Charu C. Aggarwal
IBM T J Watson Research Center
Yorktown Heights, NY

Backpropagation I: Computing Derivatives in Computational Graphs [without Backpropagation] in Exponential Time

Neural Networks and Deep Learning, Springer, 2018
Chapter 3, Section 3.2

Why Do We Need Backpropagation?

- To perform any kind of learning, we need to compute the partial derivative of the loss function with respect to each intermediate weight.
  - Simple with single-layer architectures like the perceptron.
  - Not a simple matter with multi-layer architectures.

The Complexity of Computational Graphs

- A computational graph is a directed acyclic graph in which each node computes a function of its incoming node variables.

- A neural network is a special case of a computational graph.
  - Each node computes a combination of a linear vector multiplication and a (possibly nonlinear) activation function.

- The output is a very complicated *composition* function of each intermediate weight in the network.
  - The complex composition function might be hard to express neatly in closed form.
    - Difficult to differentiate!

Recursive Nesting is Ugly!

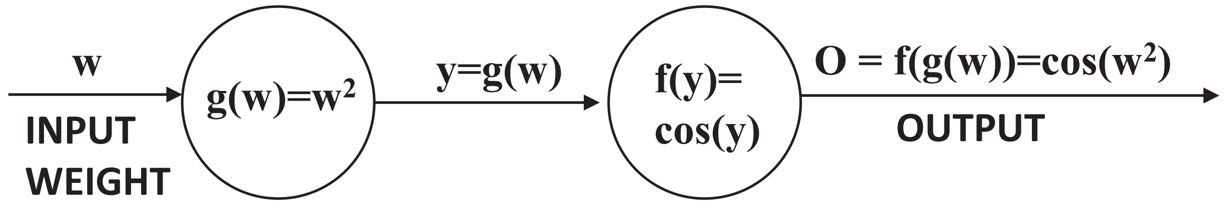
- Consider a computational graph containing two nodes in a path and input w.

- The first node computes $y = g(w)$ and the second node computes the output $o = f(y)$.
  - Overall composition function is $f(g(w))$.
  - Setting $f()$ and $g()$ to the sigmoid function results in the following:

$$f(g(w)) = \frac{1}{1 + \exp\left[-\frac{1}{1+\exp(-w)}\right]} \tag{1}$$

  - Increasing path length increases recursive nesting.

Backpropagation along Single Path (Univariate Chain Rule)



- Consider a two-node path with $f(g(w)) = \cos(w^2)$

- In the univariate chain rule, we compute product of *local* derivatives.

$$\frac{\partial f(g(w))}{\partial w} = \underbrace{\frac{\partial f(y)}{\partial y}}_{-sin(y)} \cdot \underbrace{\frac{\partial g(w)}{\partial w}}_{2w} = -2w \cdot \sin(y) = -2w \cdot \sin(w^2)$$

- Local derivatives are easy to compute because they care about their own input and output.

Backpropagation along Multiple Paths (Multivariate Chain Rule)

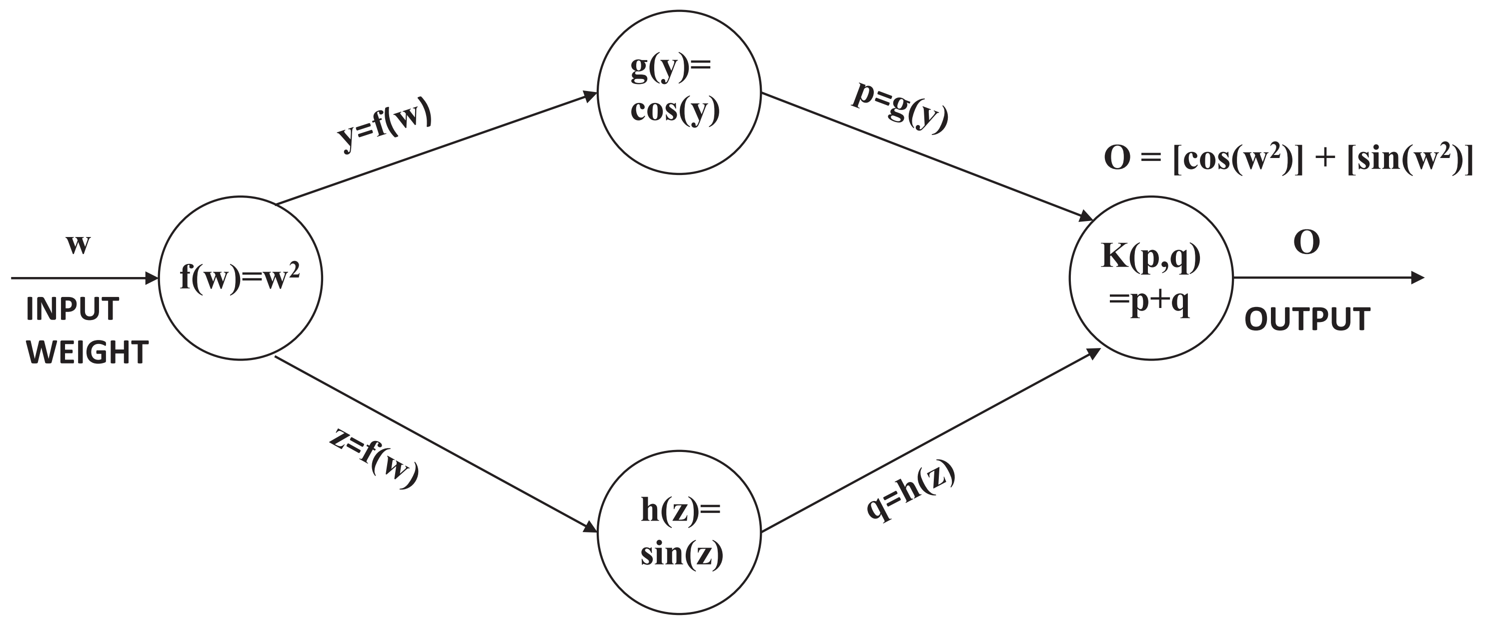
- Neural networks contain multiple nodes in each layer.

- Consider the function $f(g_1(w), \ldots g_k(w))$, in which a unit computing the *multivariate* function $f(\cdot)$ gets its inputs from k units computing $g_1(w) \ldots g_k(w)$.

- The *multivariable chain rule* needs to be used:

$$\frac{\partial f(g_1(w), \ldots g_k(w))}{\partial w} = \sum_{i=1}^{k} \frac{\partial f(g_1(w), \ldots g_k(w))}{\partial g_i(w)} \cdot \frac{\partial g_i(w)}{\partial w} \tag{2}$$

Example of Multivariable Chain Rule



$$\begin{aligned}\frac{\partial o}{\partial w} &= \underbrace{\frac{\partial K(p,q)}{\partial p}}_{1} \cdot \underbrace{g'(y)}_{\text{-sin}(y)} \cdot \underbrace{f'(w)}_{2w} + \underbrace{\frac{\partial K(p,q)}{\partial q}}_{1} \cdot \underbrace{h'(z)}_{\cos(z)} \cdot \underbrace{f'(w)}_{2w} \\ &= -2w \cdot \sin(y) + 2w \cdot \cos(z) \\ &= -2w \cdot \sin(w^2) + 2w \cdot \cos(w^2)\end{aligned}$$

- Product of local derivatives along *all* paths from w to o.

Pathwise Aggregation Lemma

- Let a non-null set $\mathcal{P}$ of paths exist from a variable w in the computational graph to output o.
  - Local gradient of node with variable $y(j)$ with respect to variable $y(i)$ for directed edge (i, j) is $z(i, j) = \frac{\partial y(j)}{\partial y(i)}$

- The value of $\frac{\partial o}{\partial w}$ is given by computing the product of the local gradients along each path in $\mathcal{P}$, and summing these products over all paths.

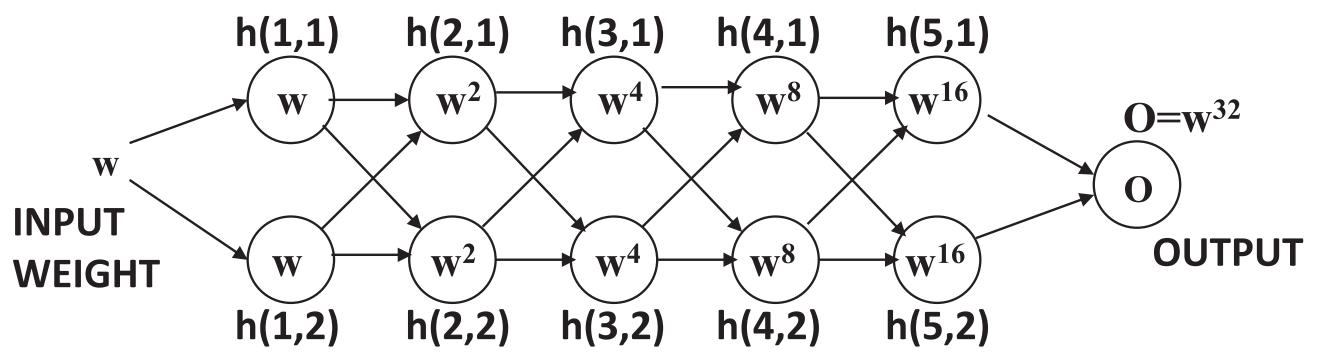
$$\frac{\partial o}{\partial w} = \sum_{P \in \mathcal{P}} \prod_{(i,j) \in P} z(i, j) \tag{3}$$

- Observation: Each $z(i, j)$ easy to compute.

An Exponential Time Algorithm for Computing Partial Derivatives

- The path aggregation lemma provides a simple way to compute the derivative with respect to intermediate variable w
  - Use computational graph to compute each value $y(i)$ of nodes i in a forward phase.
  - Compute local derivative $z(i, j) = \frac{\partial y(j)}{\partial y(i)}$ on each edge (i, j) in the network.
  - Identify the set $\mathcal{P}$ of all paths from the node with variable w to the output o.
  - For each path $P \in \mathcal{P}$ compute the product $M(P) = \prod_{(i,j)\in P} z(i, j)$ of the local derivatives on that path.
  - Add up these values over all paths $P \in \mathcal{P}$.

Example: Deep Computational Graph with Product Nodes



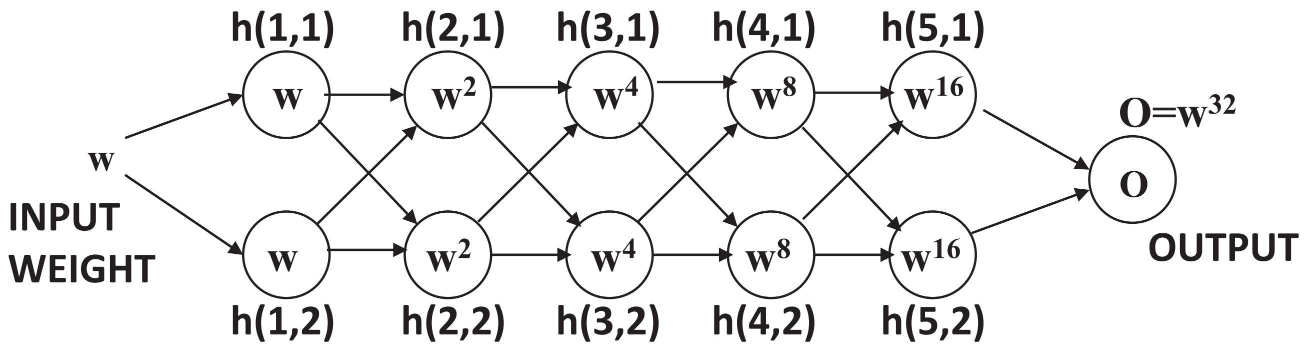
EACH NODE COMPUTES THE PRODUCT OF ITS INPUTS

- Each node computes product of its inputs $\Rightarrow$ Partial derivative of xy with respect to one input x is the other input y.

- Computing product of partial derivatives along a path is equivalent to computing product of values along the only other node disjoint path.

- Aggregative product of partial derivatives (only in this case) equals aggregating products of values.

Example of Increasing Complexity with Depth



EACH NODE COMPUTES THE PRODUCT OF ITS INPUTS

$$\frac{\partial O}{\partial w} = \sum_{j_1,j_2,j_3,j_4,j_5 \in \{1,2\}^5} \underbrace{\prod h(1,j_1)}_{w} \underbrace{h(2,j_2)}_{w^2} \underbrace{h(3,j_3)}_{w^4} \underbrace{h(4,j_4)}_{w^8} \underbrace{h(5,j_5)}_{w^{16}}$$

$$= \sum_{\text{All 32 paths}} w^{31} = 32w^{31}$$

- Impractical with increasing depth.

Observations on Exponential Time Algorithm

- Not very practical approach $\Rightarrow$ Million paths for a network with 100 nodes in each layer and three layers.

- *This is the approach of traditional machine learning with complex objective functions in closed form*.
  - For a composition function in closed form, manual differentiation explicitly traverses all paths with chain rule.
  - The algebraic expression of the derivative of a complex function might not fit the paper you write on.
  - Explains why most of traditional machine learning is a shallow neural model.

- The beautiful *dynamic programming* idea of backpropagation rescues us from complexity.

Charu C. Aggarwal
IBM T J Watson Research Center
Yorktown Heights, NY

Backpropagation II: Using Dynamic Programming [Backpropagation] to Compute Derivatives in Polynomial Time

Neural Networks and Deep Learning, Springer, 2018
Chapter 3, Section 3.2

Differentiating Composition Functions

- Neural networks compute composition functions with a lot of *repetitiveness* caused by a node appearing in multiple paths.

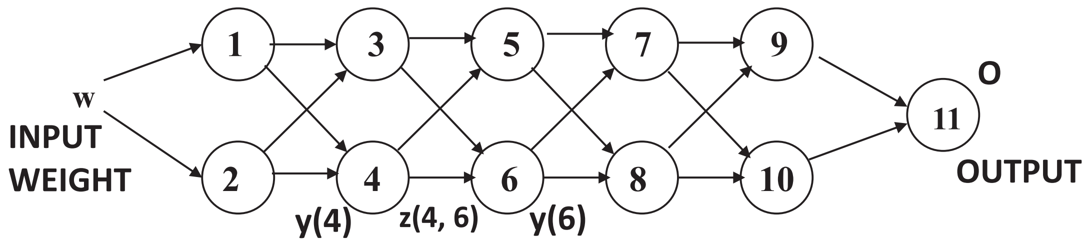
- The most natural and intuitive way to differentiate such a composition function is not the most *efficient* way to do it.

- Natural approach: Top down

$$f(w) = \sin(w^2) + \cos(w^2)$$

- We should not have to differentiate w^2 twice!

- Dynamic programming collapses repetitive computations to reduce exponential complexity into polynomial complexity!



EACH NODE i CONTAINS y(i) AND EACH EDGE BETWEEN i AND j CONTAINS z(i, j)
EXAMPLE: z(4, 6)= PARTIAL DERIVATIVE OF y(6) WITH RESPECT TO y(4)

- We want to compute the derivative of the output with respect to variable w.

- We can easily compute $z(i, j) = \frac{\partial y(j)}{\partial y(i)}$.

- Naive approach computes $S(w, o) = \frac{\partial o}{\partial w} = \sum_{P \in \mathcal{P}} \prod_{(i,j) \in P} z(i, j)$ by explicit aggregation over all paths in $\mathcal{P}$.

Dynamic Programming and Directed Acyclic Graphs

- Dynamic programming used extensively in directed acyclic graphs.
  - **Typical:** Exponentially aggregative path-centric functions between source-sink pairs.
  - **Example:** Polynomial solution to longest path problem in directed acyclic graphs (NP-hard in general).
  - **General approach:** Starts at either the source or sink and *recursively* computes the relevant function over paths of increasing length by reusing intermediate computations.
- Our path-centric function: $S(w, o) = \sum_{P \in \mathcal{P}} \prod_{(i,j) \in P} z(i, j)$.
  - Backwards direction makes more sense here because we have to compute derivative of output (sink) with respect to all variables in early layers.

Dynamic Programming Update

- Let $A(i)$ be the set of nodes at the ends of outgoing edges from node i.

- Let $S(i, o)$ be the *intermediate* variable indicating the same path aggregative function from i to o.

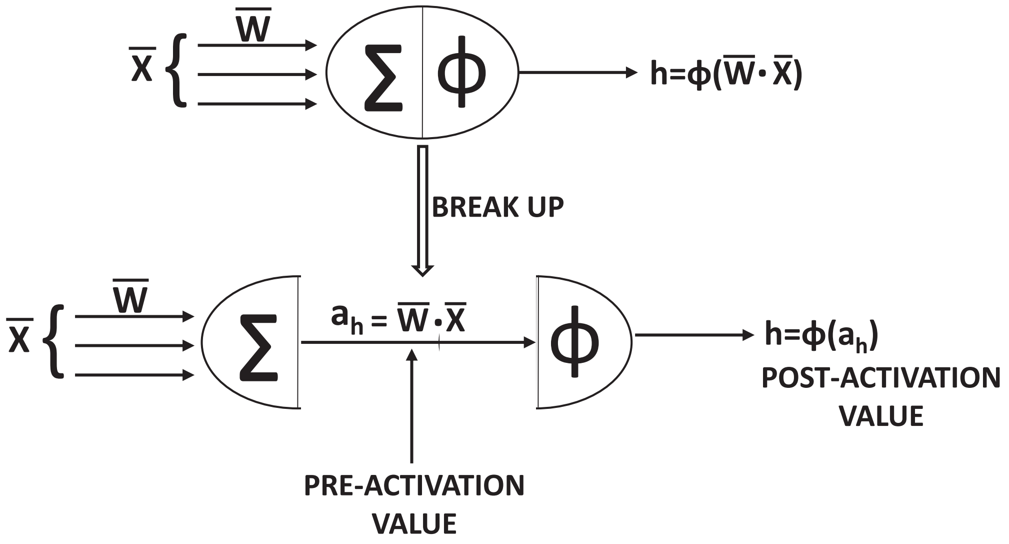
$$S(i, o) \Leftarrow \sum_{j \in A(i)} S(j, o) \cdot z(i, j) \tag{4}$$

- Initialize $S(o, o)$ to 1 and compute backwards to reach $S(w, o)$.
  - Intermediate computations like $S(i, o)$ are also useful for computing derivatives in other layers.

- Do you recognize the multivariate chain rule in Equation 4?

$$\frac{\partial o}{\partial y(i)} = \sum_{j \in A(i)} \frac{\partial o}{\partial y(j)} \cdot \frac{\partial y(j)}{\partial y(i)}$$

How Does it Apply to Neural Networks?



- A neural network is a special case of a computational graph.
  - We can define the computational graph in multiple ways.
  - Pre-activation variables or post-activation variables or both as the node variables of the computation graph?
  - The three lead to different updates but the end result is equivalent.

Pre-Activation Variables to Create Computational Graph

- Compute derivative $\delta(i,o)$ of loss L at o with respect to pre-activation variable at node i.

- We always compute loss derivatives $\delta(i,o)$ with respect to activations in *nodes* during dynamic programming rather than *weights*.
  - Loss derivative with respect to weight w_{ij} from node i to node j is given by the product of $\delta(j,o)$ and hidden variable at i (why?)

- Key points: $z(i,j) = w_{ij} \cdot \Phi'_i$, Initialize $S(o,o) = \delta(o,o) = \frac{\partial L}{\partial o}\Phi'_o$

$$\delta(i,o) = S(i,o) = \Phi'_i \sum_{j \in A(i)} w_{ij} S(j,o) = \Phi'_i \sum_{j \in A(i)} w_{ij}\delta(j,o) \tag{5}$$

Post-Activation Variables to Create Computation Graph

- The variables in the computation graph are hidden values *after* activation function application.

- Compute derivative $\Delta(i, o)$ of loss L at o with respect to post-activation variable at node i.

- Key points: $z(i, j) = w_{ij} \cdot \Phi'_j$, Initialize $S(o, o) = \Delta(o, o) = \frac{\partial L}{\partial o}$

$$\Delta(i, o) = S(i, o) = \sum_{j \in A(i)} w_{ij} S(j, o) \Phi'_j = \sum_{j \in A(i)} w_{ij} \Delta(j, o) \Phi'_j \tag{6}$$

  – Compare with pre-activation approach $\delta(i, o) = \Phi'_i \sum_{j \in A(i)} w_{ij} \delta(j, o)$

  – Pre-activation approach more common in textbooks.

Variables for Both Pre-Activation and Post-Activation Values

- Nice way of decoupling the linear multiplication and activation operations.

- Simplified approach in which each layer is treated as a single node with a vector variable.

  - Update can be computed in vector and matrix multiplications.

- Topic of discussion in next part of the backpropagation series.

Losses at Arbitrary Nodes

- We assume that the loss is incurred at a single output node.

- In case of multiple output nodes, one only has to add up the contributions of different outputs in the backwards phase.

- In some cases, penalties may be applied to hidden nodes.

- For a hidden node i, we add an "initialization value" to $S(i, o)$ just after it has been computed during dynamic programming, which is based on its penalty.

  – Similar treatment as the initialization of an output node, except that we *add* the contribution to existing value of $S(i, o)$.

Handling Shared Weights

- You saw an example in autoencoders where encoder and decoder weights are shared.

- Also happens in specialized architectures like recurrent or convolutional neural networks.

- Can be addressed with a simple application of the chain rule.

- Let $w_1 \dots w_r$ be r copies of the same weight w in the neural network.

$$\frac{\partial L}{\partial w} = \sum_{i=1}^{r} \frac{\partial L}{\partial w_i} \cdot \frac{\partial w_i}{\partial w} = \sum_{i=1}^{r} \frac{\partial L}{\partial w_i} \tag{7}$$

- Pretend all weights are different and just add!

Charu C. Aggarwal
IBM T J Watson Research Center
Yorktown Heights, NY

Backpropagation III: A Decoupled View of Vector-Centric Backpropagation

Neural Networks and Deep Learning, Springer, 2018
Chapter 3, Section 3.2

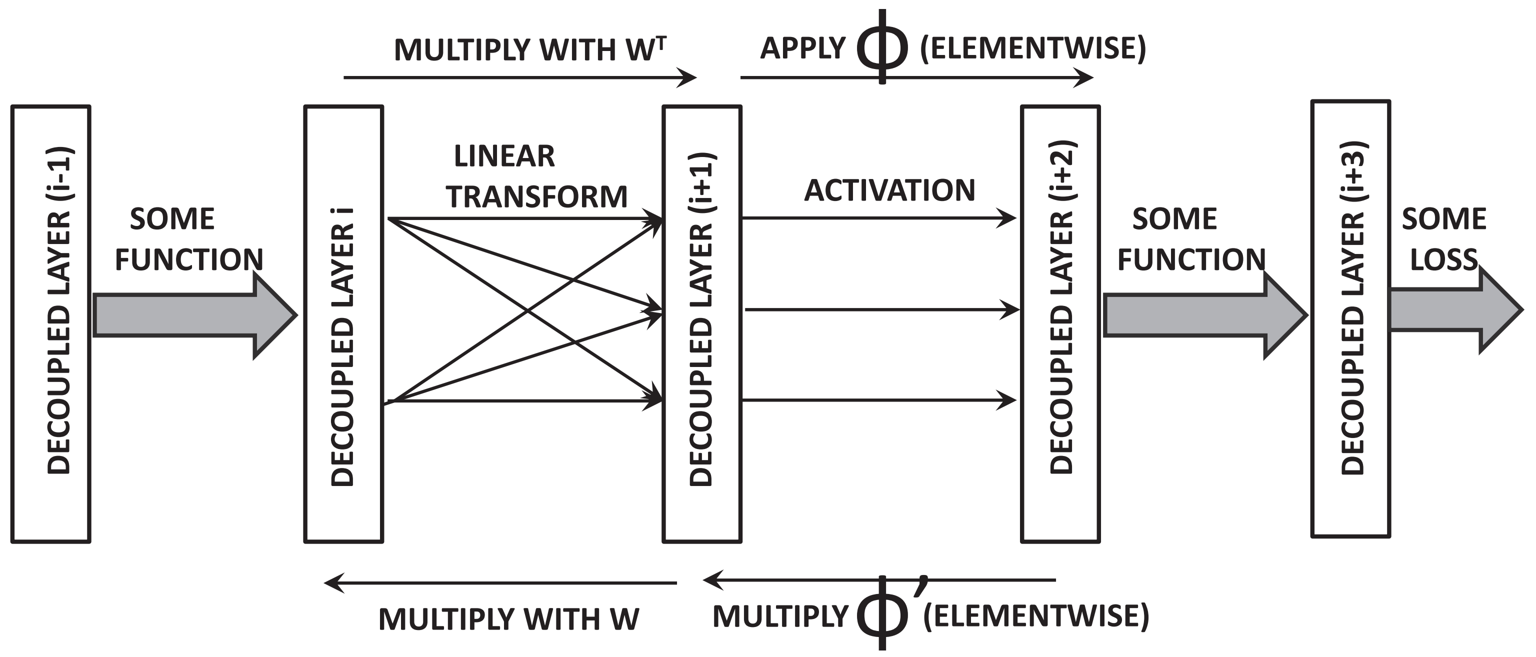
Multiple Computational Graphs from Same Neural Network

- We can create a computational graph in multiple ways from the variables in a neural network.
  - Computational graph of pre-activation variables (part II of lecture)
  - Computational graph of post-activation variables (part II of lecture)
  - Computational graph of both (this part of the lecture)
- Using both pre-activation and post-activation variables creates decoupled backpropagation updates for linear layer and for activation function.

Scalar Versus Vector Computational Graphs

- The backpropagation discussion so far uses scalar operations.
- Neural networks are constructed in layer-wise fashion.
- We can treat an entire layer as a node with a vector variable.
- We want to use layer-wise operations on vectors.
  - Most real implementations use vector and matrix multiplications.
- Want to decouple the operations of linear matrix multiplication and activation function in separate "layers."

Vector-Centric and Decoupled View of Single Layer



- Note that linear matrix multiplication and activation function are separate layers.

- Method 1 (requires knowledge of matrix calculus): You can use the vector-to-vector chain rule to backpropagate on a single path!

Converting Scalar Updates to Vector Form

- **Recap:** When the partial derivative of node q with respect to node p is $z(p,q)$, the dynamic programming update is:

$$S(p,o) = \sum_{q \in \text{Next Layer}} S(q,o) \cdot z(p,q) \tag{8}$$

- We can write the above update in vector form by creating a single column vector $\overline{g}_i$ for layer $i \Rightarrow$ Contains $S(p,o)$ for all values of p.

$$\overline{g}_i = Z\overline{g}_{i+1} \tag{9}$$

- The matrix $Z = [z(p,q)]$ is the transpose of the Jacobian!
  - We will use the notation $J = Z^T$ in further slides.

The Jacobian

- Consider layer i and layer-$(i+1)$ with activations $\overline{z}_i$ and $\overline{z}_{i+1}$.
  - The kth activation in layer-$(i+1)$ is obtained by applying an arbitrary function $f_k(\cdot)$ on the vector of activations in layer-i.

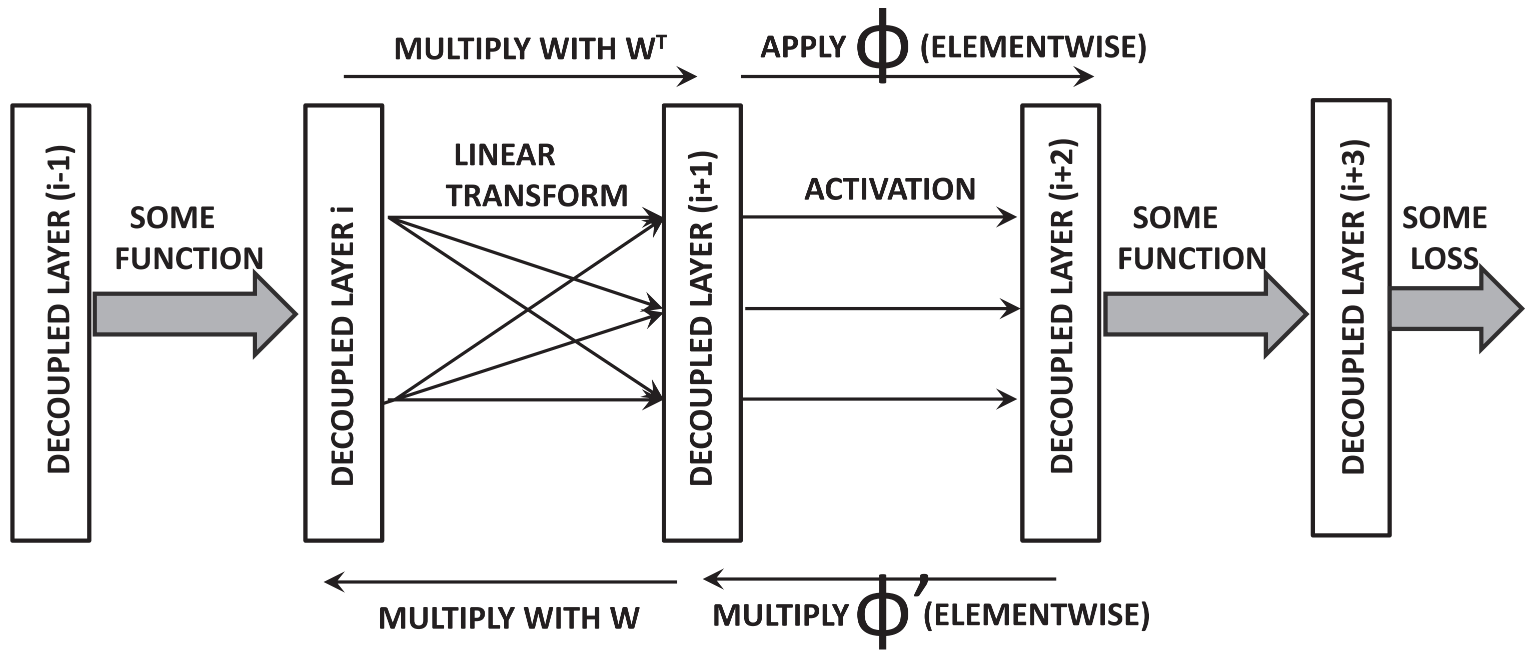
- Definition of Jacobian matrix entries:

$$J_{kr} = \frac{\partial f_k(\overline{z}_i)}{\partial \overline{z}_i^{(r)}} \tag{10}$$

- Backpropagation updates:

$$\overline{g}_i = J^T \overline{g}_{i+1} \tag{11}$$

Effect on Linear Layer and Activation Functions



- Backpropagation is multiplication with transposed weight matrix for linear layer.

- Elementwise multiplication with derivative for activation layer.

Table of Forward Propagation and Backward Propagation

Function	Forward	Backward
Linear	$\overline{z}_{i+1} = W^T \overline{z}_i$	$\overline{g}_i = W \overline{g}_{i+1}$
Sigmoid	$\overline{z}_{i+1} =$sigmoid$(\overline{z}_i)$	$\overline{g}_i = \overline{g}_{i+1} \odot \overline{z}_{i+1} \odot (1 - \overline{z}_{i+1})$
Tanh	$\overline{z}_{i+1} =$tanh$(\overline{z}_i)$	$\overline{g}_i = \overline{g}_{i+1} \odot (1 - \overline{z}_{i+1} \odot \overline{z}_{i+1})$
ReLU	$\overline{z}_{i+1} = \overline{z}_i \odot I(\overline{z}_i > 0)$	$\overline{g}_i = \overline{g}_{i+1} \odot I(\overline{z}_i > 0)$
Hard Tanh	Set to ± 1 ($\notin [-1, +1]$) Copy ($\in [-1, +1]$)	Set to 0 ($\notin [-1, +1]$) Copy ($\in [-1, +1]$)
Max	Maximum of inputs	Set to 0 (non-maximal inputs) Copy (maximal input)
Arbitrary function $f_k(\cdot)$	$\overline{z}_{i+1}^{(k)} = f_k(\overline{z}_i)$	$\overline{g}_i = J^T \overline{g}_{i+1}$ J is Jacobian (Equation 10)

- Two types of Jacobians: Linear layers are dense and activation layers are sparse.

- Maximization function used in max-pooling.

Charu C. Aggarwal
IBM T J Watson Research Center
Yorktown Heights, NY

Neural Network Training [Initialization, Preprocessing, Mini-Batching, Tuning, and Other Black Art]

Neural Networks and Deep Learning, Springer, 2018
Chapter 3, Section 3.3

How to Check Correctness of Backpropagation

- Consider a particular weight w of a randomly selected edge in the network.

- Let $L(w)$ be the current value of the loss.

- The weight of this edge is perturbed by adding a small amount $\epsilon > 0$ to it.

- Estimate of derivative:
$$\frac{\partial L(w)}{\partial w} \approx \frac{L(w + \epsilon) - L(w)}{\epsilon} \tag{12}$$

- When the partial derivatives do not match closely enough, it might be indicative of an incorrectness in implementation.

What Does "Closely Enough" Mean?

- Algorithm-determined derivative is G_e and the approximate derivative is G_a.

$$\rho = \frac{|G_e - G_a|}{|G_e + G_a|} \tag{13}$$

- The ratio should be less than 10^{-6}.

- If ReLU is used, the ratio should be less than 10^{-3}.

- Should perform the checks for a sample of the weights a few times during training.

Stochastic Gradient Descent

- We have always worked with *point-wise* loss functions so far.
  - Corresponds to stochastic gradient descent.
  - In practice, stochastic gradient descent is only a randomized approximation of the true loss function.

- True loss function is typically additive over points.
  - **Example:** Sum-of-squared errors in regression.
  - Computing gradient over a single point is like sampled gradient estimate.

Mini-batch Stochastic Gradient Descent

- One can improve accuracy of gradient computation by using a batch of instances.
  - Instead of holding a vector of activations, we hold a matrix of activations in each layer.
  - Matrix-to-matrix multiplications required for forward and backward propagation.
  - Increases the memory requirements.
- Typical sizes are powers of 2 like 32, 64, 128, 256

Why Does Mini-Batching Work?

- At early learning stages, the weight vectors are very poor.
  - Training data is highly redundant in terms of important patterns.
  - Small batch sizes gives the correct direction of gradient.
- At later learning stages, the gradient direction becomes less accurate.
  - But some amount of noise helps avoid overfitting anyway!
- Performance on out-of-sample data does not deteriorate!

Feature Normalization

- **Standardization:** Normalize to zero mean and unit variance.

- **Whitening:** Transform the data to a de-correlated axis system with principal component analysis (mean-centered SVD).

  – Truncate directions with extremely low variance.

  – Standardize the other directions.

- **Basic principle:** Assume that data is generated from Gaussian distribution and give equal importance to all directions.

Weight Initialization

- Initializations are surprisingly important.
  - Poor initializations can lead to bad convergence behavior.
  - Instability across different layers (vanishing and exploding gradients).
- More sophisticated initializations such as pretraining covered in later lecture.
- Even some simple rules in initialization can help in conditioning.

Symmetry Breaking

- Bad idea to initialize weights to the same value.
  - Results in weights being updated in lockstep.
  - Creates redundant features.
- Initializing weights to random values breaks symmetry.
- Average magnitude of the random variables is important for stability.

Sensitivity to Number of Inputs

- More inputs increase output sensitivity to the average weight.
  - Additive effect of multiple inputs: variance linearly increases with number of inputs r.
  - Standard deviation scales with the square-root of number of inputs r.
- Each weight is initialized from Gaussian distribution with standard deviation $\sqrt{1/r}$ ($\sqrt{2/r}$ for ReLU).
- **More sophisticated:** Use standard deviation of $\sqrt{2/(r_{in} + r_{out})}$.

Tuning Hyperparameters

- Hyperparameters represent the parameters like number of layers, nodes per layer, learning rate, and regularization parameter.

- Use separate validation set for tuning.

- Do not use same data set for backpropagation training as tuning.

Grid Search

- Perform grid search over parameter space.
  - Select set of values for each parameter in some "reasonable" range.
  - Test over all combination of values.
- Careful about parameters at borders of selected range.
- **Optimization:** Search over coarse grid first, and then drill down into region of interest with finer grids.

How to Select Values for Each Parameter

- Natural approach is to select uniformly distributed values of parameters.
  - Not the best approach in many cases! $\Rightarrow$ Log-uniform intervals.
  - Search uniformly in reasonable values of log-values and then exponentiate.
  - **Example:** Uniformly sample log-learning rate between -3 and -1, and then raise it to the power of 10.

Sampling versus Grid Search

- With a large number of parameters, grid search is still expensive.

- With 10 parameters, choosing just 3 values for each parameter leads to $3^{10} = 59049$ possibilities.

- Flexible choice is to sample over grid space.

- Used more commonly in large-scale settings with good results.

Large-Scale Settings

- Multiple threads are often run with sampled parameter settings.

- Accuracy tracked on a separate out-of-sample validation set.

- Bad runs are detected and killed after a certain number of epochs.

- New runs may also be started after killing threads (if needed).

- Only a few winners are trained to completion and the predictions combined in an ensemble.

Charu C. Aggarwal
IBM T J Watson Research Center
Yorktown Heights, NY

Gradient Ratios, Vanishing and Exploding Gradient Problems

Neural Networks and Deep Learning, Springer, 2018
Chapter 3, Section 3.4

Effect of Varying Slopes in Gradient Descent

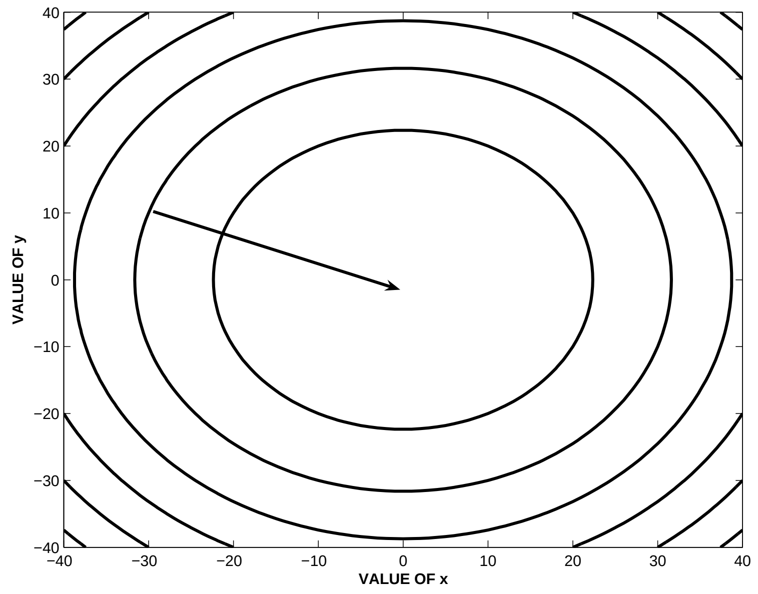
- Neural network learning is a *multivariable* optimization problem.

- Different weights have different magnitudes of partial derivatives.

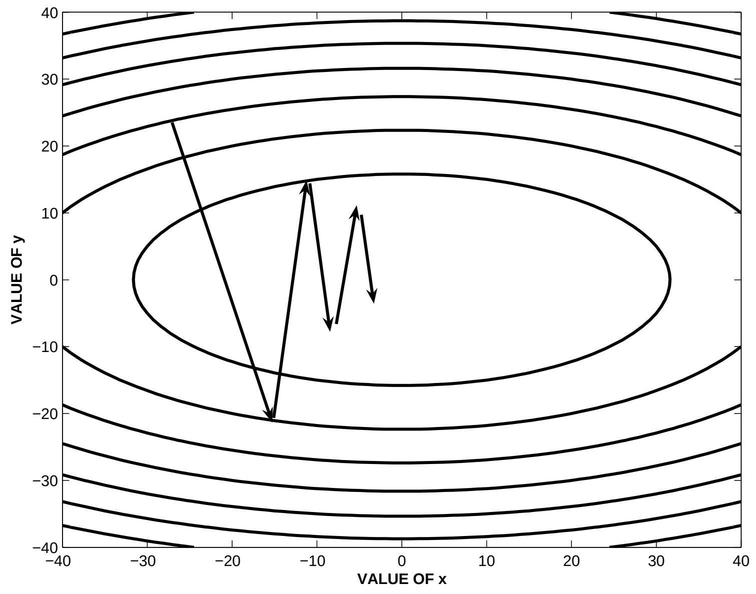
- Widely varying magnitudes of partial derivatives affect the learning.

- Gradient descent works best when the different weights have derivatives of similar magnitude.
  - *The path of steepest descent in most loss functions is only an instantaneous direction of best movement, and is not the correct direction of descent in the longer term.*

Example



(a) Loss function is circular bowl
$L = x^2 + y^2$



(b) Loss function is elliptical bowl
$L = x^2 + 4y^2$

- Loss functions with varying sensitivity to different attributes

Revisiting Feature Normalization

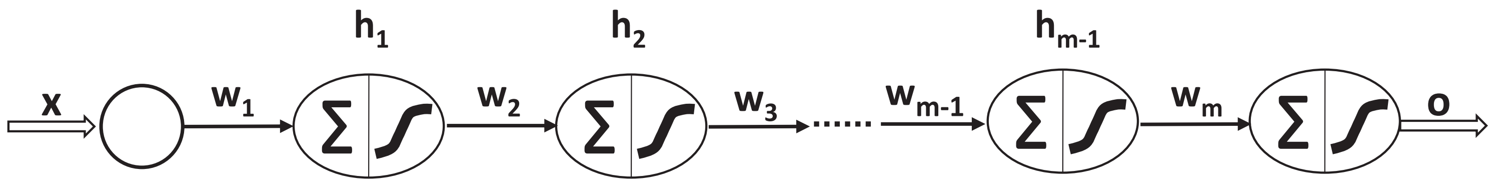
- In the previous lecture, we discussed feature normalization.
- When features have very different magnitudes, gradient ratios of different weights are likely very different.
- Feature normalization helps even out gradient ratios to some extent.
  - Exact behavior depends on target variable and loss function.

The Vanishing and Exploding Gradient Problems

- An extreme manifestation of varying sensitivity occurs in deep networks.

- The weights/activation derivatives in different layers affect the backpropagated gradient in a multiplicative way.
  - With increasing depth this effect is magnified.
  - The partial derivatives can either increase or decrease with depth.

Example



- Neural network with one node per layer.

- Forward propagation multiplicatively depends on each weight and activation function evaluation.

- Backpropagated partial derivative get multiplied by weights and activation function derivatives.

- Unless the values are exactly one, the partial derivatives will either continuously increase (explode) or decrease (vanish).

- Hard to initialize weights exactly right.

Activation Function Propensity to Vanishing Gradients

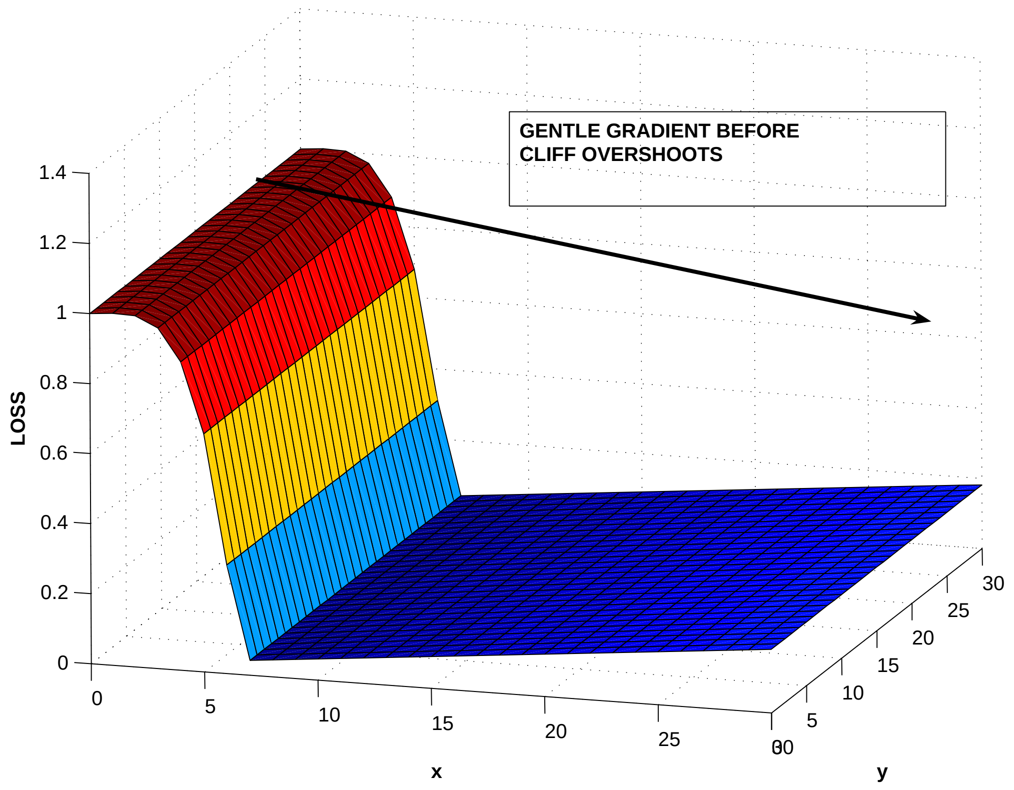
- Partial derivative of sigmoid with output $o \Rightarrow o(1 - o)$.
  - Maximum value at $o = 0.5$ of 0.25.
  - For 10 layers, the activation function alone will multiply by less than $0.25^{10} \approx 10^{-6}$.
- At extremes of output values, the partial derivative is close to 0, which is called *saturation*.
- The tanh activation function with partial derivative $(1 - o^2)$ has a maximum value of 1 at $o = 0$, but saturation will still cause problems.

Exploding Gradients

- Initializing weights to very large values to compensate for the activation functions can cause exploding gradients.

- Exploding gradients can also occur when weights across different layers are shared (e.g., recurrent neural networks).
  - The effect of a finite change in weight is extremely unpredictable across different layers.
  - Small *finite* change changes loss negligibly, but a slightly larger value might change loss drastically.

Cliffs



- Often occurs with the exploding gradient problem.

A Partial Fix to Vanishing Gradients

- The ReLU has linear activation for nonnegative values and otherwise sets outputs to 0.

- The ReLU has a partial derivative of 1 for nonnegative inputs.

- However, it can have a partial derivative of 0 in some cases and never get updated.
  - Neuron is permanently dead!

Leaky ReLU

- For negative inputs, the leaky ReLU can still propagate some gradient backwards.
  - At the reduced rate of $\alpha < 1$ times the learning case for nonnegative inputs:

$$\Phi(v) = \begin{cases} \alpha \cdot v & v \leq 0 \\ v & \text{otherwise} \end{cases} \tag{14}$$

- The value of α is a hyperparameter chosen by the user.

- The gains with the leaky ReLU are not guaranteed.

Maxout

- The activation used is $\max\{\overline{W_1}\cdot\overline{X}, \overline{W_2}\cdot\overline{X}\}$ with two coefficient vectors.

- One can view the maxout as a generalization of the ReLU.
  - The ReLU is obtained by setting one of the coefficient vectors to 0.
  - The leaky ReLU can also be simulated by setting the other coefficient vector to $\overline{W_2} = \alpha\overline{W_1}$.

- Main disadvantage is that it doubles the number of parameters.

Gradient Clipping for Exploding Gradients

- Try to make the different components of the partial derivatives more even.
  - *Value-based clipping:* All partial derivatives outside ranges are set to range boundaries.
  - *Norm-based clipping:* The entire gradient vector is normalized by the L_2-norm of the entire vector.
- One can achieve a better conditioning of the values, so that the updates from mini-batch to mini-batch are roughly similar.
- Prevents an anomalous gradient explosion during the course of training.

Other Comments on Vanishing and Exploding Gradients

- The methods discussed above are only partial fixes.

- Other fixes discussed in later lectures:
  - Stronger initializations with pretraining.
  - Second-order learning methods that make use of second-order derivatives (or *curvature* of the loss function).

Charu C. Aggarwal
IBM T J Watson Research Center
Yorktown Heights, NY

First-Order Gradient Descent Methods

Neural Networks and Deep Learning, Springer, 2018
Chapter 3, Section 3.5

First-Order Descent

- First-order methods work with steepest-descent directions.

- Modifications to basic form of steepest-descent:
  - Need to reduce step sizes with algorithm progression.
  - Need a way of avoiding local optima.
  - Need to address widely varying slopes with respect to different weight parameters.

Learning Rate Decay

- Initial learning rates should be high but reduce over time.

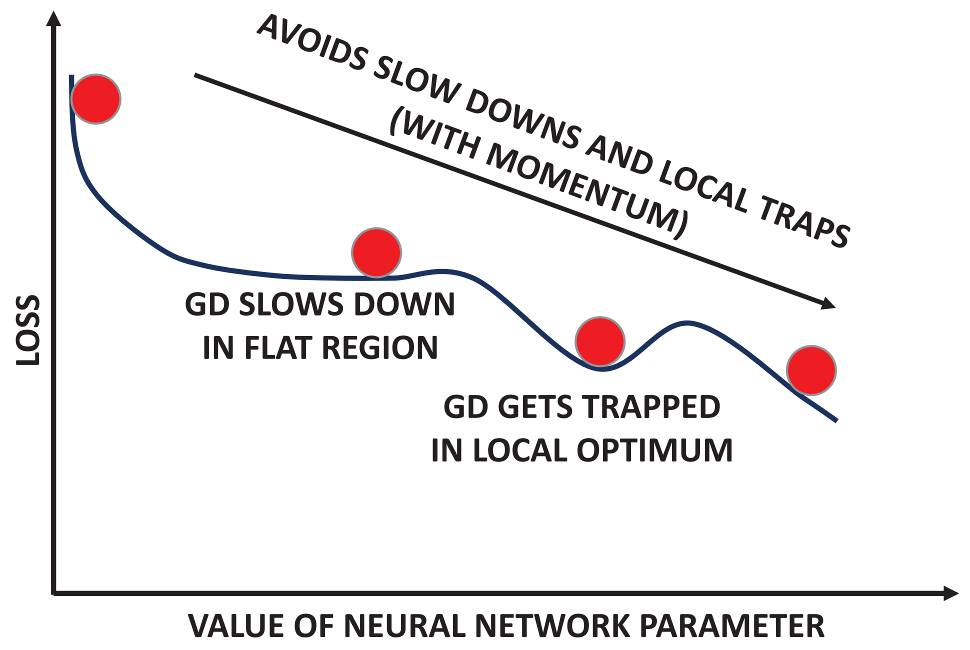
- The two most common decay functions are *exponential decay* and *inverse decay*.

- The learning rate α_t can be expressed in terms of the initial decay rate α_0 and epoch t as follows:

$$\alpha_t = \alpha_0 \exp(-k \cdot t) \quad \text{[Exponential Decay]}$$
$$\alpha_t = \frac{\alpha_0}{1 + k \cdot t} \quad \text{[Inverse Decay]}$$

  The parameter k controls the rate of the decay.

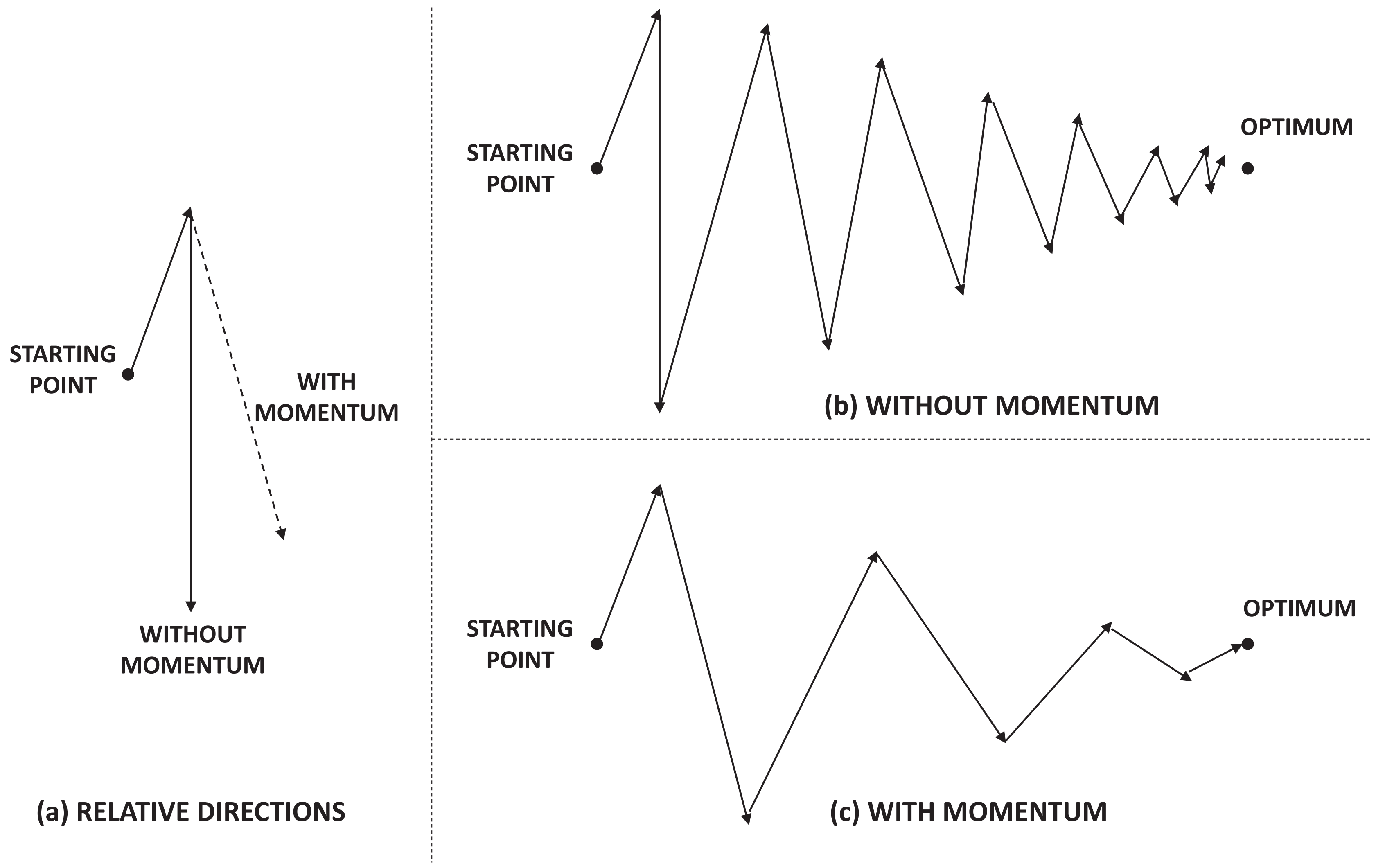
Momentum Methods: Marble Rolling Down Hill



- Use a *friction parameter* $\beta \in (0, 1)$ to gain speed in direction of movement.

$$\overline{V} \Leftarrow \beta \overline{V} - \alpha \frac{\partial L}{\partial \overline{W}}; \quad \overline{W} \Leftarrow \overline{W} + \overline{V}$$

Avoiding Zig-Zagging with Momentum



Nesterov Momentum

- Modification of the traditional momentum method in which *the gradients are computed at a point that would be reached after executing a β-discounted version of the previous step again*.

- Compute at a point reached using only the momentum portion of the current update:

$$\overline{V} \Leftarrow \underbrace{\beta\overline{V}}_{\text{Momentum}} - \alpha\frac{\partial L(\overline{W} + \beta\overline{V})}{\partial\overline{W}}; \quad \overline{W} \Leftarrow \overline{W} + \overline{V}$$

- Put on the brakes as the marble reaches near bottom of hill.

- Nesterov momentum should always be used with mini-batch SGD (rather than SGD).

AdaGrad

- *Aggregate* squared magnitude of ith partial derivative in A_i.

- The square-root of A_i is proportional to the root-mean-square slope.

  – The absolute value will increase over time.

$$A_i \Leftarrow A_i + \left(\frac{\partial L}{\partial w_i}\right)^2 \quad \forall i \tag{15}$$

- The update for the ith parameter w_i is as follows:

$$w_i \Leftarrow w_i - \frac{\alpha}{\sqrt{A_i}}\left(\frac{\partial L}{\partial w_i}\right); \quad \forall i \tag{16}$$

- Use $\sqrt{A_i + \epsilon}$ in the denominator to avoid ill-conditioning.

AdaGrad Intuition

- Scaling the derivative inversely with $\sqrt{A_i}$ encourages faster *relative* movements along gently sloping directions.
  - Absolute movements tend to slow down prematurely.
  - Scaling parameters use stale values.

RMSProp

- The RMSProp algorithm uses *exponential smoothing* with parameter $\rho \in (0, 1)$ in the relative estimations of the gradients.
  - Absolute magnitudes of scaling factors do not grow with time.
  - Problem of staleness is ameliorated.

$$A_i \Leftarrow \rho A_i + (1-\rho)\left(\frac{\partial L}{\partial w_i}\right)^2 \quad \forall i \tag{17}$$

$$w_i \Leftarrow w_i - \frac{\alpha}{\sqrt{A_i}}\left(\frac{\partial L}{\partial w_i}\right); \quad \forall i$$

- Use $\sqrt{A_i + \epsilon}$ to avoid ill-conditioning.

RMSProp with Nesterov Momentum

- Possible to combine RMSProp with Nesterov Momentum

$$v_i \Leftarrow \beta v_i - \frac{\alpha}{\sqrt{A_i}} \left(\frac{\partial L(\overline{W} + \beta \overline{V})}{\partial w_i} \right); \quad w_i \Leftarrow w_i + v_i \quad \forall i$$

- Maintenance of A_i is done with shifted gradients as well.

$$A_i \Leftarrow \rho A_i + (1-\rho) \left(\frac{\partial L(\overline{W} + \beta \overline{V})}{\partial w_i} \right)^2 \quad \forall i \tag{18}$$

AdaDelta and Adam

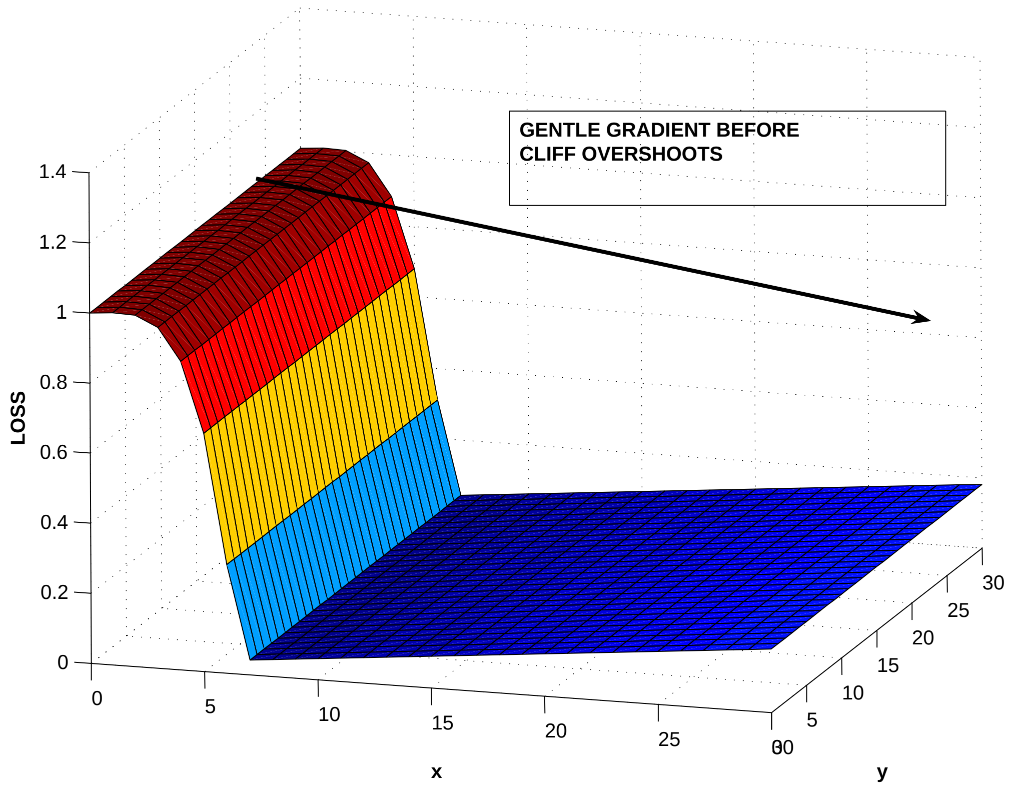
- Both methods derive intuition from RMSProp
  - AdaDelta track of an exponentially smoothed value of the *incremental changes* of weights Δw_i in previous iterations to decide parameter-specific learning rate.
  - Adam keeps track of exponentially smoothed gradients from previous iterations (in addition to normalizing like RMSProp).
- Adam is extremely popular method.

Charu C. Aggarwal
IBM T J Watson Research Center
Yorktown Heights, NY

Second-Order Gradient Descent Methods

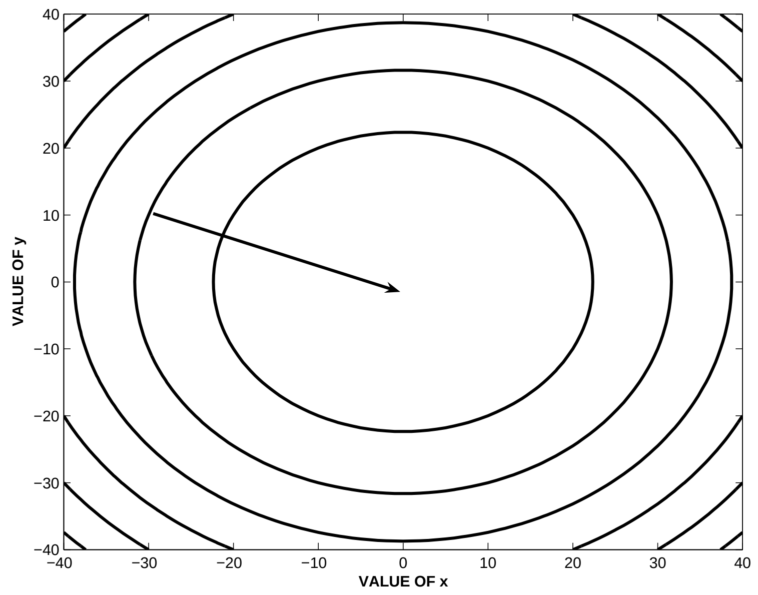
Neural Networks and Deep Learning, Springer, 2018
Chapter 3, Section 3.5.5

Why Second-Order Methods?

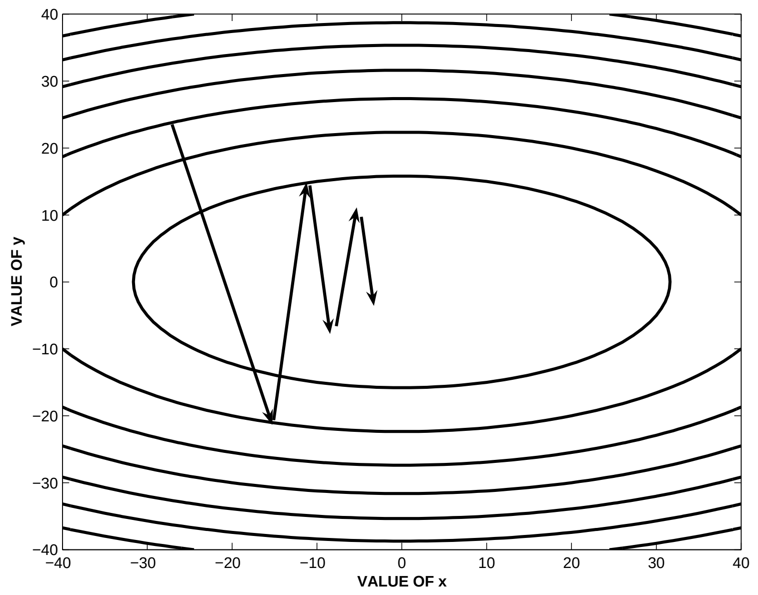


- First-order methods are not enough when there is curvature.

Revisiting the Bowl



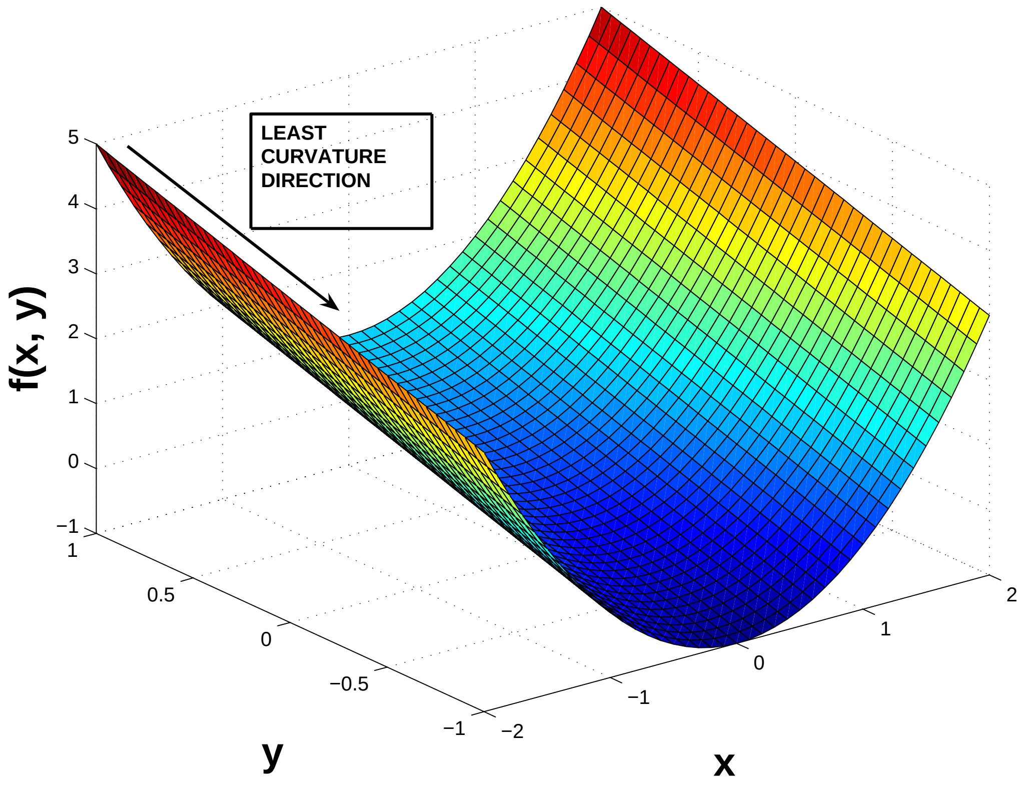
(a) Loss function is circular bowl $L = x^2 + y^2$



(b) Loss function is elliptical bowl $L = x^2 + 4y^2$

- High curvature directions cause bouncing in spite of higher gradient $\Rightarrow$ Need second-derivative for more information.

A Valley



- Gently sloping directions are better with less curvature!

The Hessian

- The second-order derivatives of the loss function $L(\overline{W})$ are of the following form:

$$H_{ij} = \frac{\partial^2 L(\overline{W})}{\partial w_i \partial w_j}$$

- The partial derivatives use all pairwise parameters in the denominator.

- For a neural network with d parameters, we have a $d \times d$ *Hessian matrix* H, for which the (i, j)th entry is H_{ij}.

Quadratic Approximation of Loss Function

- One can write a quadratic approximation of the loss function with Taylor expansion about $\overline{W_0}$:

$$L(\overline{W}) \approx L(\overline{W}_0)+(\overline{W}-\overline{W}_0)^T[\nabla L(\overline{W}_0)]+\frac{1}{2}(\overline{W}-\overline{W}_0)^T H(\overline{W}-\overline{W}_0) \tag{19}$$

- One can derive a single-step optimality condition from initial point $\overline{W_0}$ by setting the gradient to 0.

Newton's Update

- Can solve quadratic approximation in one step from initial point $\overline{W}_0$.

  $\nabla L(\overline{W}) = 0$ [Gradient of Loss Function]
  $\nabla L(\overline{W}_0) + H(\overline{W} - \overline{W}_0) = 0$ [Gradient of Taylor approximation]

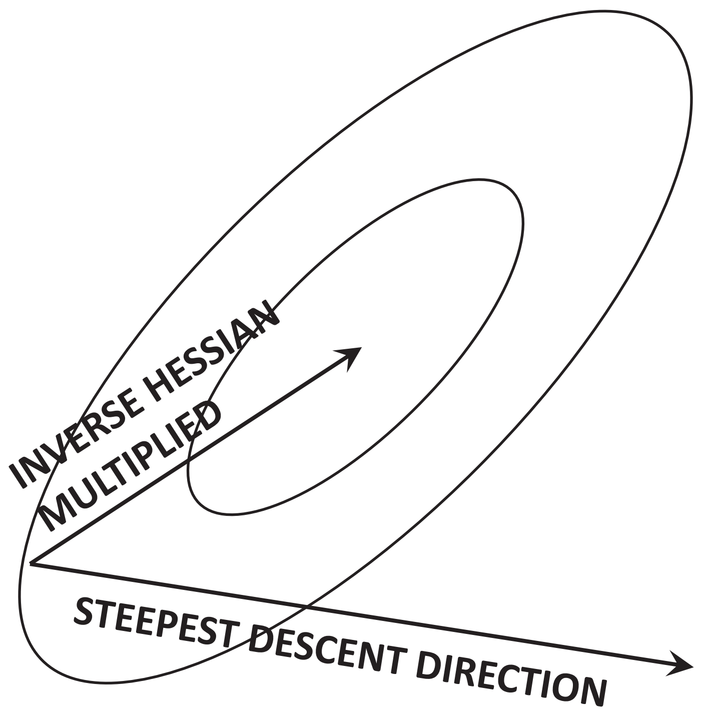
- Rearrange optimality condition to obtain Newton update:

$$\overline{W}^* \Leftarrow \overline{W}_0 - H^{-1}[\nabla L(\overline{W}_0)] \tag{20}$$

- Note the ratio of first-order to second-order $\Rightarrow$ Trade-off between speed and curvature

- Step-size not needed!

Why Second-Order Methods?



- Pre-multiplying with the inverse Hessian finds a trade-off between speed of descent and curvature.

Basic Second-Order Algorithm and Approximations

- Keep making Newton's updates to convergence (single step needed for quadratic function)
  - Even computing the Hessian is difficult!
  - Inverting it is even more difficult
- Solutions:
  - Approximate the Hessian.
  - Find an algorithm that works with projection $H\overline{v}$ for some direction $\overline{v}$.

Conjugate Gradient Method

- Get to optimal in d steps (instead of single Newton step) where d is number of parameters.

- Use optimal step-sizes to get best point along a direction.

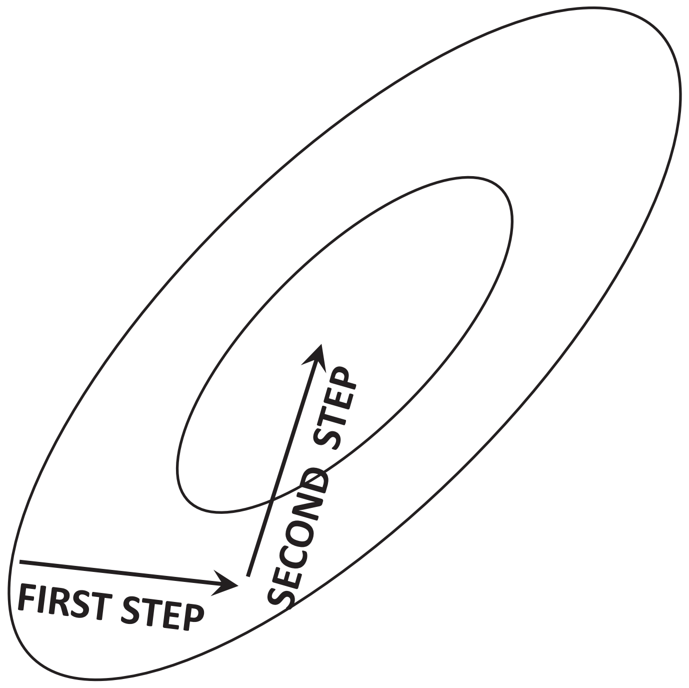
- *Thou shalt not worsen with respect to previous directions!*

- **Conjugate direction:** The gradient of the loss function on *any* point on an update direction is always orthogonal to the previous update directions.

$$\overline{q}_{t+1} = -\nabla L(\overline{W}_{t+1}) + \left(\frac{\overline{q}_t^T H [\nabla L(\overline{W}_{t+1})]}{\overline{q}_t^T H \overline{q}_t} \right) \overline{q}_t \tag{21}$$

- For quadratic function, it requires d updates instead of single update of Newton method.

Conjugate Gradients on 2-Dimensional Quadratic



- Two conjugate directions are required to reach optimality

Conjugate Gradient Algorithm

- For quadratic functions only.
  - Update $\overline{W}_{t+1} \Leftarrow \overline{W}_t + \alpha_t \overline{q}_t$. Here, the step size α_t is computed using line search.
  - Set $\overline{q}_{t+1} = -\nabla L(\overline{W}_{t+1}) + \left(\frac{\overline{q}_t^T H[\nabla L(\overline{W}_{t+1})]}{\overline{q}_t^T H \overline{q}_t}\right) \overline{q}_t$. Increment t by 1.

- For non-quadratic functions approximate loss function with Taylor expansion and perform $\ll d$ of the above steps. Then repeat.

Efficiently Computing Projection of Hessian

- The update requires computation of the *projection* of the Hessian rather than inversion of Hessian.

$$\overline{q}_{t+1} = -\nabla L(\overline{W}_{t+1}) + \left(\frac{\overline{q}_t^T H[\nabla L(\overline{W}_{t+1})]}{\overline{q}_t^T H \overline{q}_t}\right) \overline{q}_t \tag{22}$$

- Easy to perform numerically!

$$H\overline{v} \approx \frac{\nabla L(\overline{W}_0 + \delta \overline{v}) - \nabla L(\overline{W}_0)}{\delta} \tag{23}$$

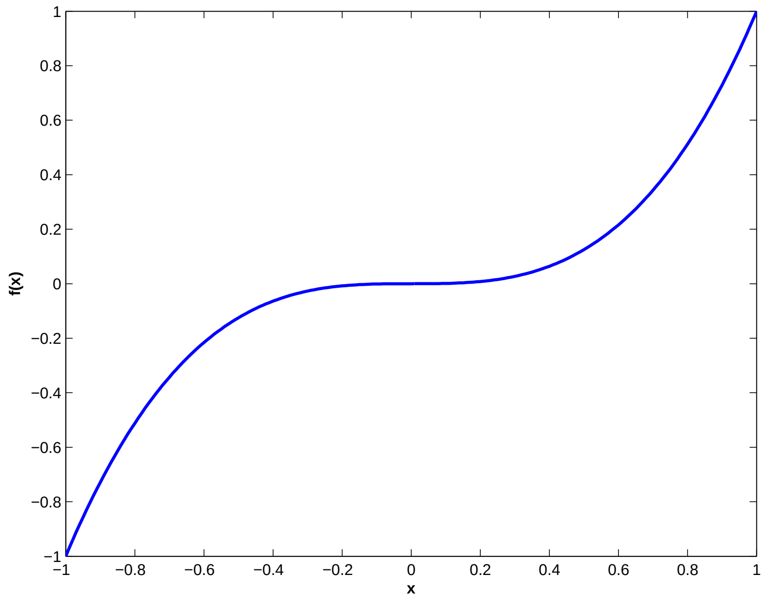
Other Second-Order Methods

- *Quasi-Newton Method:* A sequence of increasingly accurate approximations of the inverse Hessian matrix are used in various steps.

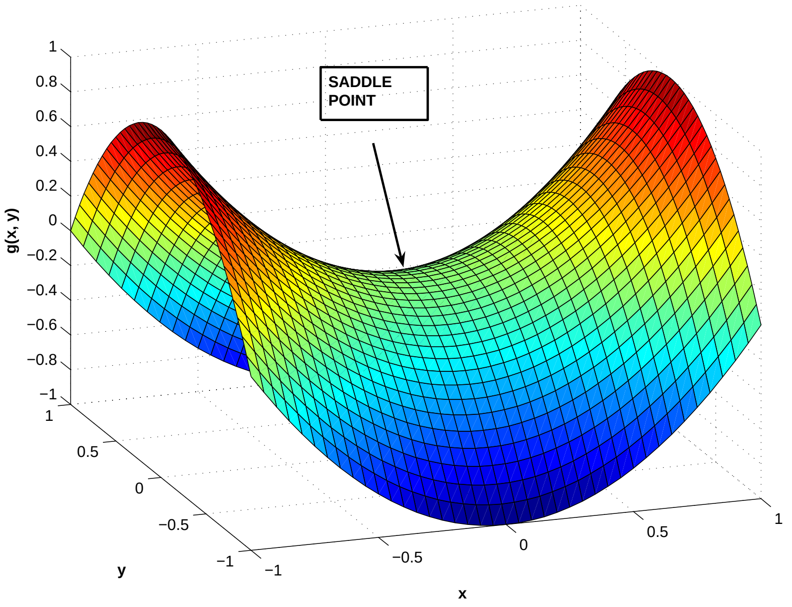
- Many variations of this approach.

- Commonly-used update is BFGS, which stands for the Broyden–Fletcher–Goldfarb–Shanno algorithm and its limited memory variant L-BFGS.

Problems with Second-Order Methods



(a) $f(x) = x^3$
Degenerate



(b) $f(x) = x^2 - y^2$
Stationary

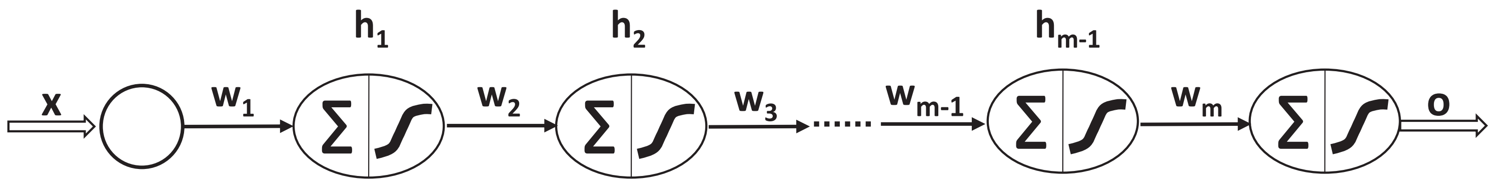
- Saddle points: Whether it is maximum or minimum depends on which direction we approach it from.

Charu C. Aggarwal
IBM T J Watson Research Center
Yorktown Heights, NY

Batch Normalization

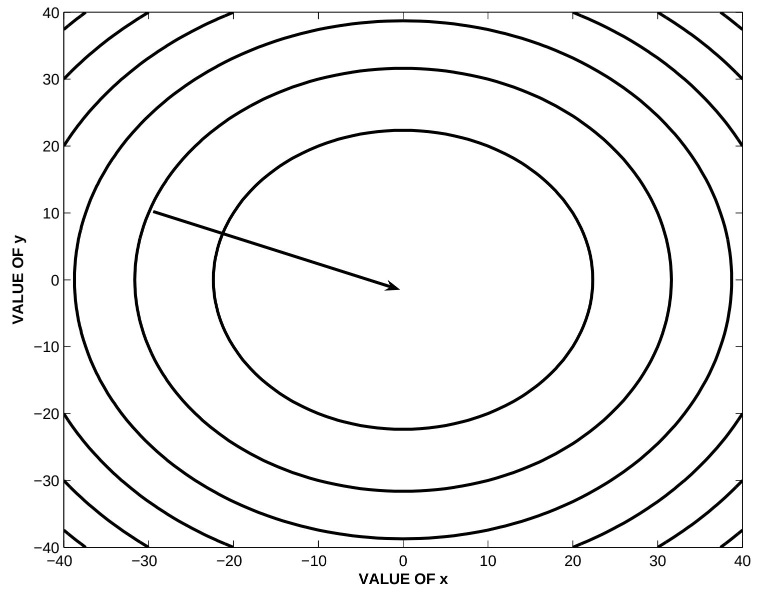
Neural Networks and Deep Learning, Springer, 2018
Chapter 3, Section 3.6

Revisiting the Vanishing and Exploding Gradient Problems

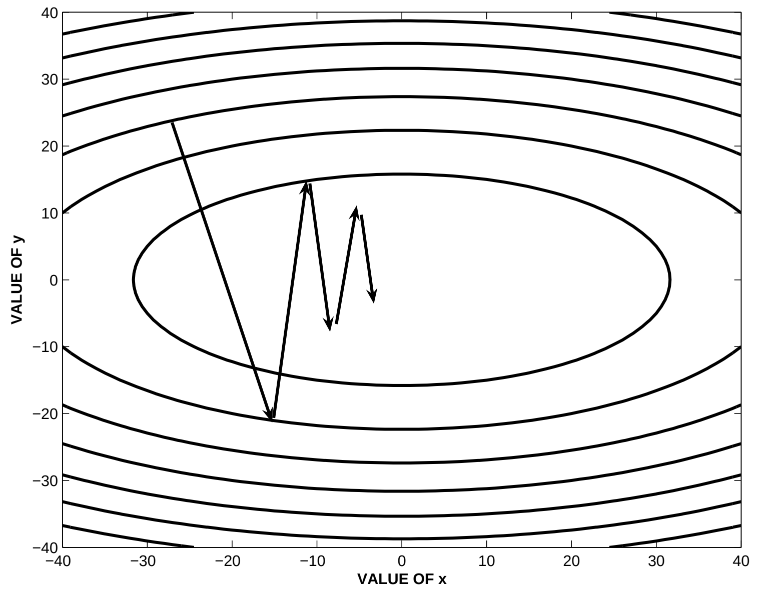


- Neural network with one node per layer.
- Forward propagation multiplicatively depends on each weight and activation function evaluation.
- Backpropagated partial derivative get multiplied by weights and activation function derivatives.
- Unless the values are exactly one, the partial derivatives will either continuously increase (explode) or decrease (vanish).
- Hard to initialize weights exactly right.

Revisiting the Bowl



(a) Loss function is circular bowl $L = x^2 + y^2$



(b) Loss function is elliptical bowl $L = x^2 + 4y^2$

- Varying scale of different parameters will cause bouncing
- Varying scale of features causes varying scale of parameters

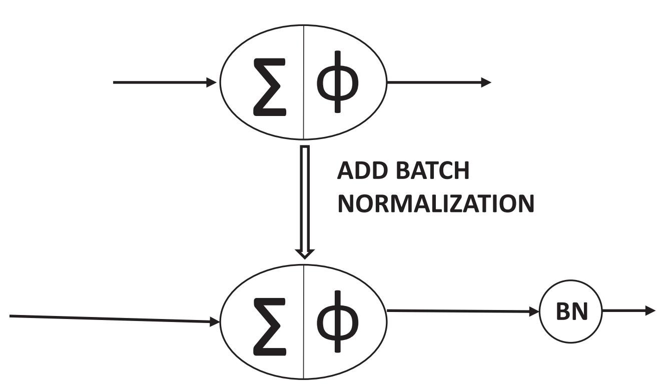
Input Shift

- One can view the input to each layer as a shifting data set of hidden activations during training.

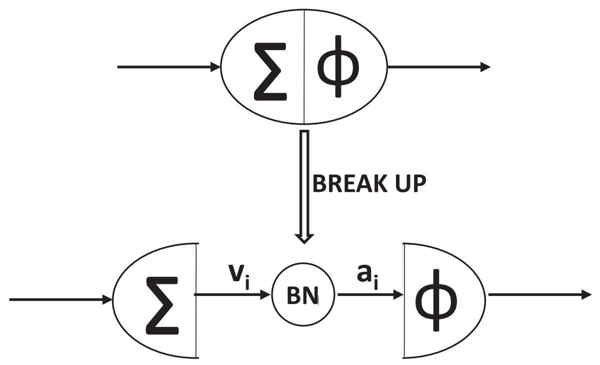
- A shifting input causes problems during learning.
  - Convergence becomes slower.
  - Final result may not generalize well because of unstable inputs.

- Batch normalization ensures (somewhat) more stable inputs to each layer.

Solution: Batch Normalization



(a) Post-activation normalization



(b) Pre-activation normalization

- Add an additional layer than normalizes in *batch-wise* fashion.

- Additional learnable parameters to ensure that optimal level of nonlinearity is used.

- Pre-activation normalization more common than post-activation normalization.

Batch Normalization Node

- The ith unit contains two parameters β_i and γ_i that need to be learned.

- Normalize over *batch* of m instances for ith unit.

$$\mu_i = \frac{\sum_{r=1}^{m} v_i^{(r)}}{m} \quad \forall i \qquad \text{[Batch Mean]}$$

$$\sigma_i^2 = \frac{\sum_{r=1}^{m} (v_i^{(r)} - \mu_i)^2}{m} + \epsilon \quad \forall i \qquad \text{[Batch Variance]}$$

$$\hat{v}_i^{(r)} = \frac{v_i^{(r)} - \mu_i}{\sigma_i} \quad \forall i, r \qquad \text{[Normalize Batch Instances]}$$

$$a_i^{(r)} = \gamma_i \cdot \hat{v}_i^{(r)} + \beta_i \quad \forall i, r \qquad \text{[Scale with Learnable Parameters]}$$

- Why do we need β_i and γ_i?
  - Most activations will be near zero (near-linear regime).

Changes to Backpropagation

- We need to backpropagate through the newly added layer of normalization nodes.
  - The BN node can be treated like any other node.
- We want to optimize the parameters β_i and γ_i.
  - The gradients with respect to these parameters are computed during backpropagation.
- Detailed derivations in book.

Issues in Inference

- The transformation parameters μ_i and σ_i depend on the batch.

- How should one compute them during testing when a *single* test instance is available?

- The values of μ_i and σ_i are computed up front using the *entire* population (of training data), and then treated as constants during testing time.

  – One can also maintain exponentially weighted averages during training.

- The normalization is a simple linear transformation during inference.

Batch Normalization as Regularizer

- Batch normalization also acts as a regularizer.
- Same data point can cause somewhat different updates depending on which batch it is included in.
- One can view this effect as a kind of noise added to the update process.
- Regularization is can be shown to be equivalent to adding a small amount of noise to the training data.
- The regularization is relatively mild.