DSC 140B

Representation Learning

Lecture 13 | Part 1

Backpropagation

Gradient of a Network

- To train a network, we need to compute the gradient of the risk.
- In the process, we 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, ℓ.
- Last time, we derived **backpropagation**: an algorithm for doing this efficiently.

Backpropagation

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

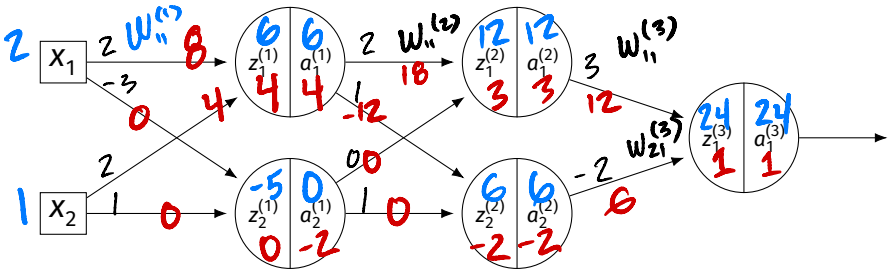
1. Evaluate the network to compute $z_i^{(\ell)}$ and $a_i^{(\ell)}$ for all nodes.
2. For each layer ℓ from last to first:
   - Compute $\frac{\partial H}{\partial a_j^{(\ell)}} = \sum_{k=1}^{n_{\ell+1}} \frac{\partial H}{\partial z_k^{(\ell+1)}} W_{jk}^{(\ell+1)}$
   - Compute $\frac{\partial H}{\partial z_j^{(\ell)}} = \frac{\partial H}{\partial a_j^{(\ell)}} g'(z_j^{\ell})$
   - Compute $\frac{\partial H}{\partial W_{ij}^{(\ell)}} = \frac{\partial H}{\partial z_j^{(\ell)}} a_i^{(\ell-1)}$
   - Compute $\frac{\partial H}{\partial b_j^{(\ell)}} = \frac{\partial H}{\partial z_j^{(\ell)}}$

This computes $\nabla_{\vec{w}} H$ and evaluates it at $\vec{x}$ and $\vec{w}$.

Example

Compute the entries of the gradient given:

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

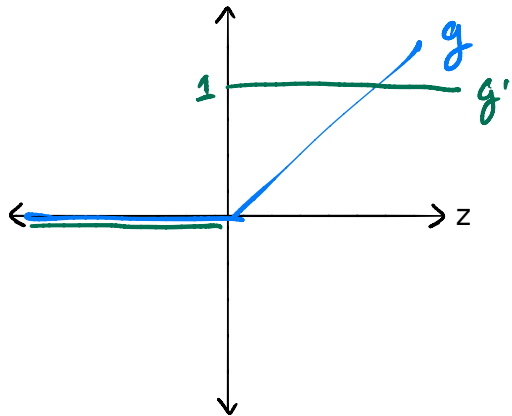


$$\frac{\partial H}{\partial a_j^{(\ell)}} = \sum_{k=1}^{n_{\ell+1}} \frac{\partial H}{\partial z_k^{(\ell+1)}} W_{jk}^{(\ell+1)} \qquad \frac{\partial H}{\partial z_j^{(\ell)}} = \frac{\partial H}{\partial a_j^{(\ell)}} g'(z_j^{\ell}) \qquad \frac{\partial H}{\partial W_{ij}^{(\ell)}} = \frac{\partial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

Aside: Derivative of ReLU

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

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



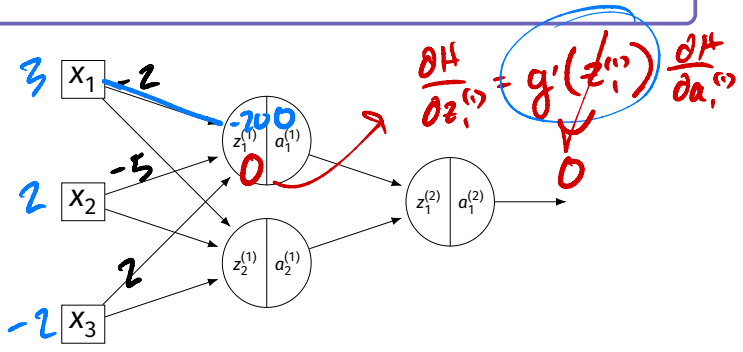
Live Q&A

Exercise

What is $\partial H / \partial W_{11}^{(1)}(\vec{x}, \vec{w})$ for the network in the previous example?

Exercise

Suppose $W_{11}^{(1)} = -2, W_{21}^{(1)} = -5, W_{31}^{(1)} = 2$ and $\vec{x} = (3, 2, -2)^T$ and all biases are 0. ReLU activations are used. What is $\partial H / \partial W_{11}^{(1)}(\vec{x}, \vec{w})$?



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 13 | Part 2

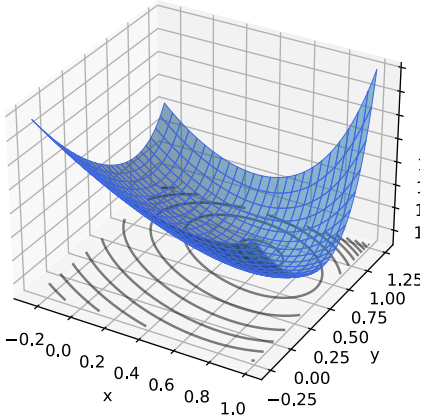
Gradient Descent for NN Training

Training Neural Networks

- We train NNs by minimizing **empirical risk**, $R(\vec{w})$.
- **Idea:** compute $\nabla_{\vec{w}} R(\vec{w})$; set to 0; solve for $\vec{w}$.
- **Problem**: for most NNs, the $R(\vec{w})$ is **complicated.**
  - I.e., we can't solve $\nabla_{\vec{w}} R(\vec{w}) = 0$ in closed form.
- Instead, we use iterative approach: **gradient descent**.

Example

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



Example

- **Goal:** minimize f.
- **First idea:** solve $\vec{\nabla} f(x, y) = 0$.
- The gradient is:

$$\vec{\nabla} f(x, y) = \begin{pmatrix} 2xe^{x^2+y^2} + 2(x-2) \\ 2ye^{x^2+y^2} + 2(y-3) \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$$

- Can we solve the system?

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

Example

- **Goal:** minimize f.
- **First idea:** solve $\vec{\nabla} f(x, y) = 0$.
- The gradient is:

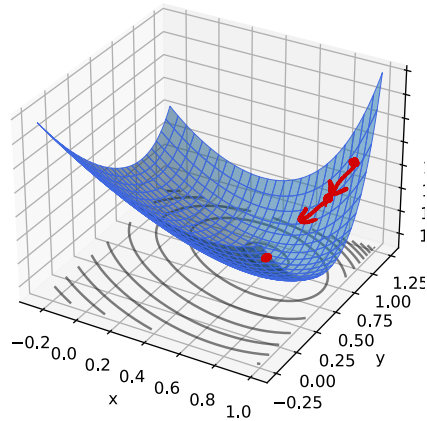
$$\vec{\nabla} f(x, y) = \begin{pmatrix} 2xe^{x^2+y^2} + 2(x-2) \\ 2ye^{x^2+y^2} + 2(y-3) \end{pmatrix}$$

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

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

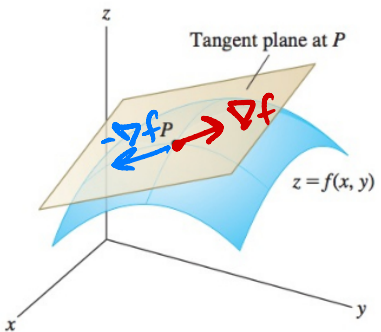
Idea

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



Which way is down?

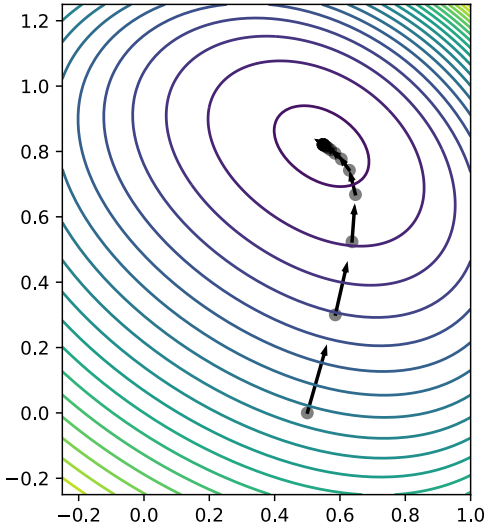
- **Claim:** $\vec{\nabla} f(\vec{x})$ points in direction of **steepest ascent**.
- It follows that $-\vec{\nabla} f(\vec{x})$ points in direction of **steepest descent**.
- $\|\vec{\nabla} f(\vec{x})\|$ measures the rate of change.



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
```


1.2
1.0
0.8
0.6
0.4
0.2
0.0
−0.2
−0.2
0.0
0.2
0.4
0.6
0.8
1.0

Gradient Descent for NN Training

- Now let's use gradient descent to train a NN.

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.

In General

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

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

- We want to minimize it using gradient descent.

Gradient of the Risk

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

Gradient of the Risk

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

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

Gradient of the Risk

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

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

Using the chain rule...

Gradient of the Risk

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

Gradient of the Risk

- In general:

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

- When H is a neural network, we compute $\nabla_{\vec{w}} H$ using **backpropagation**.

- But to move forward, we need to pick loss, ℓ.

Example: Square Loss

- Recall, the square loss is:

$$\ell(H(\vec{x}^{(i)}; \vec{w}), y_i) = (H(\vec{x}^{(i)}; \vec{w}) - y_i)^2$$

- The partial derivative of the loss w.r.t. H is:

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

Example: Square Loss

- If we use square loss, the risk is the **mean squared error**.
- The gradient of the risk is:

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

- We compute it using backpropagation.

Gradient Descent for NNs

- Pick starting weights $\vec{w}^{(0)}$, learning rate parameter $\eta > 0$, loss function ℓ.
- Until convergence, repeat:
  - Compute $\nabla_{\vec{w}} R(\vec{w}^{(0)})$ using backpropagation.
  - Update $\vec{w}^{(i+1)} = \vec{w}^{(i)} - \eta \nabla_{\vec{w}} R(\vec{w}^{(i)})$.
- When do we stop?
  - When difference between $\vec{w}^{(i)}$ and $\vec{w}^{(i+1)}$ is negligible.
  - I.e., when $\|\vec{w}^{(i)} - \vec{w}^{(i+1)}\|$ is small.

Interpretation

- We use **backpropagation** to compute $\nabla_{\vec{w}} H$.
  - And therefore to compute $\nabla_{\vec{w}} R$.
- Recall: entry i of $\nabla_{\vec{w}} H$ measures how much H changes when we change w_i.
  - I.e., how much "influence" w_i has on H.
- Intuition: if prediction is wrong, weights with greatest influence should be changed the most.

Interpretation

- This is essentially what backpropagation does.

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

- Backprop propagates errors “backwards” through the network, assigning blame to each weight.

DSC 140B

Representation Learning

Lecture 13 | 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 is the average of the gradient of the losses:

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

- The averaging is over **all training points**.

- This can take a long time when n is large.[1]

[1] Trivia: this usually takes $\Theta(nd)$ time.

Idea

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

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

- **Idea:** instead of using all n training points, randomly choose a smaller set, B:

$$\vec{\nabla} R(\vec{w}) \approx \frac{1}{|B|} \sum_{i \in B} \vec{\nabla} \ell(H(\vec{x}^{(i)}; \vec{w}), y_i)$$

Stochastic Gradient

- The smaller set B is called a **mini-batch**.
- We now compute a **stochastic gradient**:

$$\vec{\nabla} R(\vec{w}) \approx \frac{1}{|B|} \sum_{i \in B} \vec{\nabla} \ell(H(\vec{x}^{(i)}; \vec{w}), y_i)$$

- "Stochastic," because it is a random.

Stochastic Gradient

$$\vec{\nabla} R(\vec{w}) \approx \frac{1}{|B|} \sum_{i \in B} \vec{\nabla} \ell(H(\vec{x}^{(i)}; \vec{w}), y_i)$$

- The stochastic gradient is an **approximation** of the full gradient.
- When $|B| \ll n$, it is **much faster** to compute.
- But the approximation is **noisy**.

Stochastic Gradient Descent for ERM

To minimize empirical risk $R(\vec{w})$:

- Pick starting weights $\vec{w}^{(0)}$, learning rate $\eta > 0$, batch size m.
- Until convergence, repeat:
  - **Randomly sample** a batch B of m training data points.
  - **Compute stochastic gradient:**

$$\vec{g} = \frac{1}{|B|} \sum_{i \in B} \vec{\nabla} \ell(H(\vec{x}^{(i)}; \vec{w}), y_i)$$

  - **Update:** $\vec{w}^{(t+1)} = \vec{w}^{(t)} - \eta \vec{g}$
- When converged, return $\vec{w}^{(t)}$.

Note

- A **new batch** should be randomly sampled on each iteration!
- This way, the entire training set is used over time.
- Size of batch should be **small** compared to n.
  - Think: $m = 64$, $m = 32$, or even $m = 1$.

Example: Least Squares

- We can use SGD to perform least squares regression.

- Need to compute the gradient of the square loss:

$$\ell_{\text{sq}}(H(\vec{x}^{(i)}; \vec{w}), y_i) = \left(\text{Aug}(\vec{x}^{(i)}) \cdot \vec{w} - y_i\right)^2$$

Example: Least Squares

- The gradient of the square loss of a linear predictor is:

$$\begin{aligned}
&\vec{\nabla} \ell_{\text{sq}}(H(\vec{x}^{(i)}; \vec{w}), y_i) \\
&\qquad = \vec{\nabla} \left(\text{Aug}(\vec{x}^{(i)}) \cdot \vec{w} - y_i \right)^2 \\
&\qquad = 2 \left(\text{Aug}(\vec{x}^{(i)}) \cdot \vec{w} - y_i \right) \vec{\nabla} \left(\text{Aug}(\vec{x}^{(i)}) \cdot \vec{w} - y_i \right) \\
&\qquad = 2 \left(\text{Aug}(\vec{x}^{(i)}) \cdot \vec{w} - y_i \right) \text{Aug}(\vec{x}^{(i)})
\end{aligned}$$

Example: Least Squares

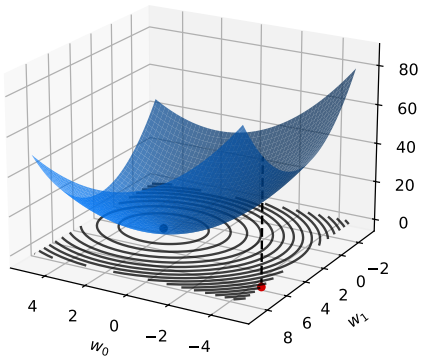
- Therefore, on each step we compute the stochastic gradient:

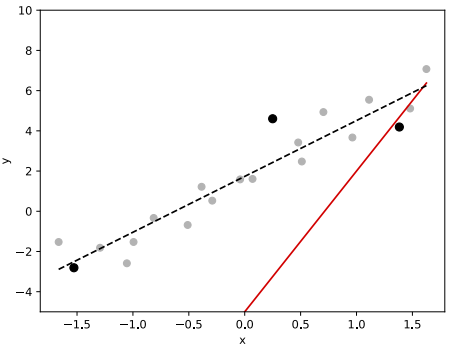
$$\vec{g} = \frac{2}{m} \sum_{i \in B} \left(\text{Aug}(\vec{x}^{(i)}) \cdot \vec{w} - y_i \right) \text{Aug}(\vec{x}^{(i)})$$

- The update rule is:

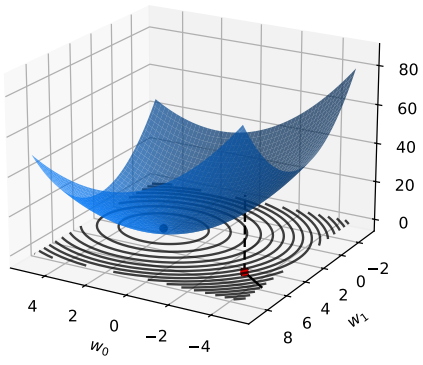
$$\vec{w}^{(t+1)} = \vec{w}^{(t)} - \eta \vec{g}$$

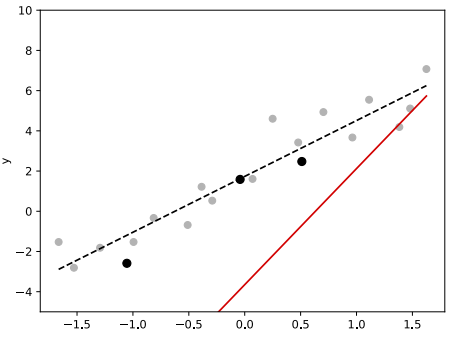
Example: SGD



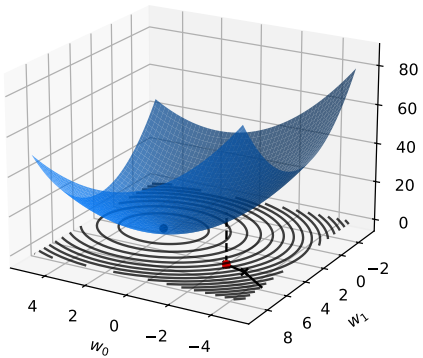


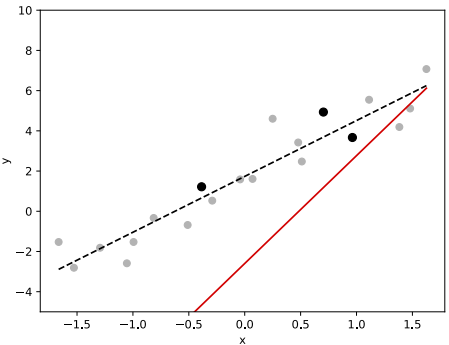
Example: SGD



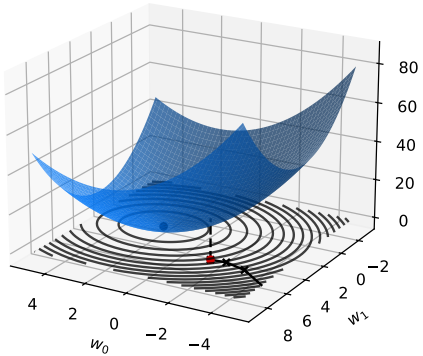


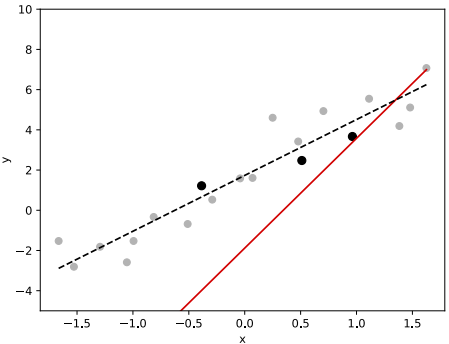
Example: SGD



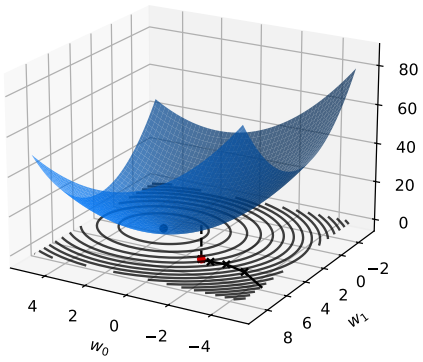


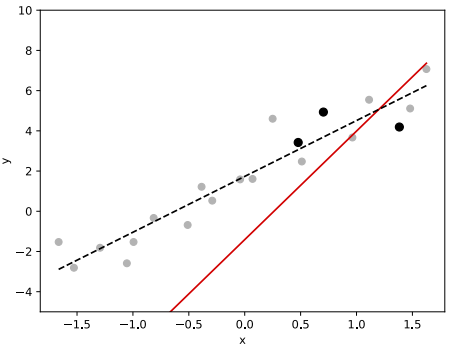
Example: SGD



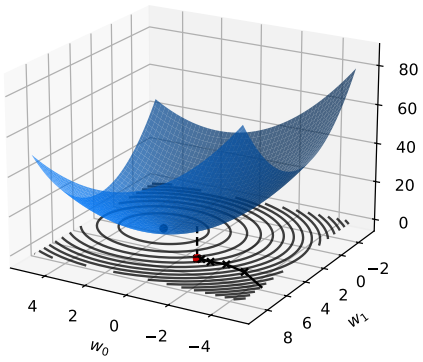


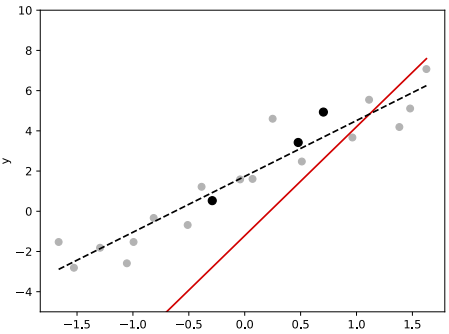
Example: SGD



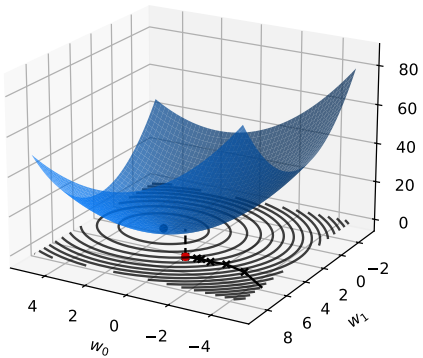


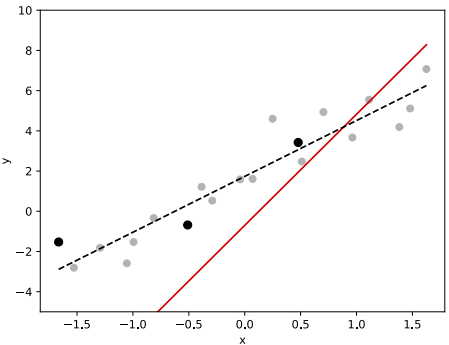
Example: SGD



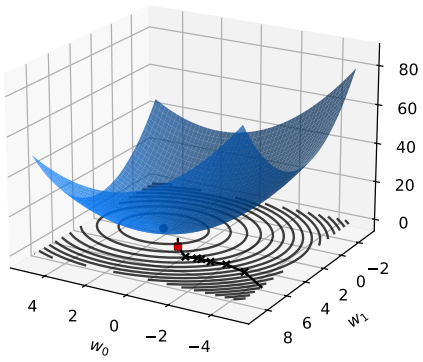


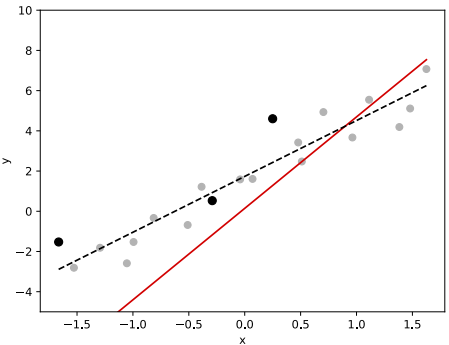
Example: SGD



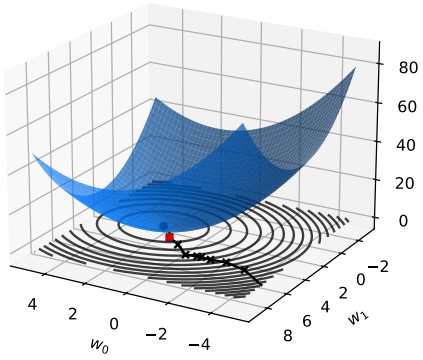


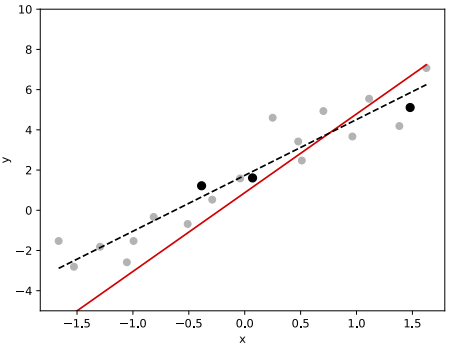
Example: SGD



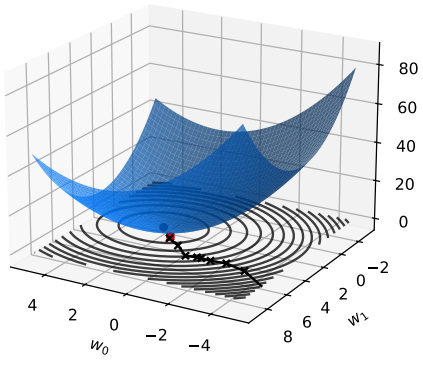


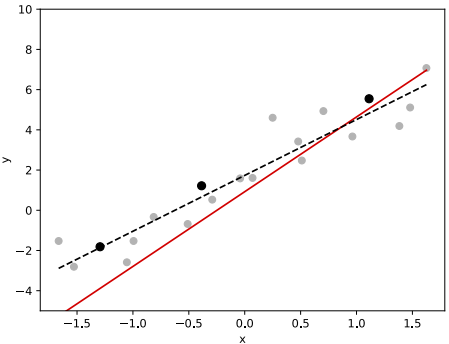
Example: SGD



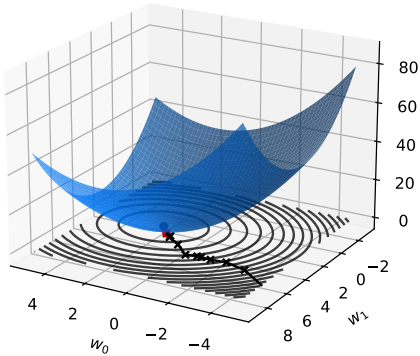


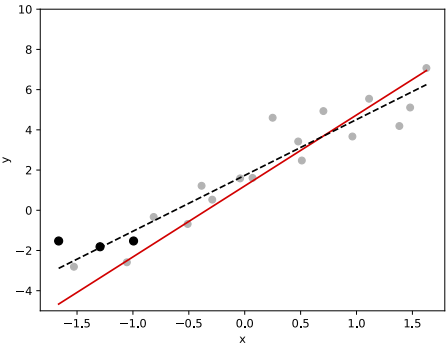
Example: SGD



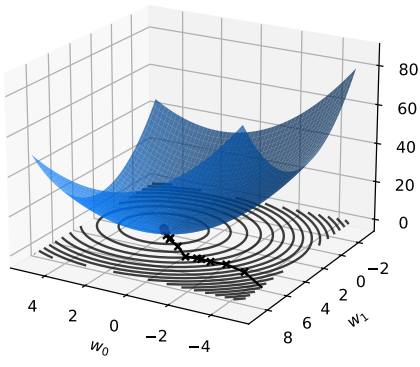


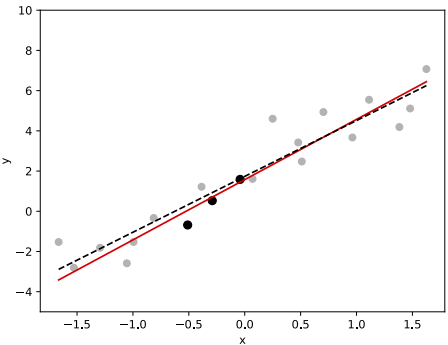
Example: SGD



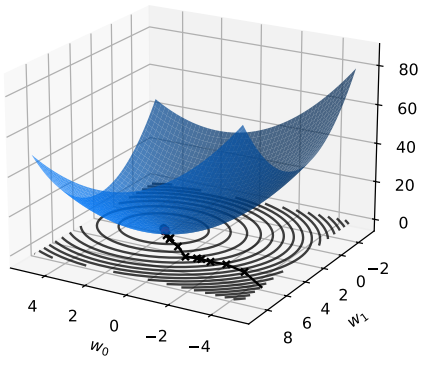


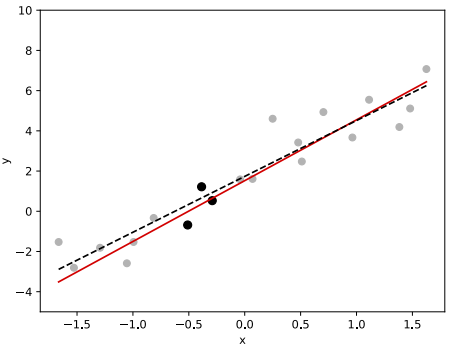
Example: SGD



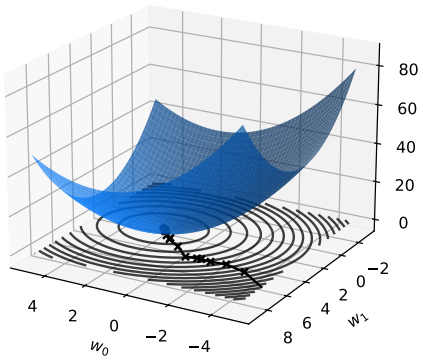


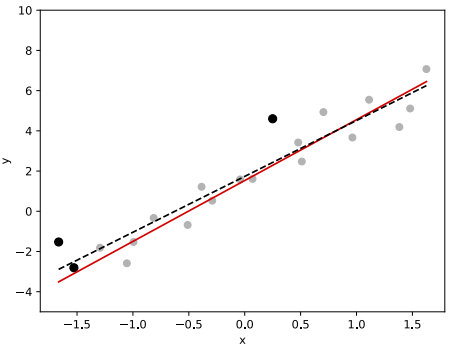
Example: SGD



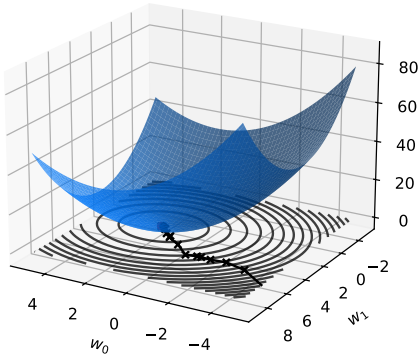


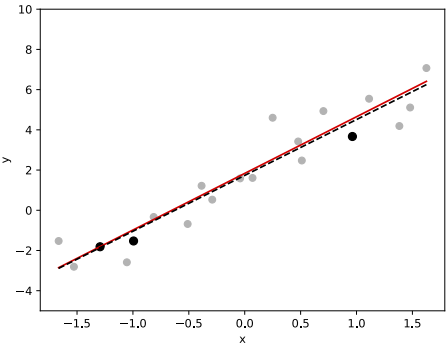
Example: SGD



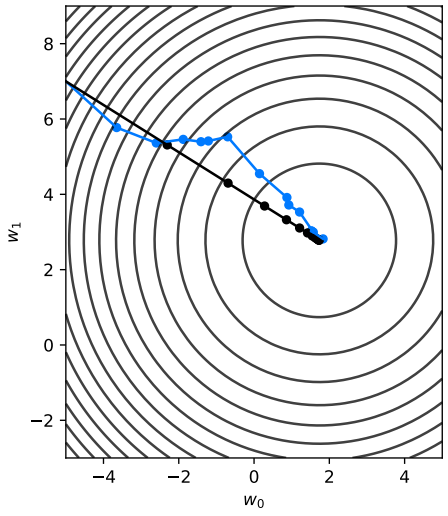


Example: SGD





SGD vs. GD



Tradeoffs

- In each step of GD, move in the “best” direction.
  - But **slowly!**
- In each step of SGD, move in a “good” direction.
  - But **quickly!**
- SGD may take more steps to converge, but can be faster overall.

Example

- Suppose you're doing **least squares regression** on a medium-to-large data set.
- Say, n = 200,000 examples, d = 5,000 features.
- Encoded as 64 bit floats, X is 8 GB.
  - Fits in your laptop's memory, but barely.
- **Example:** predict sentiment from text.

Timing

- Solving the normal equations took **30.7 seconds**.
- Gradient descent took **8.6 seconds**.
  - 14 iterations, ≈ 0.6 seconds per iteration.
- Stochastic gradient descent takes **3 seconds**.
  - Batch size $m = 16$.
  - 13,900 iterations, ≈ 0.0002 seconds per iteration.

Aside: Terminology

- Some people say “stochastic gradient descent” only when batch size is 1.
- They say “mini-batch gradient descent” for larger batch sizes.
- **In this class**: we’ll use “SGD” for any batch size, as long as it’s chosen randomly.

Aside: A Popular Variant

- One variant of SGD uses **epochs**.

- During each epoch, we:
  - Randomly shuffle the training data.
  - Divide the training data into n/m mini-batches.
  - Perform one step for each mini-batch.

Usefulness of SGD

- SGD **enables** learning on **massive** data sets.
  - Billions of training examples, or more.
- Useful even when exact solutions available.
  - E.g., least squares regression / classification.

DSC 140B

Representation Learning

Lecture 13 | Part 4

PyTorch

Training NNs in Python

- Python is **the** language of ML.
- There are several popular NN libraries for Python, but nowadays **PyTorch** is the most popular.

Demo Notebook

- A demo notebook is linked at `dsc140b.com`