Linear and Logistic Regression

CS 759/859 Natural Language Processing Lecture 7

Samuel Carton, University of New Hampshire

Last lecture

Key idea: Dimension reduction

Concepts

- Dimensionality of data
- Variance of data
- Principle components
- Matrix factorization
- SVD and PCA
- Application to clustering

Toolkits

- Scikit-learn for SVD

Parametric classification

Parametric classification

With the K-nearest neighbor classifier, we learned how to take a big pile of training vectors ***X*** with known labels **y**, and use them to classify an unknown vector x with some prediction $\hat{y}$.

But all we're really doing there is putting the training vectors in a pile, and running a nearest-neighbor search over them for each new x

- Very inefficient

Wouldn't it be nice if we could learn some mathematical formula from the training data which could just take in a vector and directly spit out a prediction, without having to go over all the data every time?

Toy dataset

Goal: Can we come up with some vector ***w***, with one number per word, where if we take the dot product of w and each vector, we get something close to the true label?

```
review_0 = "The film was a delight--I was riveted."
review_1 = "It's the most delightful and riveting movie."
review_2 = "It was a terrible flick, the worst I have ever seen."
review_3 = "I have a feeling the film was recut poorly."
reviews = [review_0, review_1, review_2, review_3]

labels = [1, 1, 0, 0]
```

	and	delight	ever	feel	film	flick	have	it	most	movi	poorli	recut	rivet	seen	terribl	the	wa	worst	label
0	0	1	0	0	1	0	0	0	0	0	0	0	1	0	0	1	2	0	1
1	1	1	0	0	0	0	0	1	1	1	0	0	1	0	0	1	0	0	1
2	0	0	1	0	0	1	1	1	0	0	0	0	0	1	1	1	1	1	0
3	0	0	0	1	1	0	1	0	0	0	1	1	0	0	0	1	1	0	0

Manual parameters/predictions

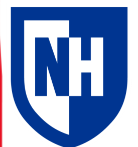
```python
import numpy as np
manual_parameters = np.array([[
    0, #and
    1, #delight
    0, #ever
    0,#feel
    0, #film
    0, #flick
    0, #have
    0, #it
    0, #most
    0, #movi
    -1, #poorli
    0, #recut
    1, #rivet
    0, #seen
    -1, #terribl
    0, #the
    0, #wa
    -1, #worst
    ]])
```

Intuitive idea: put a 1 on the words we know are positive, and a -1 on the ones we know are negative.

But how to squeeze the predictions to between 0 and 1?

```python
1  toy_vector_df['manual_labels'] = manual_predicted_labels
2  toy_vector_df
```

	and	delight	ever	feel	film	flick	have	it	most	movi	poorli	recut	rivet	seen	terribl	the	wa	worst	label	manual_labels
0	0	1	0	0	1	0	0	0	0	0	0	0	1	0	0	1	2	0	1	2
1	1	1	0	0	0	0	0	1	1	1	0	0	1	0	0	1	0	0	1	2
2	0	0	1	0	0	1	1	1	0	0	0	0	0	1	1	1	1	1	0	-2
3	0	0	0	1	1	0	1	0	0	0	1	1	0	0	0	1	1	0	0	-1



Logistic function

To solve this problem, we are going to wrap our original function, which we will call *f*, in a **logistic function**

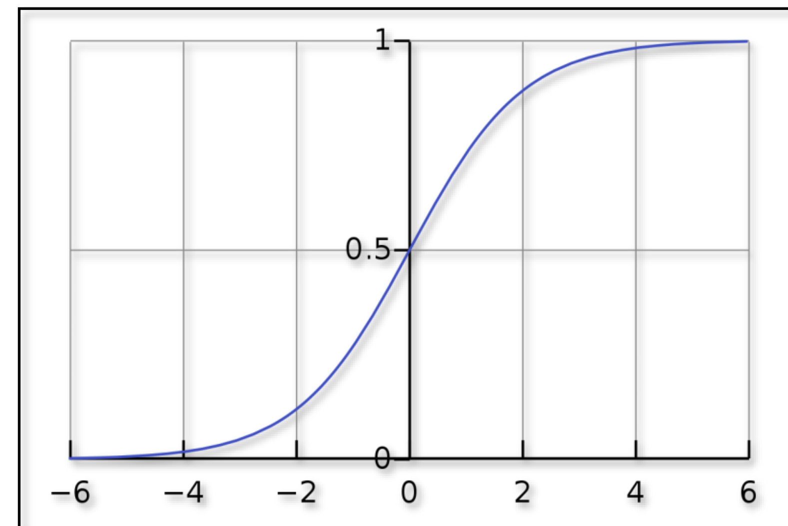
$$\sigma(x) = \frac{1}{1+e^{-x}} \qquad \sigma(f(x)) = \frac{1}{1+e^{-(Wx+b)}}$$

One nice thing about it is that it is easy to differentiate because of the property that:

$$\frac{d}{dx}\sigma(x) = \sigma(x)(1-\sigma(x))$$

So if we call our original function *f*:

$$\frac{d}{dx}\sigma(f(x)) = \sigma(f(x))\big(1-\sigma(f(x))\big)f'(x) = W\sigma(Wx+b)(1-\sigma(Wx+b))$$



https://en.wikipedia.org/wiki/Logistic_function

Sigmoid-ed manual predictions

Looking a lot better, but could we have learned this automatically from the data?

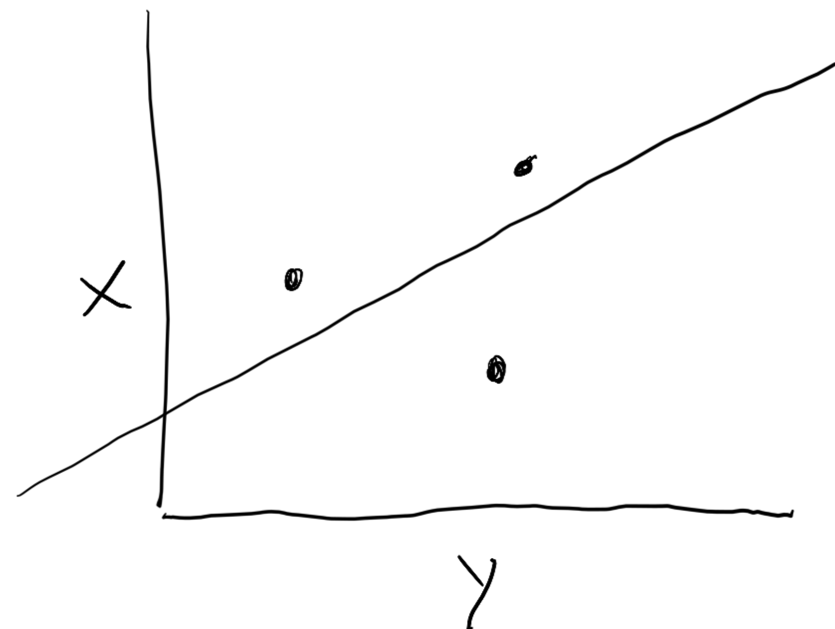
	and	delight	ever	feel	film	flick	have	it	most	movi	...	recut	rivet	seen	terribl	the	wa	worst	label	manual_labels	sigmoid_labels
0	0	1	0	0	1	0	0	0	0	0	...	0	1	0	0	1	2	0	1	2	0.880797
1	1	1	0	0	0	0	0	1	1	1	...	0	1	0	0	1	0	0	1	2	0.880797
2	0	0	1	0	0	1	1	1	0	0	...	0	0	1	1	1	1	1	0	-2	0.119203
3	0	0	0	1	1	0	1	0	0	0	...	1	0	0	0	1	1	0	0	-1	0.268941

4 rows × 21 columns

Linear regression

Linear regression

Basic idea: given some points in N-dimensional space, find a "line of best fit" that is as close as possible to those points.



When the points are text:

- N = vocabulary size
- Examples:
  - Grading essays 0-100
  - Scoring text complexity

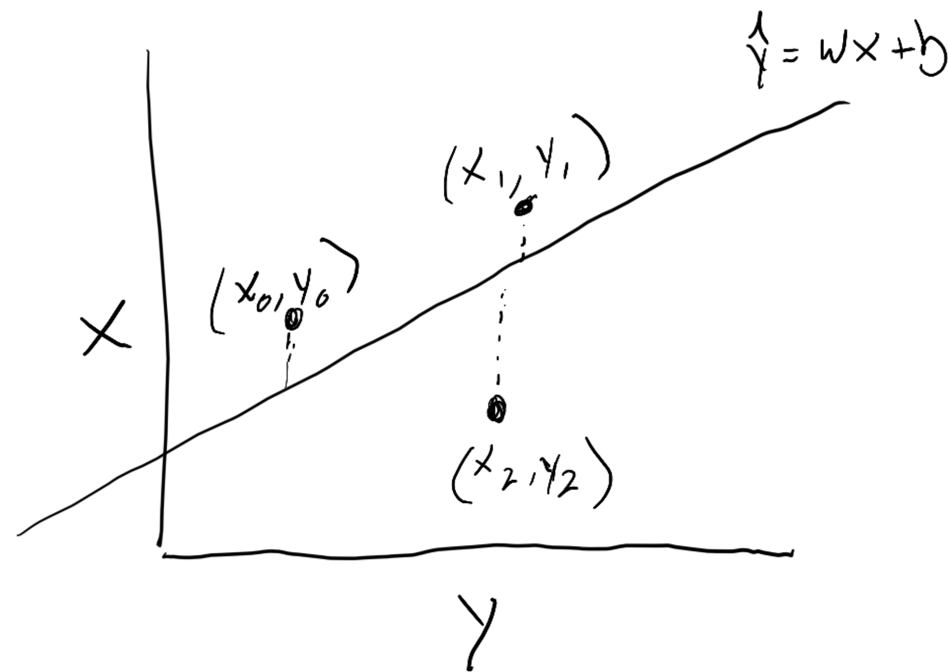
Linear regression

Mathematically, what we're trying to do is figure out some function:

$\hat{y} = Wx + b$

…where W and b are values such that $\hat{y}$ tends to be close to y for any given x.

Very common in ML to refer to predicted output as $\hat{y}$ and true output as y.



Loss function

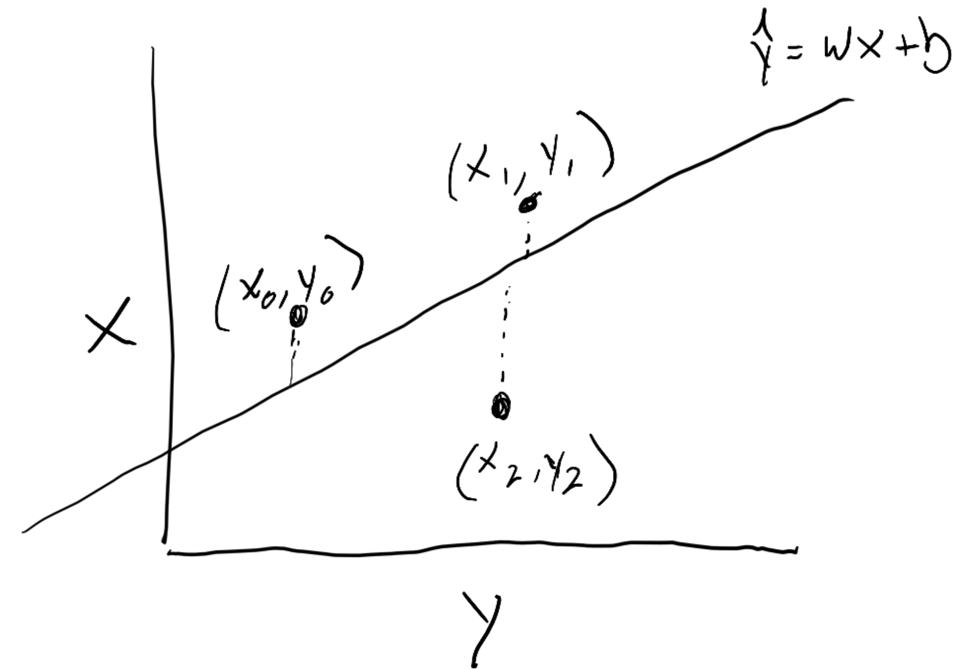
We generally articulate this goal with a **loss function** that describes the value we're trying to minimize with our choice of W and b.

AKA "Objective function"

It's very typical to minimize **squared loss** between expected and true output: $\sum_i (\hat{y}_i - y_i)^2$

That would give us a loss function of:

$$L(W, b) = \sum_i (\hat{y}_i - y_i)^2$$
$$= (\hat{y}_0 - y_0)^2 + (\hat{y}_1 - y_1)^2 + (\hat{y}_2 - y_2)^2$$
$$= (Wx_0 + b - y_0)^2 + (Wx_1 + b - y_1)^2 + (Wx_2 + b - y_2)^2$$

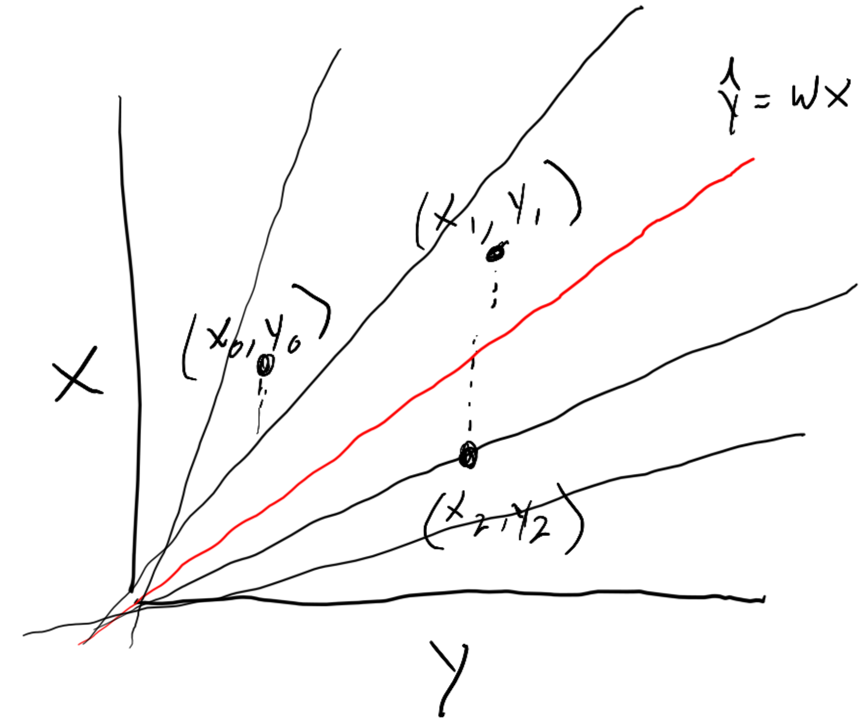


Simple example

To show how we can solve this, I'll use a simple example with **no intercept** (b)

So the loss function is:

$$L(W,b) = \sum_i (\hat{y}_i - y_i)^2$$
$$= (\hat{y}_0 - y_0)^2 + (\hat{y}_1 - y_1)^2 + (\hat{y}_2 - y_2)^2$$
$$= (Wx_0 - y_0)^2 + (Wx_1 - y_1)^2 + (Wx_2 - y_2)^2$$
$$= (W^2{x_0}^2 - 2Wx_0y_0 + {y_0}^2)$$
$$+(W^2{x_1}^2 - 2Wx_1y_1 + {y_1}^2)$$
$$+(W^2{x_2}^2 - 2Wx_2y_2 + {y_2}^2)$$



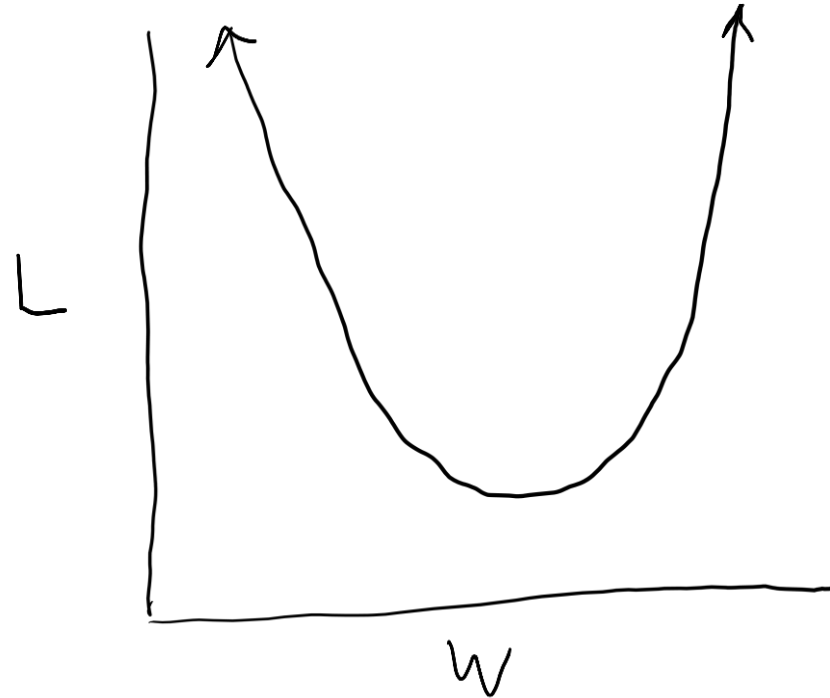
$$= W^2({x_0}^2 + {x_1}^2 + {x_2}^2) - 2W(x_0y_0 + x_1y_1 + x_2y_2) + ({y_0}^2 + {y_1}^2 + {y_2}^2)$$

Simple example

If the loss function for W is this:

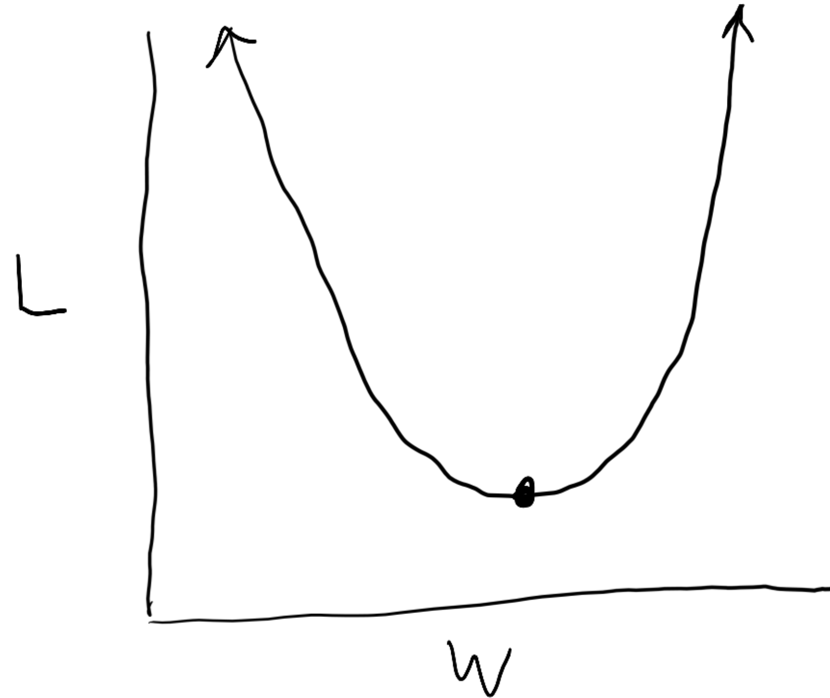
$$L(W) = W^2({x_0}^2 + {x_1}^2 + {x_2}^2)$$
$$-2W(x_0 y_0 + x_1 y_1 + x_2 y_2)$$
$$+({y_0}^2 + {y_1}^2 + {y_2}^2)$$

Then the graph of L as a function of W looks like this:



Simple example

It's pretty obvious what value of *W* will minimize the loss here.



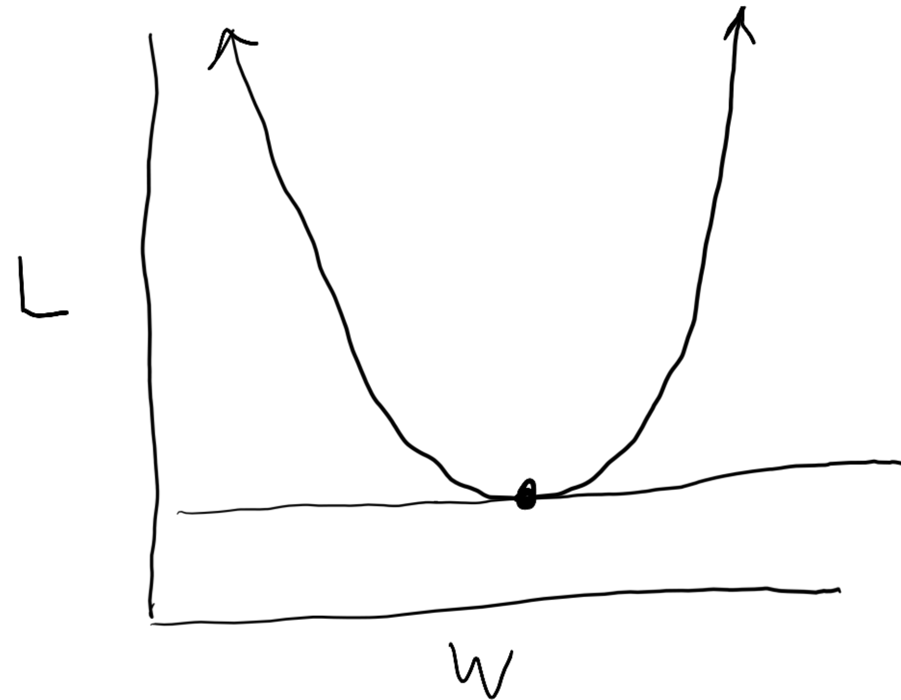
Simple example

What may be less obvious is that this also happens to be the point where the derivative of L with respect to W is 0.

$$\frac{dL}{dW} = 0$$

In some sense it is the bottom of a pit.

Gradient descent is the process of gradually following the slope of the function down to these pit-bottoms

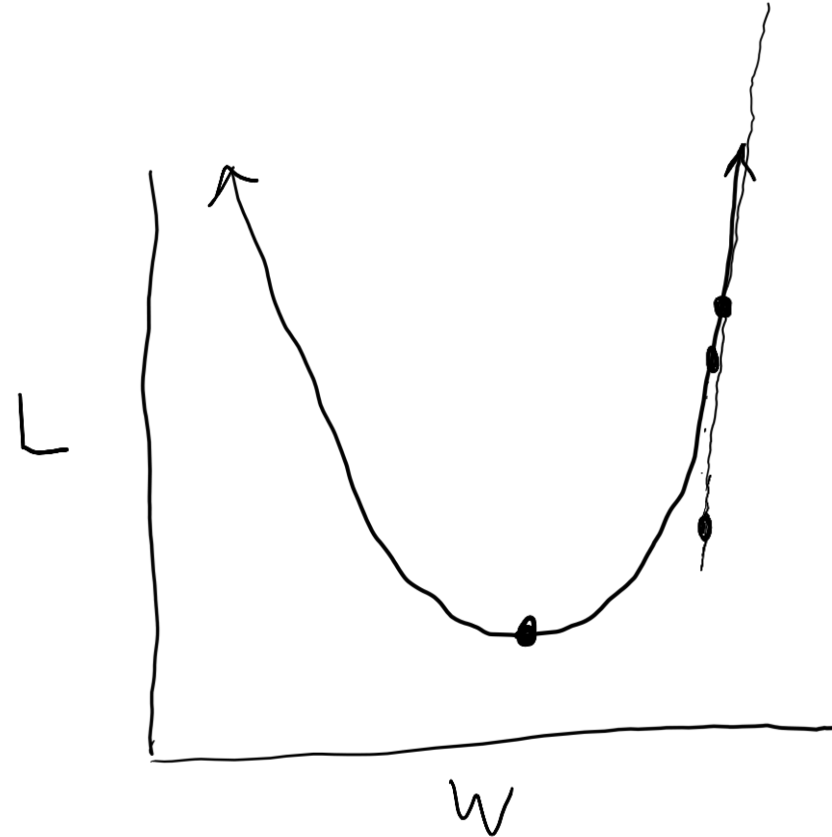


Simple example

In the most simple case, we pick a random point on the function and find the slope (derivative)

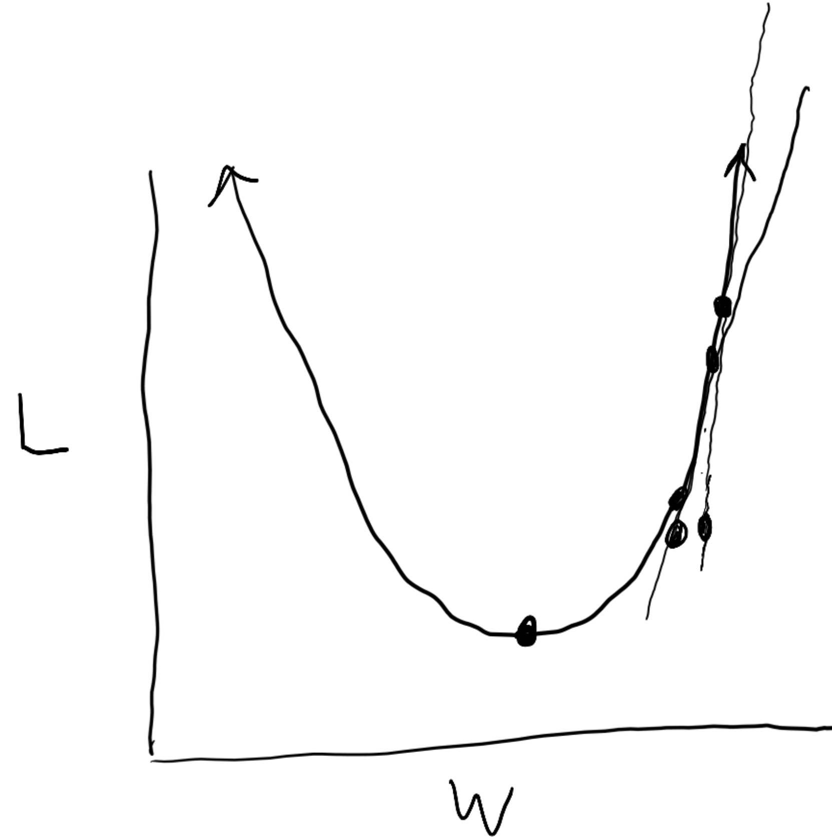
Then we move some incremental distance in the direction that **reduces** the value of L (left in this case)

This increment that we move each step is called the **learning rate**



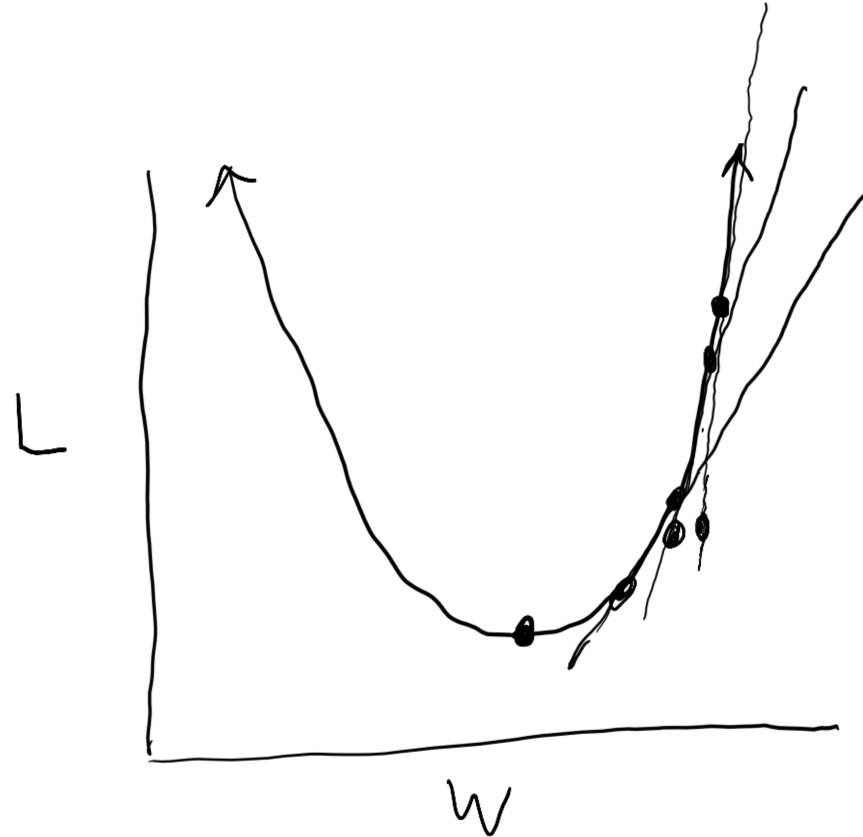
Simple example

Then we calculate the slope again at this new point and move one increment in the reducing-L direction (still left).



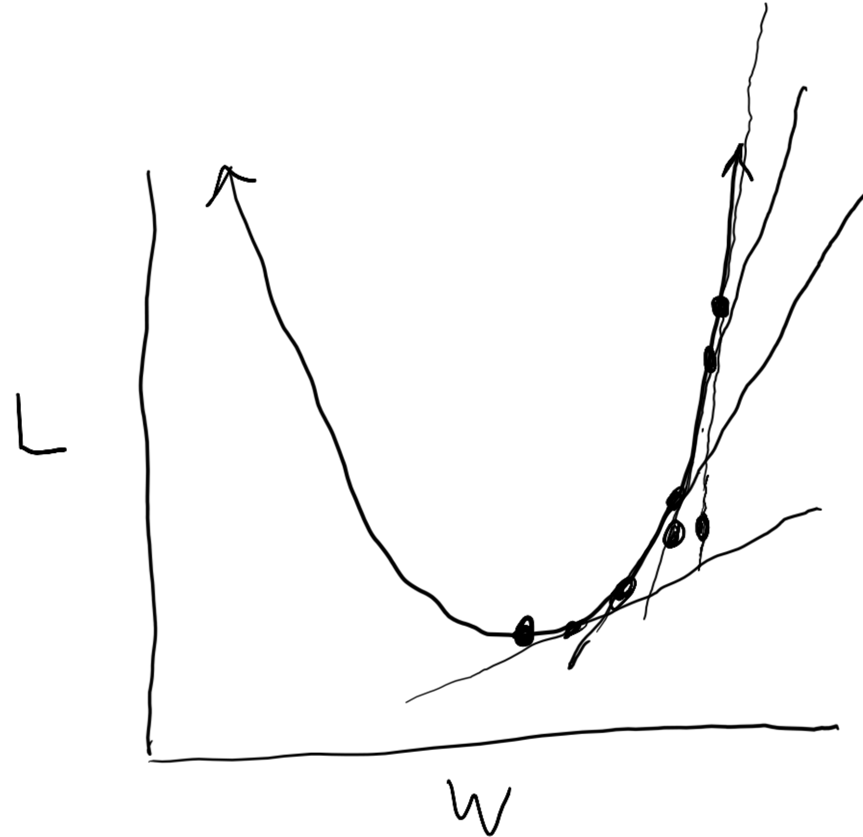
Simple example

And we keep doing that…



Simple example

And keep doing that…



Simple example

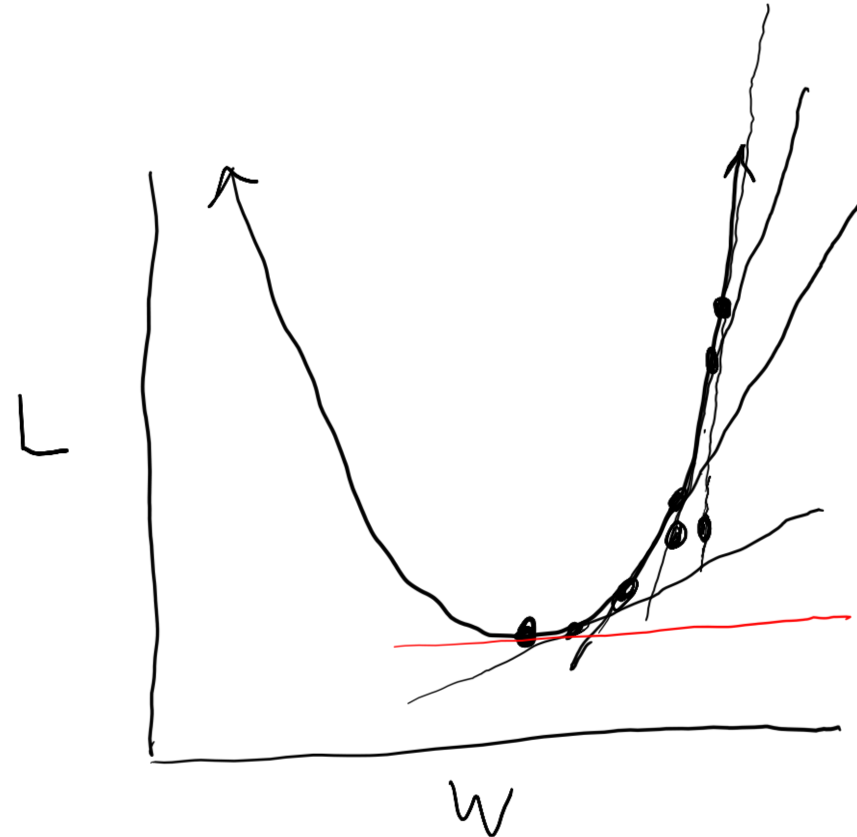
Until we hit a point on W where the slope seems to have levelled out

That is, $\frac{dL}{dW} = 0$

And we conclude that we've found the value of W that minimizes L

Challenge question:

- What if the learning rate is too high?
- What if it is too low?



Adding back the intercept

So what if our function **does** have an intercept?

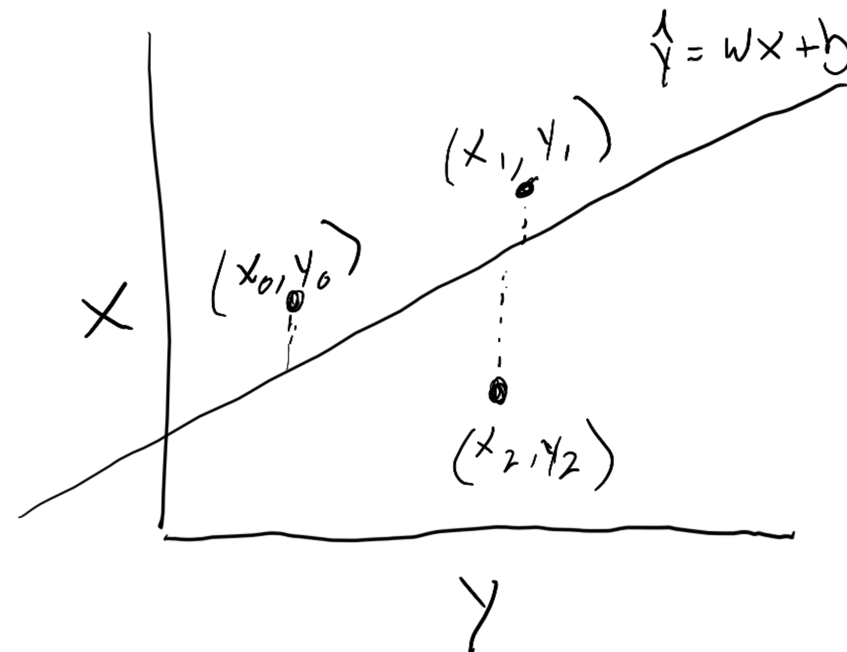
$$L(W,b) = \sum_i (\hat{y}_i - y_i)^2$$

$$= (\hat{y}_0 - y_0)^2 + (\hat{y}_1 - y_1)^2 + (\hat{y}_2 - y_2)^2$$

$$= (Wx_0 + b - y_0)^2 + (Wx_1 + b - y_1)^2 + (Wx_2 + b - y_2)^2$$

$$= (W^2{x_0}^2 - 2Wx_0y_0 + 2Wbx_0 - b2y_0 + {y_0}^2 + b^2)$$
$$+(W^2{x_1}^2 - 2Wx_1y_1 + 2Wbx_1 - b2y_1 + {y_1}^2 + b^2)$$
$$+(W^2{x_2}^2 - 2Wx_2y_2 + 2Wbx_2 - b2y_2 + {y_2}^2 + b^2)$$

$$= W^2({x_0}^2 + {x_1}^2 + {x_2}^2)$$
$$-2W(x_0y_0 + x_1y_1 + x_2y_2)$$
$$+2Wb(x_0 + x_1 + x_2)$$
$$-2b(y_0 + y_1 + y_2)$$
$$+({y_0}^2 + {y_1}^2 + {y_2}^2)$$
$$+3b^2$$

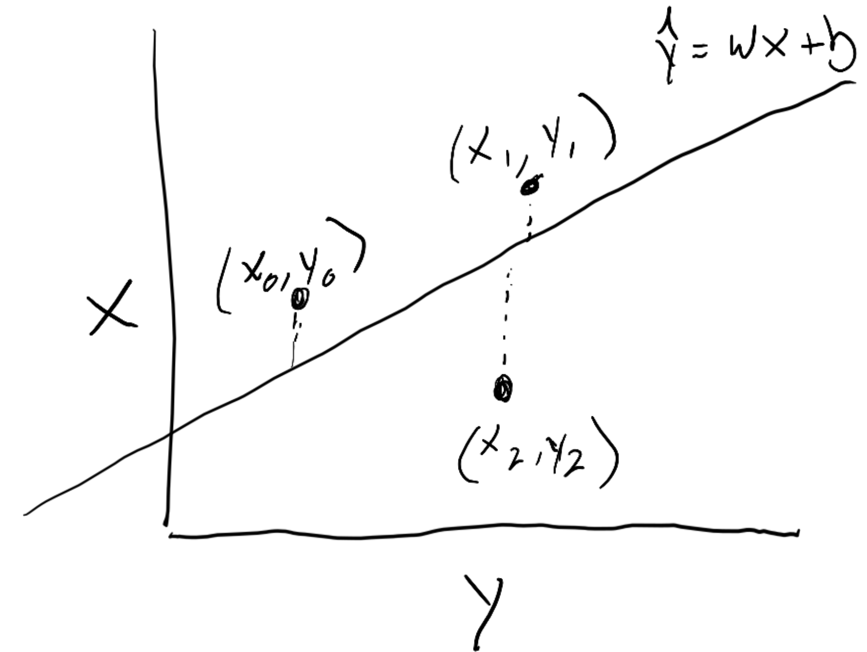


Adding back the intercept

So what if our function **does** have an intercept?

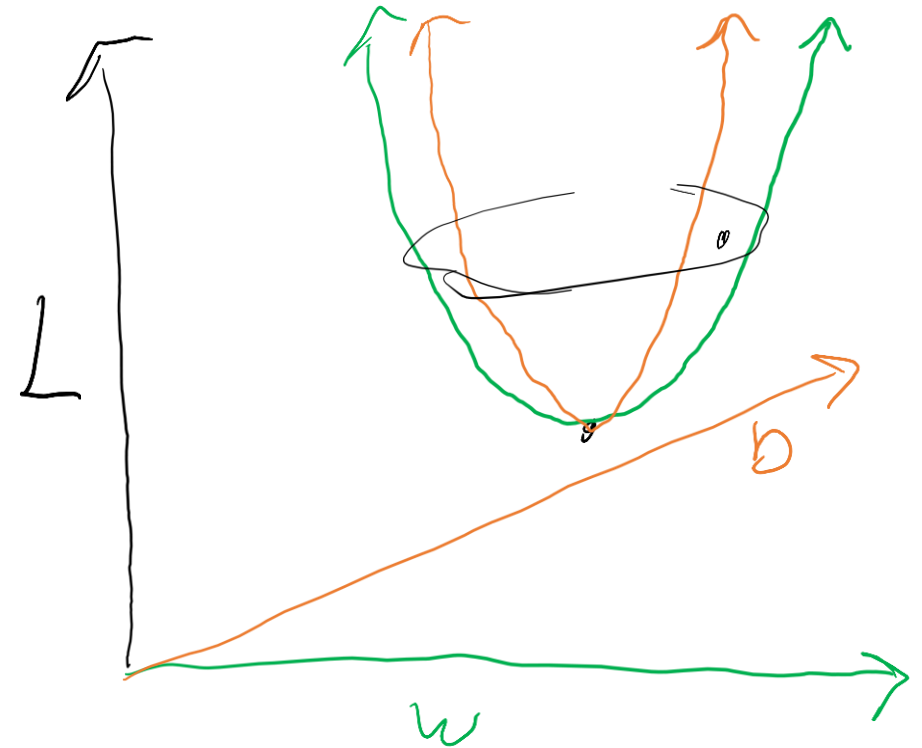
$$
\begin{aligned}
L(W,b) = {} & W^2({x_0}^2 + {x_1}^2 + {x_2}^2) \\
& -2W(x_0y_0 + x_1y_1 + x_2y_2) \\
& +2Wb(x_0 + x_1 + x_2) \\
& -2b(y_0 + y_1 + y_2) \\
& +({y_0}^2 + {y_1}^2 + {y_2}^2) \\
& +3b^2
\end{aligned}
$$

More complicated, but the key thing is that L is still just a quadratic function of W and b



Adding back the intercept

So the loss becomes a 2-dimensional function, and we're trying to find a value for W and a value for b, which, taken together, minimize L



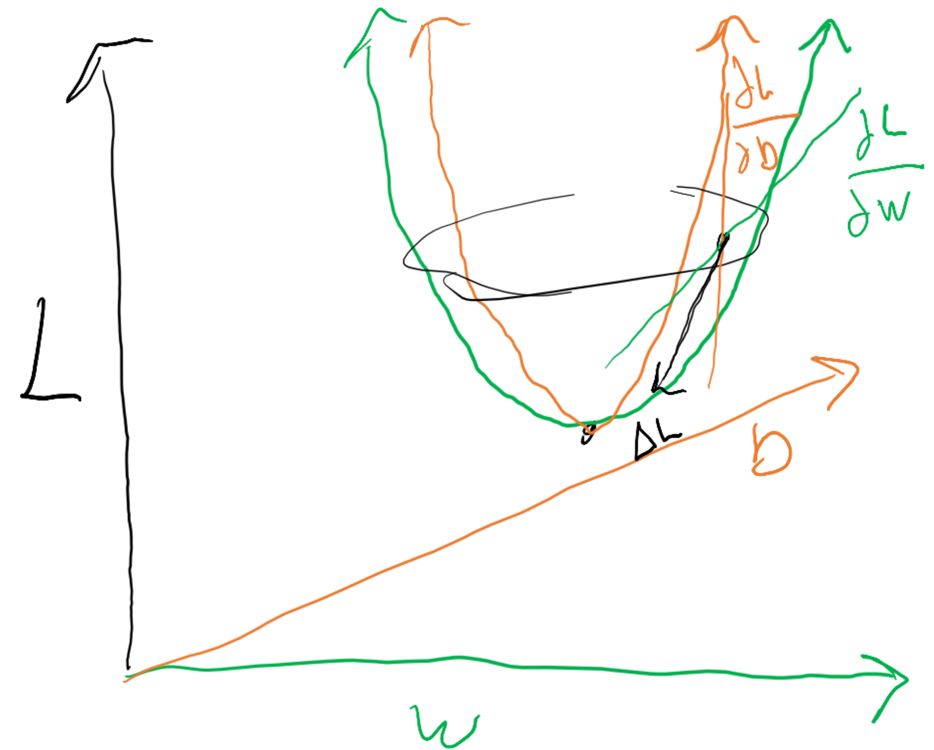
Adding back the intercept

We can still use gradient descent though!

Only now, instead of following the **derivative** $\frac{dL}{dW}$ of L with respect to W to the bottom…

We now follow a **vector** composed of the partial derivates of L with respect to W and b: $\left(\frac{\partial L}{\partial W}, \frac{\partial L}{\partial b}\right)$

We call this vector the **gradient** of L with respect to W and b, and usually denote it with the Δ symbol, e.g. Δ_L(W,b)



Gradient descent

Gradient descent is the method by which all neural nets are trained.

It works* in any situation where it is possible to calculate the gradient of the loss with respect to the model parameters

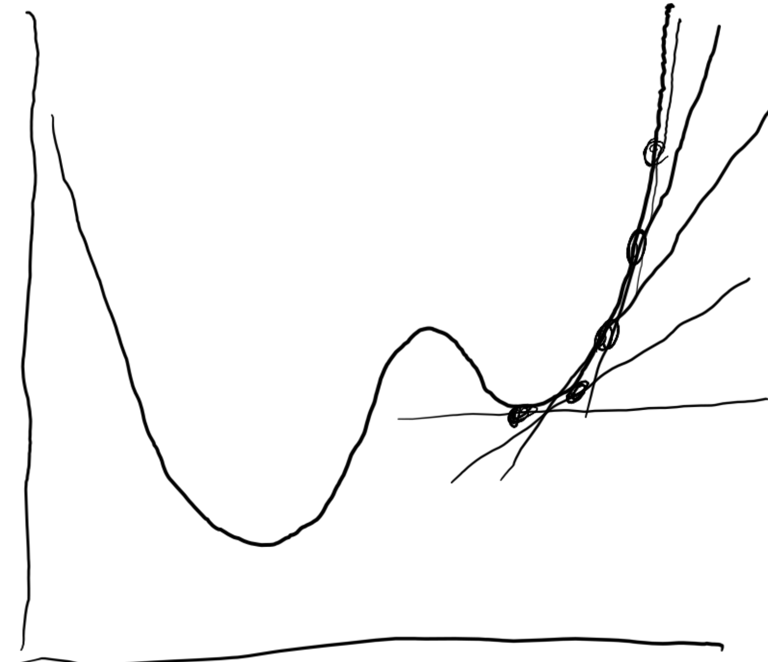
- $\hat{y} = Wx + b$ has two parameters: W and b
- ChatGPT has 175 billion parameters

*: it works more or less well depending on the shape of the loss function.

- If the function is a nice **convex** “bucket”, then it will **always** find the global minimum.
- But this is not usually true

Local minima

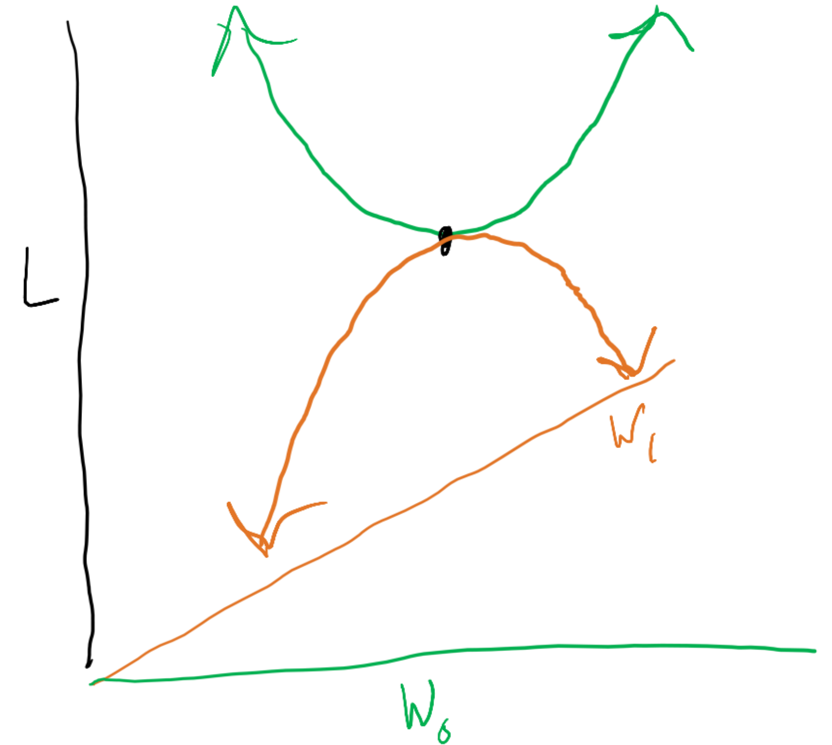
One issue in gradient descent is “local minima” which are false “dips” the gradient descent can get stuck in.



Saddle points

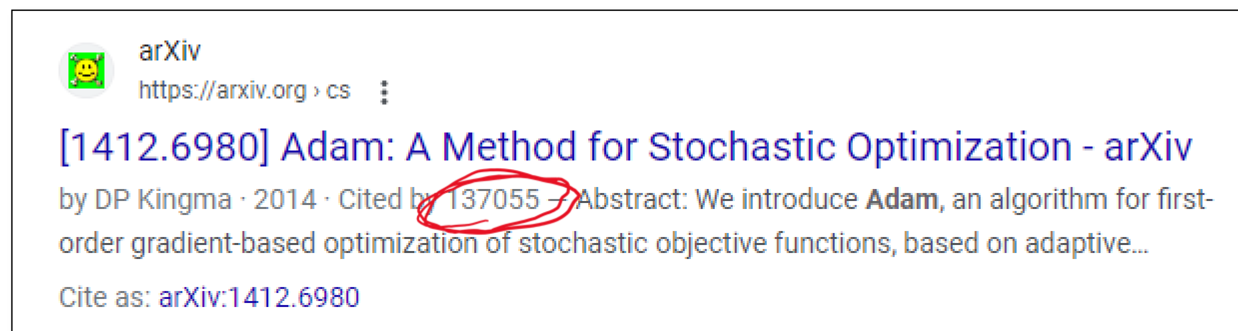
Another issue is “saddle points” which represent a minimum for one parameter but a maximum for a different one.

The gradient for both can be zero here… but it’s not necessarily a very good solution.



Advanced gradient descent

Advanced gradient descent algorithms have various tricks to help them avoid local minima and other issues



Most popular: **Adam**

- Uses "momentum" to learn adaptive learning rate for each parameter
- Generally the default choice for optimizing any arbitrary neural net

Long story short: just use Adam for everything

- Unless you have a good reason not to

Logistic regression

Logistic regression

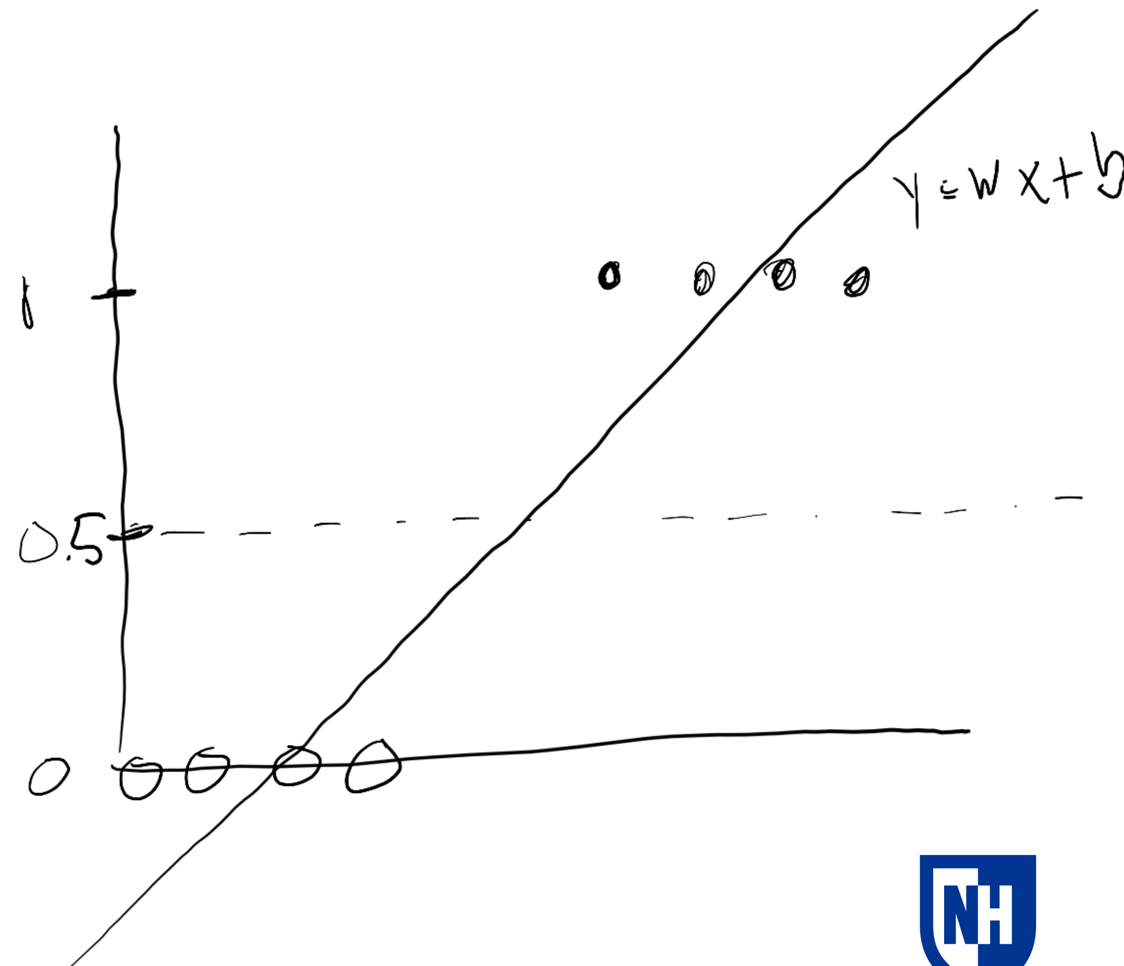
Generally in NLP we're more interested in **classification** than regression

- Mapping input x's to a discrete category rather than a continuous value

Linear regression not ideal for this

We can do it hackily with a threshold on the predicted value

- But this has problems



Logistic function

To solve this problem, we are going to wrap our original function, which we will call *f*, in a **logistic function**

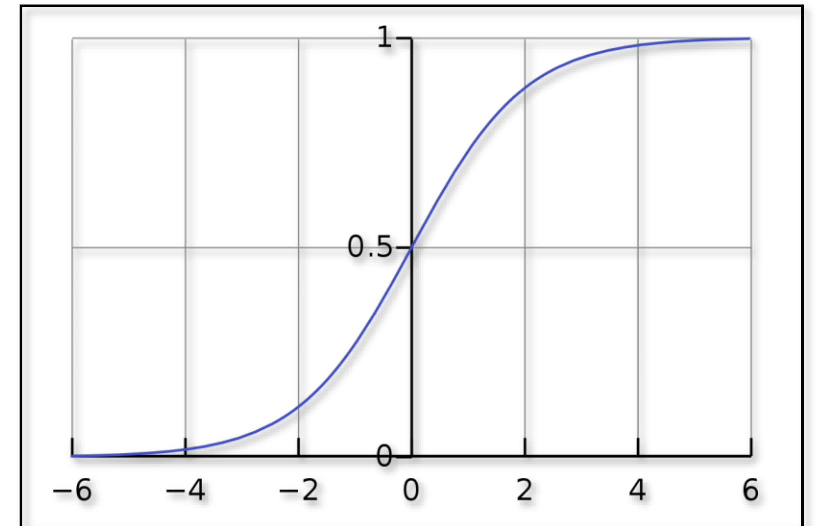
$$\sigma(x) = \frac{1}{1 + e^{-x}} \qquad \sigma(f(x)) = \frac{1}{1 + e^{-(Wx+b)}}$$

One nice thing about it is that it is easy to differentiate because of the property that:

$$\frac{d}{dx}\sigma(x) = \sigma(x)(1 - \sigma(x))$$

So if we call our original function *f*:

$$\frac{d}{dx}\sigma(f(x)) = \sigma(f(x))\big(1 - \sigma(f(x))\big)f'(x) = W\sigma(Wx + b)(1 - \sigma(Wx + b))$$

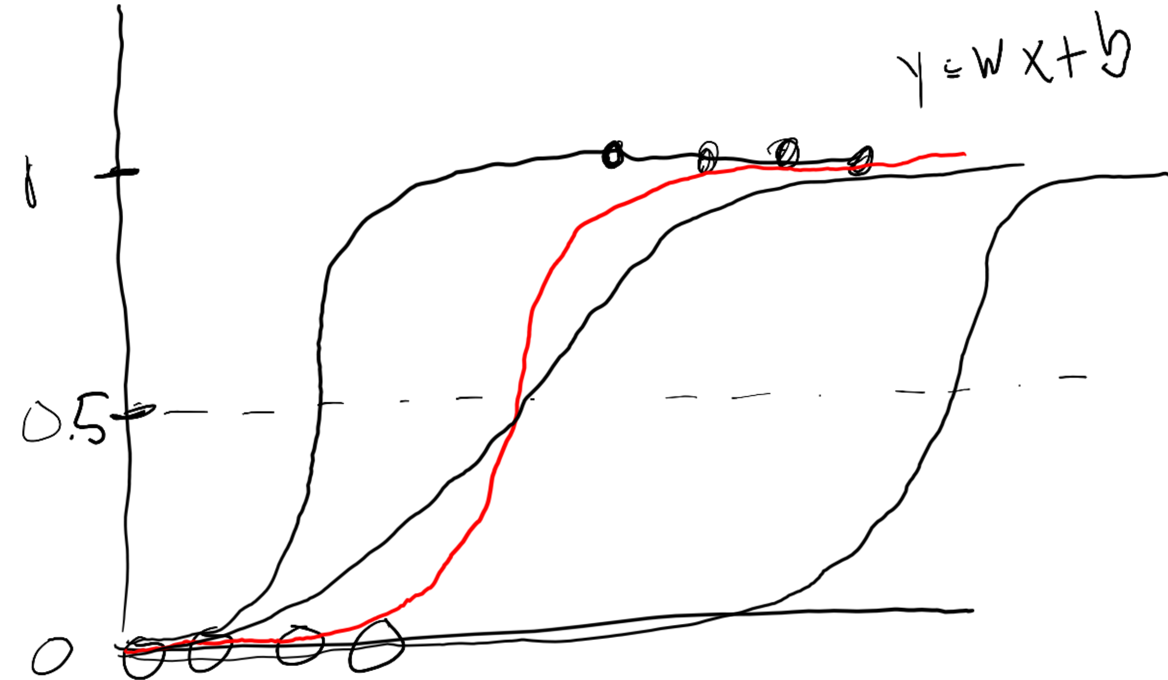


https://en.wikipedia.org/wiki/Logistic_function

Logistic regression

So now instead of trying to fit a straight line to the data, we're trying to choose W and b to fit this S-shaped logistic curve to the data

Different choices for W and b change how steep the curve is and where it is centered.



Gradient descent for logistic regression

I won't do the full derivation, but:

- The function we're trying to fit is differentiable
- Which means we can create a differentiable loss function
- Which means we can do gradient descent!

However: mean squared error is not always convex for logistic regression

So we typically use **cross-entropy** loss as our objective:

- Sum across possible classes of true value for that class multiplied by predicted log-probability of that class

$$L(y, \hat{y}) = \sum_{c} y_c \log(\hat{y}_c)$$

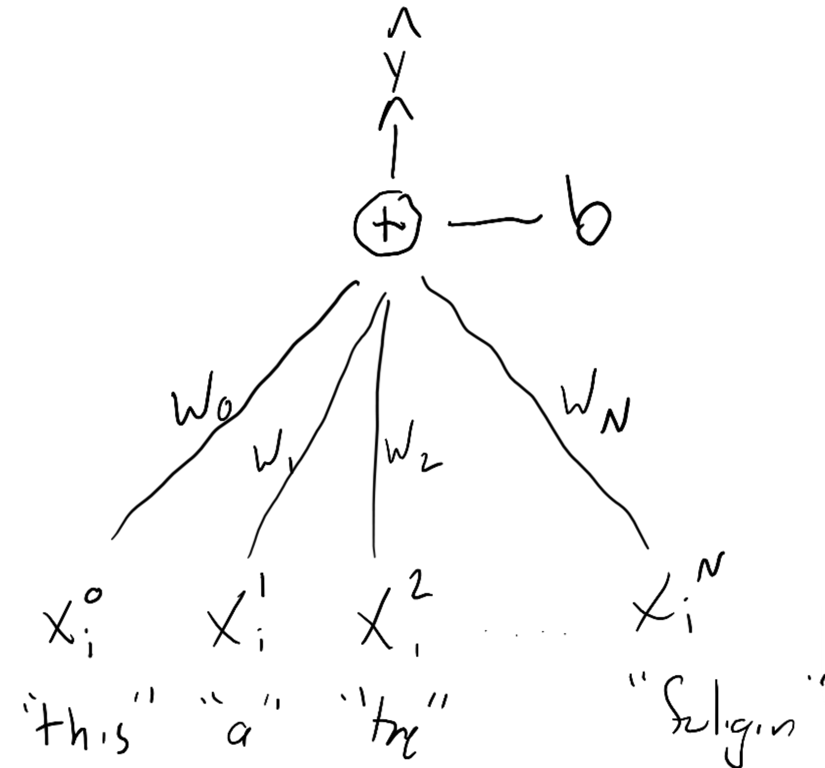
More detailed discussion available in Speech and Language Processing chapter 5: https://web.stanford.edu/~jurafsky/slp3/5.pdf

Visualizing linear regression

You can think of linear regression as a vector operation between matrices of x’s, W’s and ys

$$\begin{bmatrix} x_0^0 & x_0^1 & x_0^2 & \cdots & x_0^N \\ x_1^0 & x_1^1 & x_1^2 & \cdots & x_1^N \\ & & \ddots & & \\ x_M^0 & x_M^1 & x_M^2 & \cdots & x_M^N \end{bmatrix} \begin{bmatrix} W_0 \\ W_1 \\ \vdots \\ W_N \end{bmatrix} + b = \begin{bmatrix} \hat{y}_0 \\ \hat{y}_1 \\ \vdots \\ \hat{x}_M \end{bmatrix}$$

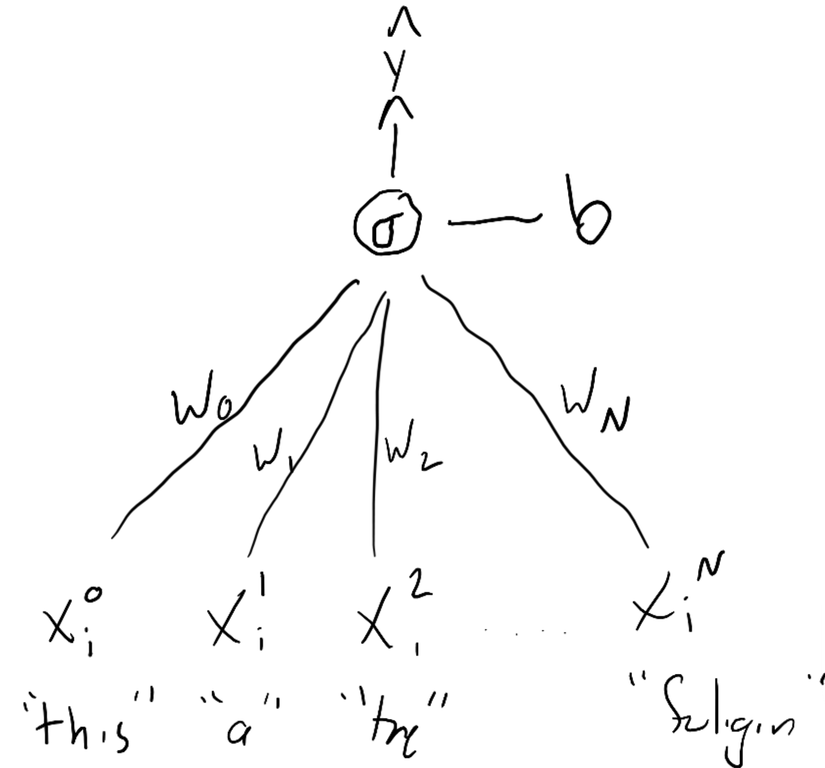
Or in a graphical form which shows how the individual x’s come together to form $\hat{y}$



Visualizing logistic regression

You can do the same thing for logistic regression by adding the σ function

$$\sigma\left(\begin{bmatrix} x_0^0 & x_0^1 & x_0^2 & \cdots & x_0^N \\ x_1^0 & x_1^1 & x_1^2 & \cdots & x_1^N \\ & & \cdots & & \\ x_M^0 & x_M^1 & x_M^2 & \cdots & x_M^N \end{bmatrix} \begin{bmatrix} w_0 \\ w_1 \\ \vdots \\ w_N \end{bmatrix} + b\right) = \begin{bmatrix} \hat{y}_0 \\ \hat{y}_1 \\ \vdots \\ \hat{y}_M \end{bmatrix}$$



Visualizing logistic regression

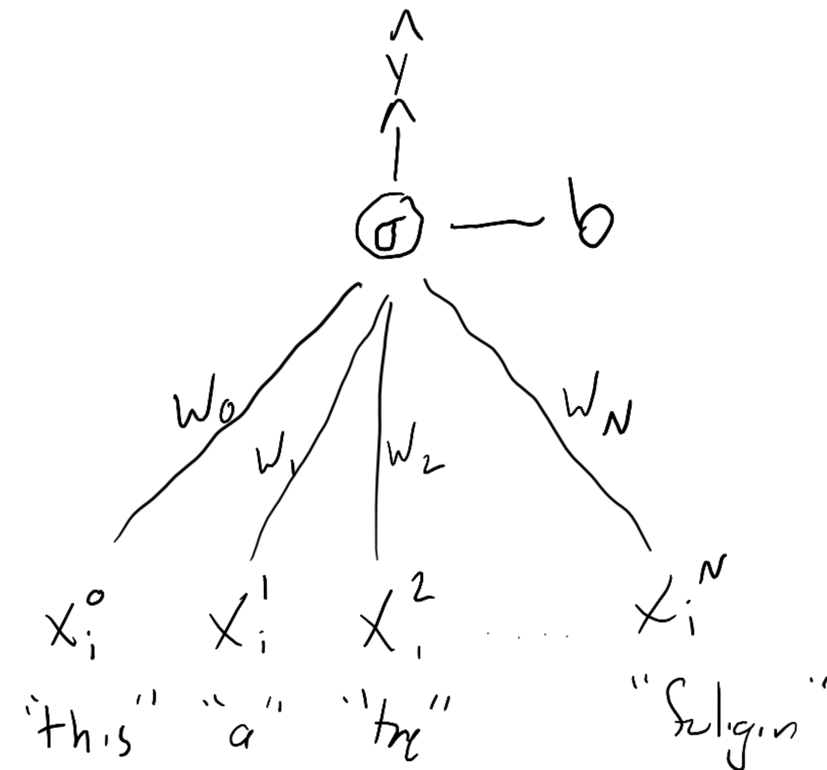
When we think of the logistic function as a final step being placed on top of the weighted sum $Wx + b$ in order to squeeze it down to [0,1], then we call it an **activation function**

There are a bunch of activation functions commonly used in neural nets:

- Rectified linear (relu), tanh, etc…

https://pytorch.org/docs/stable/nn.html#non-linear-activations-weighted-sum-nonlinearity

But logistic is the classic one



Linear vs logistic regression in practice

Linear regression with Scikit-Learn

Code description

- Read/preprocess SST-2
- Build and evaluate a Scikit-learn linear regression model
- Manually inspect some model parameters to explain predictions

Notebook headings

Linear regression
- Training
- Evaluation
- Global explanations
- Local explanations

“Kangaroo” and “Edmund”

“Kangaroo”

KANGAROO JACK

PG 2003, Comedy, 1h 29m

8%

TOMATOMETER

114 Reviews

29%

AUDIENCE SCORE

50,000+ Ratings

“Edmund”

???

Logistic regression with Scikit-Learn

Code description

- Example of training, evaluation and inspection of a Scikit-Learn logistic regression model

Notebook headings

- Binary logistic regression
  - Training
  - Evaluation
  - Global explanations
  - Local explanations

Concluding thoughts

Linear regression

- Learn $Wx + b$ from data
- Predict continuous values
- Optimize mean squared error

Logistic regression

- Learn $\sigma(Wx + b)$ from data
- Predict (close to) 0 or 1
- Optimize cross-entropy

Key concepts:

- Loss function
  - I.e. objective function
- Gradient of loss with respect to parameters
- Gradient descent
- Activation function