Training Neural Networks: Optimization

Intro to Deep Learning, Spring 2020
Quick Recap

• Gradient descent, Backprop
Quick Recap: Training a network

- Define a total “loss” over all training instances
  - Quantifies the difference between desired output and the actual output, as a function of weights
- Find the weights that minimize the loss

\[ L(W) = \frac{1}{N_X} \sum_X \text{div}(f(X;W), D(X)) \]

\[ \hat{W} = \arg \min_W L(W) \]
Quick Recap: Training networks by gradient descent

\[ L(W) = \frac{1}{N_x} \sum_X \text{div}(f(X;W), D(X)) \]

\[ \nabla_W L(W) = \frac{1}{N_x} \sum_X \nabla_W \text{div}(f(X;W), D(X)) \]

Solved through gradient descent as

\[ \hat{W} = \arg\min_W L(W) \]

\[ W_k = W_{k-1} - \eta \nabla_W L(W)^T \]

- The gradient of the total loss is the average of the gradients of the loss for the individual instances
- The total gradient can be plugged into gradient descent update to learn the network
Quick Recap: Training networks by gradient descent

\[ L(W) = \frac{1}{N_X} \sum_X \text{loss} \]

\[ \nabla_W L(W) = \frac{1}{N_X} \sum_X \nabla_W \text{loss}(f(X; W), D(X)) \]

Solved through gradient descent as

\[ \hat{W} = \arg \min_W L(W) \]

\[ W_k = W_{k-1} - \eta \nabla_W L(W)^T \]

- The gradient of the total loss is the average of the gradients of the loss for the individual instances
- The gradient can be plugged into gradient descent update to learn the network parameters
Quick Recap

• Gradient descent, Backprop

• The issues with backprop and gradient descent

  – 1. Minimizes a loss which relates to classification accuracy, but is not actually classification accuracy
     • The divergence is a continuous valued proxy to classification error
     • Minimizing the loss is expected to, but not guaranteed to minimize classification error

  – 2. Simply minimizing the loss is hard enough..
Quick recap: Problem with gradient descent

- A step size that assures fast convergence for a given eccentricity can result in divergence at a higher eccentricity
- Or result in extremely slow convergence at lower eccentricity

$$W_k = W_{k-1} - \eta \nabla_W L(W)^T$$
Quick recap: Problem with gradient descent

- The loss is a function of many weights (and biases)
  - Has different eccentricities w.r.t different weights
- A fixed step size for all weights in the network can result in the convergence of one weight, while causing a divergence of another
Story so far: Second-order methods

- Second-order methods “normalize” the variation along the components to mitigate the problem of different optimal learning rates for different components
  - But this requires computation of inverses of second-order derivative matrices
  - Computationally infeasible
  - Not stable in non-convex regions of the loss surface
  - Approximate methods address these issues, but simpler solutions may be better
Recap: 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.

- Better to start with a large (divergent) learning rate and slowly shrink it over iterations.
  - More likely to find better minima.
Story so far: Learning rate

- Divergence-causing learning rates may not be a bad thing
  - Particularly for ugly loss functions
- *Decaying* learning rates provide good compromise between escaping poor local minima and convergence

- Many of the convergence issues arise because we force the same learning rate on all parameters
Let's take a step back

- Problems arise because of requiring a fixed step size across all dimensions
  - Because step are “tied” to the gradient
- Let's try releasing this requirement
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):
    • increase 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
• Select an initial value $\hat{w}$ and compute the derivative
  – Take an initial step $\Delta w$ against the derivative
    • In the direction that reduces the function
      $\Delta w = \text{sign} \left( \frac{dE(\hat{w})}{dw} \right) \Delta w$
      $\hat{w} = \hat{w} - \Delta w$
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 longer step

\[ \alpha > 1 \]

- \( \Delta w = \alpha \Delta w \)
- \( \hat{w} = \hat{w} - \Delta w \)
• 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

\[ \alpha > 1 \]

- \( \Delta w = \alpha \Delta w \)
- \( \hat{w} = \hat{w} - \Delta w \)
• Compute the derivative in the new location
  – If the derivative has changed sign

Orange arrow shows direction of derivative, i.e. direction of increasing \( E(w) \)
Rprop

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

Orange arrow shows direction of derivative, i.e. direction of increasing $E(w)$.
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
    - $\Delta w = \beta \Delta w$
    - Take the smaller step forward
      - $\hat{w} = \hat{w} - \Delta w$

$\beta < 1$

Orange arrow shows direction of derivative, i.e. direction of increasing $E(w)$
Rprop (simplified)

• Set $\alpha = 1.2$, $\beta = 0.5$
• For each layer $l$, for each $i, j$:
  – Initialize $w_{l,i,j}$, $\Delta w_{l,i,j} > 0$
  – $prevD(l, i, j) = \frac{dErr(w_{l,i,j})}{dw_{l,i,j}}$
  – $\Delta w_{l,i,j} = \text{sign}(prevD(l, i, j)) \Delta w_{l,i,j}$
  – 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 $\text{sign}(prevD(l, i, j)) = \text{sign}(D(l, i, j))$:
      – $\Delta w_{l,i,j} = \min(\alpha \Delta w_{l,i,j}, \Delta_{\text{max}})$
      – $prevD(l, i, j) = D(l, i, j)$
    • else:
      – $w_{l,i,j} = w_{l,i,j} + \Delta w_{l,i,j}$
      – $\Delta w_{l,i,j} = \max(\beta \Delta w_{l,i,j}, \Delta_{\text{min}})$

Ceiling and floor on step
Rprop (simplified)

- Set $\alpha = 1.2$, $\beta = 0.5$
- For each layer $l$, for each $i, j$:
  - Initialize $w_{l,i,j}, \Delta w_{l,i,j} > 0$,
  - $prevD(l, i, j) = \frac{dErr(w_{l,i,j})}{dw_{l,i,j}}$
  - $\Delta w_{l,i,j} = \text{sign}(prevD(l, i, j)) \Delta w_{l,i,j}$
  - 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 sign($prevD(l, i, j)$) == sign($D(l, i, j)$):
      - $\Delta w_{l,i,j} = \alpha \Delta w_{l,i,j}$
      - $prevD(l, i, j) = D(l, i, j)$
    - 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

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

- But with two modifications
QuickProp: Modification 1

- It treats each dimension independently
- For $i = 1: N$
  \[ w_i^{k+1} = w_i^k - E''(w_i^k | w_j^k, j \neq i)^{-1} E'(w_i^k | w_j^k, j \neq i) \]
- This eliminates the need to compute and invert expensive Hessians
QuickProp: Modification 2

- It approximates the second derivative through finite differences
- For $i = 1: N$
  \[ w_i^{k+1} = w_i^k - D(w_i^k, w_i^{k-1})^{-1} E'(w_i^k | w_j^k, j \neq i) \]
- This eliminates the need to compute expensive double derivatives
**QuickProp**

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

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 \((l - 1)^{th}\) layer to node \( j \) in the \( l^{th}\) layer:

\[
  \Delta w^{(k)}_{l,ij} = \frac{\Delta w^{(k-1)}_{l,ij}}{Err'(w^{(k)}_{l,ij}) - Err'(w^{(k-1)}_{l,ij})} \cdot Err'(w^{(k)}_{l,ij}) \cdot w^{(k-1)}_{l,ij}
\]

\[
  w^{(k+1)}_{l,ij} = w^{(k)}_{l,ij} - \Delta w^{(k)}_{l,ij}
\]
QuickProp

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

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 \( (l - 1) \)th layer to node \( j \) in the \( l \)th layer:

\[
\Delta w_{l,ij}^{(k)} = \frac{\Delta w_{l,ij}^{(k-1)}}{Err'(w_{l,ij}^{(k)}) - Err'(w_{l,ij}^{(k-1)})} \cdot Err'(w_{l,ij}^{(k)})
\]

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

Computed using backprop
Quickprop

• Employs Newton updates with empirically derived derivatives

• Prone to some instability for non-convex objective functions

• But is still one of the fastest training algorithms for many problems
Story so far : Convergence

• Gradient descent can miss obvious answers
  – And this may be a *good* thing

• Vanilla gradient descent may be too slow or unstable due to the differences between the dimensions

• Second order methods can normalize the variation across dimensions, but are complex

• Adaptive or decaying learning rates can improve convergence

• Methods that decouple the dimensions can improve convergence
But...

• Try to normalize curvature in all directions
  – Second order methods, e.g. Newton’s method
  – Too expensive: require inversion of a giant Hessian

• Treat each dimension independently:
  – Rprop, quickprop
  – Works, but ignores dependence between dimensions
    • Can result in unexpected behavior
  – Can still be too slow
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

\[ \Delta W^{(k)} = \beta \Delta W^{(k-1)} - \eta \nabla_W \text{Loss}(W^{(k-1)})^T \]

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

- Typical \( \beta \) 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 $W_1, W_2, \ldots, W_K$

• Do:
  
  – For all $i, j, k$, initialize $\nabla_{W_k} \text{Loss} = 0$
  
  – For all $t = 1: T$
    
    • For every layer $k$:
      
      – Compute $\nabla_{W_k} \text{Div}(Y_t, d_t)$
      
      – Compute $\nabla_{W_k} \text{Loss} += \frac{1}{T} \nabla_{W_k} \text{Div}(Y_t, d_t)$

  – For every layer $k$:
    
    $$W_k = W_k - \eta (\nabla_{W_k} \text{Loss})^T$$

• Until Loss has converged
Training with momentum

• Initialize all weights $W_1, W_2, \ldots, W_K$

• Do:
  – For all layers $k$, initialize $\nabla_{W_k} \text{Loss} = 0$, $\Delta W_k = 0$
  – For all $t = 1: T$
    • For every layer $k$:
      – Compute gradient $\nabla_{W_k} \text{Div}(Y_t, d_t)$
      – $\nabla_{W_k} \text{Loss} += \frac{1}{T} \nabla_{W_k} \text{Div}(Y_t, d_t)$
    – For every layer $k$
      $$\Delta W_k = \beta \Delta W_k - \eta (\nabla_{W_k} \text{Loss})^T$$
      $$W_k = W_k + \Delta W_k$$

• Until $\text{Loss}$ has converged
Momentum Update

• The momentum method

\[ \Delta W^{(k)} = \beta \Delta W^{(k-1)} - \eta \nabla_W \text{Loss} (W^{(k-1)})^T \]

• At any iteration, to compute the current step:
Momentum Update

- The momentum method

\[ \Delta W^{(k)} = \beta \Delta W^{(k-1)} - \eta \nabla_W Loss(W^{(k-1)})^T \]

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

- The momentum method

\[ \Delta W^{(k)} = \beta \Delta W^{(k-1)} - \eta \nabla_w \text{Loss}(W^{(k-1)})^T \]

- 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
  \[ \Delta W^{(k)} = \beta \Delta W^{(k-1)} - \eta \nabla_W \text{Loss}(W^{(k-1)})^T \]

- 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

\[
\Delta W^{(k)} = \beta \Delta W^{(k-1)} - \eta \nabla_w \text{Loss} \left( W^{(k-1)} + \beta \Delta W^{(k-1)} \right)^T \\
W^{(k)} = W^{(k-1)} + \Delta W^{(k)}
\]
Nestorov’s Accelerated Gradient

- Comparison with momentum (example from Hinton)
- Converges much faster
Training with Nestorov

• Initialize all weights $W_1, W_2, \ldots, W_K$

• Do:
  – For all layers $k$, initialize $\nabla_{W_k} \text{Loss} = 0$, $\Delta W_k = 0$
  – For every layer $k$
    
    $$W_k = W_k + \beta \Delta W_k$$
  – For all $t = 1: T$
    • For every layer $k$:
      – Compute gradient $\nabla_{W_k} \text{Div}(Y_t, d_t)$
      – $\nabla_{W_k} \text{Loss} += \frac{1}{T} \nabla_{W_k} \text{Div}(Y_t, d_t)$
  – For every layer $k$
    $W_k = W_k - \eta (\nabla_{W_k} \text{Loss})^T$
    $\Delta W_k = \beta \Delta W_k - \eta (\nabla_{W_k} \text{Loss})^T$

• Until $\text{Loss}$ has converged
Momentum and trend-based methods.

• We will return to this topic again, very soon..
• Gradient descent can miss obvious answers
  – And this may be a *good* thing

• Vanilla gradient descent may be too slow or unstable due to the differences between the dimensions

• Second order methods can normalize the variation across dimensions, but are complex

• Adaptive or decaying learning rates can improve convergence

• Methods that decouple the dimensions can improve convergence

• Momentum methods which emphasize directions of steady improvement are demonstrably superior to other methods
Quick Summary

• Gradient descent, Backprop
• The issues with backprop and gradient descent
  • Momentum methods..
Momentum methods: principle

- Ideally: Have component-specific step size
  - But the resulting updates will not be against the gradient and do not guarantee descent
- Adaptive solution: Start with a common step size
  - Shrink step size in directions where the weight oscillates
  - Expand step size in directions where the weight moves consistently in one direction

\[ W_k = W_{k-1} - \eta \nabla_w L(W)^T \]

Increase stepsize because previous updates consistently moved weight right

Decrease stepsize because previous updates kept changing direction

Stepsize shrinks along \( w_2 \) but increases along \( w_1 \)
Quick recap: Momentum methods

Momentum

\[ \Delta W^{(k)} = \beta \Delta W^{(k-1)} - \eta \nabla_{W} \text{Loss}(W^{(k-1)})^T \]

Nestorov

\[ W^{(k)}_{\text{extend}} = W^{(k-1)} + \beta \Delta W^{(k-1)} \]
\[ \Delta W^{(k)} = \beta \Delta W^{(k-1)} - \eta \nabla_{W} \text{Loss}(W^{(k)}_{\text{extend}})^T \]
\[ W^{(k)} = W^{(k-1)} + \Delta W^{(k)} \]

• Momentum: Retain gradient value, but *smooth out* gradients by maintaining a running average
  – Cancels out steps in directions where the weight value oscillates
  – Adaptively increases step size in directions of consistent change
Recap

- Neural networks are universal approximators
- We must *train* them to approximate any function
- Networks are trained to minimize total “error” on a training set
  - We do so through empirical risk minimization
- We use variants of gradient descent to do so
  - Gradients are computed through backpropagation
Recap

• Vanilla gradient descent may be too slow or unstable

• Better convergence can be obtained through
  – Second order methods that normalize the variation across dimensions
  – Adaptive or decaying learning rates that can improve convergence
  – Methods like Rprop that decouple the dimensions can improve convergence
  – Momentum methods which emphasize directions of steady improvement and deemphasize unstable directions
Moving on…

• Incremental updates
• Revisiting “trend” algorithms
• Generalization
• Tricks of the trade
  – Divergences..
  – Activations
  – Normalizations
Moving on: Topics for the day

• Incremental updates
• Revisiting “trend” algorithms
• Generalization
• Tricks of the trade
  – Divergences..
  – Activations
  – Normalizations
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 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
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
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
• 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: 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

• 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 \(W_1, W_2, \ldots, W_K\)

• Do:
  – For all \(t = 1: T\)
    • For every layer \(k\):
      – Compute \(\nabla_{W_k} \text{Div}(Y_t, d_t)\)
      – Update
        \[ W_k = W_k - \eta \nabla_{W_k} \text{Div}(Y_t, d_t)^T \]

• Until \textbf{Loss} 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
Incremental Update: Stochastic Gradient Descent

• Given \((X_1, d_1), (X_2, d_2),..., (X_T, d_T)\)

• Initialize all weights \(W_1, W_2, ..., W_K\)

• Do:

  – For all \(t = 1: T\)
    • For every layer \(k\):
      – Compute \(\nabla_{W_k} \text{Div}(Y_t, d_t)\)
      – Update \(W_k = W_k - \eta \nabla_{W_k} \text{Div}(Y_t, d_t)^T\)

• Until \(\text{Loss}\) has converged
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
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: 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: 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.
• 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: 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.
Incremental Update: Stochastic Gradient Descent

• Given \((X_1, d_1), (X_2, d_2), \ldots, (X_T, d_T)\)
• Initialize all weights \(W_1, W_2, \ldots, W_K\)
• Do:
  — Randomly permute \((X_1, d_1), (X_2, d_2), \ldots, (X_T, d_T)\)
  — For all \(t = 1: T\)
    • For every layer \(k\):
      — Compute \(\nabla_{W_k} \text{Div}(Y_t, d_t)\)
      — Update
        \[W_k = W_k - \eta \nabla_{W_k} \text{Div}(Y_t, d_t)^T\]
• Until \textit{Loss} has converged
• In any gradient descent optimization problem, presenting training instances incrementally can be more effective than presenting them all at once
  – Provided training instances are provided 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?

• For “why”: first consider a simplistic explanation that’s often given
  – Look at an extreme example
The expected behavior of the gradient

\[
dE(W^{(1)}, W^{(2)}, ..., W^{(K)}) \bigg/ \bigg[ \frac{d}{dw_{l,j}} \bigg]^{(k)} = \frac{1}{T} \sum_i dDiv(Y(X_i), d_i; W^{(1)}, W^{(2)}, ..., W^{(K)}) \bigg/ \bigg[ \frac{d}{dw_{l,j}} \bigg]^{(k)}
\]

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

\[ X_1 = X_2 = \cdots = X_T \]

- Extreme instance of data clotting: all the training instances are exactly the same
The expected behavior of the gradient

\[
\frac{dE}{dw_{i,j}^{(k)}} = \frac{1}{T} \sum_i \frac{dDiv(Y(X_i), d_i)}{dw_{i,j}^{(k)}} = \frac{dDiv(Y(X_i), d_i)}{dw_{i,j}^{(k)}}
\]

- The individual training instance contribute identical directions to the overall gradient
  - The final gradient points is simply the gradient for an individual instance
Batch vs SGD

- Batch gradient descent operates over $T$ training instances to get a single update.
- SGD gets $T$ updates for the same computation.

$X_1 = X_2 = \ldots = X_T$
Clumpy data..

- Also holds if all the data are not identical, but are tightly clumped together

\[X_1 \approx X_2 \approx \ldots \approx X_T\]
Clumpy data..

- As data get increasingly diverse, the benefits of incremental updates decrease, but do not entirely vanish
When does it work

• What are the considerations?

• And how well does it work?
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
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 \(\nabla_{W_k} \text{Div}(Y_t, d_t)\)
      - Update
        \[
        W_k = W_k - \eta_j \nabla_{W_k} \text{Div}(Y_t, d_t)^T
        \]
  
- Until \textbf{Loss} has converged
Incremental Update: Stochastic Gradient Descent

- Given \((X_1, d_1), (X_2, d_2),..., (X_T, d_T)\)
- Initialize all weights \(W_1, W_2, ..., W_K; \ j = 0\)
- Do:
  - Randomly permute \((X_1, d_1), (X_2, d_2),..., (X_T, d_T)\)
  - For all \(t = 1:T\)
    - \(j = j + 1\)
    - For every layer \(k:\)
      - Compute \(\nabla_{W_k} \text{Div}(Y_t, d_t)\)
      - Update
        \[
        W_k = W_k - \eta_j \nabla_{W_k} \text{Div}(Y_t, d_t)^T
        \]
  - Until Loss has converged
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 local minimum
SGD convergence

- We will define convergence in terms of the number of iterations taken to get within $\epsilon$ of the optimal solution
  - $|f(W^{(k)}) - f(W^*)| < \epsilon$
  - Note: $f(W)$ here is the error on the entire training data, although SGD itself updates after every training instance

- Using the optimal learning rate $1/k$, for strongly convex functions,
  $$|W^{(k)} - W^*| < \frac{1}{k} |W^{(0)} - W^*|$$
  - Strongly convex $\rightarrow$ Can be placed inside a quadratic bowl, touching at any point
  - Giving us the iterations to $\epsilon$ convergence as $O\left(\frac{1}{\epsilon}\right)$

- For generically convex (but not strongly convex) function, various proofs report an $\epsilon$ convergence of $\frac{1}{\sqrt{k}}$ using a learning rate of $\frac{1}{\sqrt{k}}$. 
Batch gradient convergence

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

\[ |W^{(k)} - W^*| < c^k |W^{(0)} - W^*| \]

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

• For generic convex functions, iterations to \( \epsilon \) convergence is \( O\left(\frac{1}{\epsilon}\right) \)

• Batch gradients converge “faster”
  – But SGD performs \( T \) updates for every batch update
SGD Convergence: Loss value

If:

• $f$ is $\lambda$-strongly convex, and
• at step $t$ we have a noisy estimate of the subgradient $\hat{g}_t$ with $\mathbb{E}[\|\hat{g}_t\|^2] \leq G^2$ for all $t$,
• and we use step size $\eta_t = \frac{1}{\lambda t}$

Then for any $T > 1$:

$$
\mathbb{E}[f(w_T) - f(w^*)] \leq \frac{17G^2(1 + \log(T))}{\lambda T}
$$
SGD Convergence

• We can bound the expected difference between the loss over our data using the optimal weights $w^*$ and the weights $w_T$ at any single iteration 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 Convergence and weight averaging

Polynomial Decay Averaging:

\[
\bar{w}_t^\gamma = \left( 1 - \frac{\gamma + 1}{t + \gamma} \right) \bar{w}_{t-1}^\gamma + \frac{\gamma + 1}{t + \gamma} w_t
\]

With \(\gamma\) some small positive constant, e.g. \(\gamma = 3\)

Achieves \(O\left(\frac{1}{T}\right)\) (strongly convex) and \(O\left(\frac{1}{\sqrt{T}}\right)\) (convex) convergence
A simpler problem: K-means
Note: SGD converges slower
Also note the rather large variation between runs
  – Lets try to understand these results..
Recall: Modelling a function

- To learn a network $f(X; W)$ to model a function $g(X)$ we minimize the expected divergence

$$
\bar{W} = \arg\min_W \int_X \text{div}(f(X; W), g(X))P(X)dx
= \arg\min_W E\left[\text{div}(f(X; W), g(X))\right]
$$
Recall: The *Empirical* risk

- In practice, we minimize the *empirical risk* (or loss)

\[
\text{Loss}(f(X; W), g(X)) = \frac{1}{N} \sum_{i=1}^{N} \text{div}(f(X_i; W), d_i)
\]

\[
\hat{W} = \text{argmin}_{W} \text{Loss}(f(X; W), g(X))
\]

- The *expected value* of the *empirical risk* is actually the *expected divergence*

\[
E[\text{Loss}(f(X; W), g(X))] = E[\text{div}(f(X; W), g(X))]
\]
Recall: The Empirical risk

- In practice, we minimize the empirical risk (or loss)

\[ \text{Loss}(f(X; W), g(X)) = \frac{1}{N} \sum_{i=1}^{N} \text{div}(f(X_i; W), d_i) \]

The empirical risk is an unbiased estimate of the expected loss

Though there is no guarantee that minimizing it will minimize the expected loss

\[ E[\text{Loss}(f(X; W), g(X))] = E[\text{div}(f(X; W), g(X))] \]
Recall: The *Empirical risk*

In practice, we minimize the empirical risk $\mathcal{L} = \frac{1}{N} \sum_{i=1}^{N} \text{div}(f(X_i; W), d_i)$.

The expected value of the empirical risk is actually the expected divergence $E[\mathcal{L}] = E[\text{div}(f(X; W), g(X))]$.

The empirical risk is an *unbiased* estimate of the expected loss.

Though there is no guarantee that minimizing it will minimize the expected loss.

The variance of the estimator is proportional to $1/N$.

The larger this variance, the greater the likelihood that the $W$ that minimizes the empirical risk will differ significantly from the $W$ that minimizes the expected loss.

The variance of the empirical risk: $\text{var}(\mathcal{L}) = \frac{1}{N} \text{var}(\text{div})$.
• At each iteration, SGD focuses on the divergence of a single sample \( \text{div}(f(X_i; W), d_i) \)

• The expected value of the sample error is still the expected divergence \( E[\text{div}(f(X; W), g(X))] \)
SGD

- At each iteration, SGD focuses on the divergence of a single sample $\text{div}(f(X_i; W), d_i)$
- The expected value of the sample error is still the expected divergence $\mathbb{E}[\text{div}(f(X; W), g(X))]$
At each iteration, SGD focuses on the divergence of a single sample $\text{div}(f(X_i; W), d_i)$.

The sample error is also an unbiased estimate of the expected error $E\left[\text{div}(f(X; W), g(X))\right]$. 

The variance of the sample error is the variance of the divergence itself: $\text{var}(\text{div})$. This is $N$ times the variance of the empirical average minimized by batch update.
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
  - The divergence is a function of the error
  - We want to find the $W$ that minimizes the average divergence
Explaining the variance

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

\[ f(x) \]

\[ g(x; W) \]
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
• 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 example

- A simpler problem: K-means
- Note: SGD converges slower
- Also has large variation between runs
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
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
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
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 \((X_1, d_1), (X_2, d_2), \ldots, (X_T, d_T)\)
- Initialize all weights \(W_1, W_2, \ldots, W_K; \quad j = 0\)
- Do:
  - Randomly permute \((X_1, d_1), (X_2, d_2), \ldots, (X_T, d_T)\)
  - 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 \(\nabla W_k Div(Y_t, d_t)\)
        - \(\Delta W_k = \Delta W_k + \frac{1}{b} \nabla W_k Div(Y_t, d_t)^T\)
  - Update
    - For every layer \(k\):
      \[W_k = W_k - \eta_j \Delta W_k\]
- Until \(Err\) has converged
**Incremental Update: Mini-batch update**

- Given \((X_1, d_1), (X_2, d_2), ..., (X_T, d_T)\)
- Initialize all weights \(W_1, W_2, ..., W_K\); \(j = 0\)
- 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 \(\nabla_{W_k} \text{Div}(Y_t, d_t)\)
        - \(\Delta W_k = \Delta W_k + \frac{1}{b} \nabla_{W_k} \text{Div}(Y_t, d_t)^T\)
      - Update
        - For every layer \(k\):
          \[ W_k = W_k - \eta_j \Delta W_k \]
  - Until \(Err\) has converged
Mini Batches

- Mini-batch updates compute and minimize a batch loss

\[ \text{BatchLoss}(f(X;W), g(X)) = \frac{1}{b} \sum_{i=1}^{b} \text{div}(f(X_i;W), d_i) \]

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

\[ E[\text{BatchLoss}(f(X;W), g(X))] = E[\text{div}(f(X;W), g(X))] \]
Mini Batches

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

- Mini-batch updates compute and minimize a batch loss

\[
\text{BatchLoss}(f(X; W), g(X)) = \frac{1}{b} \sum_{i=1}^{b} \text{div}(f(X_i; W), d_i)
\]

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

\[
E[\text{BatchLoss}(f(X; W), g(X))] = E[\text{div}(f(X; W), g(X))]
\]
Mini Batches

The variance of the batch loss: \( \text{var}(\text{BatchLoss}) = \frac{1}{b} \text{var}(\text{div}) \)

This will be much smaller than the variance of the sample error in SGD

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

- Mini-batch updates compute and minimize a batch loss

\[
\text{BatchLoss}(f(X; W), g(X)) = \frac{1}{b} \sum_{i=1}^{b} \text{div}(f(X_i; W), d_i)
\]

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

\[
E[\text{BatchLoss}(f(X; W), g(X))] = E[\text{div}(f(X; W), g(X))]
\]
Minibatch convergence

• For convex functions, convergence rate for SGD is \( O \left( \frac{1}{\sqrt{k}} \right) \).

• For mini-batch updates with batches of size \( b \), the convergence rate is \( O \left( \frac{1}{\sqrt{bk}} + \frac{1}{k} \right) \)
  
  – 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
Measuring Loss

• Convergence is generally defined in terms of the overall training loss
  – Not sample or batch loss

• Infeasible to actually measure the overall training loss after each iteration

• More typically, we estimate is as
  – Divergence or classification error on a held-out set
  – Average sample/batch loss over the past \( N \) samples/batches
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
• 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
Training and minibatches

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

- Incremental updates
- Revisiting “trend” algorithms
- Generalization
- Tricks of the trade
  - Divergences..
  - Activations
  - Normalizations
Recall: Momentum

• The momentum method
  \[ \Delta W^{(k)} = \beta \Delta W^{(k-1)} - \eta \nabla_W \text{Err}(W^{(k-1)}) \]

• Updates using a running average of the gradient
The momentum method

$$\Delta W^{(k)} = \beta \Delta W^{(k-1)} - \eta \nabla_w \text{Loss}(W^{(k-1)})^T$$

Incremental SGD and mini-batch gradients tend to have high variance

Momentum smooths out the variations
  – Smoother and faster convergence
Incremental Update: **Mini-batch update**

- Given \((X_1, d_1), (X_2, d_2), \ldots, (X_T, d_T)\)
- Initialize all weights \(W_1, W_2, \ldots, W_K; \ j = 0, \Delta W_k = 0\)
- Do:
  - Randomly permute \((X_1, d_1), (X_2, d_2), \ldots, (X_T, d_T)\)
  - For \(t = 1 : b: T\)
    - \(j = j + 1\)
    - For every layer \(k\):
      - \(\nabla_{W_k} \text{Loss} = 0\)
    - For \(t' = t : t+b-1\)
      - For every layer \(k\):
        - Compute \(\nabla_{W_k} \text{Div}(Y_t, d_t)\)
        - \(\nabla_{W_k} \text{Loss} += \frac{1}{b} \nabla_{W_k} \text{Div}(Y_t, d_t)\)
  - Update
    - For every layer \(k\):
      \[
      \Delta W_k = \beta \Delta W_k - \eta_j (\nabla_{W_k} \text{Loss})^T
      \]
      \[
      W_k = W_k + \Delta W_k
      \]
- Until **Loss** has converged
Nestorov’s Accelerated Gradient

• At any iteration, to compute the current step:
  – First extend the previous step
  – Then compute the gradient at the resultant position
  – Add the two to obtain the final step

• This also applies directly to incremental update methods
  – The accelerated gradient smooths out the variance in the gradients
Nestorov’s Accelerated Gradient

- Nestorov’s method
  \[ \Delta W^{(k)} = \beta \Delta W^{(k-1)} - \eta \nabla_W \text{Loss}(W^{(k-1)} + \beta \Delta W^{(k-1)})^T \]
  \[ W^{(k)} = W^{(k-1)} + \Delta W^{(k)} \]
Incremental Update: Mini-batch update

- Given \((X_1, d_1), (X_2, d_2), ..., (X_T, d_T)\)
- Initialize all weights \(W_1, W_2, ..., W_K; \ j = 0, \Delta W_k = 0\)
- 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 \(k:\)
      - \(W_k = W_k + \beta \Delta W_k\)
      - \(\nabla_{W_k} \text{Loss} = 0\)
    - For \(t' = t: t+b-1\)
      - For every layer \(k:\)
        - Compute \(\nabla_{W_k} \text{Div}(Y_t, d_t)\)
        - \(\nabla_{W_k} \text{Loss} += \frac{1}{b} \nabla_{W_k} \text{Div}(Y_t, d_t)\)
  - Update
    - For every layer \(k:\)
      - \(W_k = W_k - \eta j \nabla_{W_k} \text{Loss}^T\)
      - \(\Delta W_k = \beta \Delta W_k - \eta j \nabla_{W_k} \text{Loss}^T\)

- Until \(\text{Loss}\) has converged
Still higher-order methods

• Momentum and Nestorov’s method improve convergence by normalizing the mean of the derivatives

• More recent methods take this one step further by also considering their variance
  – RMS Prop
  – Adagrad
  – AdaDelta
  – ADAM: very popular in practice
  – ...

• All roughly equivalent in performance
Smoothing the trajectory

• Simple gradient and acceleration methods still demonstrate oscillatory behavior in some directions
  – Depends on magic step size parameters
• Observation: Steps in “oscillatory” directions show large total movement
  – In the example, total motion in the vertical direction is much greater than in the horizontal direction
• Improvement: Dampen step size in directions with high motion
  – *Second order term*
Normalizing steps by second moment

- 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
  - According to their variation (and not just their average)
- A variety of algorithms have been proposed on this premise
  - We will see a popular example
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 $\partial_w^2 D = (\partial_w D)^2$
    • Short-hand notation represents the squared derivative, not the second derivative
  – The mean squared derivative is a running estimate of the average squared derivative. We will show this as $E[\partial_w^2 D]$

• Modified update rule: We want to
  – scale down 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

• 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

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

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

**Note similarity to RPROP**
The magnitude of the derivative is being normalized out
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, \( E[\partial^2_wD]_k = 0 \)
  – For all \( t = 1: B: T \) (incrementing in blocks of \( B \) inputs)
    • For all weights in all layers initialize \( (\partial_wD)_k = 0 \)
    • For \( b = 0: B - 1 \)
      – Compute
        » Output \( Y(X_{t+b}) \)
        » Compute gradient \( \frac{dDiv(Y(X_{t+b}),d_{t+b})}{dw} \)
        » Compute \( (\partial_wD)_k += \frac{1}{B} \frac{dDiv(Y(X_{t+b}),d_{t+b})}{dw} \)

• update:

\[
E[\partial^2_wD]_k = \gamma E[\partial^2_wD]_{k-1} + (1 - \gamma)(\partial^2_wD)_k
\]

\[
w_{k+1} = w_k - \frac{\eta}{\sqrt{E[\partial^2_wD]_k + \epsilon}} \partial_wD
\]

• \( k = k + 1 \)

• Until \( E(W^{(1)}, W^{(2)}, \ldots, W^{(K)}) \) 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

\[
\begin{align*}
m_k &= \delta m_{k-1} + (1 - \delta)(\partial_w D)_k \\
v_k &= \gamma v_{k-1} + (1 - \gamma)(\partial_w^2 D)_k \\
\hat{m}_k &= \frac{m_k}{1 - \delta^k}, \quad \hat{v}_k = \frac{v_k}{1 - \gamma^k} \\
w_{k+1} &= w_k - \frac{\eta}{\sqrt{\hat{v}_k + \epsilon}} \hat{m}_k
\end{align*}
\]
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

**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

\[
\begin{align*}
m_k &= \delta m_{k-1} + (1 - \delta)(\partial_w D)_k \\
v_k &= \gamma v_{k-1} + (1 - \gamma)(\partial_w^2 D)_k \\
\hat{m}_k &= \frac{m_k}{1 - \delta^k}, \quad \hat{v}_k = \frac{v_k}{1 - \gamma^k}
\end{align*}
\]

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

Ensures that the $\delta$ and $\gamma$ terms do not dominate in early iterations
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

Visualizing the optimizers: Long Valley

Visualizing the optimizers: Saddle Point

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 more advanced techniques