Machine Larning 1 - Linear Classifiers

Linear predictors

Weight vector $\mathbf{w} \in \mathbb{R}^d$

Feature vector $\phi(x) \in \mathbb{R}^d$

For binary classification:

Definition: (binary) linear classifier

$$f_{\mathbf{w}}(x) = \text{sign}(\mathbf{w} \cdot \phi(x)) = \begin{cases} +1 & \text{if } \mathbf{w} \cdot \phi(x) > 0 \\ -1 & \text{if } \mathbf{w} \cdot \phi(x) < 0 \\ ? & \text{if } \mathbf{w} \cdot \phi(x) = 0 \end{cases}$$

Sign means you gotta commit, which side are you on?

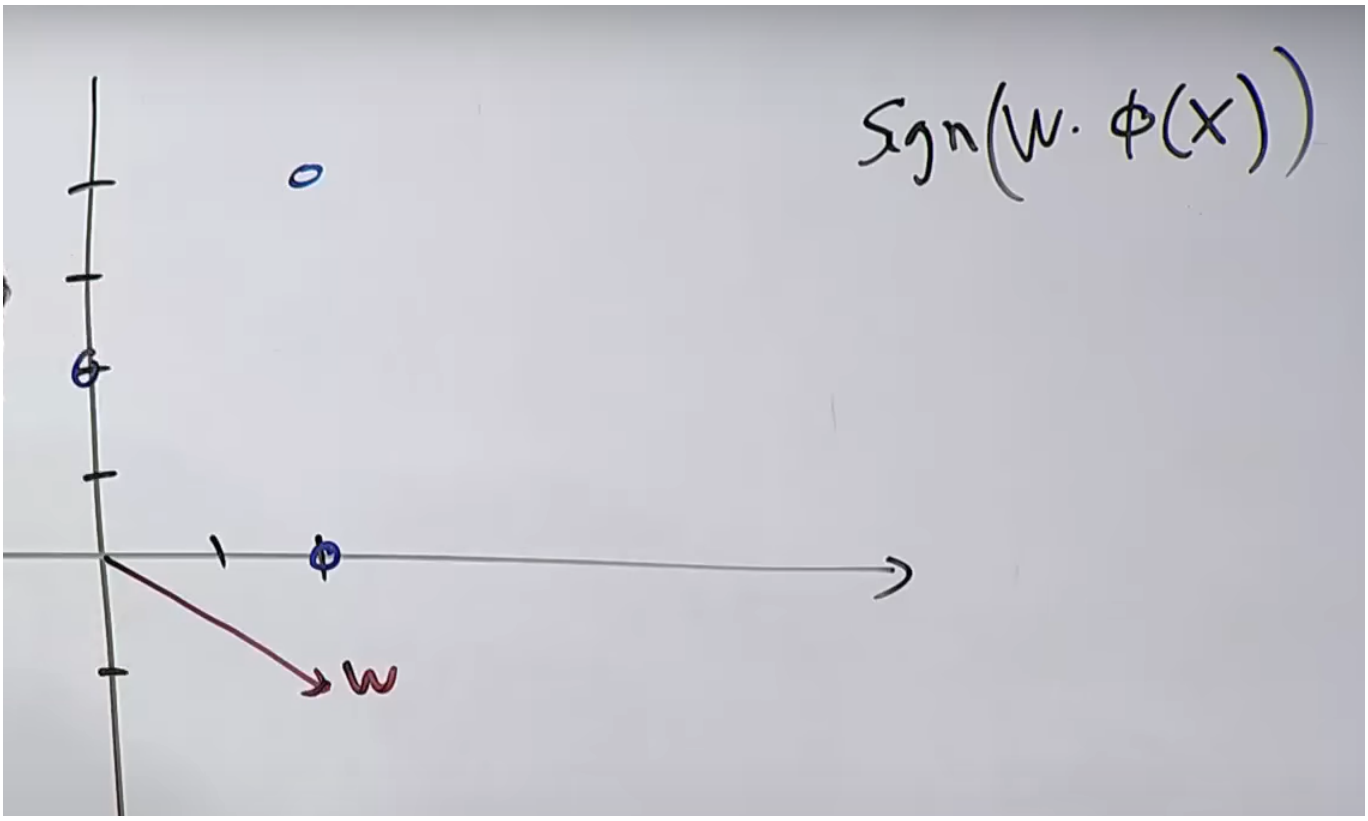
Visual intuition:

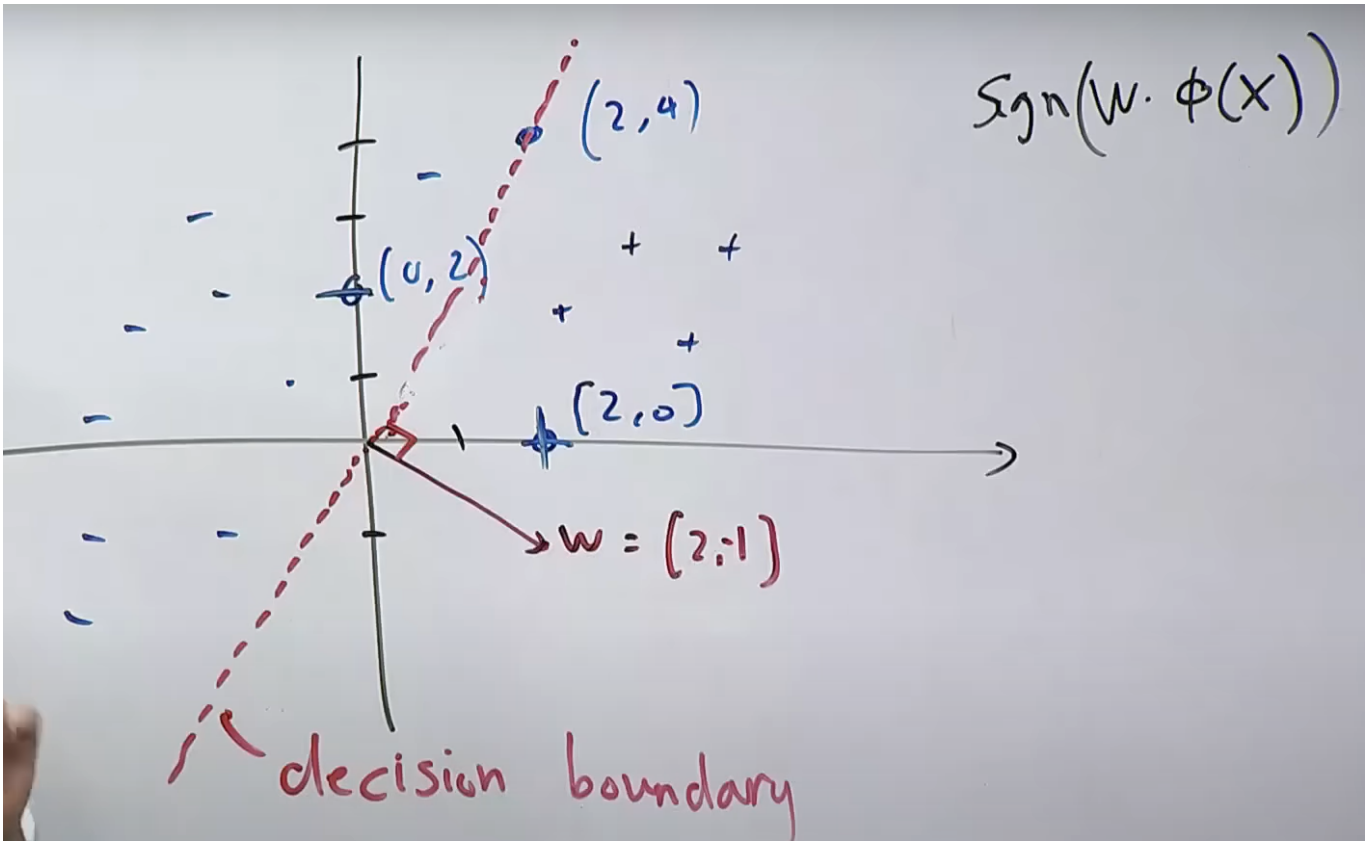
Geometric intuition

Example:

$$\mathbf{w} = [2, -1]$$

$$\phi(x) \in \{[2, 0], [0, 2], [2, 4]\}$$





Decision boundary

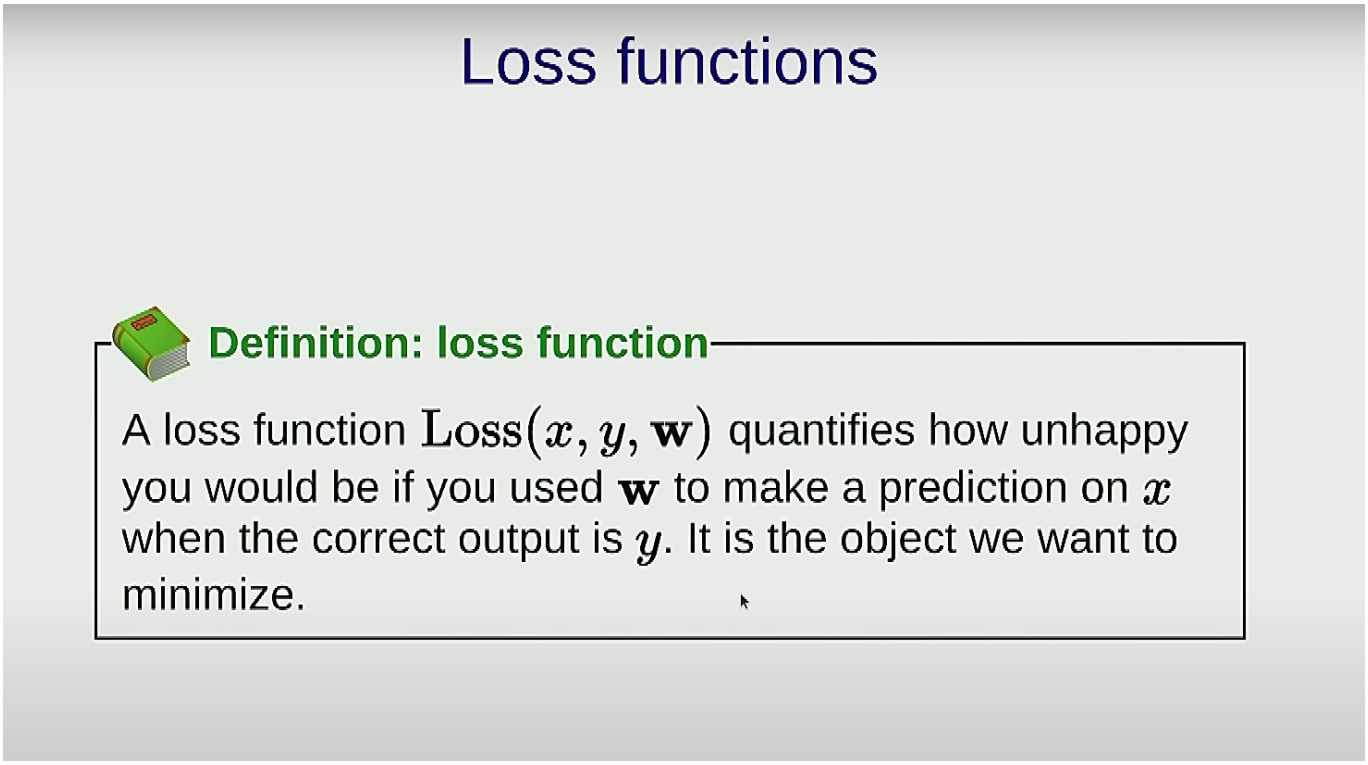
Loss Minimization

The leaner take some data and retruns a predictor

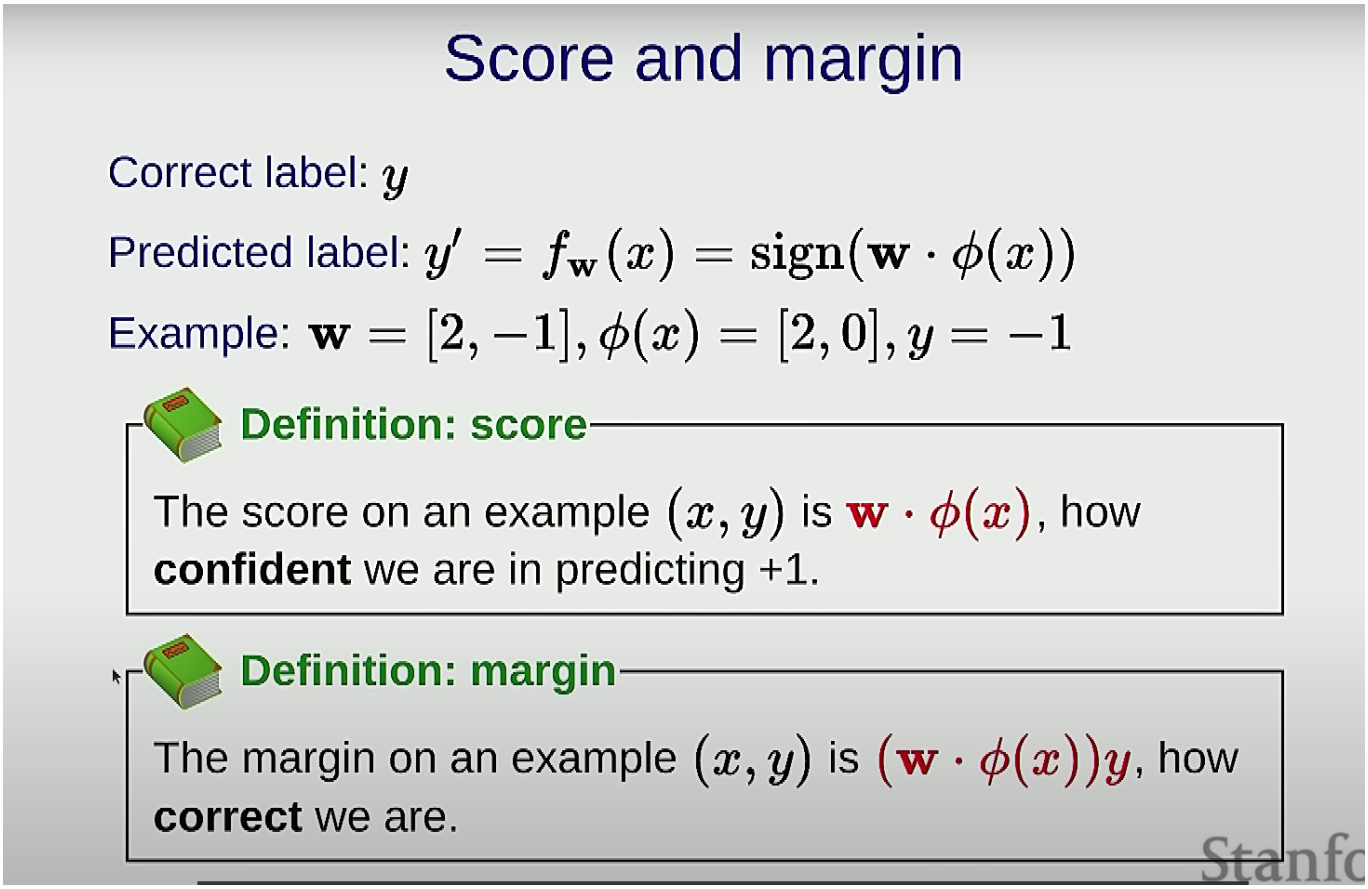
From what we wanna compute to how we wanna compute

Loss Function

Loss(x,y,w)



Score and Margin



When margin is less than 0, it means the prediction is confidently wrong (The score and actual Y has different signs)

Recall the binary classifier:

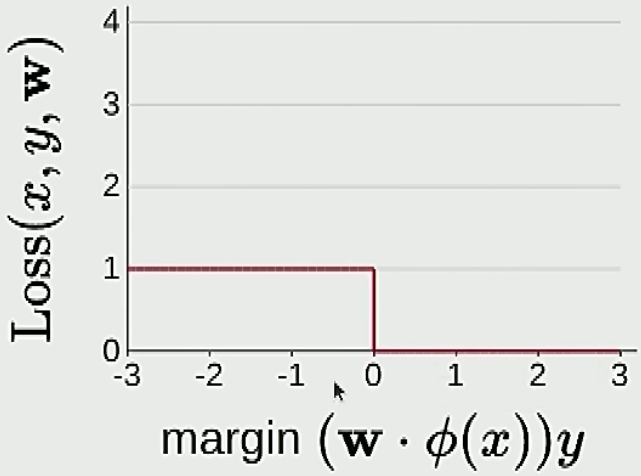
$$f_{\mathbf{w}}(x) = \text{sign}(\mathbf{w} \cdot \phi(x))$$

Definition: zero-one loss

$$\begin{aligned}\text{Loss}_{0\text{-}1}(x, y, \mathbf{w}) &= \mathbf{1}[f_{\mathbf{w}}(x) \neq y] \\ &= \mathbf{1}[\underbrace{(\mathbf{w} \cdot \phi(x))y}_{\text{margin}} \leq 0]\end{aligned}$$

The Loss function means: did u make mistake or not [f(w) = y] : this returns 0 or 1 depends on the statement is true or false

Binary classification

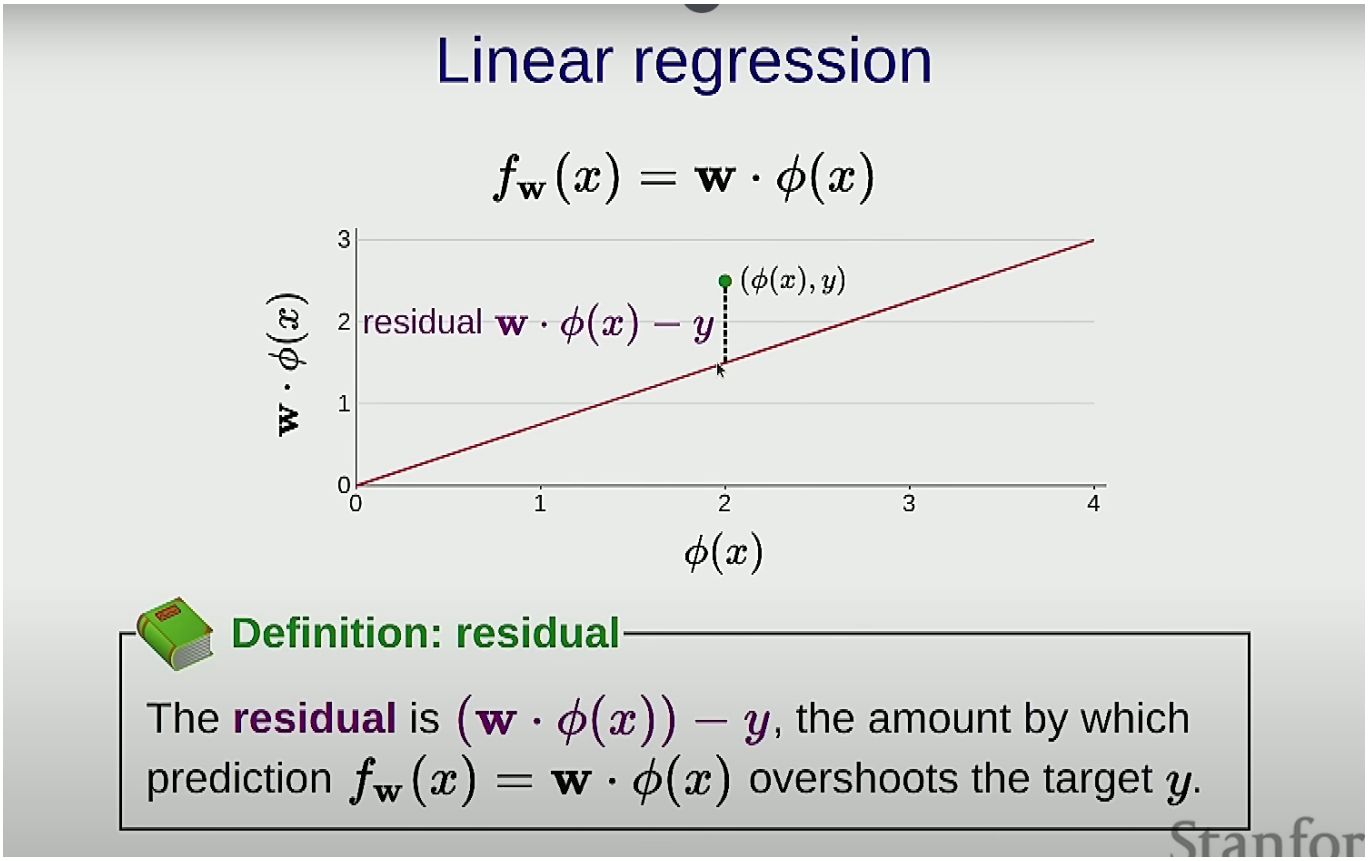


$$\text{Loss}_{0\text{-}1}(x, y, \mathbf{w}) = \mathbf{1}[(\mathbf{w} \cdot \phi(x))y \leq 0]$$

Linear Regression

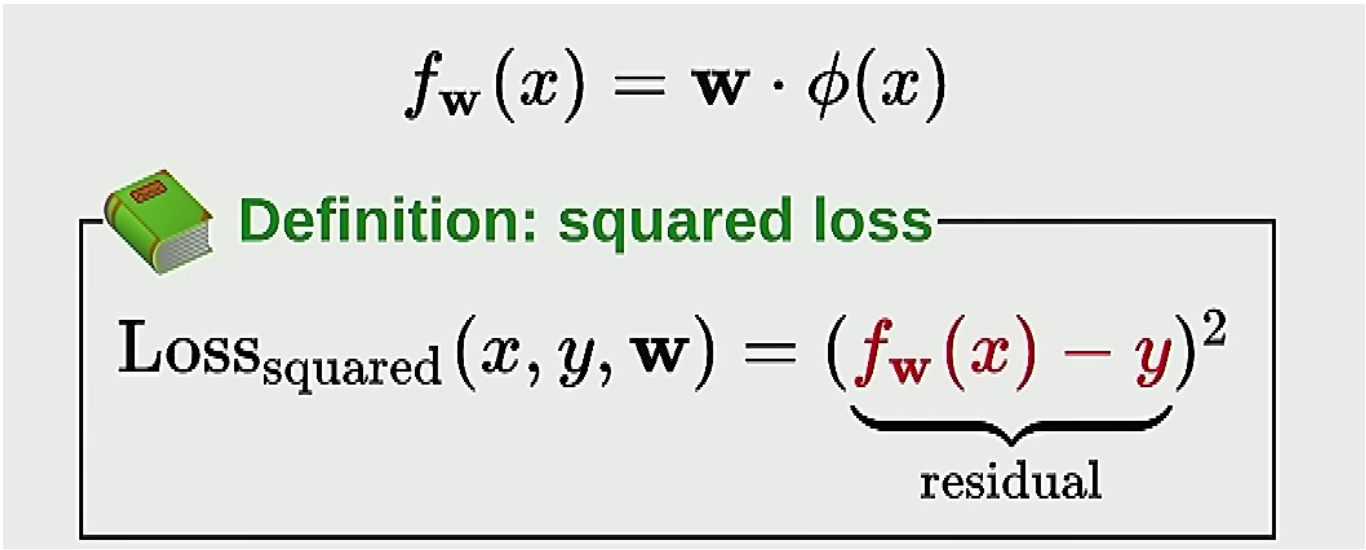
Residual

Try to catch how much did I wrong

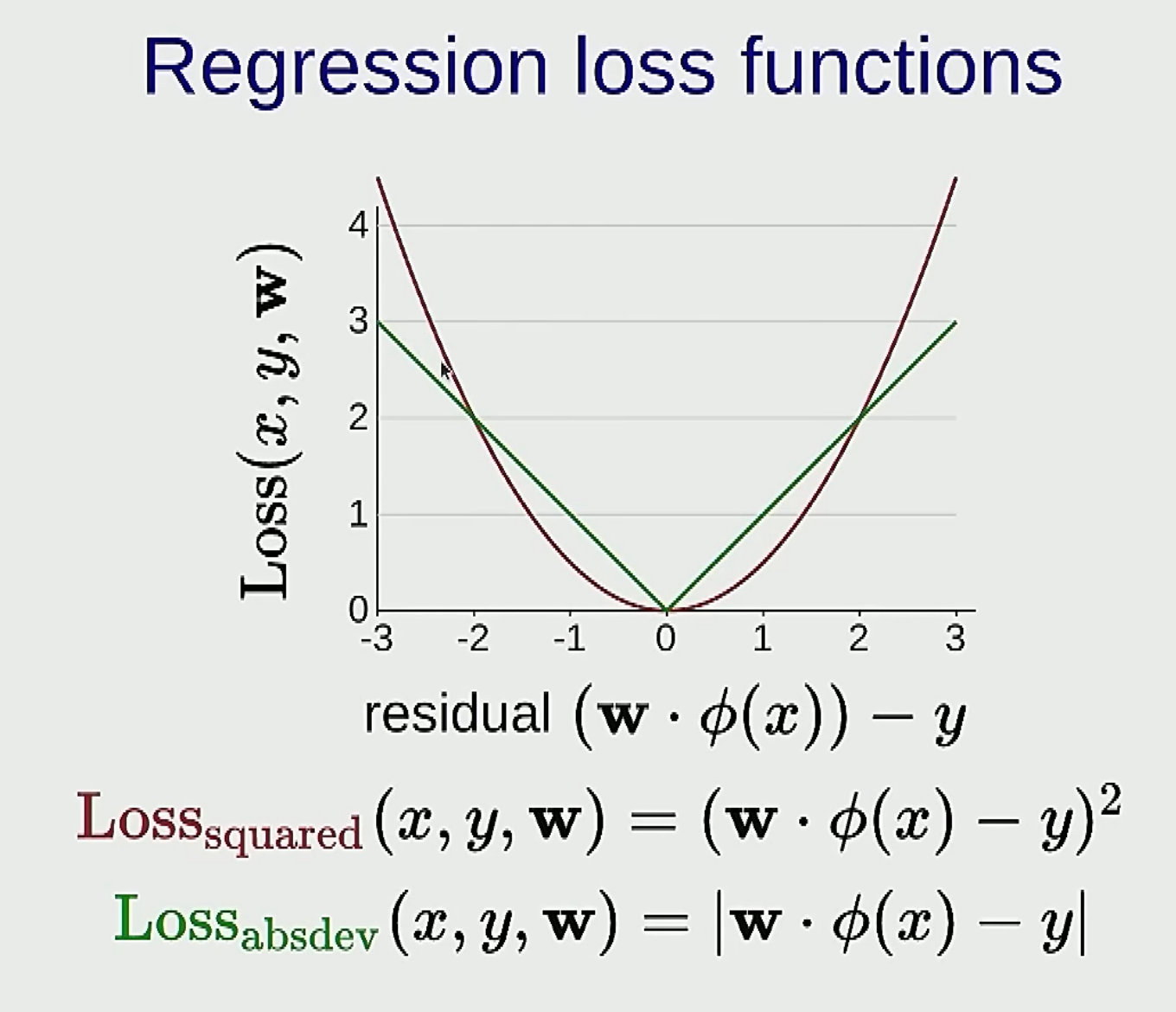


Redidual is the distance between the true value y and the predicted value y

Squared Loss



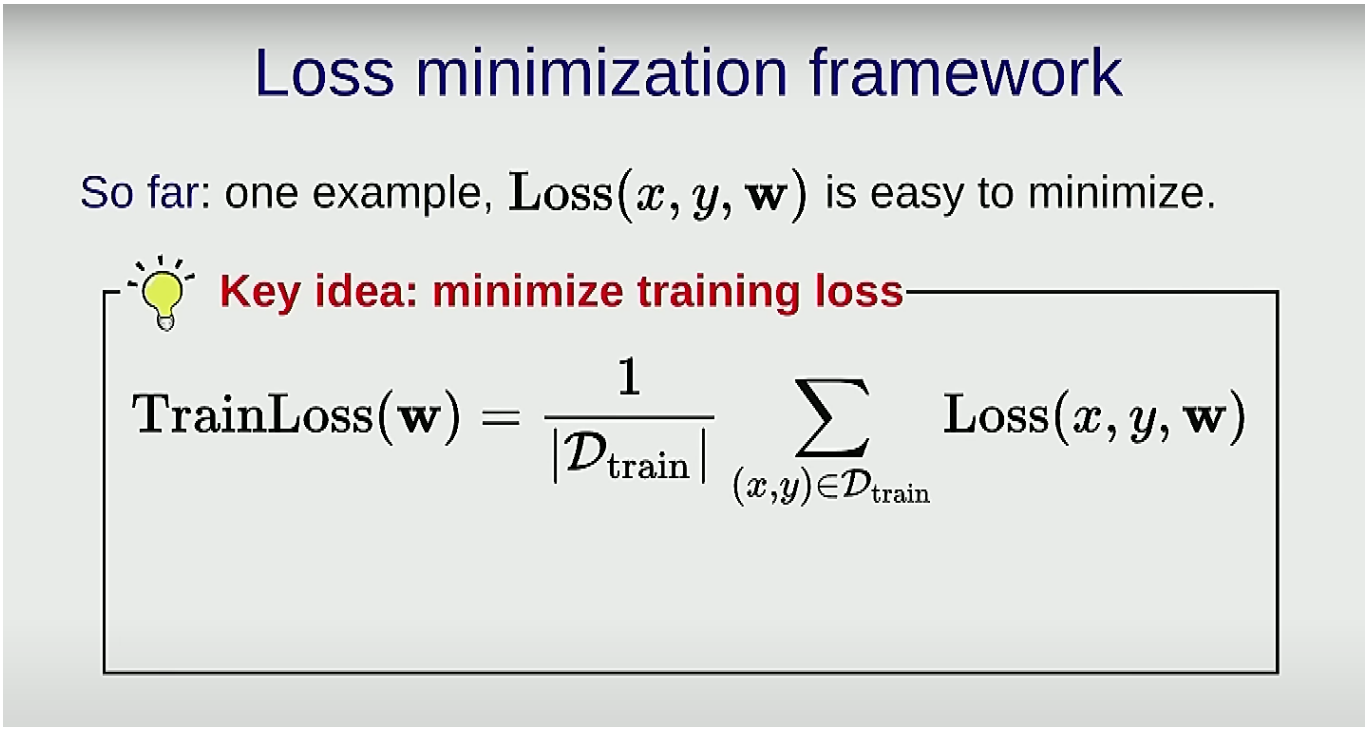
Squared Residual



Which one to choose? The squared loss is kinda like "mean", where its value may got affected by outliers extremely

For standerded deviation, you can think that as medium, which less affected by outliers

Minimize Loss => Train Loss



The sum of all loss averanged by the number of training data

This is a function of w(weight) because we wanna find a weight vector that best fit for all the data(points)

Stochastic gradient descent

Optimization Algorithm

Terrin map(等高线)

Gredinent

Where the value is increase the most

How to optimize?

Definition: gradient

The gradient $\nabla_{\mathbf{w}} \text{TrainLoss}(\mathbf{w})$ is the direction that increases the loss the most.

Algorithm: gradient descent

Initialize $\mathbf{w} = [0, \ldots, 0]$

For $t = 1, \ldots, T$:

$$\mathbf{w} \leftarrow \mathbf{w} - \underbrace{\eta}_{\text{step size}} \underbrace{\nabla_{\mathbf{w}} \text{TrainLoss}(\mathbf{w})}_{\text{gradient}}$$

Least squares regression

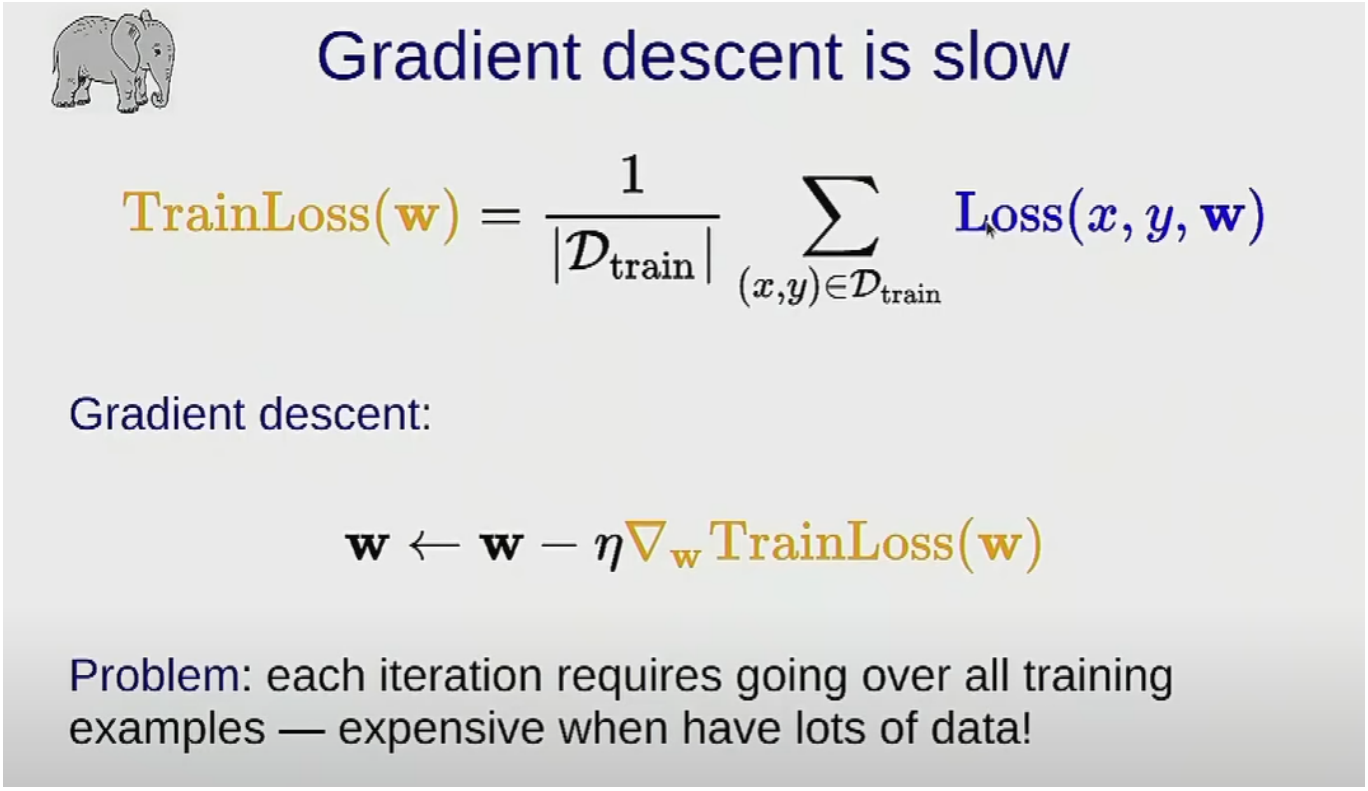
Objective function:

$$\text{TrainLoss}(\mathbf{w}) = \frac{1}{|\mathcal{D}_{\text{train}}|} \sum_{(x,y) \in \mathcal{D}_{\text{train}}} (\mathbf{w} \cdot \phi(x) - y)^2$$

Gradient (use chain rule):

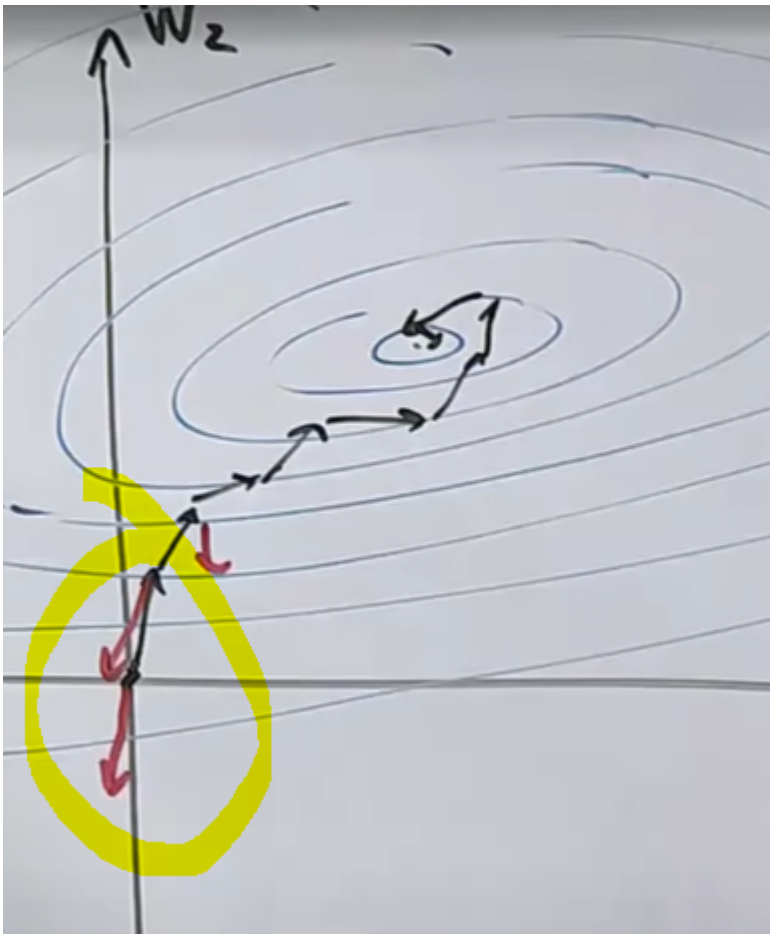
$$\nabla_{\mathbf{w}} \text{TrainLoss}(\mathbf{w}) = \frac{1}{|\mathcal{D}_{\text{train}}|} \sum_{(x,y) \in \mathcal{D}_{\text{train}}} 2(\underbrace{\mathbf{w} \cdot \phi(x) - y}_{\text{prediction}-\text{target}})\phi(x)$$

The gradient decent is great, but for morden examples with millions of examples, this can be really slow

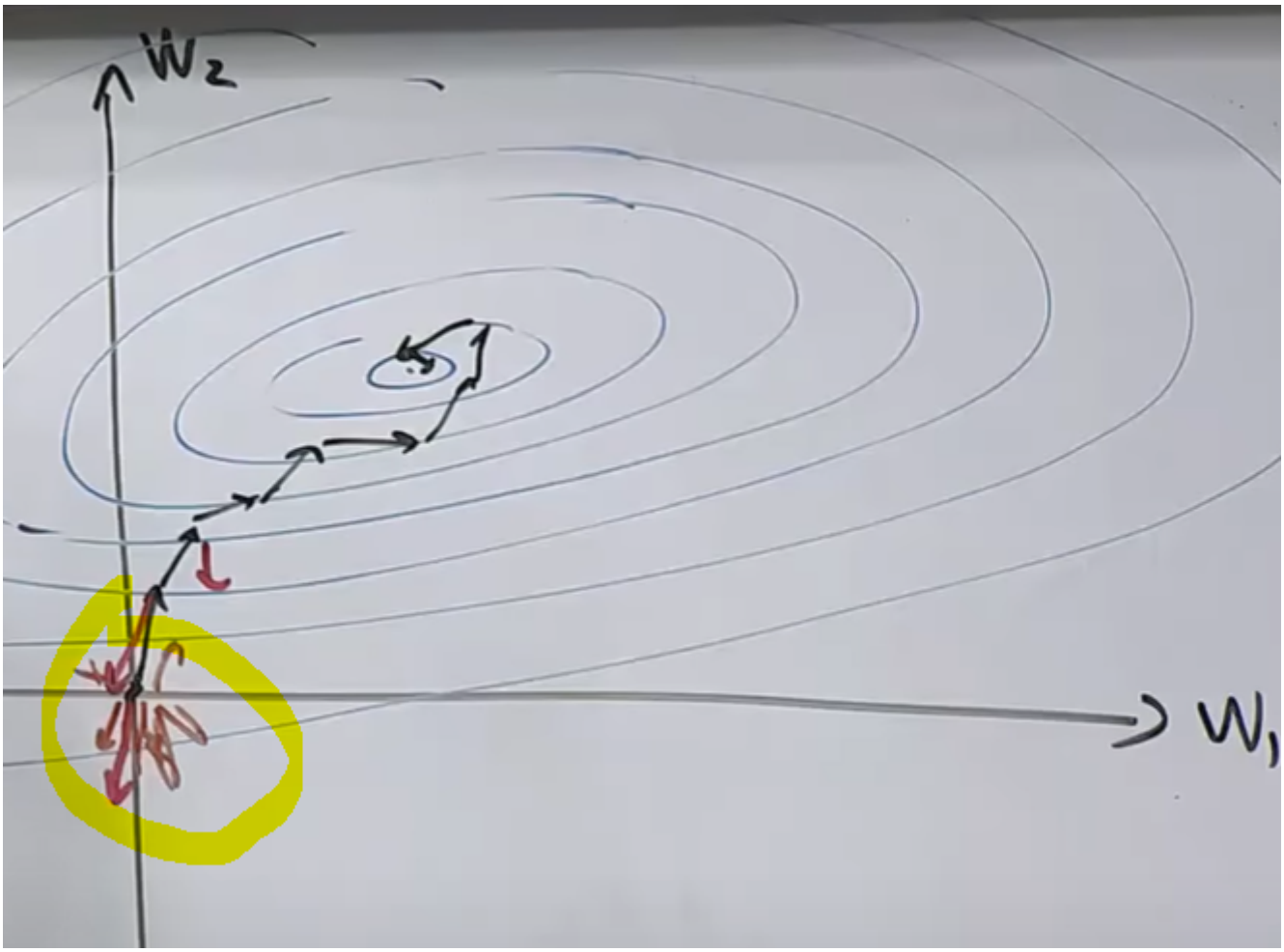


It's slow because We have go through all the points for each iterations

The idea for `Stochastic gradient descent` is that, maybe we don't have to do it



This grediant is from all the examples from all points in your trainning set.



It is a sum of different things pointing to different directions, which all averange out to this direction.

So maybe we can not agarange all of them, maybe just a couple of them, or even just pick one of them

The key idea is **It not about quality, it's about quantity**

Step size

Choose wisely, either too big or too small won't be good

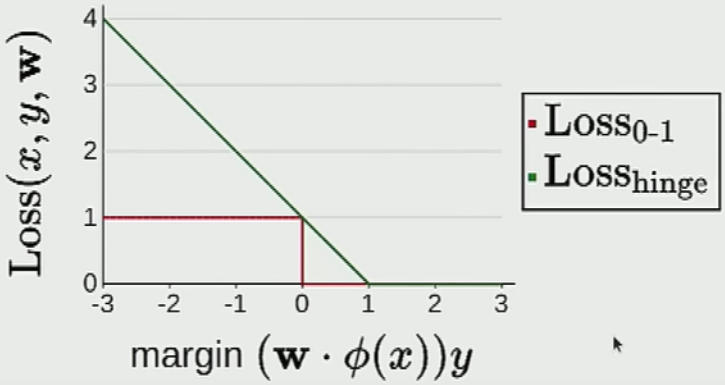
Gradient descent for zero-one classification

It won't work because the gradient is 0 lol

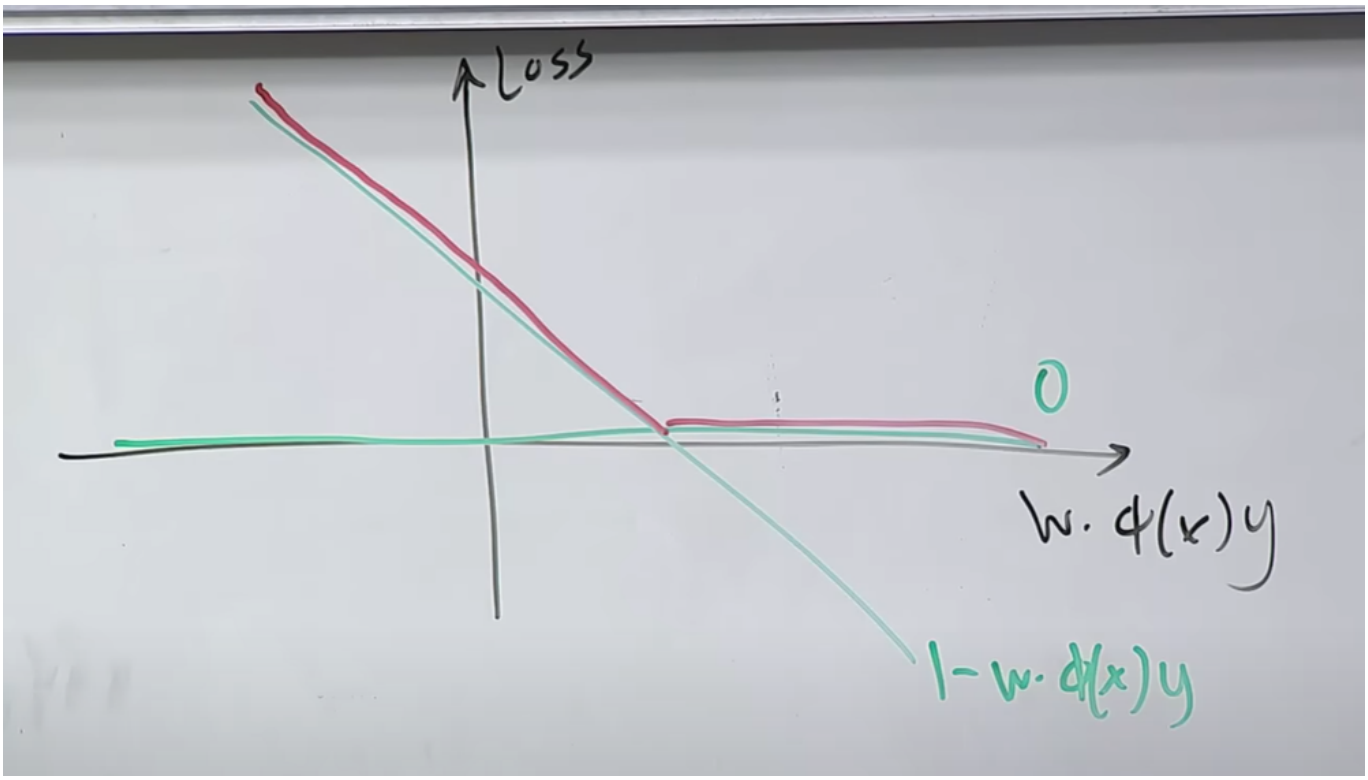
We can try to make it non-0 though

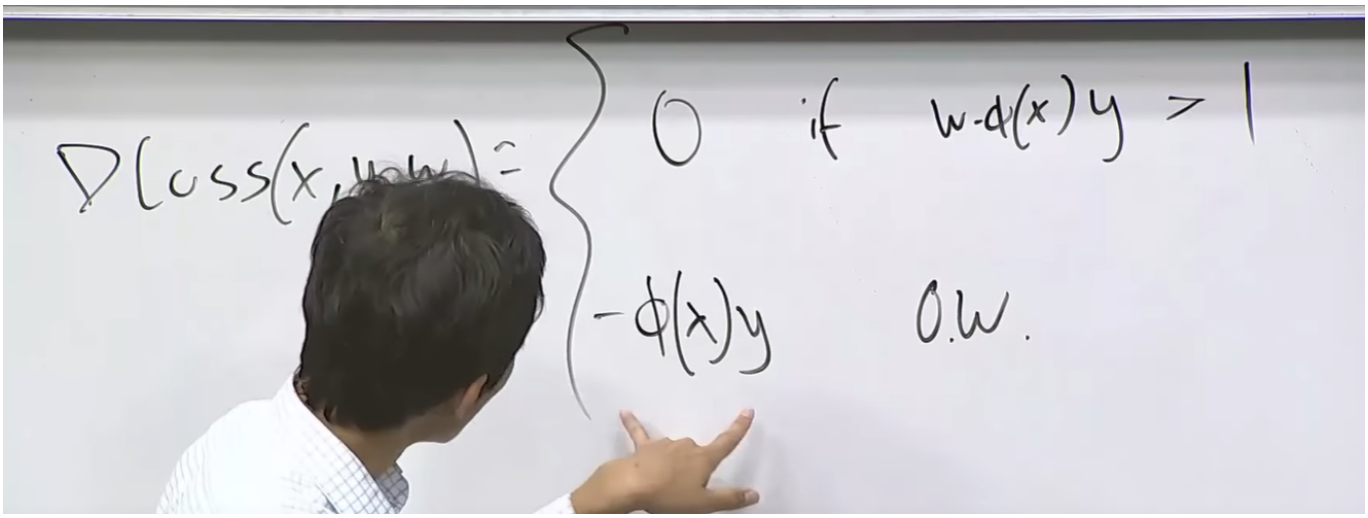
Hinge loss (SVMs)

$$\text{Loss}_{\text{hinge}}(x, y, \mathbf{w}) = \max\{1 - (\mathbf{w} \cdot \phi(x))y, 0\}$$

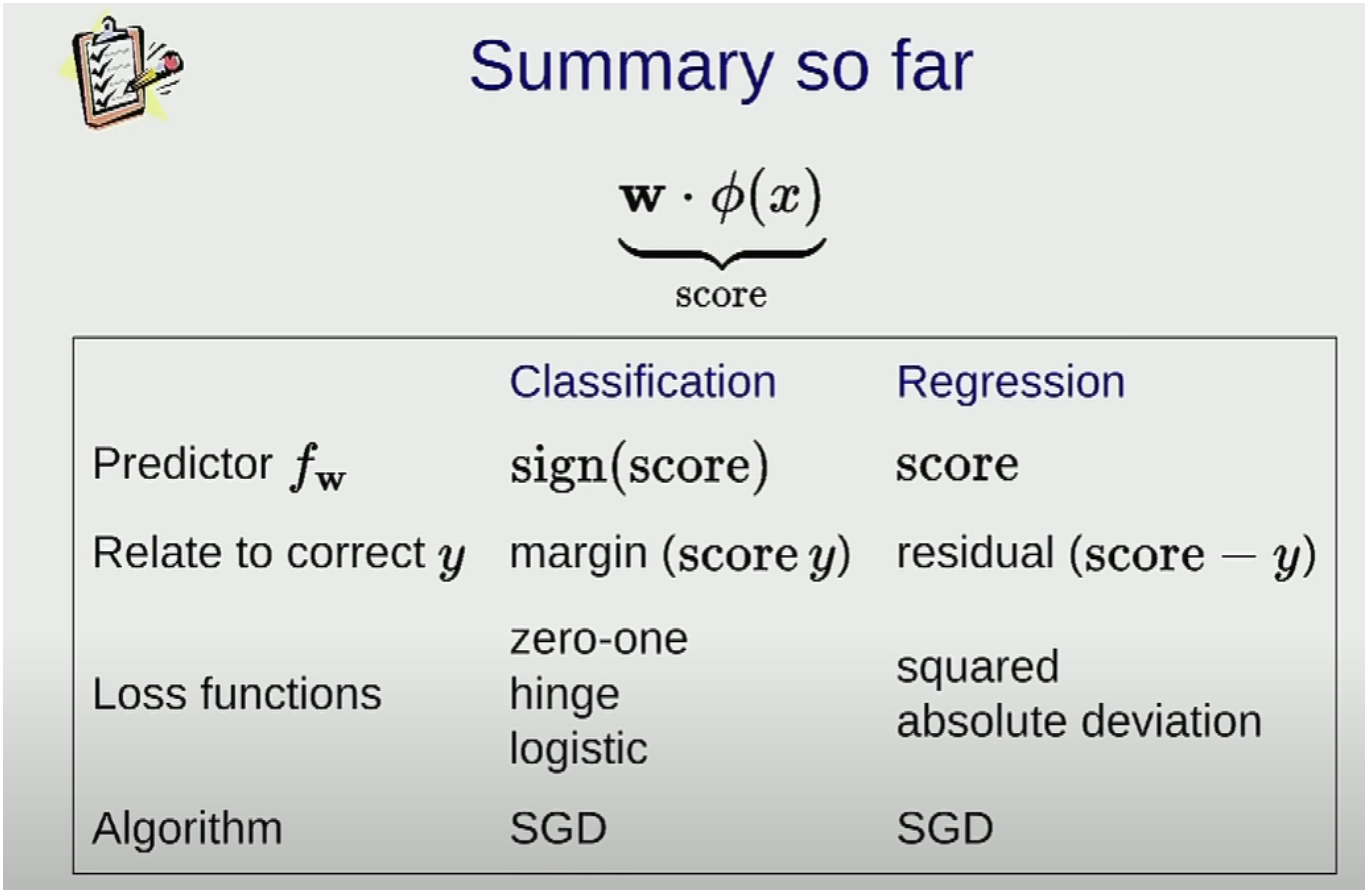


- Intuition: hinge loss upper bounds 0-1 loss, has non-trivial gradient





Summary



- To train, you have to access how well you are doing:
  - You can use margin or residual for classification or Regression respectively
- You can define loss functions
- Optimizing with SGD(which turns out to be a lot faster than GD)

```
import numpy as np

######################################################################
# Modeling: what we wanna compute
# points = [(np.array([2]),4), (np.array([4]),2)]
# d = 1
```

```
# Generate artificial data
# Let's first decide what the right answer is
true_w = np.array([1, 2, 3, 4, 5])
d = len(true_w)
points = []
for i in range(10000):
    x = np.random.randn(d)
    # Add some noise lol
    y = true_w.dot(x) + np.random.randn()
    points.append((x, y))

def F(w):
    return sum((w.dot(x) - y)**2 for x, y in points) / len(points)

def dF(w):
    #外层求导 后内层求导
    return sum(2*(w.dot(x) - y)*x for x, y in points) / len(points)

def stochasticF(w, i):
    x, y = points[i]
    return (w.dot(x) - y)**2

def stochasticDF(w, i):
    x, y = points[i]
    return 2*(w.dot(x) - y)*x
######################################################################
# Algorithm: How we compute it

# the gradient descent depends on a function, the gradient, and dementionality
def gradientDescent(F, dF, dementionality):

    w = np.zeros(dementionality)

    stepSize = 0.01
    for t in range(100000):
        errorSum = F(w)
        gradient = dF(w)
        #Gradient tells me where the function is increasing so we move to the
oppisite direction
        w = w - gradient * stepSize

        print("iteration {}: w = {}, ErrorSum F(w) = {}".format(t, w, errorSum))

def stochasticGradientDescent(sF, sDF, dementionality, n):

    w = np.zeros(dementionality)

    stepSize = 0.01
    for t in range(1000):
        for i in range(n):
            errorSum = sF(w, i)
            gradient = sDF(w, i)
            #Gradient tells me where the function is increasing so we move to the
```

```
    oppisite direction
                w = w - gradient * stepSize
                # Somehow this is not work lol
                # stepSize = 1.0/(t+1)

            print("iteration {}: w = {}, ErrorSum F(w) = {}".format(t, w, errorSum))

    # gradientDescent(F, dF, d)
    stochasticGradientDescent(stochasticF, stochasticDF, d, 10)
```