4/2

Optimization:

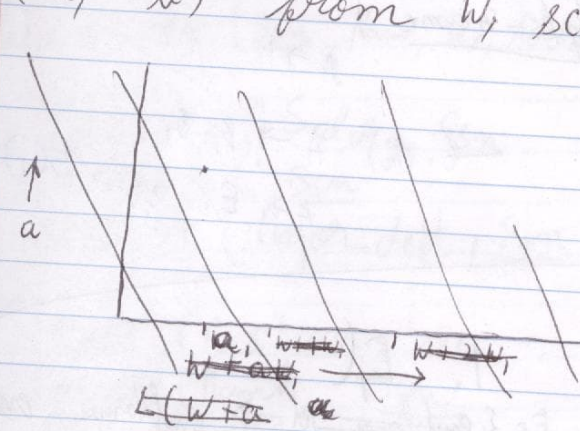
☆ This is where we calculate W!

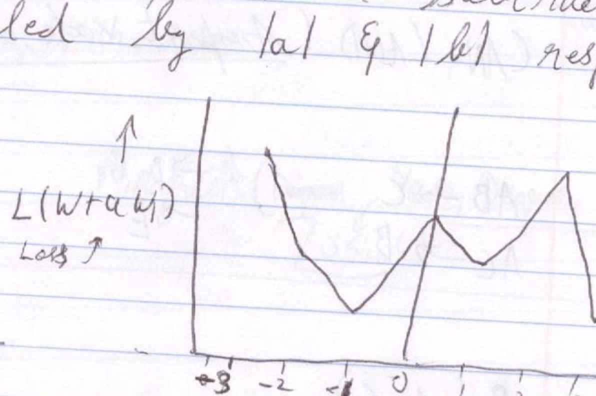
• Visualising loss function: Although this seems confusing, the plot of a loss function is just:

i) $W + aW_1$ (1-D line)
ii) $W + aW_1 + bW_2$ (2-D, colorful heatmap-like plot)

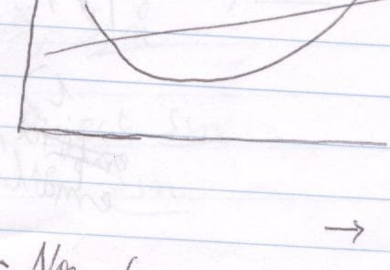
W is the initial weight. It can be given represented by a single point in space (just have it correspond to some point in space through a bijective function.)

W_1 & W_2 are matrices of same dimensions as W with values that we add (+a, +b) or subtract ~~from~~ (-a, -b) from W_1 scaled by |a| & |b| respectively.

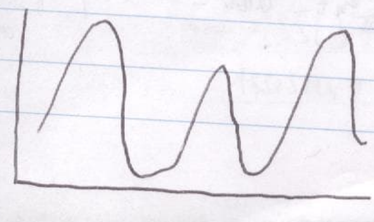




• Convex Function (Side Note)



A line segment is always above or on the graph



Non-Convex

→ Has global minimum only!

☆ Optimization Strategies:

1. ~~to~~ Random Search : Randomly search for W

```
repeat 1.. n
    W = randomWeights()
    loss = Loss(W)

    If loss < bestLoss :
        bestLoss = loss
        bestW = W
```

2. Random Local Search : Go in Random direction from W, update if W_{new} is lower loss.

```
W = randomWeights()
repeat 1.. n
    W1 = randomWeights() * stepSize        (stepSize = 0.0001)
    Wnew = W + W1
    loss = Loss(Wnew)
    If loss < bestLoss
        bestLoss = loss
        W = Wnew
```

3. Gradient Descent : Mathematically compute best direction along which W should be changed → Find direction of steepest Descent of Loss function.

Time for some calculus, kids!

ﾐ4

☆ Gradient: · Vector of partial derivatives ~~of~~ a vector of numbers
· Slopes in each of the dimensions.

$$\frac{d}{dx} f(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

☆ Calculating gradient:

1. For all values in a vector, add a small value h, see if change is negative or positive. (~~Analytical~~ Numerical Method)

```
gradient(x, f):
    original = f(x)          // f is our loss function.
    grad = []
    h = 0.00001              // h has to be very very small
                                                  (h -> 0)
    for i in 1... len(x):
        old-value = x[i]
        ~~fxh = f~~
        x[i] = x[i]+h
        fxh = f(x)           // Updated vector with x[i]+h
        grad[i] = (fxh - original) / h
        x[i] = old-value     // Revert
    return grad
```

☆ Note: X is not input data here, just some variable. Usually, (it is W, weights, we update.)

• Note how each time we update only one value in vector x by ~~th~~ h, and then get $f(x)$ with this updated vector. This is equivalent to partial derivative of loss function along that value - ~~update~~ slope for that value, in that dimension.

☆ <u>How to Use</u>:

```
W = random_weights()
df = gradient(W, loss_func)    // From prev. page

W_new = W-StepSize * df    // StepSize is variable
                           // df tells direction in which
                           loss_func increases, so we
                           move in opposite direction
```

☆ StepSize - also known as <u>"Learning Rate"</u>
Hyperparameter we tune.
Gradient tells us <u>direction</u> of <u>steepest increase</u>, but <u>not how far</u> we must <u>step in that direction</u>.

2. <u>Calculus</u> (<u>Analytical Method</u>)

$\frac{d}{dW_{y_i}} L_i$ $\quad$ i = data point ~~for~~ we are calculating loss for.

y_i = true class

W_{y_i} = Weights for true class

<u>Example</u>: SVM Hinge Loss Gradient:

$$\nabla_{W_{y_i}} L_i = \frac{d}{dW_{y_i}} \left[\sum_{y_i \neq j} \max(0, w_j^T x_i - w_{y_i}^T x_i + \Delta)\right]$$

$$= -\left(\sum_{j \neq y_i} \mathbb{I}(w_j^T x_i - w_{y_i}^T x_i + \Delta > 0)\right) x_i$$

Mini-batch Gradient Descent: Compute gradient over batches / samples of training data, in large-scale applications.

→ May also be referred to as "Stochastic Gradient Descent".