DSC 140A

Probabilistic Modeling & Machine Learning

Lecture 4 | Part 1

Linear Classification

Classification

- We've been considering **regression**. What about **classification**?

Empirical Risk Minimization

- Step 1: choose a **hypothesis class**
  - Let's assume we've chosen linear predictors
- Step 2: choose a **loss function**
- Step 3: minimize **expected loss (empirical risk)**

Exercise

Can we use the square loss for classification?

$$(H(\vec{x}^{(i)}) - y_i)^2$$

Square Loss for Classification

- **Yes!** We can use the square loss, but it may not be the best choice.
- E.g., suppose $H(\vec{x}^{(i)}) = 11$ and $y_i = 1$.
  - Square loss: 100. **Large!**
- E.g., suppose $H(\vec{x}^{(i)}) = -9$ and $y_i = 1$.
  - Square loss:

Main Idea

While the square loss can be used for classification, it may not be the best choice because it penalizes predictions that are “very correct”.

It is designed for regression.

Least Squares Classifier

- **Least squares classification** is performed *exactly* the same as least squares regression.
  - I.e., solve the normal equations: $\vec{w}^* = (X^T X)^{-1} X^T \vec{y}$

- Except the prediction is thresholded:

$$H(\vec{x}) = \text{sign}\left(\text{Aug}\left(\vec{x}\right) \cdot \vec{w}^*\right)$$

Least Squares and Outliers

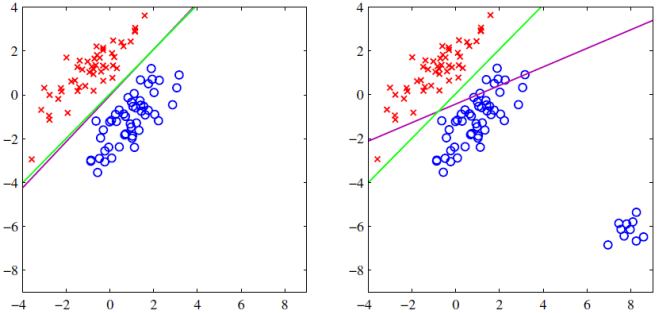


Figure 4.4 The left plot shows data from two classes, denoted by red crosses and blue circles, together with the decision boundary found by least squares (magenta curve) and also by the logistic regression model (green curve), which is discussed later in Section 4.3.2. The right-hand plot shows the corresponding results obtained when extra data points are added at the bottom left of the diagram, showing that least squares is highly sensitive to outliers, unlike logistic regression.

[1]Bishop, *Pattern Recognition and Machine Learning*

Another Loss Function

- What about the **0-1 loss**?
  - Loss = 0 if prediction is **correct**.
  - Loss = 1 if prediction is **incorrect**.

- More formally:

$$L_{0\text{-}1}(H(\vec{x}^{(i)}), y_i) = \begin{cases} 0 & \text{if } \text{sign}(H(\vec{x}^{(i)})) = y_i \\ 1 & \text{if } \text{sign}(H(\vec{x}^{(i)})) \neq y_i \end{cases}$$

Expected 0-1 Loss

- The expected 0-1 loss (empirical risk) has a nice interpretation:

$$R_{0\text{-}1}(H) = \frac{1}{n}\sum_{i=1}^{n}\begin{cases}0 & \text{if } \text{sign}(H(\vec{x}^{(i)})) = y_i \\ 1 & \text{if } \text{sign}(H(\vec{x}^{(i)})) \neq y_i\end{cases}$$

Exercise

What is it?

Answer

- The empirical risk with respect to the 0-1 loss is (1 - the **accuracy**) of the classifier.

$$R_{0\text{-}1}(H) = \frac{1}{n}\sum_{i=1}^{n}\begin{cases}0 & \text{if } \text{sign}(H(\vec{x}^{(i)})) = y_i \\ 1 & \text{if } \text{sign}(H(\vec{x}^{(i)})) \neq y_i\end{cases}$$

$$= \frac{\#\text{ of } \textbf{incorrect} \text{ predictions}}{n}$$

ERM for the 0-1 Loss

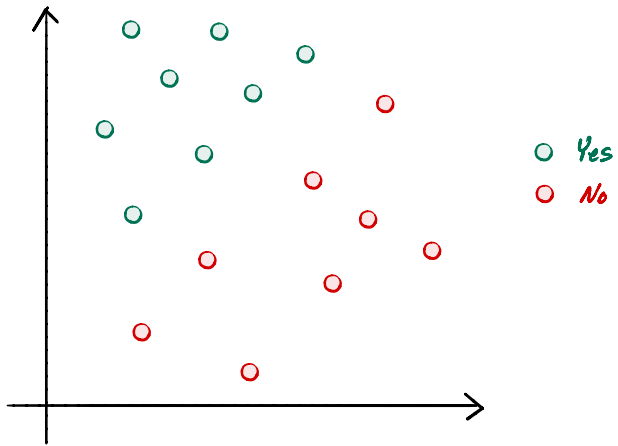
- Assume the 0-1 loss, linear prediction functions.
- Next step: find linear H minimizing the empirical risk.
- That is: find H which maximizes the **accuracy** on the training set.

Problem

- The 0-1 loss is not differentiable.
- Can’t even use gradient descent...

Why?

- The 0-1 risk surface is flat almost everywhere.



Computationally Difficult

- It is **not feasible** to minimize 0-1 risk in general.
- More formally: NP-Hard to optimize expected 0-1 loss in general.[2]

[2] It is efficiently doable if the classes are linearly separable by finding convex hulls of each class. If non-separable, it is difficult.

Main Idea

It is computationally difficult (NP-Hard) to maximize the accuracy of a linear classifier.

Surrogate Loss

- Instead of the 0-1 loss, we use a **surrogate loss**.
- That is, a loss that is similar in spirit, but has better mathematical properties.

The Perceptron Loss

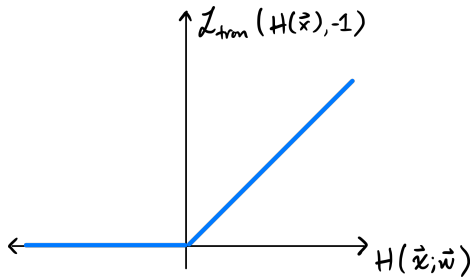
- The **perceptron loss** is designed for binary classification.
  - Loss = 0 if prediction is correct.
  - Loss > 0 if prediction incorrect.
  - Loss increases with distance from boundary.

- More formally:

$$L_{\text{tron}}(H(\vec{x}), y) = \begin{cases} 0, & \text{sign}(H(\vec{x})) = \text{sign}(y) \\ |H(\vec{x})|, & \text{sign}(H(\vec{x})) \neq \text{sign}(y) \end{cases}$$

Perceptron Loss

$$L_{\text{tron}}(H(\vec{x};\vec{w}), y) = \begin{cases} 0, & \text{sign}(H(\vec{x})) = \text{sign}(y) \\ |H(\vec{x})|, & \text{sign}(H(\vec{x})) \neq \text{sign}(y) \end{cases}$$



ERM for the Perceptron

- **Goal:** minimize the empirical expected perceptron loss (risk):

$$L_{\text{tron}}(H(\vec{x}), y) = \begin{cases} 0, & \text{sign}(H(\vec{x})) = \text{sign}(y) \\ |H(\vec{x})|, & \text{sign}(H(\vec{x})) \neq \text{sign}(y) \end{cases}$$

Optimization

- The perceptron risk is:

$$R_{\text{tron}}(\vec{w}) = \frac{1}{n}\sum_{i=1}^{n} L_{\text{tron}}(H(\vec{x}), y)$$

$$= \frac{1}{n}\sum_{i=1}^{n} \begin{cases} 0, & \text{sign}(H(\vec{x})) = \text{sign}(y) \\ |H(\vec{x})|, & \text{sign}(H(\vec{x})) \neq \text{sign}(y) \end{cases}$$

Exercise

Suppose the training data are **linearly separable**. What is the minimum possible value of R_{tron}?

Optimization

- To optimize, solve $\vec{\nabla} R_{\text{tron}}(\vec{w}) = 0$?
- The gradient of the risk *would* be:

$$\begin{aligned}\vec{\nabla} R_{\text{tron}}(\vec{w}) &= \vec{\nabla} \frac{1}{n} \sum_{i=1}^{n} L_{\text{tron}}(H(\vec{x}), y) \\ &= \frac{1}{n} \sum_{i=1}^{n} \vec{\nabla} L_{\text{tron}}(H(\vec{x}), y)\end{aligned}$$

- However, the perceptron loss is **not differentiable**.

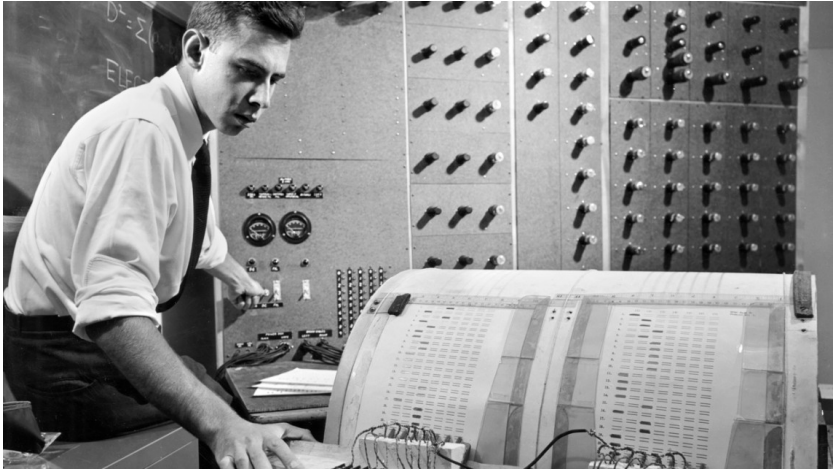
Problem

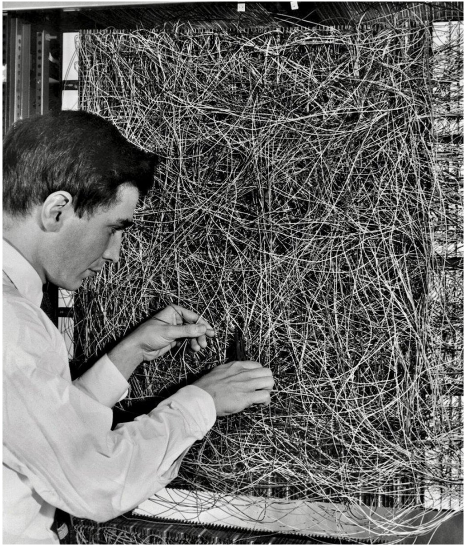
- The risk is **not differentiable**.
- However:
  - it is **not flat**!
  - its derivative exists almost everywhere.
- We can train iteratively using **subgradient descent** (as we'll see).

Some History

- Perceptrons were one of the first “machine learning” models.
- The basis of modern neural networks.

Rosenblatt's Perceptron





DSC 140A

Probabilistic Modeling & Machine Learning

Lecture 4 | Part 2

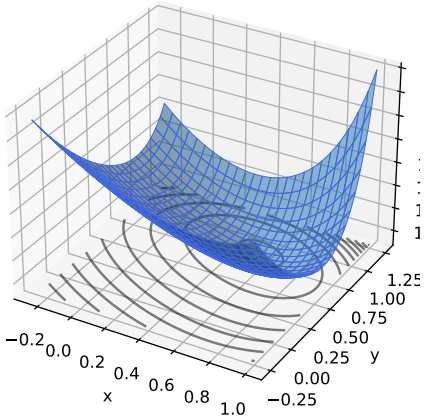
Gradient Descent

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?

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:

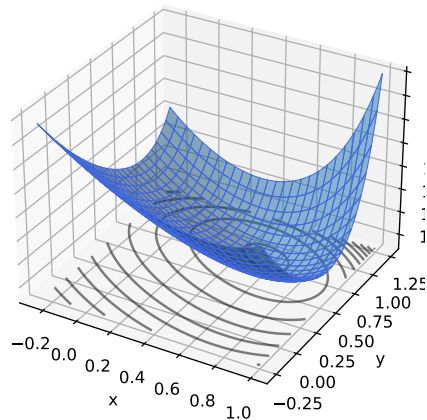
$$\vec{\nabla} f(x, y) = \begin{pmatrix} 2xe^{x^2+y^2} + 2(x-2) \\ 2ye^{x^2+y^2} + 2(y-3) \end{pmatrix}$$

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

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

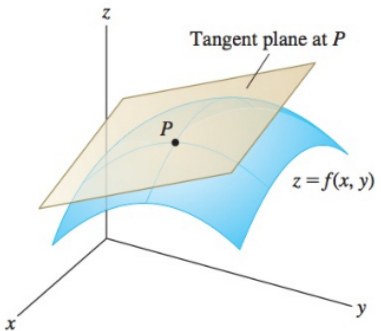
Idea

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

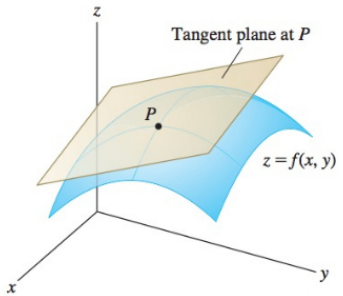


Which way is down?

- 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.



Which way is down?



- Linear approximation:

$$f(x_0 + \delta_x, y_0 + \delta_y) \approx f(x_0, y_0) + \qquad +$$

Which way is down?

- Define $\vec{\delta} = (\delta_x, \delta_y)^T$
- Define the **gradient**: $\vec{\nabla} f(x, y) = \left(\frac{\partial f}{\partial x}(x, y), \frac{\partial f}{\partial y}(x, y)\right)^T$
- Then the linear approximation becomes:

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

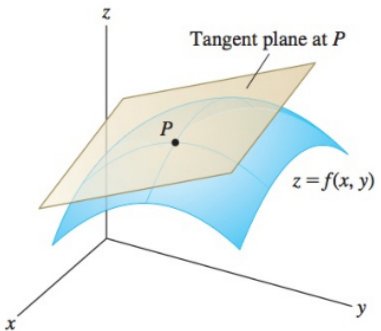
Which way is up?

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

- Suppose δ is a unit vector (a **direction**).
- Which direction results in the greatest **increase**?
- Recall $\vec{\delta} \cdot \vec{\nabla} f(x_0, y_0) = \|\vec{\delta}\| \|\vec{\nabla} f(x_0, y_0)\| \cos\theta$
- Answer: choosing δ in the direction of gradient.

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).



The Gradient

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

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

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

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
```

Example

- **Goal:** minimize $f(x, y) = e^{x^2+y^2} + (x - 2)^2 + (y - 3)^2$ with gradient descent.
- Initial location $\vec{x}^{(0)} = (\frac{1}{2}, 0)^T$; learning rate $\eta = 0.05$
- Recall:

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

Example: First Step

- Compute gradient at $\vec{x}^{(0)} = (\frac{1}{2}, 0)^T$:

$$\vec{\nabla} f(1/2, 0) = \begin{pmatrix} 2 \cdot (1/2) \cdot e^{(1/2)^2+0^2} + 2((1/2) - 2) \\ 2 \cdot 0 \cdot e^{(1/2)^2+0^2} + 2(0 - 3) \end{pmatrix} \approx \begin{pmatrix} -1.71 \\ -6 \end{pmatrix}$$

- Update:

$$\begin{aligned} \vec{x}^{(1)} &= \vec{x}^{(0)} - \eta \vec{\nabla} f(\vec{x}^{(0)}) \\ &= (1/2, 0)^T - .05 \cdot \left(-1.71, -6\right)^T \\ &= (1/2, 0)^T - \left(-.08, -.3\right)^T \\ &= (.58, .3)^T \end{aligned}$$

Example: Second Step

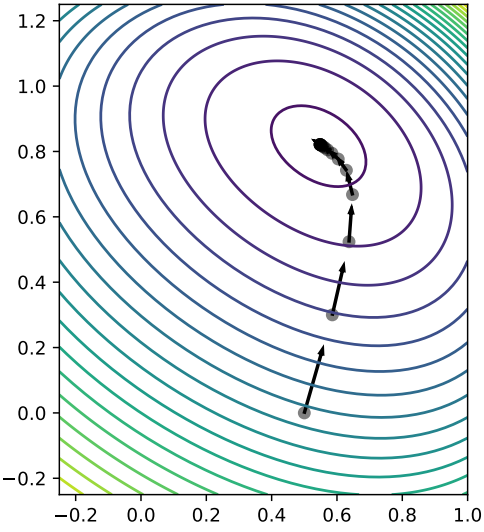
- Compute $\vec{\nabla} f$ at $\vec{x}^{(1)}$:

$$\vec{\nabla} f(\vec{x}^{(1)}) = \vec{\nabla} f(.58, .3)^T \approx (-1.06, -4.48)^T$$

- Update:

$$\begin{aligned} \vec{x}^{(2)} &= \vec{x}^{(1)} - \eta \vec{\nabla} f(\vec{x}^{(1)}) \\ &= (.58, .3)^T - .05 \cdot (-1.06, -4.48)^T \\ &= (.63, .52)^T \end{aligned}$$

- Repeat...


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

DSC 140A

Probabilistic Modeling & Machine Learning

Lecture 4 | Part 3

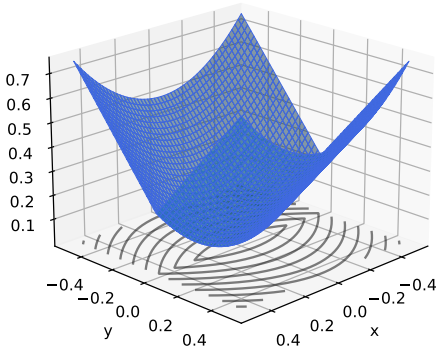
Subgradient Descent

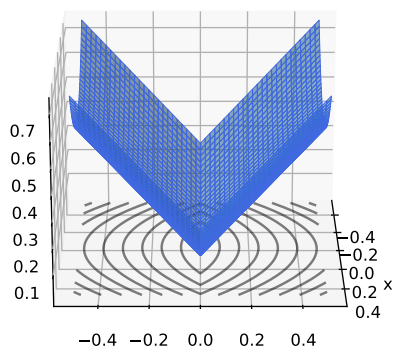
A Problem

- To perform gradient descent, function must be **differentiable**.
- What if it isn't?

Example

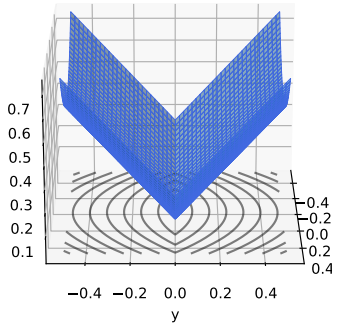
- $f(x, y) = x^2 + |y|$





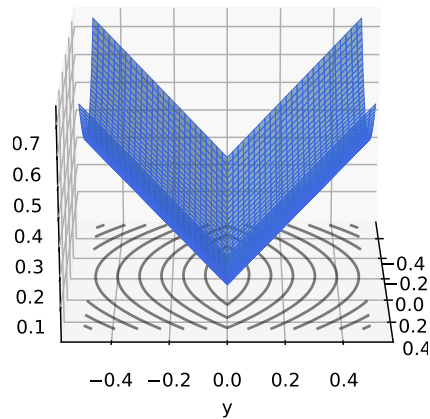
Exercise

Where is the function **not** differentiable? That is, where is there not a well-defined tangent plane?



Answer

- $\vec{\nabla} f(x, y)$ is defined everywhere except along $y = 0$.
- If $y > 0$,
  $f(x, y) = x^2 + |y| = x^2 + y$.
  - $\vec{\nabla} f(x, y) = (2x, 1)^T$
- If $y < 0$,
  $f(x, y) = x^2 + |y| = x^2 - y$.
  - $\vec{\nabla} f(x, y) = (2x, -1)^T$

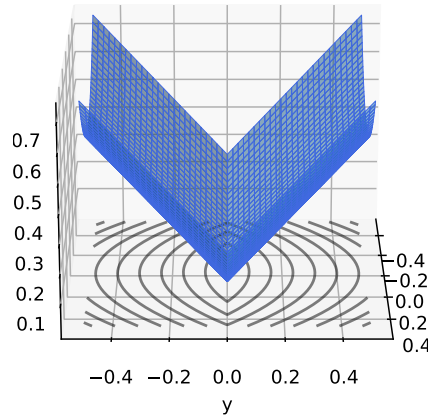


Answer

- Piecewise gradient:

$$\vec{\nabla} f(x, y) = \begin{cases} (2x, 1)^T & \text{, if } y > 1, \\ (2x, -1)^T & \text{, if } y < 1. \end{cases}$$

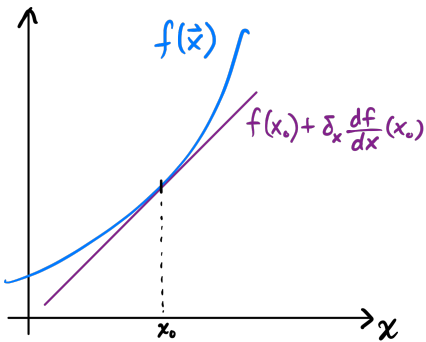
- **Not defined** along $y = 0$.

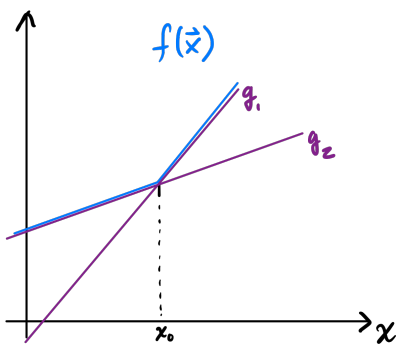


Problem

- Gradient descent will *almost* work on $f(x, y) = x^2 + |y|$.
- What if it reaches a point where $y = 0$.
- Which direction should it go?

Intuition





Subgradient

- A **subgradient** of f at $\vec{x}^{(0)}$ is a vector $\vec{g}$ such that:

$$f(\vec{x}^{(0)}) + \vec{\delta} \cdot \vec{g}$$

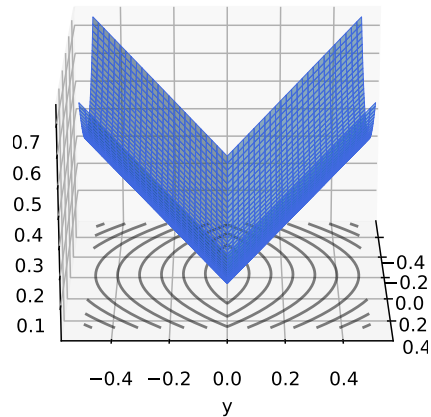
  lies below f for any choice of $\vec{\delta}$.

- There are possibly (infinitely) many subgradients.

Example

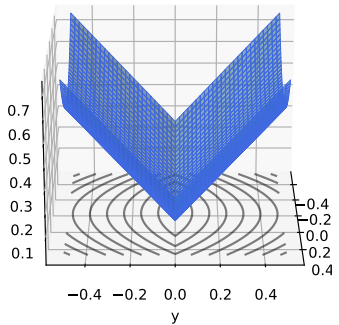
- $f(x, y) = x^2 + |y|$
- A subgradient:

$$\vec{g}(x, y) = \begin{cases} (2x, 1)^T & \text{, if } y > 1, \\ (2x, -1)^T & \text{, if } y < 1, \\ (2x, 0)^T & \text{, if } y = 0. \end{cases}$$



Exercise

Find another subgradient of the function $f(x, y) = x^2 + |y|$.



Subgradient Descent

- **Idea:** use subgradient in place of gradient in gradient descent.

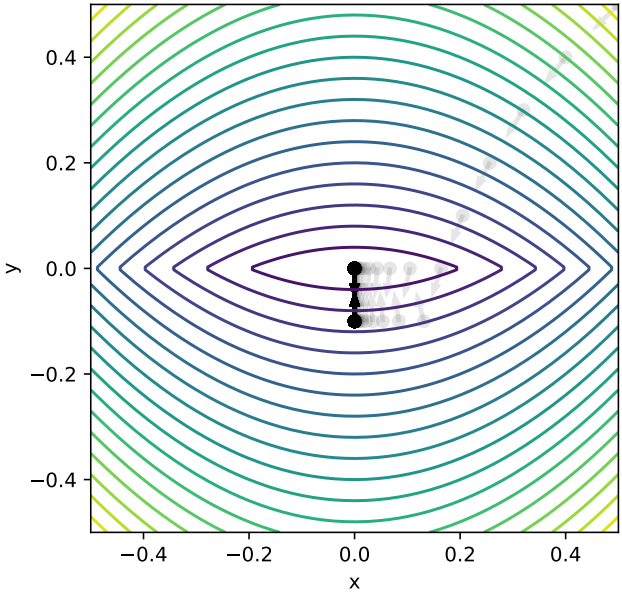
- Subgradient is not unique; choose an arbitrary one.

Subgradient Descent

- Pick arbitrary starting point $\vec{x}^{(0)}$, **learning rate** parameter $\eta > 0$.
- Until convergence, repeat:
  - Compute a **subgradient** of f at $\vec{x}^{(i)}$.
  - 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.

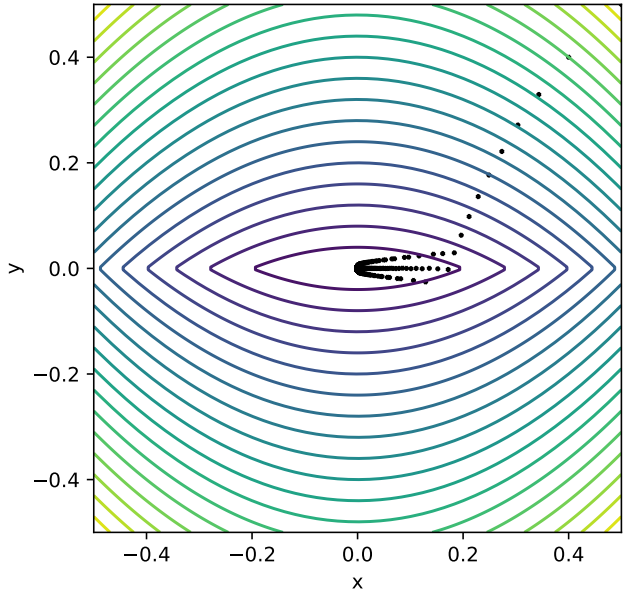
Example

- Subgradient descent on $f(x, y) = x^2 + |y|$
- Starting point: $(1/2, 1/2)^T$
- Learning rate: $\eta = 0.1$.


0.4
0.2
0.0
−0.2
−0.4
y
−0.4
−0.2
0.0
0.2
0.4
x

Problem

- Does not converge! Why?
- **Fix:** decrease learning rate with each iteration.
- Theory: choose $\eta = O(1/\sqrt{i})$, where i is iteration #.


0.4
0.2
0.0
−0.2
−0.4
y
−0.4
−0.2
0.0
0.2
0.4
x

DSC 140A

Probabilistic Modeling & Machine Learning

Lecture 4 | Part 4

(Sub)gradient Descent for ERM

ERM for the Perceptron

- **Goal:** minimize the empirical expected perceptron loss (risk):

$$L_{\text{tron}}(H(\vec{x}), y) = \begin{cases} 0, & \text{sign}(H(\vec{x})) = \text{sign}(y) \\ |H(\vec{x})|, & \text{sign}(H(\vec{x})) \neq \text{sign}(y) \end{cases}$$

Optimization

- The perceptron risk is:

$$R_{\text{tron}}(\vec{w}) = \frac{1}{n}\sum_{i=1}^{n} L_{\text{tron}}(H(\vec{x}), y)$$

$$= \frac{1}{n}\sum_{i=1}^{n} \begin{cases} 0, & \text{sign}(H(\vec{x})) = \text{sign}(y) \\ |H(\vec{x})|, & \text{sign}(H(\vec{x})) \neq \text{sign}(y) \end{cases}$$

- To optimize, solve $\vec{\nabla} R_{\text{tron}}(\vec{w}) = 0$?

Optimization

- The gradient of the risk *would* be:

$$\begin{aligned}\vec{\nabla} R_{\text{tron}}(\vec{w}) &= \vec{\nabla} \frac{1}{n} \sum_{i=1}^{n} L_{\text{tron}}(H(\vec{x}), y) \\ &= \frac{1}{n} \sum_{i=1}^{n} \vec{\nabla} L_{\text{tron}}(H(\vec{x}), y)\end{aligned}$$

- However, the perceptron loss is **not differentiable**.

Idea

- Instead of solving $\vec{\nabla} R_{\text{tron}}(\vec{w}) = 0$, use **subgradient descent**.
- We need to compute a **subgradient** of the perceptron loss.

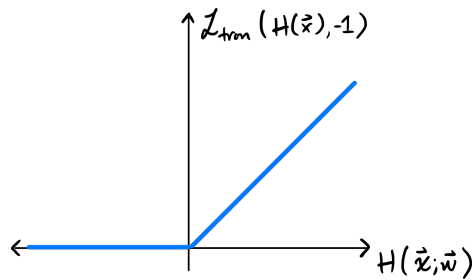
Subgradient of Perceptron Loss

$$L_{\text{tron}}(H(\vec{x};\vec{w}),y) = \begin{cases} 0, & \text{sign}(H(\vec{x})) = \text{sign}(y) \\ |\vec{x}\cdot\vec{w}|, & \text{sign}(H(\vec{x})) \neq \text{sign}(y) \end{cases}$$

- If prediction is **correct**, $\vec{\nabla} L_{\text{tron}} = 0$.
- If prediction is **incorrect** with $\vec{x}\cdot\vec{w} > 0$,
  - then $|\vec{x}\cdot\vec{w}| = \vec{x}\cdot\vec{w}$.
  - $\vec{\nabla} L_{\text{tron}} = \frac{d}{d\vec{w}}(\vec{x}\cdot\vec{w} - y) = \vec{x}$
- If prediction is **incorrect** with $\vec{x}\cdot\vec{w} < 0$,
  - then $|\vec{x}\cdot\vec{w}| = -\vec{x}\cdot\vec{w}$.
  - $\vec{\nabla} L_{\text{tron}} = \frac{d}{d\vec{w}}(-\vec{x}\cdot\vec{w} - y) = -\vec{x}$

Subgradient of Perceptron Loss

- Gradient is not defined when $H(\vec{x}; \vec{w}) = 0$.
- Can choose any subgradient.
- $\vec{0}$ works.



Subgradient

- A subgradient of the perceptron loss:

$$\vec{g}(\vec{w}, \vec{x}, y) = \begin{cases} 0, & \text{if } \operatorname{sign}(\vec{w} \cdot \vec{x}) = \operatorname{sign}(y) \text{ or } \vec{x} \cdot \vec{w} = 0, \\ \vec{x}, & \text{if } \operatorname{sign}(\vec{w} \cdot \vec{x}) \neq \operatorname{sign}(y) \text{ and } \vec{x} \cdot \vec{w} > 0, \\ -\vec{x}, & \text{if } \operatorname{sign}(\vec{w} \cdot \vec{x}) \neq \operatorname{sign}(y) \text{ and } \vec{x} \cdot \vec{w} < 0. \end{cases}$$

- Subgradient of the risk

$$\vec{g}_R(\vec{w}) = \frac{1}{n} \sum_{i=1}^{n} g(\vec{w}, \vec{x}^{(i)}, y_i)$$

Training a Perceptron

- A perceptron can be training with subgradient descent, using $\vec{g}_R$ as the subgradient.
- If the data are linearly separable[3], will obtain perfect accuracy on training data.

[3] and the learning rate is appropriately chosen

Gradient Descent for ERM

- Suppose L is a **differentiable** loss function (e.g., square loss).
- Then the gradient of the risk, $R(\vec{w})$, is:

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

Subgradient Descent for ERM

- If L is not differentiable, we may still be able to use subgradient descent if a subgradient exists.

- If $\vec{g}(\vec{w}, \vec{x}, y)$ is a subgradient of L, then a subgradient of the risk, R, is:

$$\frac{1}{n} \sum_{i=1}^{n} g(\vec{w}, \vec{x}^{(i)}, y_i)$$

What's left?

- Does (sub)gradient descent always work? **No**.
  - When is it guaranteed to work?

- Computing the full gradient can be costly.

DSC 140A

Probabilistic Modeling & Machine Learning

Lecture 4 | Part 5

Perceptron Demo: MNIST

Demo: MNIST

- MNIST is a classic machine learning data set.
- Many images of handwritten digits, 0-9.
- Multiclass classification problem.
- But we can make it binary: 3 vs. 7.

Example MNIST Digit

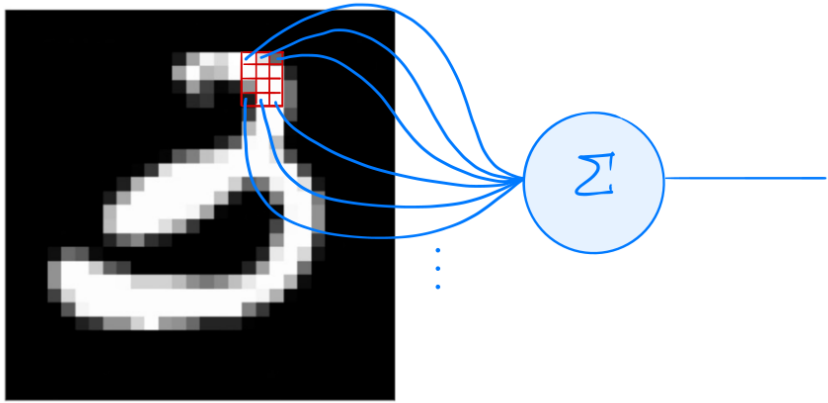
- Grayscale
- 28 x 28 pixels

MNIST Feature Vectors

- $28 \times 28 = 784$ pixels
- Each image is a vector in $\mathbb{R}^{784}$
- Each feature is intensity of single pixel
  - black $\rightarrow$ 0, white $\rightarrow$ 255
- A **very** simple representation.

Demo: MNIST

- Use only images of 3s and 7s.
- 4132 training images.
- 680 testing images.
- Some minor tuning.
  - Added random noise for robustness.
  - Picked classification threshold automatically.


Σ

Percepton Learning

- Linear prediction function parameterized by $\vec{w}$.
- In this case, we can “reshape” $\vec{w}$ to be same size as input image.

Weight Vector

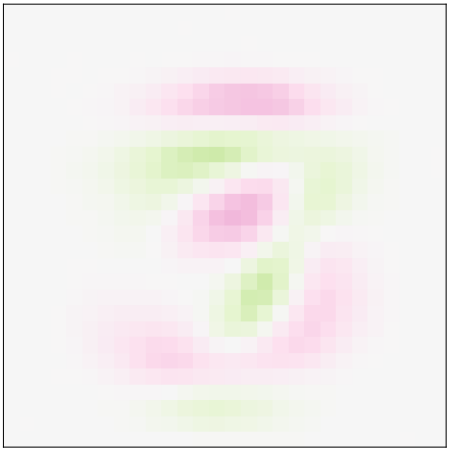
- Recall that the prediction is a **weighted vote**:

$$H(\vec{x}) = \text{sign}(w_0 + w_1 x_1 + w_2 x_2 + \ldots w_{784} x_{784})$$

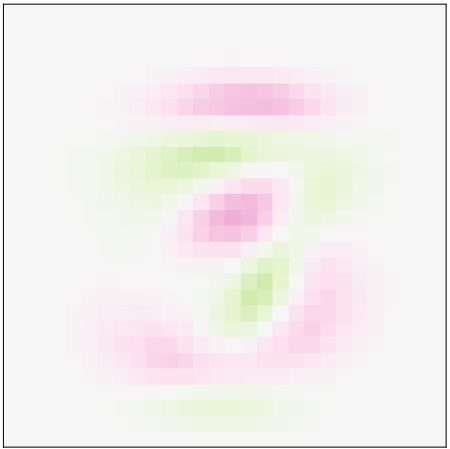
- Positive $\rightarrow$ 7, Negative $\rightarrow$ 3

- w_i is the weight of pixel i
  - positive: if this pixel is bright, I think this is a 7
  - negative: if this pixel is bright, I think this is a 3
  - magnitude: confidence in prediction

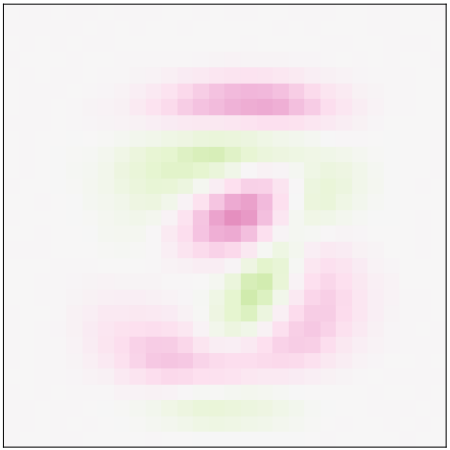
Perceptron Training



Perceptron Training



Perceptron Training



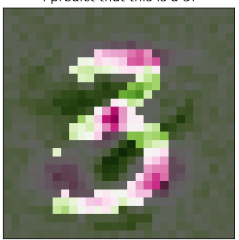
Perceptron Training

Perceptron Training

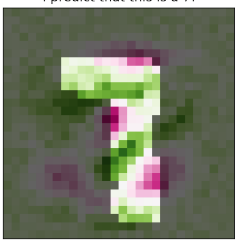
Perceptron Training

Perceptron Weight Vector

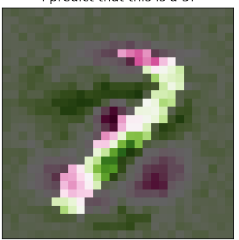
I predict that this is a 3!



I predict that this is a 7!



I predict that this is a 3!



Perceptron Results

- Test accuracy: 97.3%

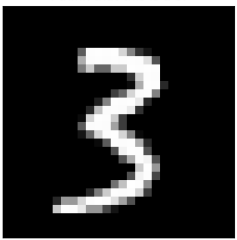
Square Loss for Classification

- What if we use square loss for classification?
- We *can*, but will it work well?

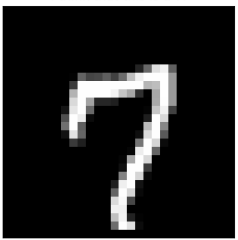
Results: Least Squares

- Test Accuracy: 96.7% (marginally worse)

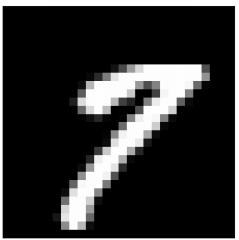
I think that this is a 3.



I think that this is a 7.



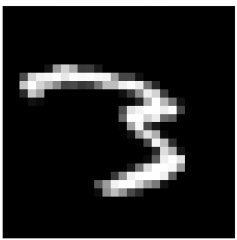
I think that this is a 7.



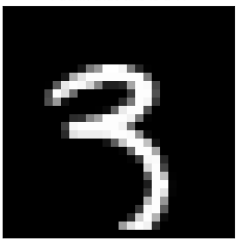
Results: Least Squares

- Misclassifications are telling.

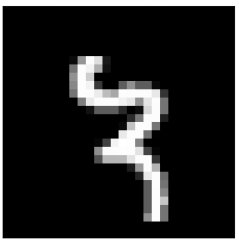
I think that this is a 7.



I think that this is a 7.

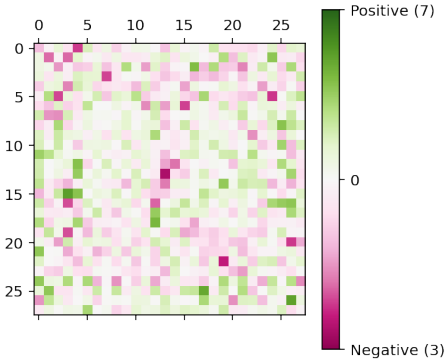


I think that this is a 7.



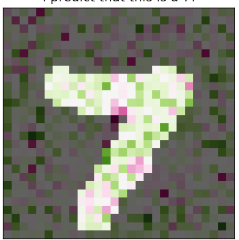
Least Squares Weight Vector

- Can visualize weight of each pixel as an image.



Least Squares Weight Vector

I predict that this is a 7!



I predict that this is a 3!

I predict that this is a 7!