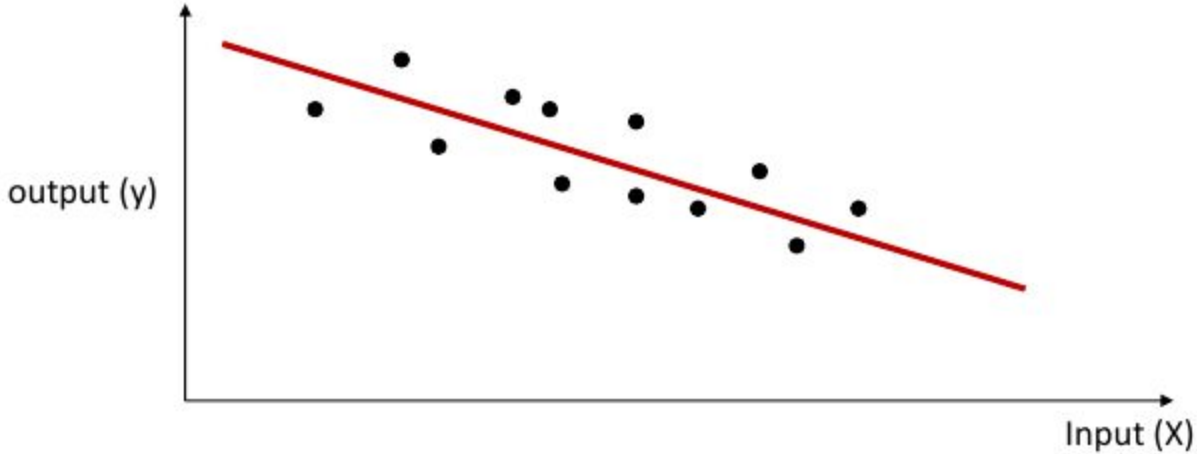
- If we loop through the samples in the same order, we may get *cyclic* behavior

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

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
    - Correction for individual instances with the eventual miniscule learning rates will not modify the function

When does SGD work

- 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 optimally determined
- If the loss is convex, SGD converges to the optimal solution
- For non-convex losses SGD converges to a local minimum

Batch gradient convergence

- In contrast, using the batch update method, for *strongly convex* functions,

$$\left|W^{(k)} - W^*\right| < c^k \left|W^{(0)} - W^*\right|$$

  - Giving us the iterations to ϵ convergence as $O\left(log\left(\frac{1}{\epsilon}\right)\right)$

- For generic convex functions, the ϵ convergence is $O\left(\frac{1}{\epsilon}\right)$

- Batch gradients converge "faster"
  - But SGD performs T updates for every batch update

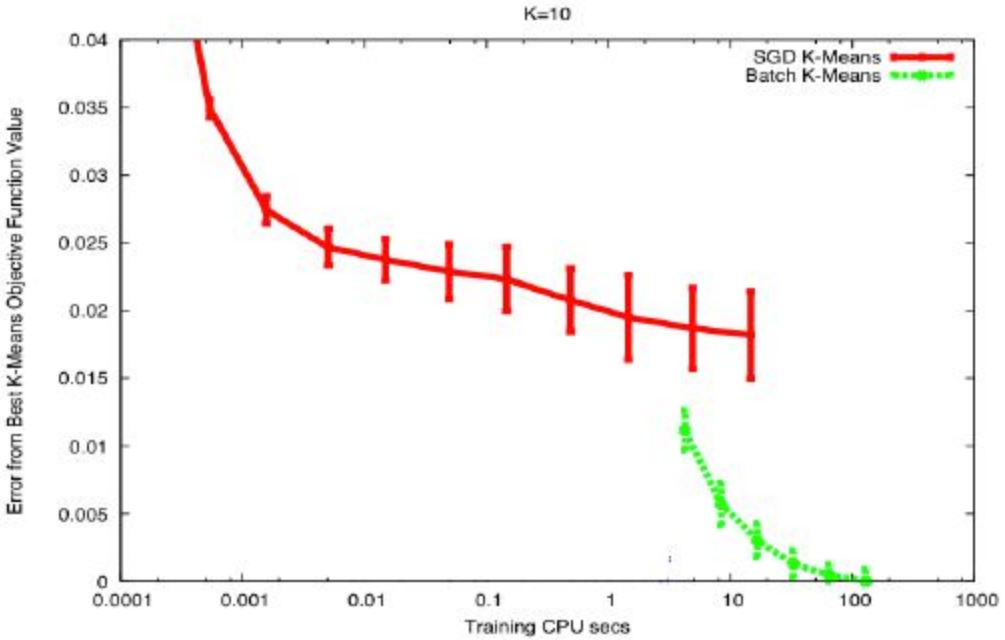
SGD Convergence

- We can bound the expected difference between the loss over our data using the optimal weights, w^*, and the weights at any single iteration, w_T, to $\mathcal{O}\left(\frac{\log(T)}{T}\right)$ for strongly convex loss or $\mathcal{O}\left(\frac{\log(T)}{\sqrt{T}}\right)$ for convex loss

- Averaging schemes can improve the bound to $\mathcal{O}\left(\frac{1}{T}\right)$ and $\mathcal{O}\left(\frac{1}{\sqrt{T}}\right)$

- Smoothness of the loss is not required

SGD example



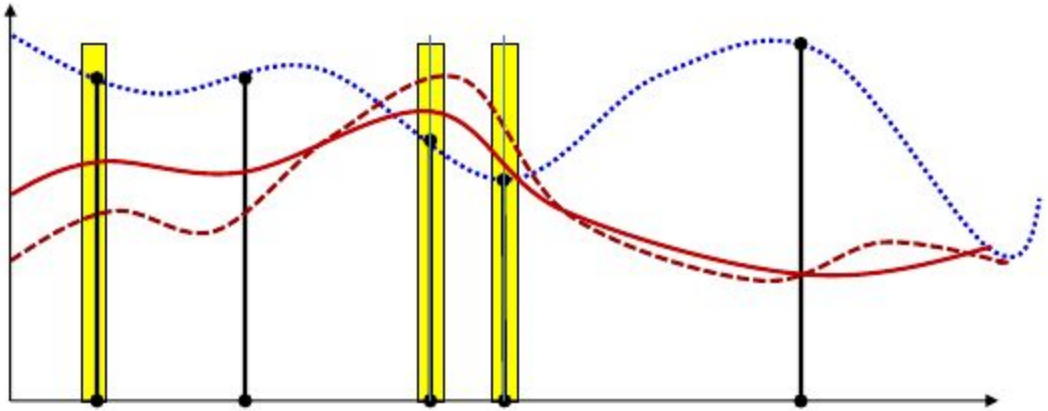
- A simpler problem: K-means
- Note: SGD converges slower
- Also note the rather large variation between runs
  - Lets try to understand these results..

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?

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

Incremental Update: Mini-batch update

- Given (X_1, d_1), (X_2, d_2),..., (X_T, d_T)
- Initialize all weights $W_1, W_2, \ldots, W_K$; $j = 0$
- Do:
  - Randomly permute (X_1, d_1), (X_2, d_2),..., (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 $\nabla_{W_k} Div(Y_t, d_t)$
        - $\Delta W_k = \Delta W_k + \nabla_{W_k} Div(Y_t, d_t)$
    - Update
      - For every layer k:

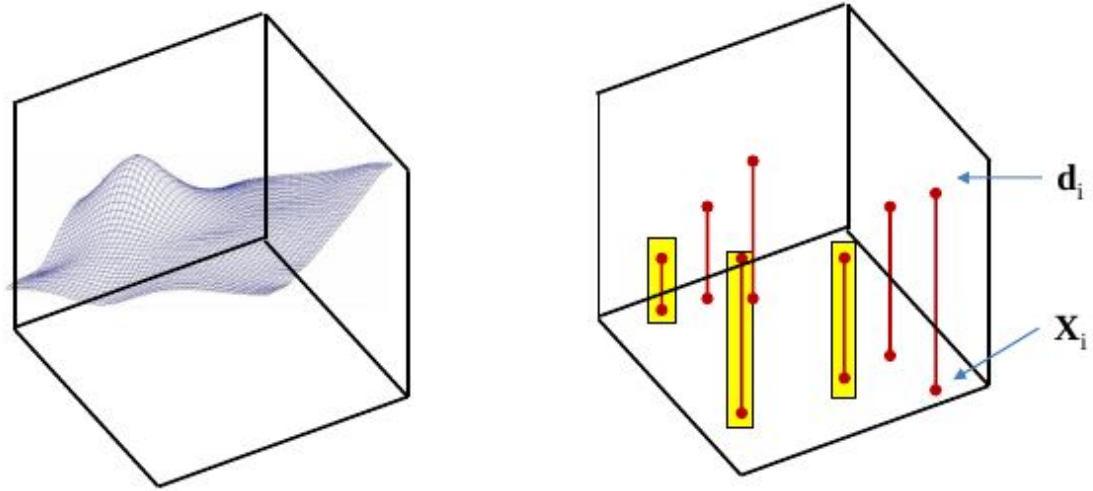
$$W_k = W_k - \eta_j \Delta W_k$$

Mini-batch size

Shrinking step size

- Until Err has converged

Mini Batches



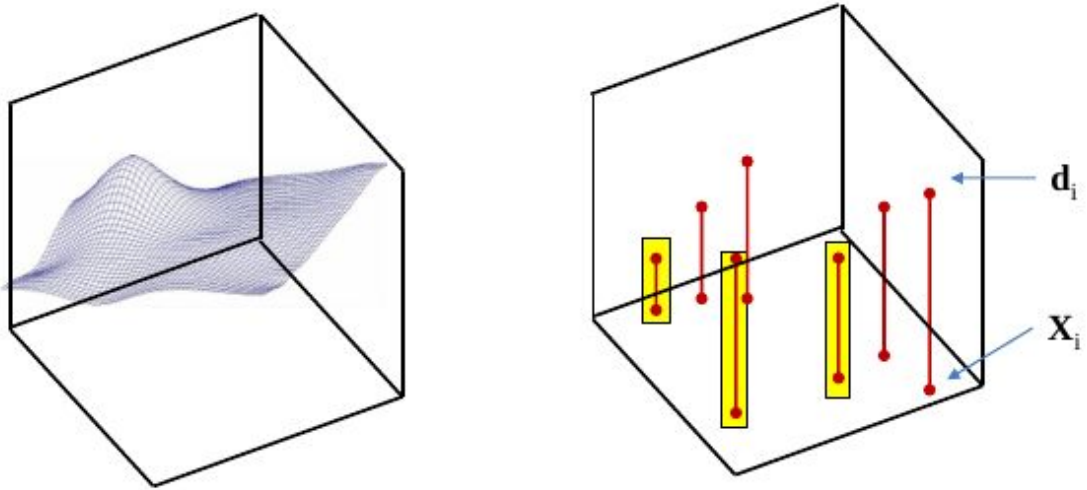
- Mini-batch updates compute and minimize a *batch error*

$$BatchErr\big(f(X;W),g(X)\big) = \frac{1}{b}\sum_{i=1}^{b} div(f(X_i;W),d_i)$$

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

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

Mini Batches



- Mini-batch updates computes an *empirical batch error*

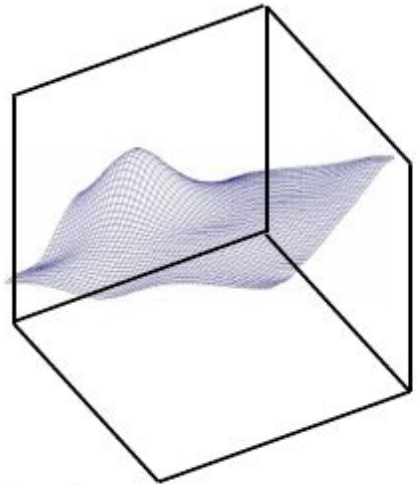
$$BatchErr\big(f(X;W),g(X)\big) = \frac{1}{b}\sum^{b} div\big(f(X_i;W),d_i\big)$$

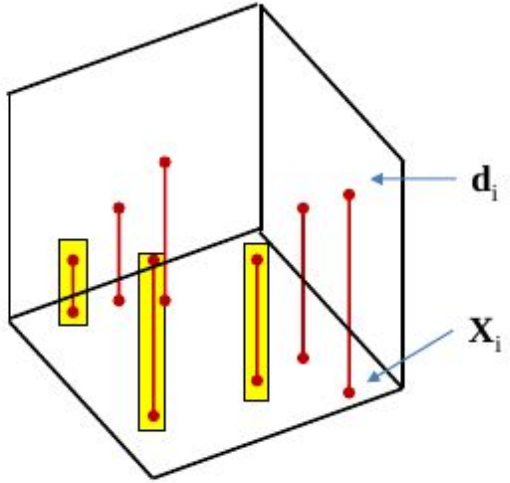
The batch error is also an unbiased estimate of the expected error

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

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

Mini Batches





- Mini-batch updates computes an *empirical batch error*

The variance of the batch error: var(Err) = 1/b var(div)
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

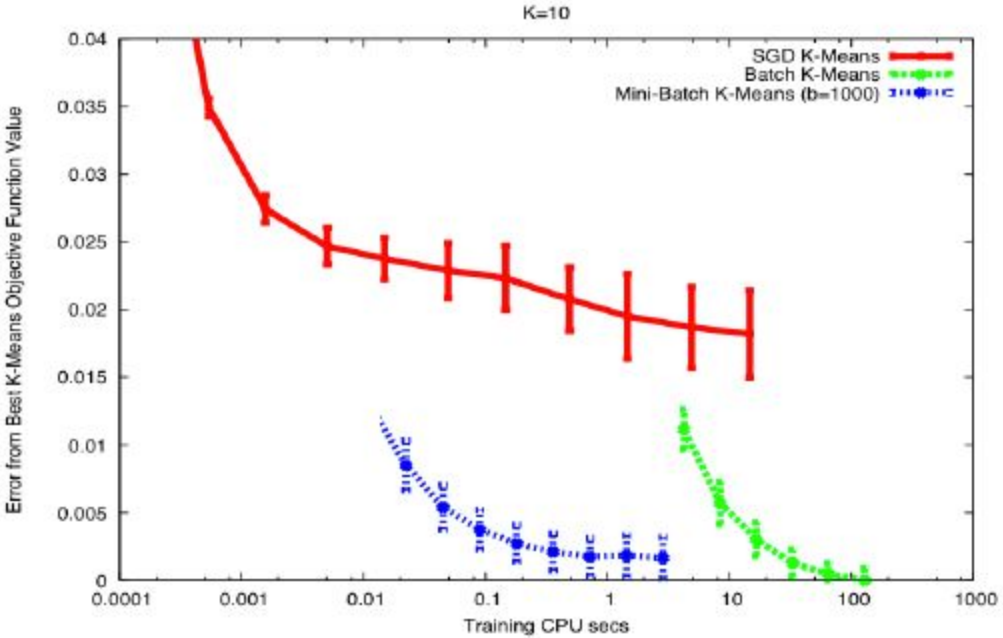
- The *expected value* of the *batch error* is also the *expected divergence*

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

Minibatch convergence

- For convex functions, convergence rate for SGD is $\mathcal{O}\left(\frac{1}{\sqrt{k}}\right)$.
- For *mini-batch* updates with batches of size b, the convergence rate is $\mathcal{O}\left(\frac{1}{\sqrt{bk}}+\frac{1}{k}\right)$
  - Apparently an improvement of $\sqrt{b}$ over SGD
  - But since the batch size is b, we perform b times as many computations per iteration as SGD
  - We actually get a *degradation* of $\sqrt{b}$
- However, in practice
  - The objectives are generally not convex; mini-batches are more effective with the right learning rates
  - We also get additional benefits of vector processing

SGD example



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

Training and minibatches

- In practice, training is usually performed using mini-batches
  - The mini-batch size is a hyper parameter to be optimized
- 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

More recent methods

- Several newer methods have been proposed that follow the general pattern of enhancing long-term trends to smooth out the variations of the mini-batch gradient
  - RMS Prop
  - ADAM: very popular in practice
  - Adagrad
  - AdaDelta
  - …
- All roughly equivalent in performance

Variance-normalized step

- In recent past
  - Total movement in *Y* component of updates is high
  - Movement in *X* components is lower
- Current update, modify usual gradient-based update:
  - Scale *down* Y component
  - Scale *up* X component
- A variety of algorithms have been proposed on this premise

RMS Prop

- Notation:
  - Updates are ***by parameter***
  - Sum derivative of divergence w.r.t any individual parameter w is shown as $\partial_w D$
  - The ***squared*** derivative is $\partial_w^2 D = (\partial_w D)^2$
  - 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 updates with large mean squared derivatives
  - scale up updates with small mean squared derivatives

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 update of the parameter by the *inverse* of the *root mean squared* derivative

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

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

RMS Prop (updates are for each weight of each layer)

- Do:
  - Randomly shuffle inputs to change their order
  - Initialize: $k = 1$; for all weights w in all layers, $E[\partial_w^2 D]_k = 0$
  - For all $t = 1:B:T$ (incrementing in blocks of B inputs)
    - For all weights in all layers initialize $(\partial_w D)_k = 0$
    - For $b = 0:B-1$
      - Compute
        - » Output $\boldsymbol{Y}(\boldsymbol{X}_{t+b})$
        - » Compute gradient $\frac{\boldsymbol{dDiv}(\boldsymbol{Y}(\boldsymbol{X}_{t+b}),\boldsymbol{d}_{t+b})}{\boldsymbol{dw}}$
        - » Compute $(\partial_w D)_k \mathrel{+}= \frac{\boldsymbol{dDiv}(\boldsymbol{Y}(\boldsymbol{X}_{t+b}),\boldsymbol{d}_{t+b})}{\boldsymbol{dw}}$
    - update:

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

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

    - $k = k + 1$
- Until $E(\boldsymbol{W}^{(1)}, \boldsymbol{W}^{(2)}, \ldots, \boldsymbol{W}^{(K)})$ has converged

ADAM: RMSprop with momentum

- RMS prop only considers a second-moment normalized version of the current gradient
- ADAM utilizes a smoothed version of the *momentum-augmented* gradient
  - Considers both first and second moments

- **Procedure:**
  - Maintain a running estimate of the mean derivative for each parameter
  - Maintain a running estimate of the mean squared value of derivatives for each parameter
  - Scale update of the parameter by the *inverse* of the *root mean squared* derivative

$$m_k = \delta m_{k-1} + (1-\delta)(\partial_w D)_k$$

$$v_k = \gamma v_{k-1} + (1-\gamma)(\partial_w^2 D)_k$$

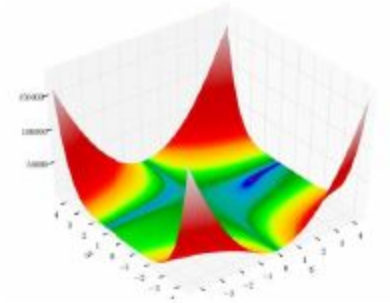
$$\hat{m}_k = \frac{m_k}{1-\delta^k}, \qquad \hat{v}_k = \frac{v_k}{1-\gamma^k}$$

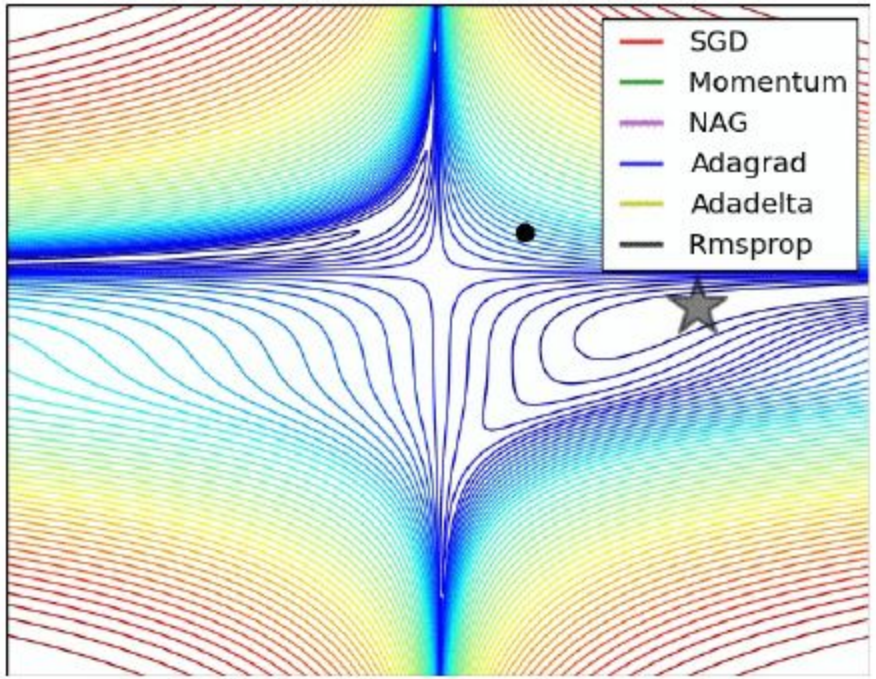
$$w_{k+1} = w_k - \frac{\eta}{\sqrt{\hat{v}_k + \epsilon}}\hat{m}_k$$

Other variants of the same theme

- Many:
  - Adagrad
  - AdaDelta
  - ADAM
  - AdaMax
  - …
- Generally no explicit learning rate to optimize
  - But come with other hyper parameters to be optimized
  - Typical params:
    - RMSProp: $\eta = 0.001$ $\gamma = 0.9$
    - ADAM: $\eta = 0.001$ $\delta = 0.9$ $\gamma = 0.999$

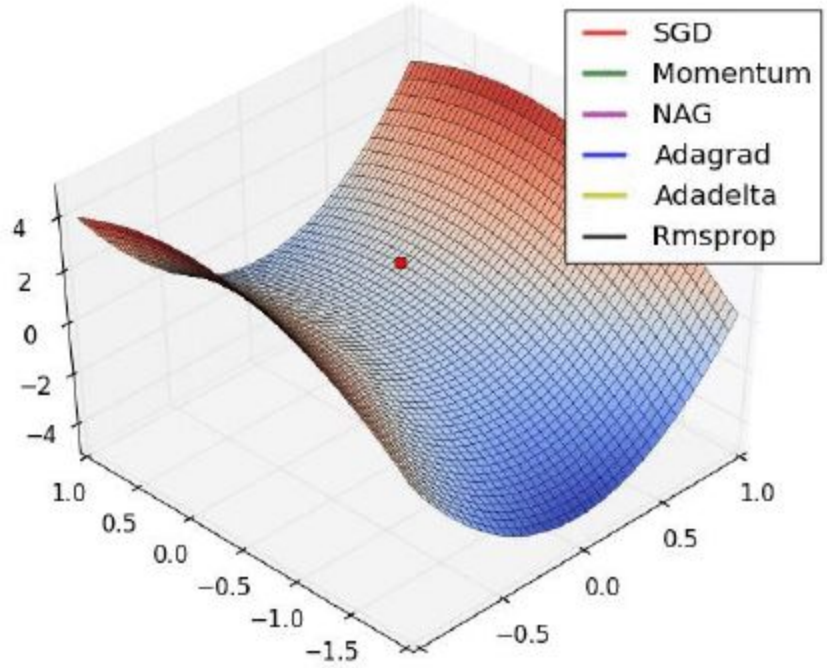
Visualizing the optimizers: Beale's Function





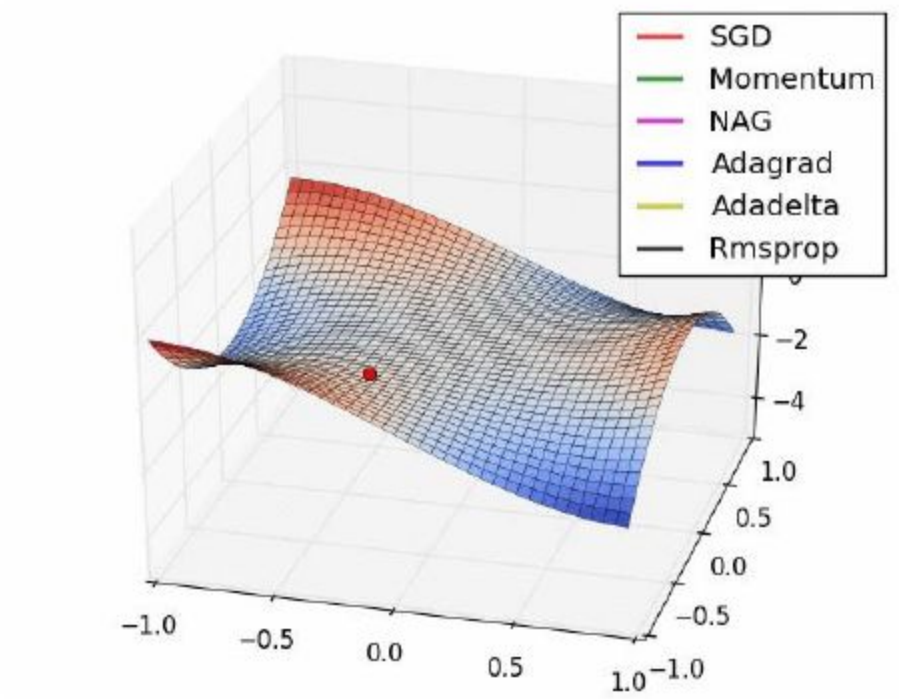
- http://www.denizyuret.com/2015/03/alec-radfords-animations-for.html

Visualizing the optimizers: Long Valley



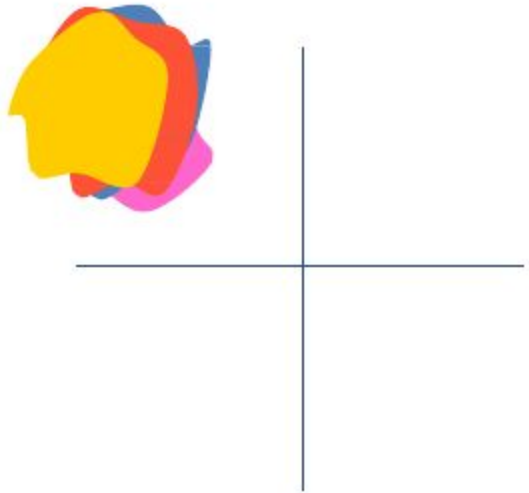
- http://www.denizyuret.com/2015/03/alec-radfords-animations-for.html

Visualizing the optimizers: Saddle Point



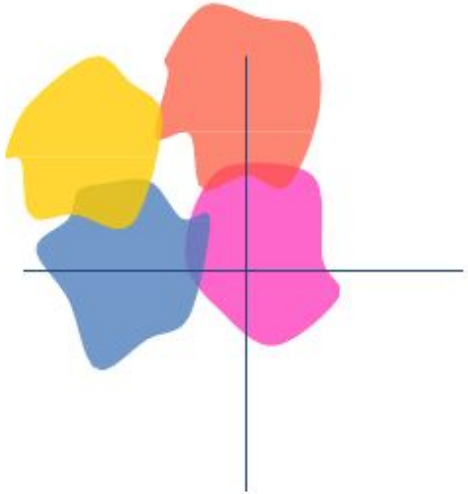
- http://www.denizyuret.com/2015/03/alec-radfords-animations-for.html

The problem of covariate shifts



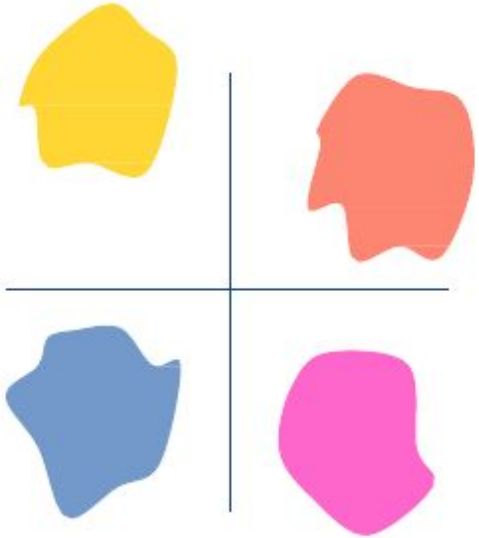
- Training assumes the training data are all similarly distributed
  - Minibatches have similar distribution

The problem of covariate shifts



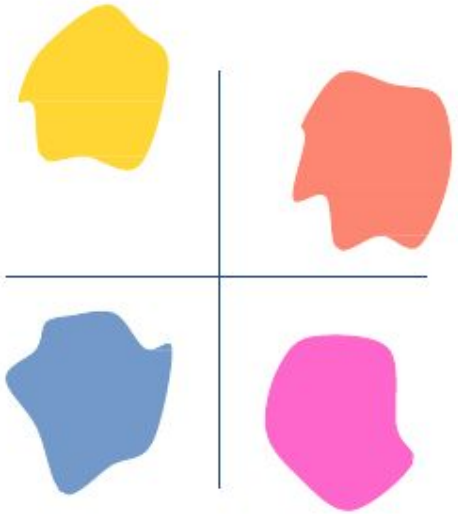
- Training assumes the training data are all similarly distributed
  - Minibatches have similar distribution
- In practice, each minibatch may have a different distribution
  - A “covariate shift”
  - Which may occur in *each* layer of the network

The problem of covariate shifts



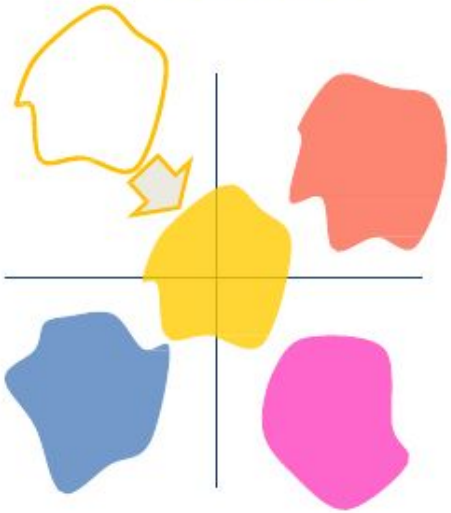
- Training assumes the training data are all similarly distributed
  - Minibatches have similar distribution
- In practice, each minibatch may have a different distribution
  - A “covariate shift”
- Covariate shifts can be large!
  - All covariate shifts can affect training badly

Solution: Move all subgroups to a "standard" location



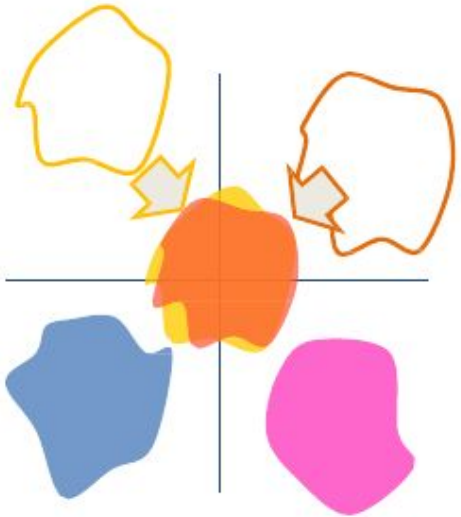
- "Move" all batches to have a mean of 0 and unit standard deviation
  - Eliminates covariate shift between batches

Solution: Move all subgroups to a “standard” location



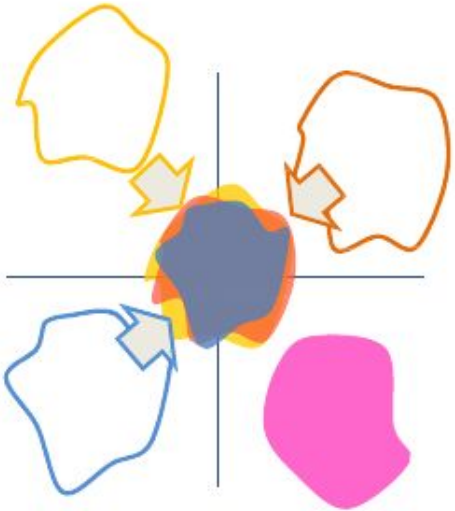
- “Move” all batches to have a mean of 0 and unit standard deviation
  - Eliminates covariate shift between batches

Solution: Move all subgroups to a "standard" location



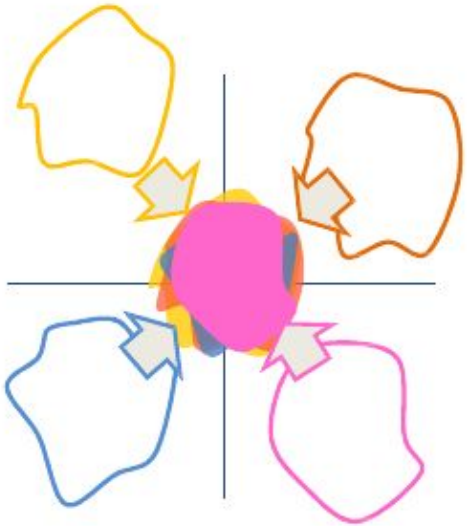
- "Move" all batches to have a mean of 0 and unit standard deviation
  - Eliminates covariate shift between batches

Solution: Move all subgroups to a "standard" location



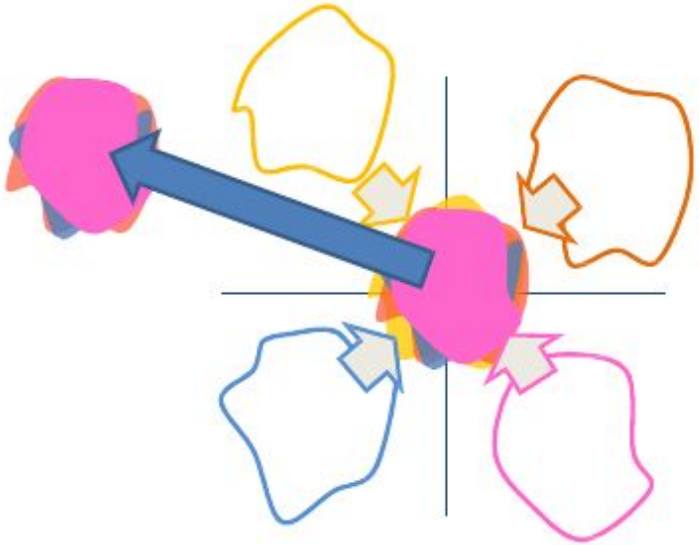
- "Move" all batches to have a mean of 0 and unit standard deviation
  - Eliminates covariate shift between batches

Solution: Move all subgroups to a “standard” location



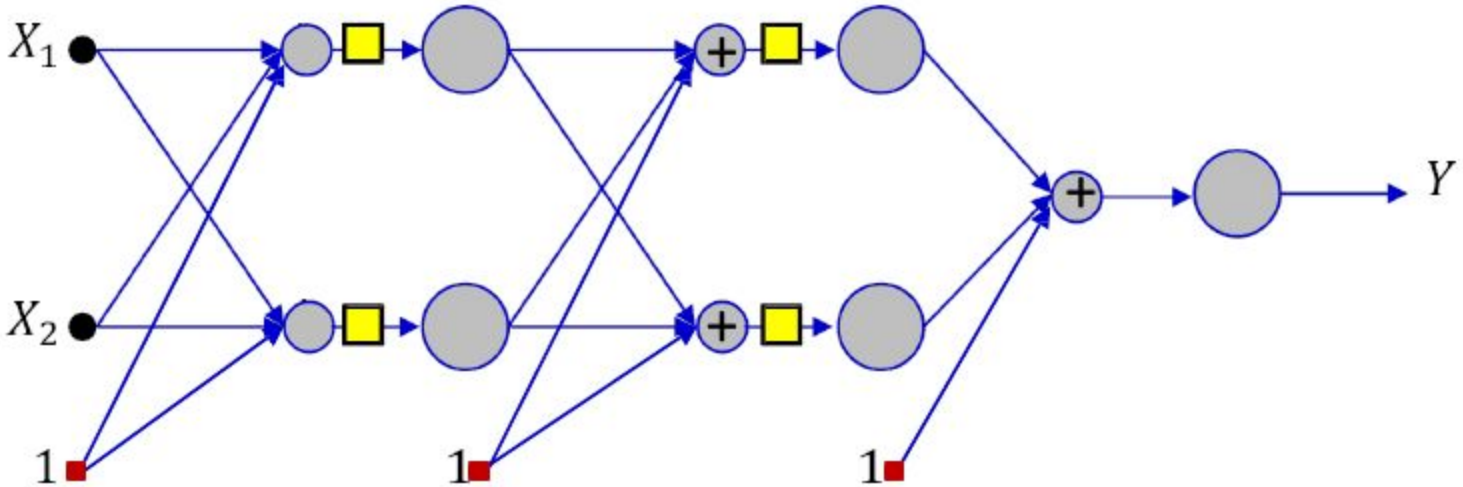
- “Move” all batches to have a mean of 0 and unit standard deviation
  - Eliminates covariate shift between batches

Solution: Move all subgroups to a "standard" location



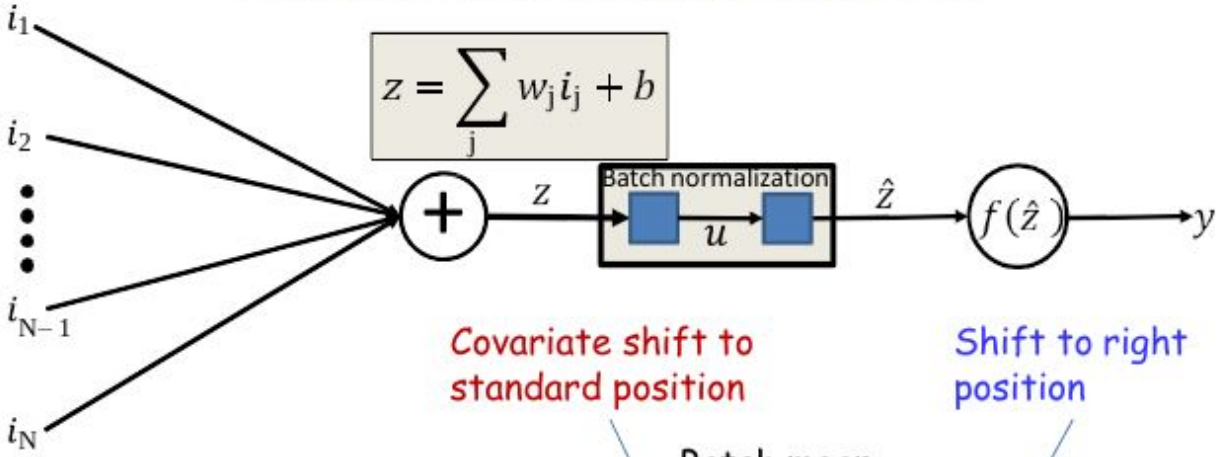
- "Move" all batches to have a mean of 0 and unit standard deviation
  - Eliminates covariate shift between batches
  - Then move the entire collection to the appropriate location

Batch normalization



- Batch normalization is a covariate adjustment unit that happens after the weighted addition of inputs but before the application of activation
  - Is done independently for each unit, to simplify computation
- **Training: The adjustment occurs over individual minibatches**

Batch normalization



Covariate shift to standard position

Shift to right position

Batch mean

$$u_i = \frac{z_i - \mu_B}{\sigma_B}$$

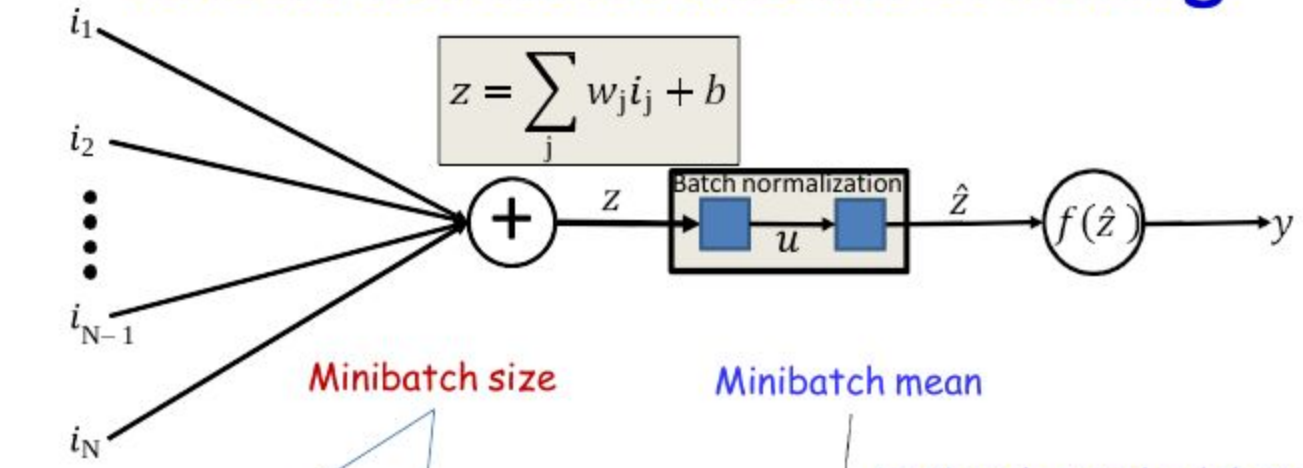
$$\hat{z}_i = \gamma u_i + \beta$$

Batch standard deviation

Neuron-specific terms

- BN aggregates the statistics over a minibatch and normalizes the batch by them
- Normalized instances are "shifted" to a *unit-specific* location

Batch normalization: Training



Minibatch size

Minibatch mean

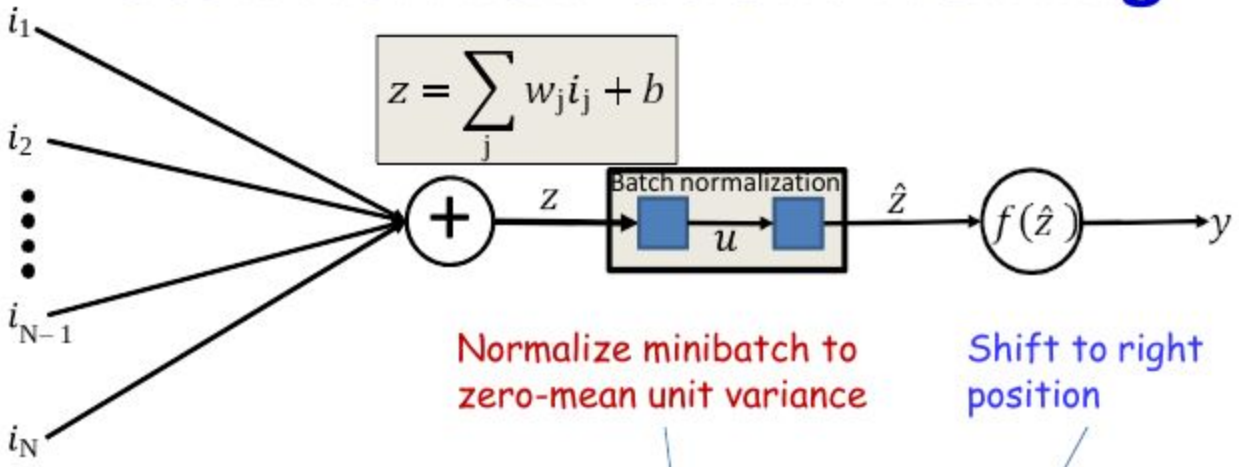
Minibatch standard deviation

$$\mu_B = \frac{1}{B}\sum_{i=1}^{B} z_i \qquad \sigma_B^2 = \frac{1}{B}\sum_{i=1}^{B}(z_i - \mu_B)^2$$

$$u = \frac{z - \mu}{\sqrt{\sigma^2 + \epsilon}} \qquad \hat{z} = \gamma u_i + \beta$$

- BN aggregates the statistics over a minibatch and normalizes the batch by them
- Normalized instances are "shifted" to a *unit-specific* location

Batch normalization: Training

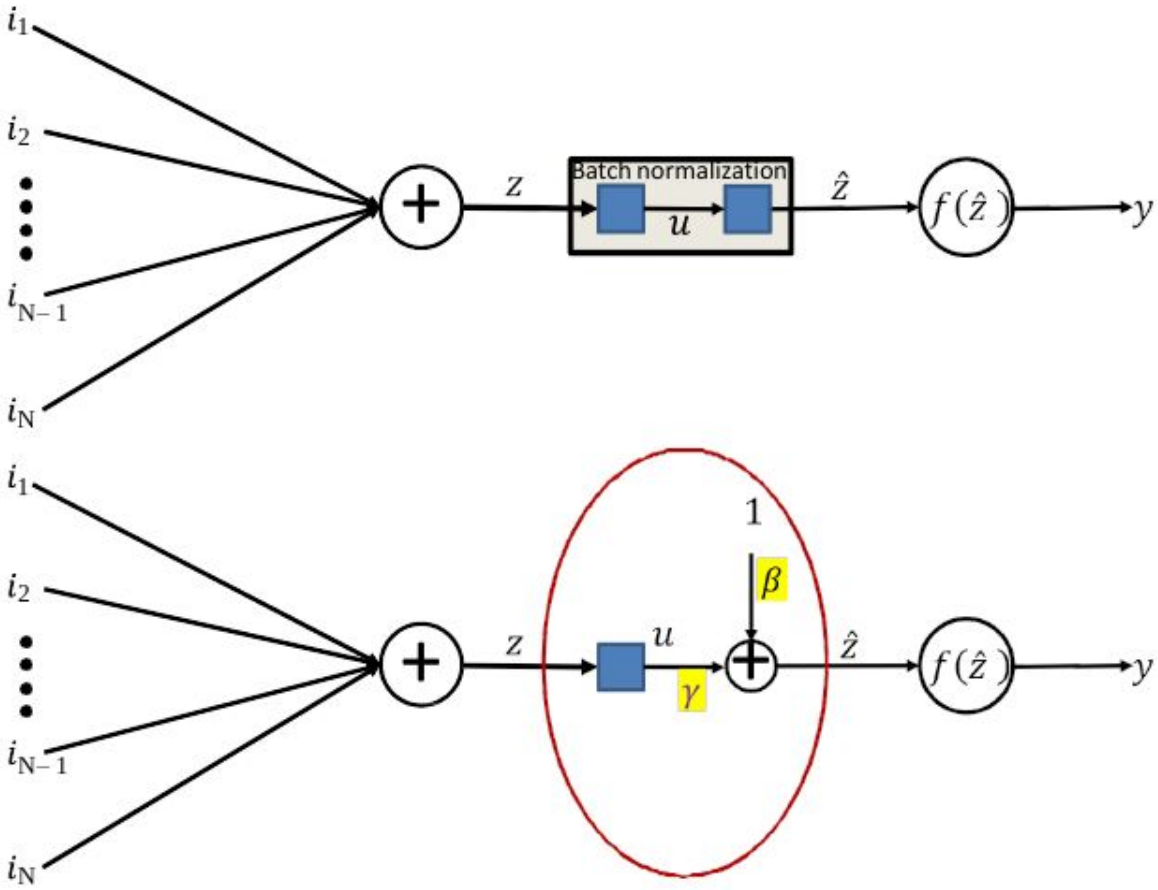


$$\mu_B = \frac{1}{B}\sum_{i=1}^{B} z_i \qquad \sigma_B^2 = \frac{1}{B}\sum_{i=1}^{B}(z_i - \mu_B)^2$$

$$u_i = \frac{z_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}} \qquad \hat{z}_i = \gamma u_i + \beta$$

- BN aggregates the statistics over a minibatch and normalizes the batch by them
- Normalized instances are "shifted" to a *unit-specific* location

A better picture for batch norm



A note on derivatives

- In conventional learning, we attempt to compute the derivative of the divergence for *individual* training instances w.r.t. parameters

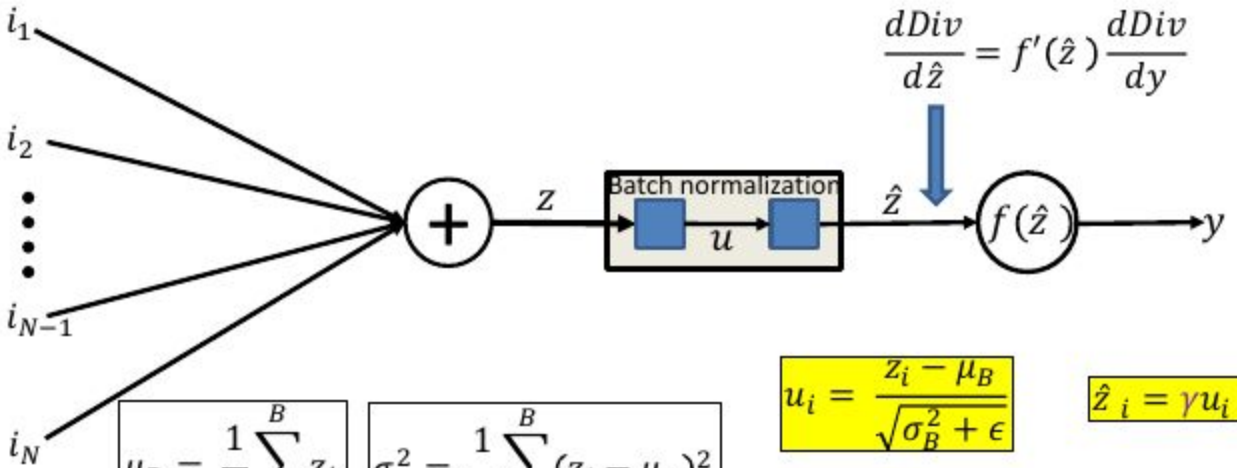
- This is based on the following relations

$$Div(minibatch) = \frac{1}{B}\sum_{t} Div(Y_t(X_t), d_t(X_t))$$

$$\frac{dDiv(minibatch)}{dw_{i,j}^{(k)}} = \frac{1}{B}\sum_{t} \frac{dDiv(Y_t(X_t), d_t(X_t))}{dw_{i,j}^{(k)}}$$

- If we use Batch Norm, the above relation gets a little complicated

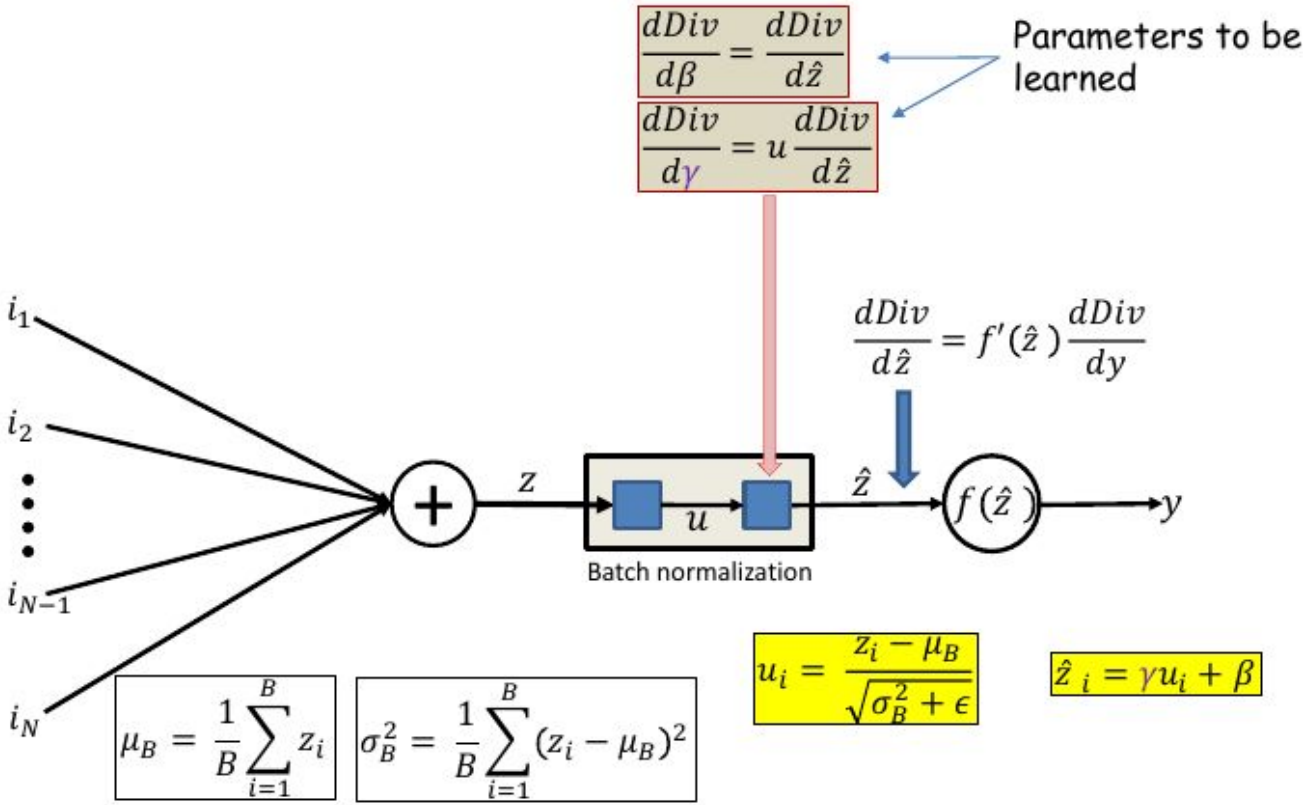
Backpropagation



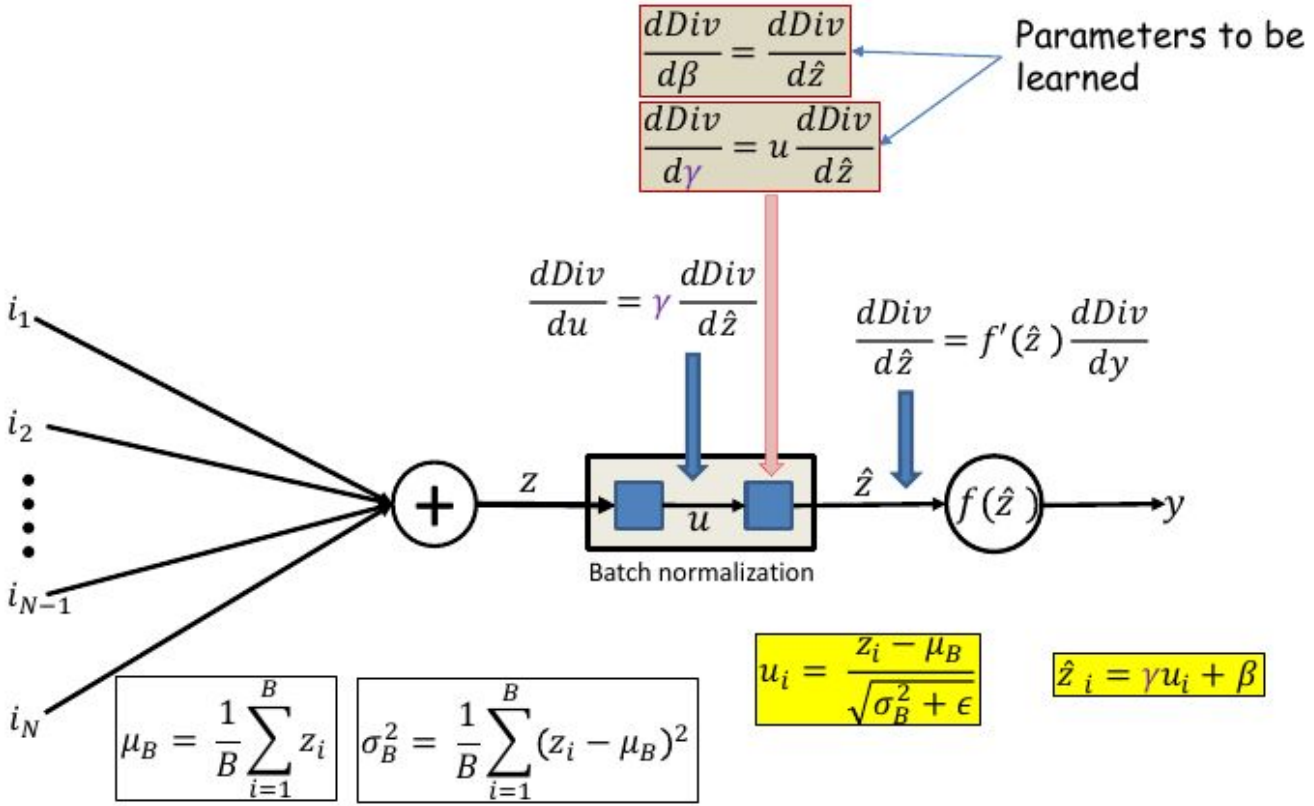
$$\mu_B = \frac{1}{B}\sum_{i=1}^{B} z_i \qquad \sigma_B^2 = \frac{1}{B}\sum_{i=1}^{B}(z_i - \mu_B)^2$$

$$u_i = \frac{z_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}} \qquad \hat{z}_i = \gamma u_i + \beta$$

Batch normalization: Backpropagation

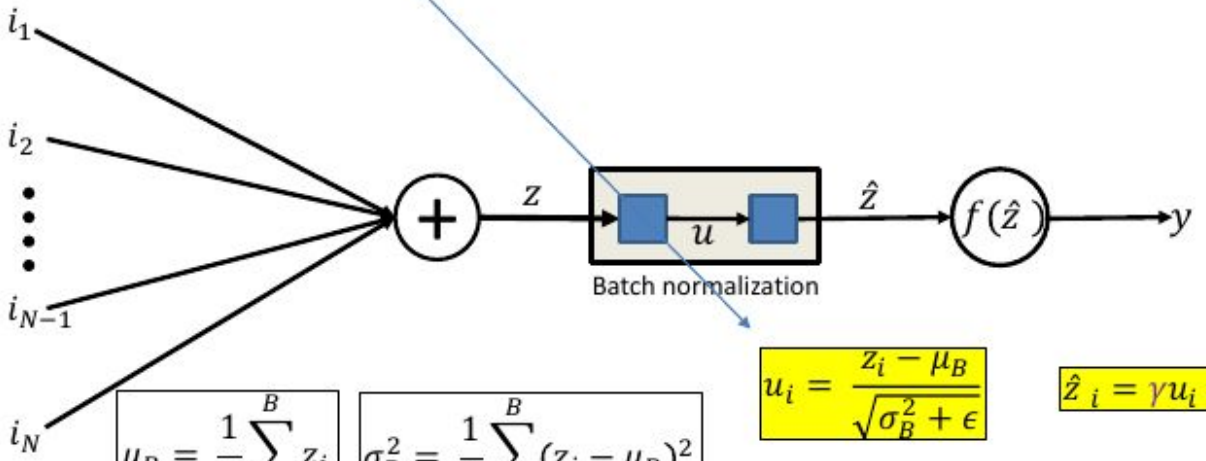


Batch normalization: Backpropagation



Batch normalization: Backpropagation

$$\frac{\partial Div}{\partial \sigma_B^2} = \sum_{i=1}^{B} \frac{\partial Div}{\partial u_i}(z_i - \mu_B) \cdot \frac{-1}{2}(\sigma_B^2 + \epsilon)^{-3/2}$$



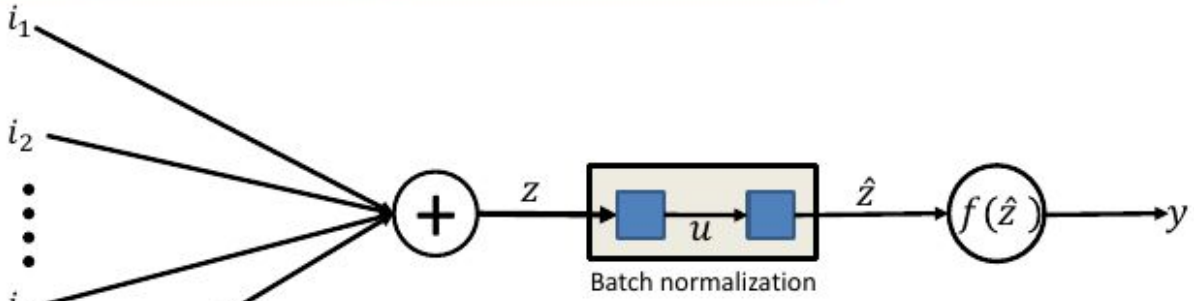
$$\mu_B = \frac{1}{B}\sum_{i=1}^{B} z_i \qquad \sigma_B^2 = \frac{1}{B}\sum_{i=1}^{B}(z_i - \mu_B)^2$$

$$u_i = \frac{z_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}} \qquad \hat{z}_i = \gamma u_i + \beta$$

Batch normalization: Backpropagation

$$\frac{\partial Div}{\partial \sigma_B^2} = \sum_{i=1}^{B} \frac{\partial Div}{\partial u_i}(z_i - \mu_B) \cdot \frac{-1}{2}(\sigma_B^2 + \epsilon)^{-3/2}$$

$$\frac{\partial Div}{\partial \mu_B} = \left(\sum_{i=1}^{B} \frac{\partial Div}{\partial u_i} \cdot \frac{-1}{\sqrt{\sigma_B^2 + \epsilon}} \right) + \frac{\partial Div}{\partial \sigma_B^2} \cdot \frac{\sum_{i=1}^{B} -2(z_i - \mu_B)}{B}$$



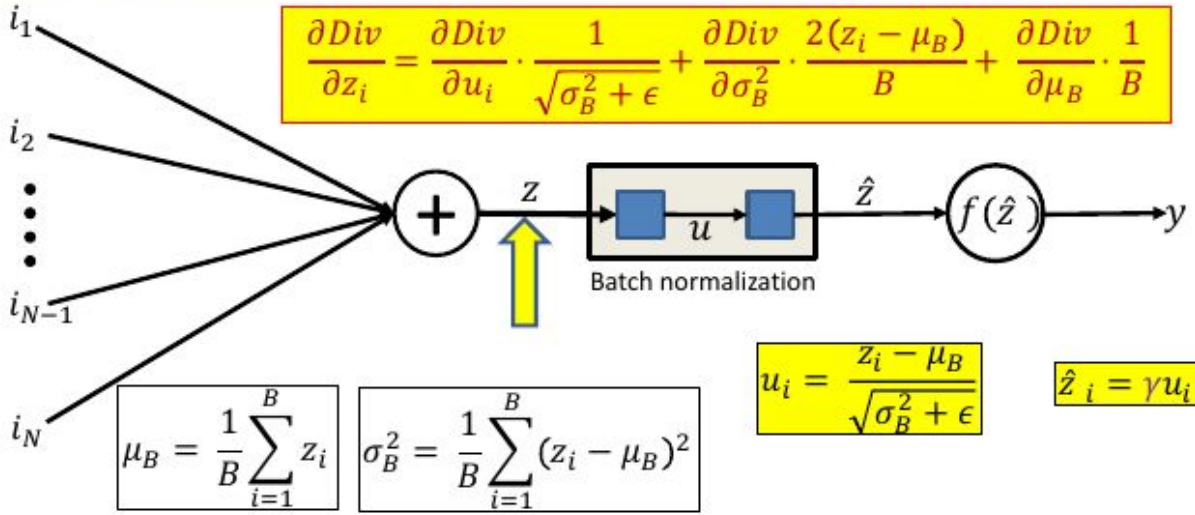
$$\mu_B = \frac{1}{B}\sum_{i=1}^{B} z_i \qquad \sigma_B^2 = \frac{1}{B}\sum_{i=1}^{B}(z_i - \mu_B)^2$$

$$u_i = \frac{z_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}} \qquad \hat{z}_i = \gamma u_i + \beta$$

Batch normalization: Backpropagation

$$\frac{\partial Div}{\partial \sigma_B^2} = \sum_{i=1}^{B} \frac{\partial Div}{\partial u_i}(z_i - \mu_B) \cdot \frac{-1}{2}(\sigma_B^2 + \epsilon)^{-3/2}$$

$$\frac{\partial Div}{\partial \mu_B} = \left(\sum_{i=1}^{B} \frac{\partial Div}{\partial u_i} \cdot \frac{-1}{\sqrt{\sigma_B^2 + \epsilon}} \right) + \frac{\partial Div}{\partial \sigma_B^2} \cdot \frac{\sum_{i=1}^{B} -2(z_i - \mu_B)}{B}$$



$$\frac{\partial Div}{\partial z_i} = \frac{\partial Div}{\partial u_i} \cdot \frac{1}{\sqrt{\sigma_B^2 + \epsilon}} + \frac{\partial Div}{\partial \sigma_B^2} \cdot \frac{2(z_i - \mu_B)}{B} + \frac{\partial Div}{\partial \mu_B} \cdot \frac{1}{B}$$

$$\mu_B = \frac{1}{B}\sum_{i=1}^{B} z_i \qquad \sigma_B^2 = \frac{1}{B}\sum_{i=1}^{B}(z_i - \mu_B)^2$$

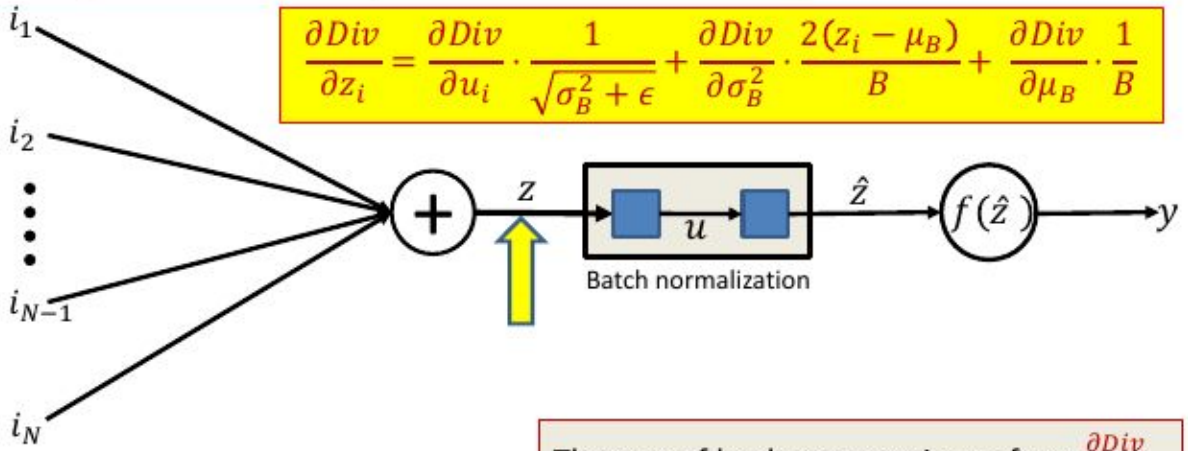
$$u_i = \frac{z_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}} \qquad \hat{z}_i = \gamma u_i + \beta$$

Batch normalization: Backpropagation

$$\frac{\partial Div}{\partial \sigma_B^2} = \sum_{i=1}^{B} \frac{\partial Div}{\partial u_i}(z_i - \mu_B) \cdot \frac{-1}{2}(\sigma_B^2 + \epsilon)^{-3/2}$$

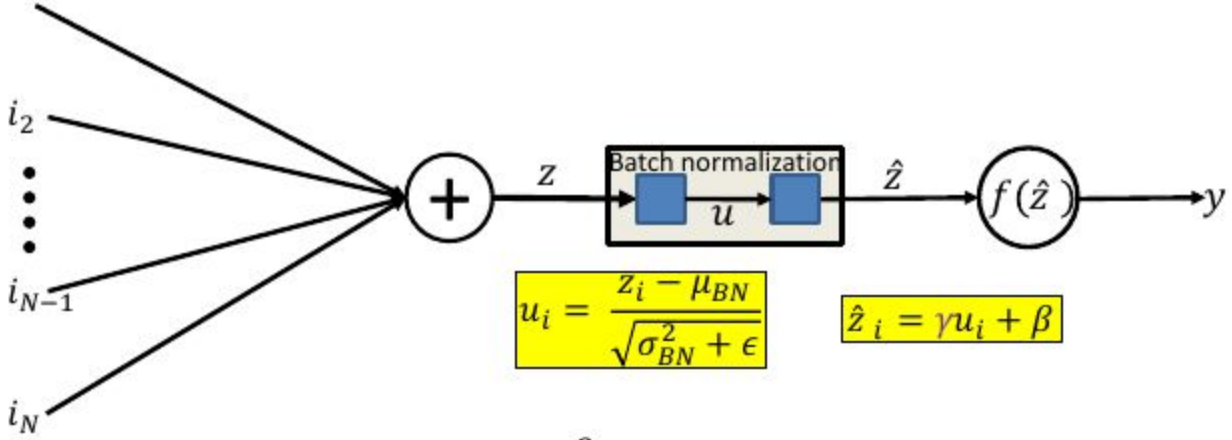
$$\frac{\partial Div}{\partial \mu_B} = \left(\sum_{i=1}^{B} \frac{\partial Div}{\partial u_i} \cdot \frac{-1}{\sqrt{\sigma_B^2 + \epsilon}} \right) + \frac{\partial Div}{\partial \sigma_B^2} \cdot \frac{\sum_{i=1}^{B} -2(z_i - \mu_B)}{B}$$

$$\frac{\partial Div}{\partial z_i} = \frac{\partial Div}{\partial u_i} \cdot \frac{1}{\sqrt{\sigma_B^2 + \epsilon}} + \frac{\partial Div}{\partial \sigma_B^2} \cdot \frac{2(z_i - \mu_B)}{B} + \frac{\partial Div}{\partial \mu_B} \cdot \frac{1}{B}$$



The rest of backprop continues from $\frac{\partial Div}{\partial z_i}$

Batch normalization: Inference



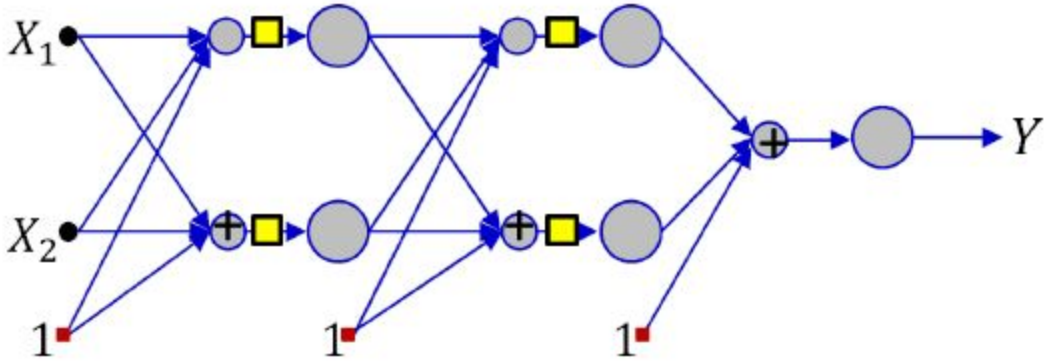
- On test data, BN requires μ_B and σ_B^2.
- We will use the *average over all training minibatches*

$$\mu_{BN} = \frac{1}{Nbatches} \sum_{batch} \mu_B(batch)$$

$$\sigma_{BN}^2 = \frac{B}{(B-1)Nbatches} \sum_{batch} \sigma_B^2(batch)$$

- Note: these are *neuron-specific*
  - $\mu_B(batch)$ and $\sigma_B^2(batch)$ here are obtained from the *final converged network*
  - The $B/(B-1)$ term gives us an unbiased estimator for the variance

Batch normalization



- Batch normalization may only be applied to *some* layers
  - Or even only selected neurons in the layer
- Improves both convergence rate and neural network performance
  - Anecdotal evidence that BN eliminates the need for dropout
  - To get maximum benefit from BN, learning rates must be increased and learning rate decay can be faster
    - Since the data generally remain in the high-gradient regions of the activations
  - Also needs better randomization of training data order

Story so far

- Gradient descent can be sped up by incremental updates
- Convergence can be improved using smoothed updates

- The choice of divergence affects both the learned network and results
- Covariate shift between training and test may cause problems and may be handled by batch normalization