

# DSC 140B

## Representation Learning

Lecture 15 | Part 1

**Backpropagation**

# Gradient of a Network

- ▶ We want to compute the gradient  $\nabla_{\vec{w}} H$ .
  - ▶ That is,  $\partial H / \partial W_{ij}^{(\ell)}$  and  $\partial H / \partial b_i^{(\ell)}$  for all valid  $i, j, \ell$ .
- ▶ A network is a composition of functions.
- ▶ We'll make good use of the **chain rule**.

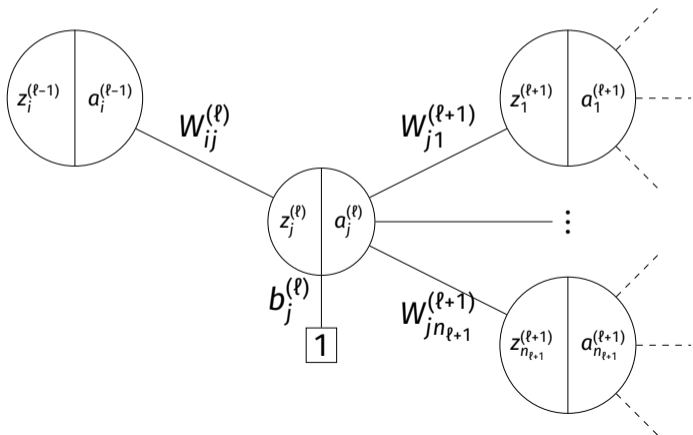
## Recall: The Chain Rule

$$\begin{aligned}\frac{d}{dx}f(g(x)) &= \frac{df}{dg} \frac{dg}{dx} \\ &= f'(g(x))g'(x)\end{aligned}$$

## Some Notation

- ▶ We'll consider an arbitrary node in layer  $\ell$  of a neural network.
- ▶ Let  $g$  be the activation function.
- ▶  $n_\ell$  denotes the number of nodes in layer  $\ell$ .

# Arbitrary Node

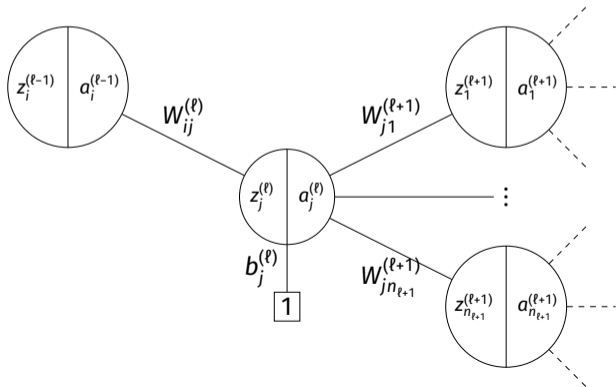


►  $\frac{\partial H}{\partial W_{ij}^{(\ell)}}$  ?

►  $\frac{\partial H}{\partial b_j^{(\ell)}}$  ?

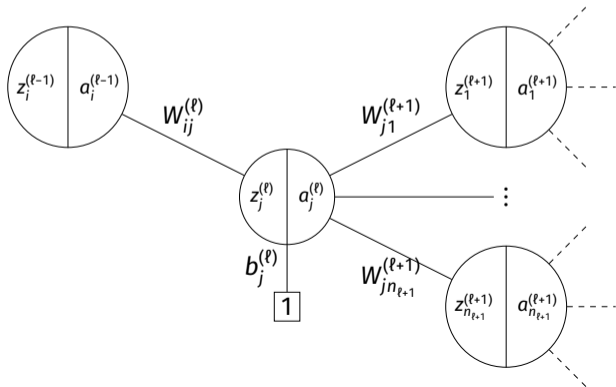
# Claim #1

$$\frac{\partial H}{\partial W_{ij}^{(\ell)}} = \frac{\partial H}{\partial z_j^{(\ell)}} a_i^{(\ell-1)}$$



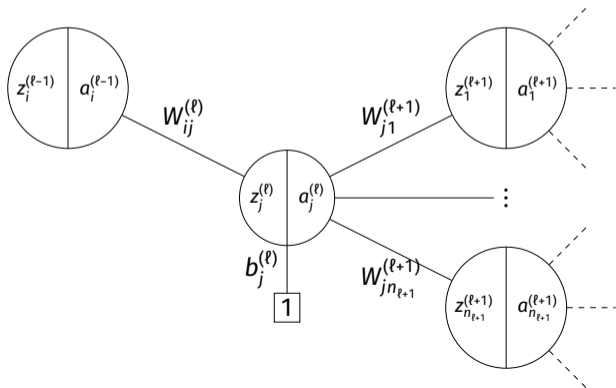
# Claim #2

$$\frac{\partial H}{\partial z_j^{(\ell)}} = \frac{\partial H}{\partial a_j^{(\ell)}} g'(z_j^\ell)$$



# Claim #3

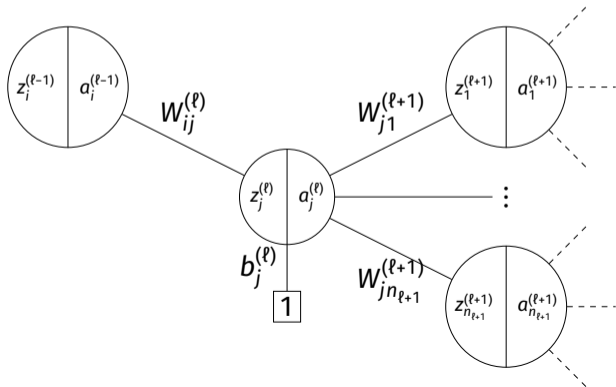
$$\frac{\partial H}{\partial a_j^{(\ell)}} = \sum_{k=1}^{n_{\ell+1}} \frac{\partial H}{\partial z_k^{(\ell+1)}} W_{jk}^{(\ell+1)}$$





## Exercise

What is  $\partial H / \partial b_j^{(\ell)}$ ?



# General Formulas

- ▶ For any node in any neural network<sup>1</sup>, we have the following recursive formulas:

- ▶ 
$$\frac{\partial H}{\partial a_j^{(\ell)}} = \sum_{k=1}^{n_{\ell+1}} \frac{\partial H}{\partial z_k^{(\ell+1)}} W_{jk}^{(\ell+1)}$$

- ▶ 
$$\frac{\partial H}{\partial z_j^{(\ell)}} = \frac{\partial H}{\partial a_j^{(\ell)}} g'(z_j^{(\ell)})$$

- ▶ 
$$\frac{\partial H}{\partial W_{ij}^{(\ell)}} = \frac{\partial H}{\partial z_j^{(\ell)}} a_i^{(\ell-1)}$$

- ▶ 
$$\frac{\partial H}{\partial b_j^{(\ell)}} = \frac{\partial H}{\partial z_j^{(\ell)}}$$

---

<sup>1</sup>Fully-connected, feedforward network

## Main Idea

The derivatives in layer  $\ell$  depend on derivatives in layer  $\ell + 1$ .

# Backpropagation

- ▶ **Idea:** compute the derivatives in last layers, first.
- ▶ That is:
  - ▶ Compute derivatives in last layer,  $\ell$ ; store them.
  - ▶ Use to compute derivatives in layer  $\ell - 1$ .
  - ▶ Use to compute derivatives in layer  $\ell - 2$ .
  - ▶ ...

# Backpropagation

Given an input  $\vec{x}$  and a current parameter vector  $\vec{w}$ :

1. Evaluate the network to compute  $z_j^{(\ell)}$  and  $a_j^{(\ell)}$  for all nodes.

2. For each layer  $\ell$  from last to first:

a) Compute  $\frac{\partial H}{\partial a_j^{(\ell)}} = \sum_{k=1}^{n_{\ell+1}} \frac{\partial H}{\partial z_k^{(\ell+1)}} W_{jk}^{(\ell+1)}$

b) Compute  $\frac{\partial H}{\partial z_j^{(\ell)}} = \frac{\partial H}{\partial a_j^{(\ell)}} g'(z_j^{(\ell)})$

c) Compute  $\frac{\partial H}{\partial W_{ij}^{(\ell)}} = \frac{\partial H}{\partial z_j^{(\ell)}} a_i^{(\ell-1)}$

d) Compute  $\frac{\partial H}{\partial b_j^{(\ell)}} = \frac{\partial H}{\partial z_j^{(\ell)}}$

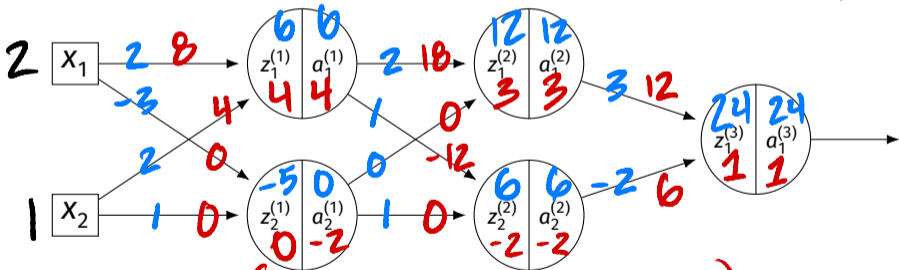
# Example

$$H = a_1^{(3)}$$

$g(z) = \text{linear}$

Compute the entries of the gradient given:

$$W^{(1)} = \begin{pmatrix} 2 & -3 \\ 2 & 1 \end{pmatrix} \quad W^{(2)} = \begin{pmatrix} 2 & 1 \\ 0 & 1 \end{pmatrix} \quad W^{(3)} = \begin{pmatrix} 3 \\ -2 \end{pmatrix} \quad \vec{x} = (2, 1)^T \quad g(z) = \text{ReLU} \text{ for hidden}$$



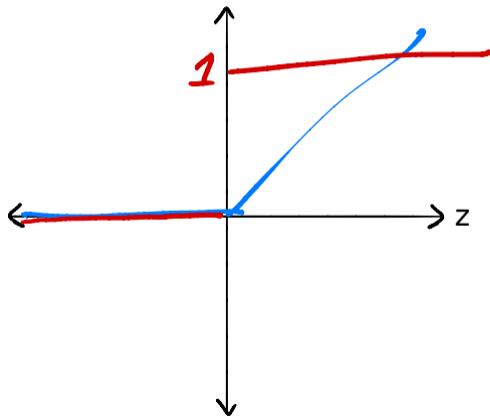
$$\nabla_{\vec{w}} H(\vec{x}; \vec{w}) = (8, 4, 0, 0, 18, 0, -12, \dots)$$

$$\frac{\partial H}{\partial a_j^{(\ell)}} = \sum_{k=1}^{n_{\ell+1}} \frac{\partial H}{\partial z_k^{(\ell+1)}} W_{jk}^{(\ell+1)} \quad \frac{\partial H}{\partial z_j^{(\ell)}} = \frac{\partial H}{\partial a_j^{(\ell)}} g'(z_j^{(\ell)}) \quad \frac{\partial H}{\partial W_{ij}^{(\ell)}} = \frac{\partial H}{\partial z_j^{(\ell)}} a_i^{(\ell-1)}$$

## Aside: Derivative of ReLU

$$g(z) = \max\{0, z\}$$

$$g'(z) = \begin{cases} 0, & z \leq 0 \\ 1, & z > 0 \end{cases}$$



## Summary: Backprop

- ▶ **Backprop** is an algorithm for efficiently computing the gradient of a neural network
- ▶ It is not an algorithm **you** need to carry out by hand: your NN library can do it for you.



# DSC 140B

## Representation Learning

Lecture 15 | Part 2

**Gradient Descent for NN Training**

# Empirical Risk Minimization

0. Collect a training set,  $\{(\vec{x}^{(i)}, y_i)\}$
1. Pick the form of the prediction function,  $H$ .
  - ▶ E.g., a neural network,  $H$ .
2. Pick a loss function.
3. Minimize the empirical risk w.r.t. that loss.

# Minimizing Risk

- ▶ To minimize risk, we often use **vector calculus**.
  - ▶ Either set  $\nabla_{\vec{w}} R(\vec{w}) = 0$  and solve...
  - ▶ Or use gradient descent: walk in opposite direction of  $\nabla_{\vec{w}} R(\vec{w})$ .
  
- ▶ Recall,  $\nabla_{\vec{w}} R(\vec{w}) = (\partial R / \partial w_0, \partial R / \partial w_1, \dots, \partial R / \partial w_d)^T$

# In General

- ▶ Let  $\ell$  be the loss function, let  $H(\vec{x}; \vec{w})$  be the prediction function.
- ▶ The empirical risk:

$$R(\vec{w}) = \frac{1}{n} \sum_{i=1}^n \ell(H(\vec{x}^{(i)}; \vec{w}), y_i)$$

- ▶ Using the chain rule:

$$\nabla_{\vec{w}} R(\vec{w}) = \frac{1}{n} \sum_{i=1}^n \frac{\partial \ell}{\partial H} \nabla_{\vec{w}} H(\vec{x}^{(i)}; \vec{w})$$

$$\vec{w} = (X^T X)^{-1} X^T \vec{y}$$

# Training Neural Networks

- ▶ For neural networks with nonlinear activations, the risk  $R(\vec{w})$  is typically **complicated**.
- ▶ The minimizer cannot be found directly.
- ▶ Instead, we use iterative methods, such as **gradient descent**.

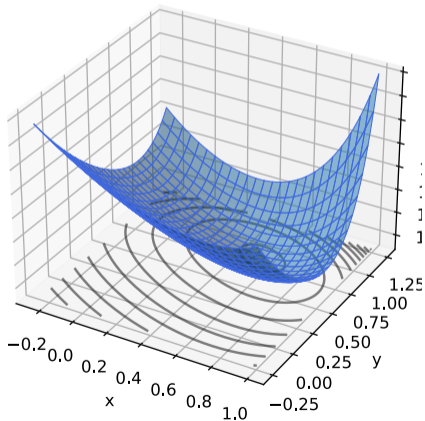
# Iterative Optimization

- ▶ To minimize a function  $f(\vec{x})$ , we may try to compute  $\vec{\nabla} f(\vec{x})$ ; set to 0; solve.
- ▶ Often, there is **no closed-form solution**.
- ▶ How do we minimize  $f$ ?

$$\vec{x} = (x, y)^T$$

# Example

- ▶ Consider  $f(x, y) = e^{x^2+y^2} + (x - 2)^2 + (y - 3)^2$ .



# Example

▶ Try solving  $\vec{\nabla} f(x, y) = 0$ .

▶ The gradient is:

$$\vec{\nabla} f(x, y) = \begin{pmatrix} 2xe^{x^2+y^2} + 2(x-2) \\ 2ye^{x^2+y^2} + 2(y-3) \end{pmatrix}$$

▶ Can we solve the system?

$$2xe^{x^2+y^2} + 2(x-2) = 0$$

$$2ye^{x^2+y^2} + 2(y-3) = 0$$



# Example

- ▶ Try solving  $\vec{\nabla} f(x, y) = 0$ .

- ▶ The gradient is:

$$\vec{\nabla} f(x, y) = \begin{pmatrix} 2xe^{x^2+y^2} + 2(x-2) \\ 2ye^{x^2+y^2} + 2(y-3) \end{pmatrix}$$

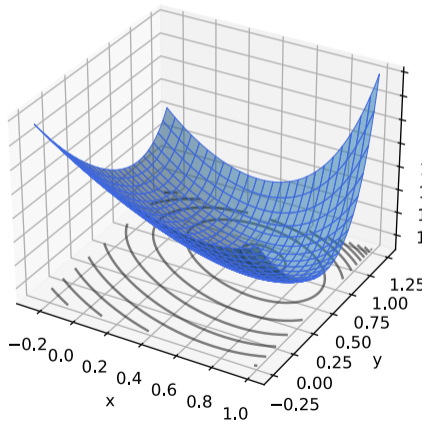
- ▶ Can we solve the system? **Not in closed form.**

$$2xe^{x^2+y^2} + 2(x-2) = 0$$

$$2ye^{x^2+y^2} + 2(y-3) = 0$$

# Idea

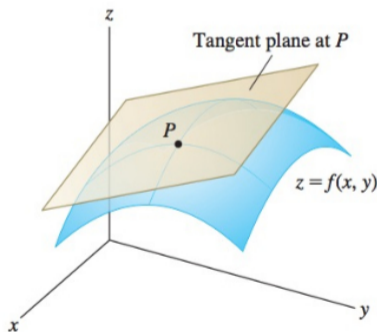
- ▶ Apply an iterative approach.
- ▶ Start at an arbitrary location.
- ▶ “Walk downhill”, towards minimum.



# Which way is down?

- ▶ Consider a differentiable function  $f(x, y)$ .
- ▶ We are standing at  $P = (x_0, y_0)$ .
- ▶ In a small region around  $P$ ,  $f$  looks like a plane.
- ▶ Slope of plane in  $x, y$  directions:

$$\frac{\partial f}{\partial x}(x_0, y_0) \quad \frac{\partial f}{\partial y}(x_0, y_0)$$



# The Gradient

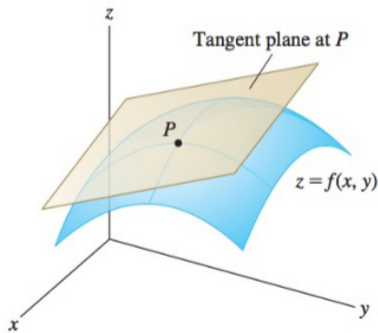
- ▶ Let  $f : \mathbb{R}^d \rightarrow \mathbb{R}$  be differentiable. The **gradient** of  $f$  at  $\vec{x}$  is defined:

$$\vec{\nabla} f(\vec{x}) = \left( \frac{\partial f}{\partial x_1}(\vec{x}), \frac{\partial f}{\partial x_2}(\vec{x}), \dots, \frac{\partial f}{\partial x_d}(\vec{x}) \right)^T$$

- ▶ **Note:**  $\vec{\nabla} f(\vec{x})$  is a **function** mapping  $\mathbb{R}^d \rightarrow \mathbb{R}^d$ .

# Which way is down?

- ▶  $\vec{\nabla}f(x_0, y_0)$  points in direction of steepest **ascent** at  $(x_0, y_0)$ .
- ▶  $-\vec{\nabla}f(x_0, y_0)$  points in direction of steepest **descent** at  $(x_0, y_0)$ .



# Gradient Properties

- ▶ The gradient is used in the linear approximation of  $f$ :

$$f(x_0 + \delta_x, y_0 + \delta_y) \approx f(x_0, y_0) + \vec{\delta} \cdot \vec{\nabla} f(x_0, y_0)$$

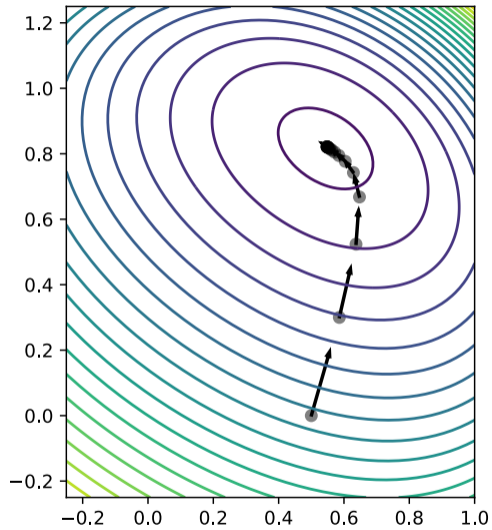
- ▶ Important properties:
  - ▶  $\vec{\nabla} f(\vec{x})$  points in direction of **steepest ascent** at  $\vec{x}$ .
  - ▶  $-\vec{\nabla} f(\vec{x})$  points in direction of **steepest descent** at  $\vec{x}$ .
  - ▶ In directions orthogonal to  $\vec{\nabla} f(\vec{x})$ ,  $f$  does not change!
  - ▶  $\|\vec{\nabla} f(\vec{x})\|$  measures steepness of ascent

# Gradient Descent

- ▶ Pick arbitrary starting point  $\vec{x}^{(0)}$ , **learning rate** parameter  $\eta > 0$ .
- ▶ Until convergence, repeat:
  - ▶ Compute gradient of  $f$  at  $\vec{x}^{(i)}$ ; that is, compute  $\vec{\nabla}f(\vec{x}^{(i)})$ .
  - ▶ Update  $\vec{x}^{(i+1)} = \vec{x}^{(i)} - \eta \vec{\nabla}f(\vec{x}^{(i)})$ .
- ▶ When do we stop?
  - ▶ When difference between  $\vec{x}^{(i)}$  and  $\vec{x}^{(i+1)}$  is negligible.
  - ▶ I.e., when  $\|\vec{x}^{(i)} - \vec{x}^{(i+1)}\|$  is small.

```
def gradient_descent(
    gradient, x, learning_rate=.01,
    threshold=.1e-4
):
    while True:
        x_new = x - learning_rate * gradient(x)
        if np.linalg.norm(x - x_new) < threshold:
            break
        x = x_new
    return x
```





# Backprop Revisited

- ▶ The weights of a neural network can be trained using **gradient descent**.
- ▶ This requires the gradient to be calculated repeatedly; this is where **backprop** enters.
- ▶ Sometimes people use “backprop” to mean “backprop + SGD”, but this is not strictly correct.

# Backprop Revisited $\ell = (H - y)^2$

- ▶ Consider training a NN using the square loss:

$$\begin{aligned}\nabla_{\vec{w}} R(\vec{w}) &= \frac{1}{n} \sum_{i=1}^n \frac{\partial \ell}{\partial H} \nabla_{\vec{w}} H(\vec{x}^{(i)}; \vec{w}) \\ &= \frac{2}{n} \sum_{i=1}^n (H(\vec{x}^{(i)}) - y_i) \nabla_{\vec{w}} H(\vec{x}^{(i)}; \vec{w})\end{aligned}$$

# Backprop Revisited

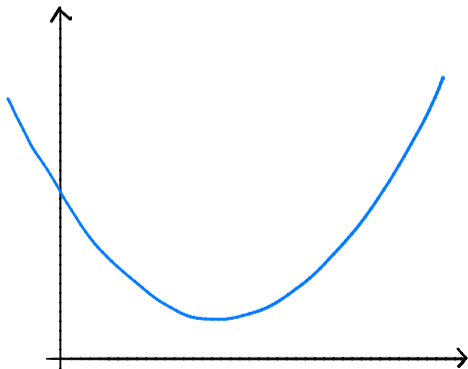
- ▶ Interpretation:

$$\nabla_{\vec{w}} R(\vec{w}) = \frac{2}{n} \sum_{i=1}^n \underbrace{(H(\vec{x}^{(i)}) - y_i)}_{\text{Error}} \underbrace{\nabla_{\vec{w}} H(\vec{x}^{(i)}; \vec{w})}_{\text{Blame}}$$

- ▶ When used in SGD, backprop “propagates error backward” in order to update weights.

# Difficulty of Training NNs

- ▶ Gradient descent is guaranteed to find optimum when objective function is **convex**.<sup>2</sup>

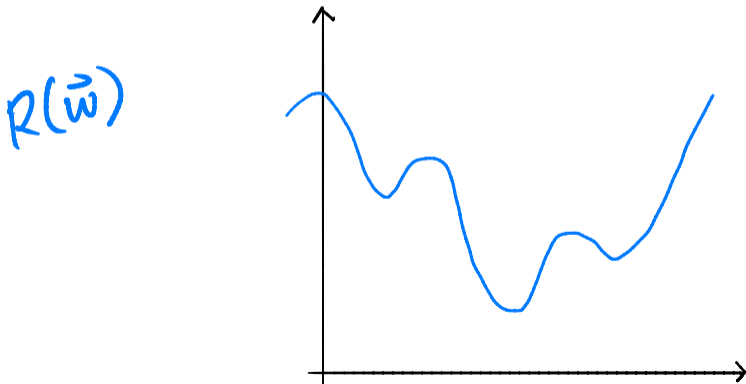


---

<sup>2</sup>Assuming it is properly initialized

# Difficulty of Training NNs

- ▶ When activations are non-linear, neural network risk is **highly non-convex**:



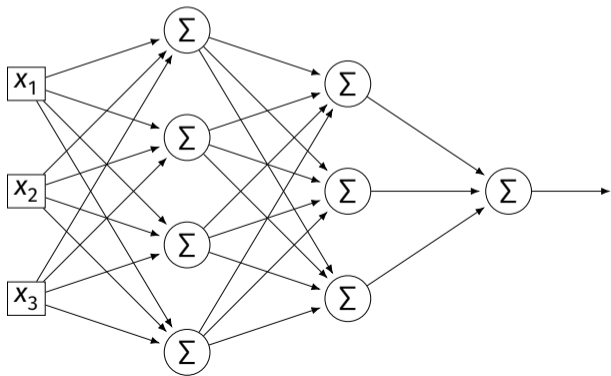
35480  
10000

## Non-Convexity

- ▶ When  $R$  is non-convex, GD can get “stuck” in local minima.
  - ▶ Solution depends on initialization.
- ▶ More sophisticated optimizers, using momentum, adaptation, better initialization, etc.
  - ▶ Adagrad, RMSprop, Adam, etc.

# Difficulty of Training (Deep) NNs

- ▶ Deep networks can suffer from the problem of **vanishing gradients**: if  $w$  is a weight at the “front” of the network,  $\partial H / \partial w$  can be very small



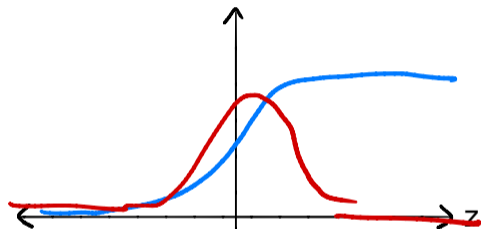


# Vanishing Gradients

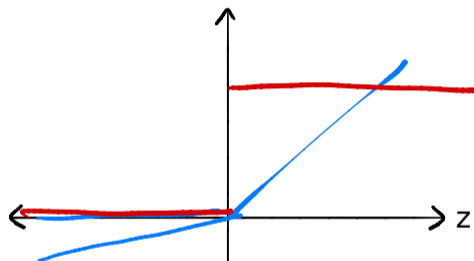
- ▶ If  $\partial H / \partial w$  is always close to zero,  $w$  is updated **very slowly** by gradient descent.
- ▶ In short: early layers are slower to train.
- ▶ One mitigation: use ReLU instead of sigmoid.

# Vanishing Gradients

$g'(z)$



Sigmoid



ReLU

$g'(z)$

# DSC 140B

*Representation Learning*

Lecture 15 | Part 3

**Stochastic Gradient Descent**

# Gradient Descent for Minimizing Risk

- ▶ In ML, we often want to minimize a **risk function**:

$$R(\vec{W}) = \frac{1}{n} \sum_{i=1}^n \ell(H(\vec{X}^{(i)}; \vec{W}), y_i)$$

# Observation

- ▶ The gradient of the risk function is a sum of gradients:

$$\vec{\nabla}R(\vec{w}) = \frac{1}{n} \sum_{i=1}^n \vec{\nabla} \ell(H(\vec{x}^{(i)}; \vec{w}), y_i)$$

- ▶ One term for each point in training data.

# Problem

- ▶ In machine learning, the number of training points  $n$  can be **very large**.
- ▶ Computing the gradient can be **expensive** when  $n$  is large.
- ▶ Therefore, each step of gradient descent can be **expensive**.

# Idea

- ▶ The (full) gradient of the risk uses all of the training data:

$$\nabla R(\vec{w}) = \frac{1}{n} \sum_{i=1}^n \nabla \ell(H(\vec{x}^{(i)}; \vec{w}), y_i)$$

- ▶ It is an average of  $n$  gradients.
- ▶ **Idea:** instead of using all  $n$  points, randomly choose  $\ll n$ .

# Stochastic Gradient

- ▶ Choose a random subset (**mini-batch**)  $B$  of the training data.
- ▶ Compute a **stochastic gradient**:

$$\nabla R(\vec{w}) \approx \sum_{i \in B} \vec{\nabla} \ell(H(\vec{x}^{(i)}; \vec{w}), y_i)$$



# Stochastic Gradient

$$\nabla R(\vec{w}) \approx \sum_{i \in B} \vec{\nabla} \ell(H(\vec{x}^{(i)}; \vec{w}), y_i)$$

- ▶ **Good:** if  $|B| \ll n$ , this is much faster to compute.
- ▶ **Bad:** it is a (random) approximation of the full gradient, noisy.

# Stochastic Gradient Descent (SGD) for ERM

- ▶ Pick arbitrary starting point  $\vec{x}^{(0)}$ , **learning rate** parameter  $\eta > 0$ , batch size  $m \ll n$ .
- ▶ Until convergence, repeat:
  - ▶ Randomly sample a batch  $B$  of  $m$  training data points (on each iteration).
  - ▶ Compute stochastic gradient of  $f$  at  $\vec{x}^{(i)}$ :

$$\vec{g} = \sum_{i \in B} \vec{\nabla} \ell(H(\vec{x}^{(i)}; \vec{w}), y_i)$$

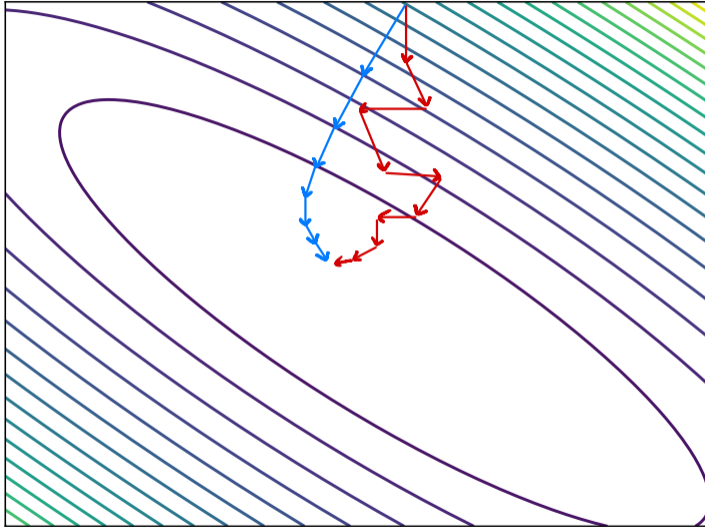
- ▶ Update  $\vec{x}^{(i+1)} = \vec{x}^{(i)} - \eta \vec{g}$

# Idea

- ▶ In practice, a stochastic gradient often works well enough.
- ▶ It is better to take many noisy steps quickly than few exact steps slowly.

# Batch Size

- ▶ Batch size  $m$  is a parameter of the algorithm.
- ▶ The larger  $m$ , the more reliable the stochastic gradient, but the more time it takes to compute.
- ▶ Extreme case when  $m = 1$  will still work.



# Usefulness of SGD

- ▶ SGD allows learning on **massive** data sets.
- ▶ Useful even when exact solutions available.
  - ▶ E.g., least squares regression / classification.

# Training NNs in Practice

- ▶ There are several Python packages for training NNs:
  - ▶ PyTorch
  - ▶ Tensorflow / Keras
- ▶ This week's discussion was a Tensorflow tutorial.