Learning as Function Approximation: Linear Models for Regression

CS771: Introduction to Machine Learning

Story so far, and the road ahead

- Seen learning approaches such as LwP, KNN, and decision trees

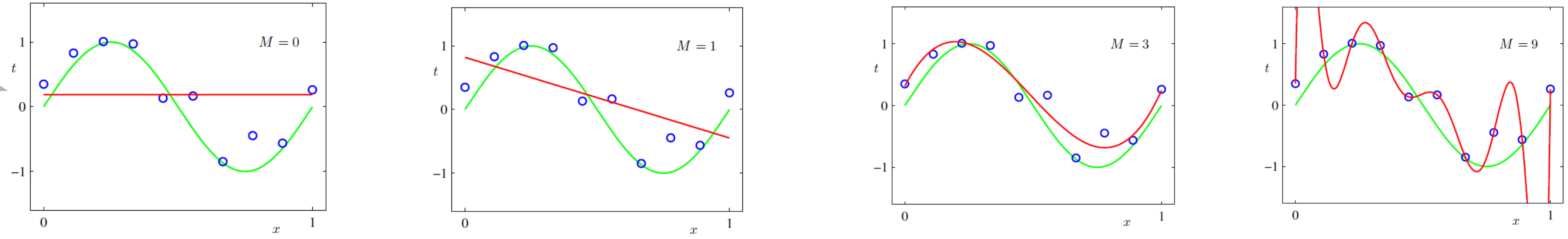
- These methods learn some rules from data, e.g.,
  - Compute means of classes and predict based on proximity from each class mean
  - Predict based on what the labels of the training set neighbors are
  - Learn to partition the data into homogeneous regions and do "region-wise prediction"

- We can also formulate learning as an optimization problem
  - Assume some model defined by some function
  - Find the optimal (best) function that has a good fit on training data

The function is defined by some parameters (or "weights) whose optimal value has to be learned from data

But also expected to have good generalization on test data

In this example, the class of functions is polynomials of various degrees



Supervised Learning as Function Approximation

- Given: Training data with N input-output pairs $\{(\boldsymbol{x}_n, y_n)\}_{n=1}^N$,
- Goal: Learn a model to predict the output for new test inputs
- Assume the model is defined by a function f that approximates I/O relationship

$$y_n \approx f(\boldsymbol{x}_n) \quad \forall n$$

And not just for the training data but also for future test data

So f must generalize well

- We can define the total "loss" that f incurs on this training data as follows

ℓ denotes some loss function which measures how close the true label y_n and the prediction $f(\boldsymbol{x}_n)$ are

$$L(f) = \sum_{n=1}^{N} \ell(y_n, f(x_n))$$

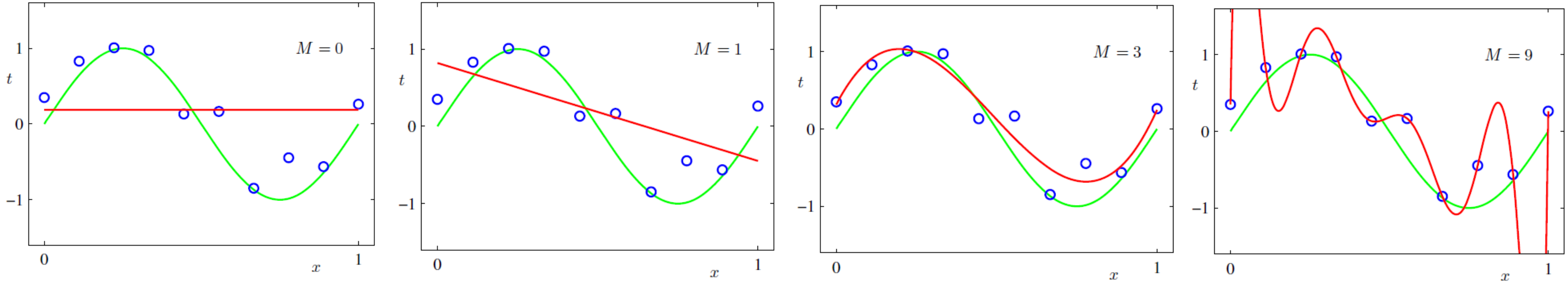
Finding the optimal function approximation

- We can find the f which minimizes the total loss by solving an optimization problem

$$\hat{f} = \text{argmin}_f \; L(f) = \text{argmin}_f \sum_{n=1}^{N} \ell(y_n, f(x_n))$$

(subject to some constraints on f)

Usually to keep f "simple" in some sense otherwise it may overfit on training data which is not desirable



Function Approximation using a Linear Model

- Assuming each output a scalar, define $f(\boldsymbol{x}) = \boldsymbol{w}^\top \boldsymbol{x}$,

Linear combination of features

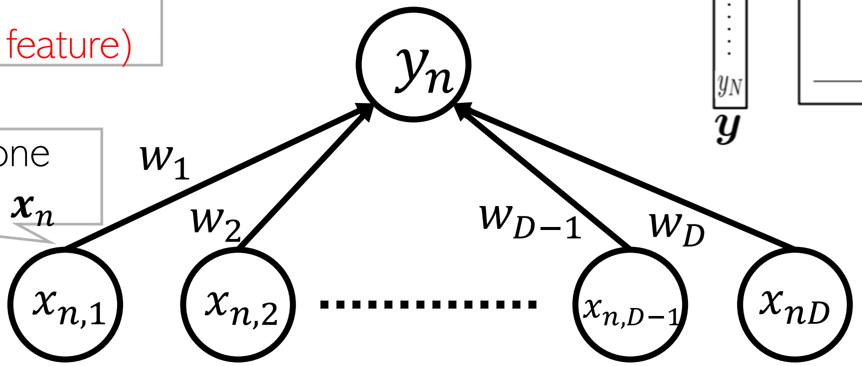
Function f has been defined by D "weights" or "parameters" (one per feature)

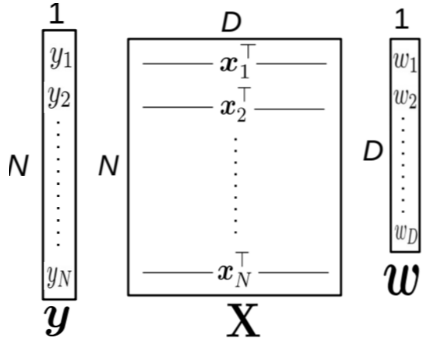
$$y_n \approx \boldsymbol{w}^\top \boldsymbol{x}_n = \sum_{i=1}^{D} w_i x_{ni}$$

Each node represents one of the features of input $\boldsymbol{x}_n$

Same linear model for all input-output pairs $(\boldsymbol{x}_n, y_n)$

Compactly: $\boldsymbol{y} \approx \boldsymbol{X}\boldsymbol{w}$





- The total loss on training data $\{(\boldsymbol{x}_n, y_n)\}_{n=1}^{N}$ will be

We need to learn these weights $\boldsymbol{w} = [w_1, w_2, \ldots, w_D]$ by solving an optimization problem

$$L(\boldsymbol{w}) = \sum_{n=1}^{N} \ell(y_n, \boldsymbol{w}^\top \boldsymbol{x}_n)$$

Will see examples of loss functions for classification (where y_n is discrete) later

- Some popular loss functions for regression (when the output is scalar)

Squared loss function

$$\ell(y_n, \boldsymbol{w}^\top \boldsymbol{x}_n) = (y_n - \boldsymbol{w}^\top \boldsymbol{x}_n)^2$$

Absolute loss function

$$\ell(y_n, \boldsymbol{w}^\top \boldsymbol{x}_n) = |y_n - \boldsymbol{w}^\top \boldsymbol{x}_n|$$

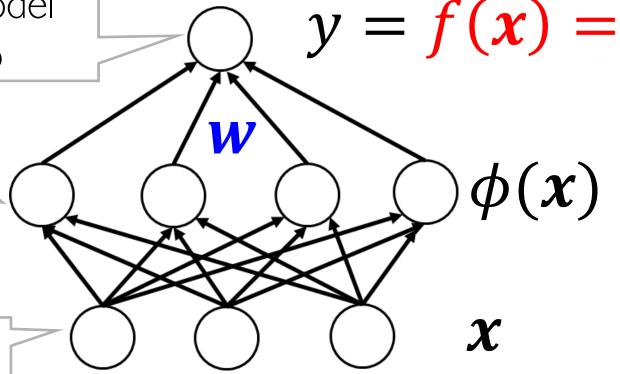
Function Approximation using a Nonlinear Model

- We can also define nonlinear models using linear models as sub-modules, e.g.,
  - Pre-defined nonlinear transformations of the inputs, followed by a linear model

Linear model at the top

Features extracted by some pre-defined non-linear transformation of input features (example: kernel methods)

Input features



$$y = f(\boldsymbol{x}) = \boldsymbol{w}^\top \phi(\boldsymbol{x})$$

Even with nonlinear models, we can use the standard loss functions such as squared or absolute loss, e.g.,

$$\ell(y_n, f(\boldsymbol{x}_n)) = (y_n - f(\boldsymbol{x}_n))^2$$
$$\ell(y_n, f(\boldsymbol{x}_n)) = |y_n - f(\boldsymbol{x}_n)|$$

  - Deep learning: A sequence of learned nonlinear transformations of the inputs, followed by a linear model

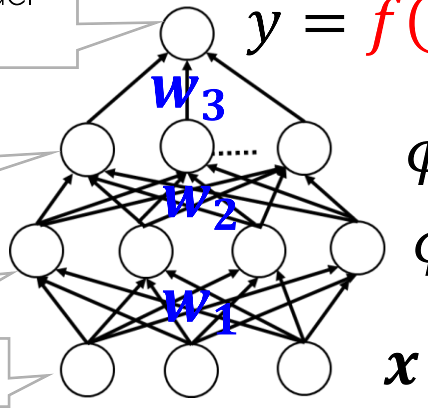
Effectively, this is a deep neural network with two hidden layers

Linear model at the top

Features extracted by some learned non-linear transformation of previous layer features

Features extracted by some learned non-linear transformation of input features

Input features



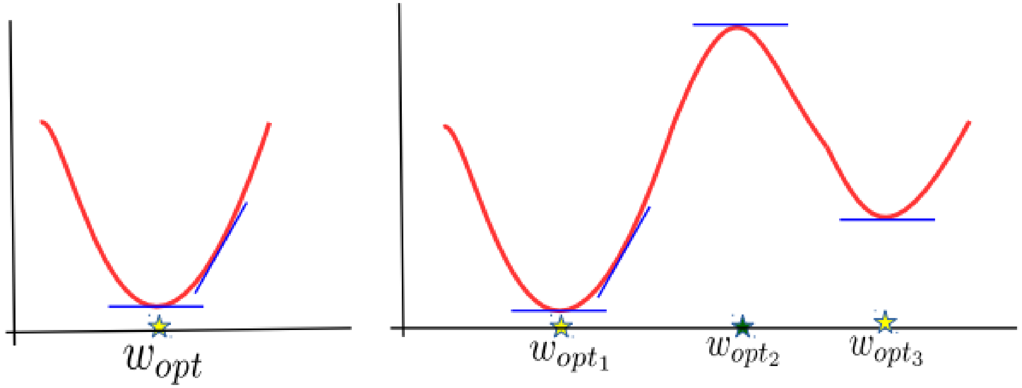
$$y = f(\boldsymbol{x}) = \boldsymbol{w}_3^\top \phi_2(\boldsymbol{x}) = \boldsymbol{w}_3^\top f_2(f_1(\boldsymbol{x}, \boldsymbol{w}_1), \boldsymbol{w}_2)$$

$$\phi_2(\boldsymbol{x}) = f_2(\phi_1(\boldsymbol{x}), \boldsymbol{w}_2) = f_2(f_1(\boldsymbol{x}, \boldsymbol{w}_1), \boldsymbol{w}_2)$$

$$\phi_1(\boldsymbol{x}) = f_1(\boldsymbol{x}, \boldsymbol{w}_1)$$

Loss Func. Minimization via First-Order Optimality

- Use calculus to find minima of the total loss function $L(\boldsymbol{w}) = \sum_{n=1}^{N} \ell(y_n, \boldsymbol{w}^\top \boldsymbol{x}_n)$



Called "first order" since only gradient is used and gradient provides the first order info about the function being optimized

The first order optimality method used only for very simple problems where the objective is convex and there are no constraints on the values the weight $\boldsymbol{w}$ can take

- First order optimality: The gradient $\boldsymbol{g}$ must be equal to zero at the optima

$$\boldsymbol{g} = \nabla_{\boldsymbol{w}}[L(\boldsymbol{w})] = \mathbf{0}$$

- Sometimes, setting $\boldsymbol{g} = \mathbf{0}$ and solving for $\boldsymbol{w}$ gives a closed form solution
- If closed form solution is not available, the gradient $\boldsymbol{g}$ can still be used in iterative optimization algos, like gradient descent (GD)
  - Note: Even if closed-form solution is possible, GD can sometimes be more efficient

Loss Func. Minimization via Iterative Optimization

Can I used this approach to solve maximization problems?

For maximization, we can use gradient ascent $\boldsymbol{w}^{(t+1)} = \boldsymbol{w}^{(t)} + \eta_t \boldsymbol{g}^{(t)}$

Iterative since it requires several steps/iterations to find the optimal solution

Fact: Gradient gives the direction of steepest change in function's value

Will move in the direction of the gradient

For convex functions, GD will converge to the global minima

Good initialization needed for non-convex functions

Gradient Descent

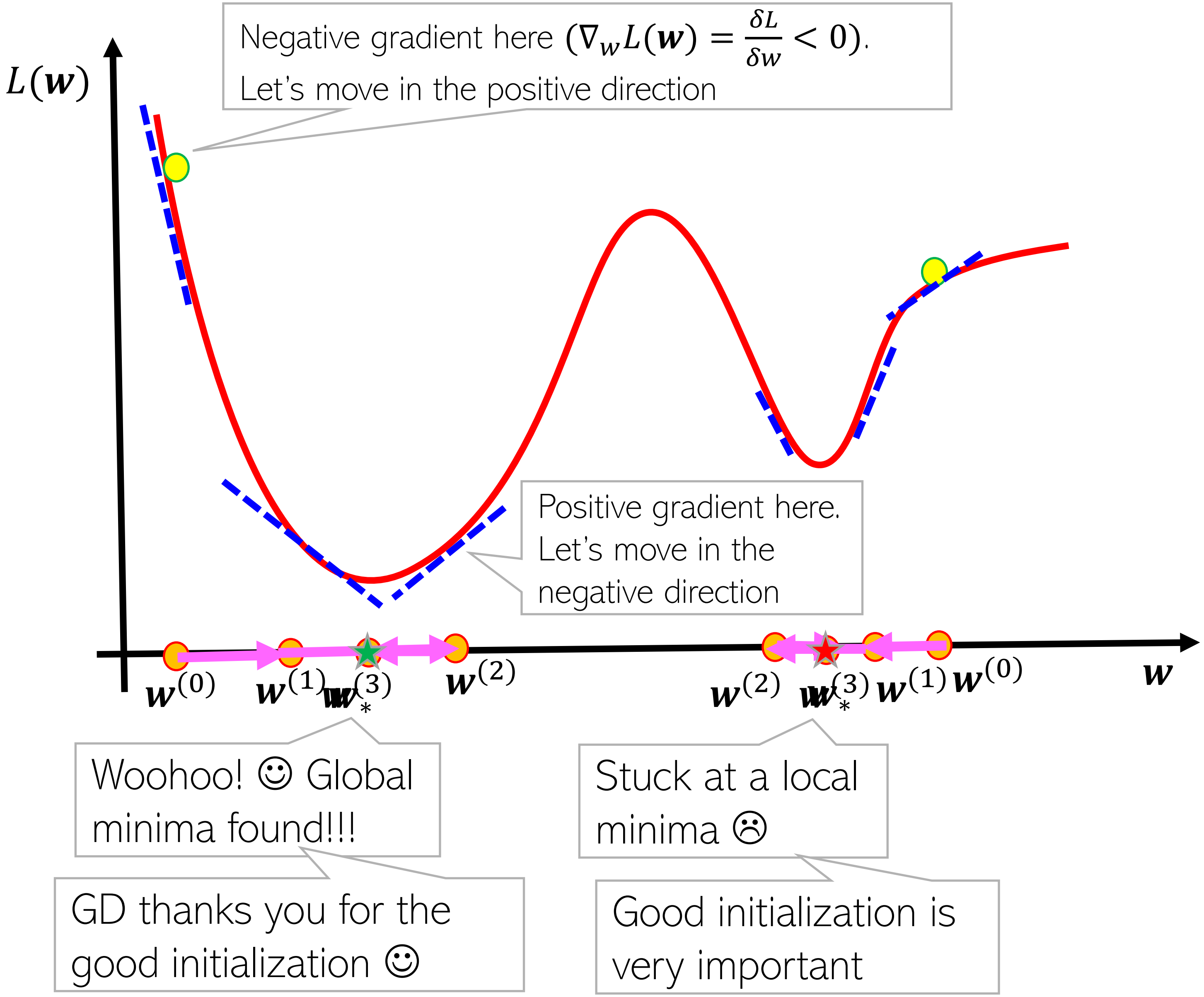
- Initialize $\boldsymbol{w}$ as $\boldsymbol{w}^{(0)}$
- For iteration $t = 0,1,2,\ldots$ (or until convergence)
  - Calculate the gradient $\boldsymbol{g}^{(t)}$ using the current iterates $\boldsymbol{w}^{(t)}$
  - Set the learning rate η_t
  - Move in the opposite direction of gradient

$$\boldsymbol{w}^{(t+1)} = \boldsymbol{w}^{(t)} - \eta_t \boldsymbol{g}^{(t)}$$

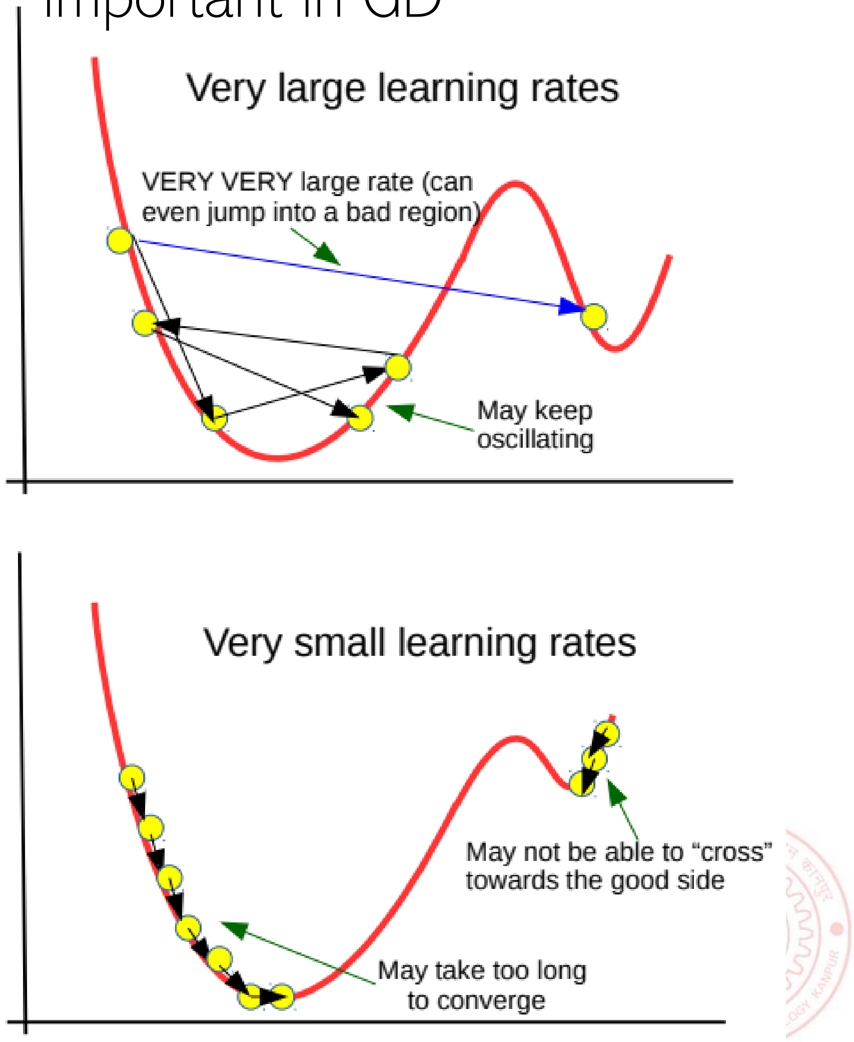
The learning rate very imp. Should be set carefully (fixed or chosen adaptively). Will discuss some strategies later

Sometimes may be tricky to to assess convergence. Will see some methods later

Gradient Descent – An Illustration

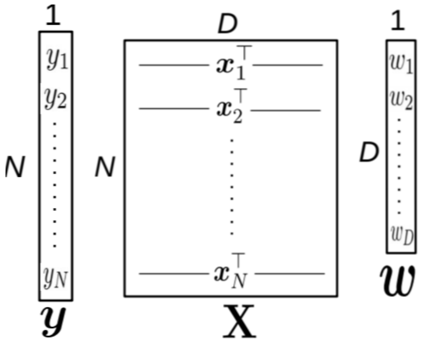


Learning rate (step-size) is very important in GD



Linear Models for Regression

Regression using a Linear Model



- The most popular linear model for regression uses the squared loss

$$L(\boldsymbol{w}) = \sum_{n=1}^{N} (y_n - \boldsymbol{w}^\top \boldsymbol{x}_n)^2 = ||\boldsymbol{y} - \boldsymbol{X}\boldsymbol{w}||^2$$

The "least squares" (LS) problem Gauss-Legendre, 18th century)

- To find the optimal weights $\boldsymbol{w}$, we minimize the above w.r.t. $\boldsymbol{w}$

$$\boldsymbol{w}_{LS} = \text{argmin}_{\boldsymbol{w}} \sum_{n=1}^{N} (y_n - \boldsymbol{w}^\top \boldsymbol{x}_n)^2$$

$$= \text{argmin}_{\boldsymbol{w}} ||\boldsymbol{y} - \boldsymbol{X}\boldsymbol{w}||^2$$

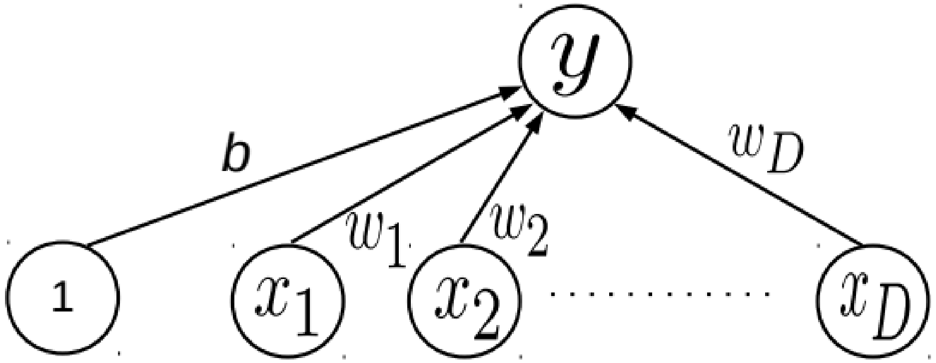
- Minimizing the above using first-order optimality (FOO) gives

Luckily FOO has given us a closed form (exact) solution in this case!

$$\boxed{\boldsymbol{w}_{LS} = \left(\sum_{n=1}^{N} \boldsymbol{x}_n \boldsymbol{x}_n^\top\right)^{-1} \left(\sum_{n=1}^{N} y_n \boldsymbol{x}_n\right) = (\boldsymbol{X}^\top \boldsymbol{X})^{-1} \boldsymbol{X}^\top \boldsymbol{y}}$$

A side note: the bias term

- We usually also have a bias term $\boldsymbol{b}$ in addition to the weights $\boldsymbol{w} = [w_1, w_2, \dots, w_D]$

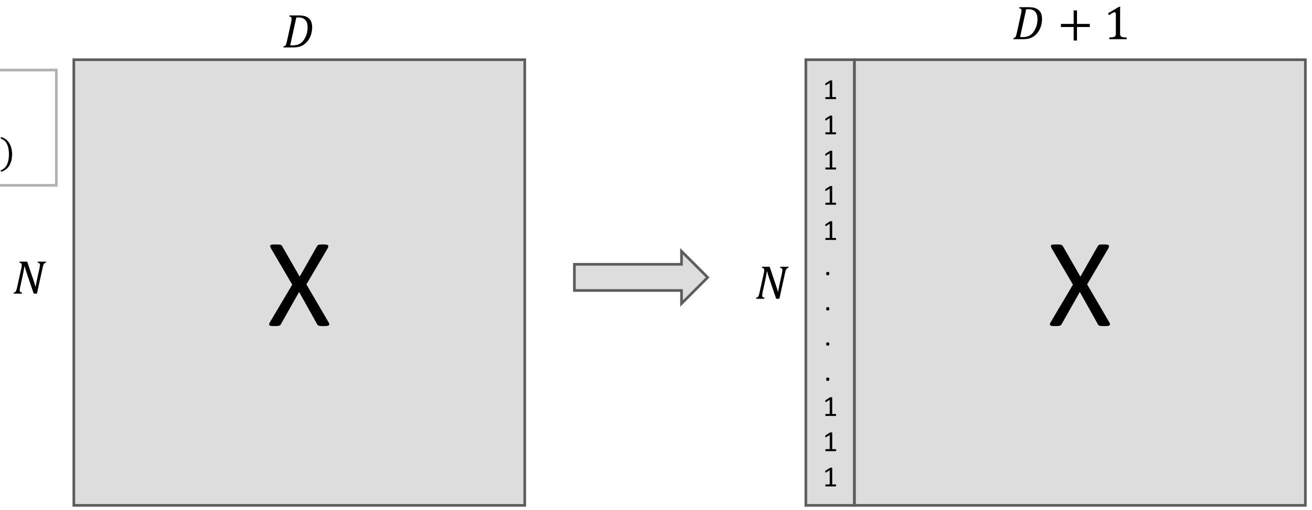


Can append a constant feature "1" for each input and again rewrite as $y = \tilde{\boldsymbol{w}}^\top \tilde{\boldsymbol{x}}$ where now both $\tilde{\boldsymbol{x}} = [1, \boldsymbol{x}]$ and $\tilde{\boldsymbol{w}} = [b, \boldsymbol{w}]$ are in $\mathbb{R}^{D+1}$

We will assume the same and omit the explicit bias for simplicity of notation

$$y = \sum_{d=1}^{D} w_d x_d + b = \boldsymbol{w}^\top \boldsymbol{x} + b$$

Note: Subscript '$\boldsymbol{n}$' omitted here from the input output pair $(\boldsymbol{x}_n, \boldsymbol{y}_n)$



Proof of the least squares solution

- We wanted to find the minima of $L(\boldsymbol{w}) = \sum_{n=1}^{N}(y_n - \boldsymbol{w}^\top \boldsymbol{x}_n)^2$

- Apply first order optimality – taking first derivative of $L(\boldsymbol{w})$ and setting it to zero

Chain rule of calculus

$$\frac{\partial L(\boldsymbol{w})}{\partial \boldsymbol{w}} = \frac{\partial}{\partial \boldsymbol{w}} \sum_{n=1}^{N}(y_n - \boldsymbol{w}^\top \boldsymbol{x}_n)^2 = \sum_{n=1}^{N} 2(y_n - \boldsymbol{w}^\top \boldsymbol{x}_n)\frac{\partial}{\partial \boldsymbol{w}}(y_n - \boldsymbol{w}^\top \boldsymbol{x}_n) = 0$$

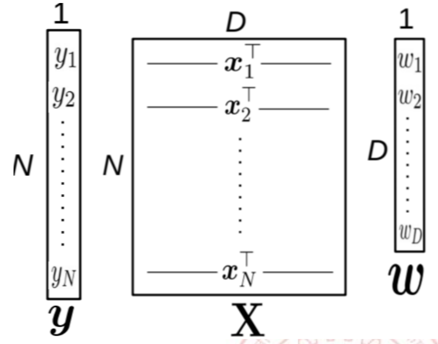
Partial derivative of dot product w.r.t each element of $\boldsymbol{w}$

Result of this derivative is $\boldsymbol{x}_n$ - same size as $\boldsymbol{w}$

- Using the fact $\frac{\partial}{\partial \boldsymbol{w}} \boldsymbol{w}^\top \boldsymbol{x}_n = \boldsymbol{x}_n$, we get $\sum_{n=1}^{N} 2(y_n - \boldsymbol{w}^\top \boldsymbol{x}_n)\boldsymbol{x}_n = 0$

- To separate $\boldsymbol{w}$ to get a solution, we write the above as

$$\sum_{n=1}^{N} 2\boldsymbol{x}_n(y_n - \boldsymbol{x}_n^\top \boldsymbol{w}) = 0 \Rightarrow \sum_{n=1}^{N} y_n \boldsymbol{x}_n - \boldsymbol{x}_n \boldsymbol{x}_n^\top \boldsymbol{w} = 0$$



$$\boldsymbol{w}_{LS} = \left(\sum_{n=1}^{N} \boldsymbol{x}_n \boldsymbol{x}_n^\top\right)^{-1}\left(\sum_{n=1}^{N} y_n \boldsymbol{x}_n\right) = (\boldsymbol{X}^\top \boldsymbol{X})^{-1} \boldsymbol{X}^\top \boldsymbol{y}$$

Problem(s) with the least squares solution!

- We minimized the objective $L(\boldsymbol{w}) = \sum_{n=1}^{N}(y_n - \boldsymbol{w}^\top \boldsymbol{x}_n)^2$ w.r.t. $\boldsymbol{w}$ and got

$$\boldsymbol{w}_{LS} = \left(\sum_{n=1}^{N} \boldsymbol{x}_n \boldsymbol{x}_n^\top\right)^{-1} \left(\sum_{n=1}^{N} y_n \boldsymbol{x}_n\right) = (\boldsymbol{X}^\top \boldsymbol{X})^{-1} \boldsymbol{X}^\top \boldsymbol{y}$$

- Problem: The matrix $\boldsymbol{X}^\top \boldsymbol{X}$ may not be invertible
  - This may lead to non-unique solutions for $\boldsymbol{w}_{opt}$
- Problem: Overfitting since we only minimized loss defined on training data
  - Weights $\boldsymbol{w} = [w_1, w_2, \ldots, w_D]$ may become arbitrarily large to fit training data perfectly
  - Such weights may perform poorly on the test data however

$R(\boldsymbol{w})$ is called the Regularizer and measures the "magnitude" of $\boldsymbol{w}$

- One Solution: Minimize a **regularized objective** $\boxed{L(\boldsymbol{w}) + \lambda R(\boldsymbol{w})}$
  - The reg. will prevent the elements of $\boldsymbol{w}$ from becoming too large
  - Reason: Now we are minimizing training error + magnitude of vector $\boldsymbol{w}$

$\lambda \geq 0$ is the reg. hyperparam. Controls how much we wish to regularize (needs to be tuned via cross-validation)

Regularized Least Squares (a.k.a. Ridge Regression)

- Recall that the regularized objective is of the form $L_{reg}(\boldsymbol{w}) = L(\boldsymbol{w}) + \lambda R(\boldsymbol{w})$
- One possible/popular regularizer: the squared Euclidean (ℓ_2 squared) norm of $\boldsymbol{w}$

$$R(\boldsymbol{w}) = \|\boldsymbol{w}\|_2^2 = \boldsymbol{w}^\top \boldsymbol{w}$$

- With this regularizer, we have the regularized least squares problem as

Look at the form of the solution. We are adding a small value λ to the diagonals of the DxD matrix $\boldsymbol{X}^\top \boldsymbol{X}$ (like adding a ridge/mountain to some land)

Why is the method called “ridge” regression

$$\boldsymbol{w}_{ridge} = \arg\min_{\boldsymbol{w}} L(\boldsymbol{w}) + \lambda R(\boldsymbol{w})$$

$$= \arg\min_{\boldsymbol{w}} \sum_{n=1}^{N} (y_n - \boldsymbol{w}^\top \boldsymbol{x}_n)^2 + \lambda \boldsymbol{w}^\top \boldsymbol{w}$$

- Proceeding just like the LS case, we can find the optimal $\boldsymbol{w}$ which is given by

$$\boxed{\boldsymbol{w}_{ridge} = \left(\sum_{n=1}^{N} \boldsymbol{x}_n \boldsymbol{x}_n^\top + \lambda I_D\right)^{-1} \left(\sum_{n=1}^{N} y_n \boldsymbol{x}_n\right) = (\boldsymbol{X}^\top \boldsymbol{X} + \lambda I_D)^{-1} \boldsymbol{X}^\top \boldsymbol{y}}$$

Why regularization helps?

Remember – in general, weights with large magnitude are bad since they can cause overfitting on training data and may not work well on test data

- The regularized objective we minimized is

$$L_{reg}(\boldsymbol{w}) = \sum_{n=1}^{N} (y_n - \boldsymbol{w}^\top \boldsymbol{x}_n)^2 + \lambda \boldsymbol{w}^\top \boldsymbol{w}$$

- Minimizing $L_{reg}(\boldsymbol{w})$ w.r.t. $\boldsymbol{w}$ gives a solution for $\boldsymbol{w}$ that
  - Keeps the training error small
  - Has a small ℓ_2 squared norm $\boldsymbol{w}^\top \boldsymbol{w} = \sum_{d=1}^{D} w_d^2$

Good because, consequently, the individual entries of the weight vector $\boldsymbol{w}$ are also prevented from becoming too large

- Small entries in $\boldsymbol{w}$ are good since they lead to "smooth" models

Not a "smooth" model since its test data predictions may change drastically even with small changes in some feature's value

$\boldsymbol{x}_n =$	1.2	0.5	2.4	0.3	0.8	0.1	0.9	2.1
$\boldsymbol{x}_m =$	1.2	0.5	2.4	0.3	$0.8+\epsilon$	0.1	0.9	2.1

Exact same feature vectors only differing in just one feature by a small amount

$y_n = 0.8$

$y_m = 100$

Very different outputs though (maybe one of these two training ex. is an outlier)

A typical $\boldsymbol{w}$ learned without ℓ_2 reg.

3.2	1.8	1.3	2.1	10000	2.5	3.1	0.1

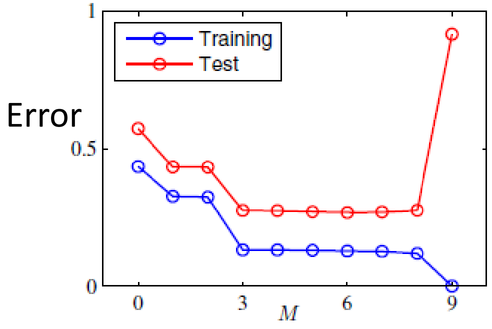
Just to fit the training data where one of the inputs was possibly an outlier, this weight became too big. Such a weight vector will possibly do poorly on normal test inputs

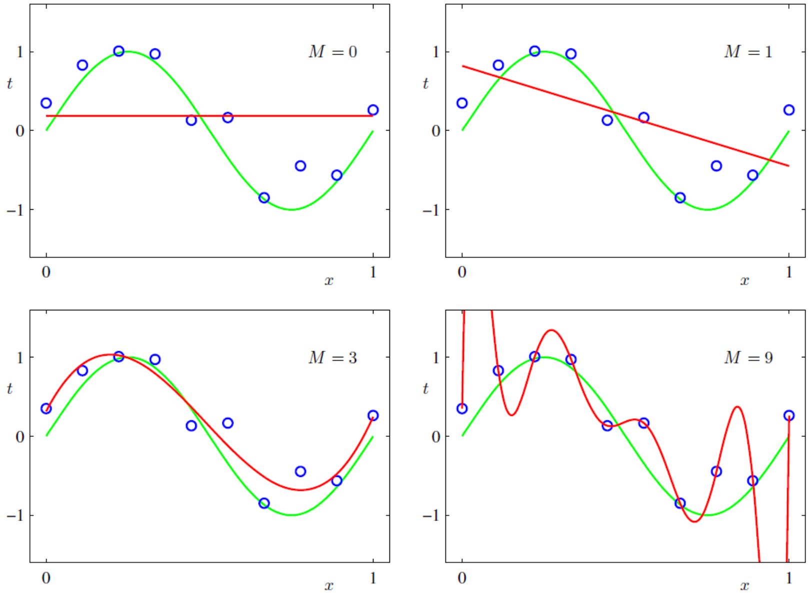
Large magnitude weights are bad: Another illustration

- Consider a polynomial regression model using a single scalar feature x

$$y = w_0 + w_1 x + w_2 x^2 + \cdots + w_M x^M$$

- Can also think of this as a linear model $y = \boldsymbol{w}^\top \phi(x)$
  - Pre-defined feature mapping $\phi(x) = [1, x, x^2, \ldots, x^M]$
  - Weight vector $\boldsymbol{w} = [w_0, w_1, \ldots, w_M]$





	$M = 0$	$M = 1$	$M = 3$	$M = 9$
$w_0^\star$	0.19	0.82	0.31	0.35
$w_1^\star$		-1.27	7.99	232.37
$w_2^\star$			-25.43	-5321.83
$w_3^\star$			17.37	48568.31
$w_4^\star$				-231639.30
$w_5^\star$				640042.26
$w_6^\star$				-1061800.52
$w_7^\star$				1042400.18
$w_8^\star$				-557682.99
$w_9^\star$				125201.43

Learning this model using high order (M) of polynomial gives the model too many degrees of freedom (with all or most of M weights taking very large values, resulting in overfitting on training data and quite likely giving a poor model on test data)

Regularization of $\boldsymbol{w}$ will help shrink w_i's of the "unnecessary" higher order terms to very small values

Other Ways to Control Overfitting

Note that optimizing loss functions with such regularizers is usually harder than ridge reg. but several advanced techniques exist (we will see some of those later)

- Use a regularizer $R(\boldsymbol{w})$ defined by other norms, e.g.,

Use them if you have a very large number of features but many irrelevant features. These regularizers can help in automatic feature selection

ℓ_1 norm regularizer

$$\|\boldsymbol{w}\|_1 = \sum_{d=1}^{D} |w_d|$$

When should I used these regularizers instead of the ℓ_2 regularizer?

Automatic feature selection? Wow, cool!!! But how exactly?

$$\|\boldsymbol{w}\|_0 = \#\mathrm{nnz}(\boldsymbol{w})$$

ℓ_0 norm regularizer (counts number of nonzeros in $\boldsymbol{w}$

Using such regularizers gives a sparse weight vector $\boldsymbol{w}$ as solution (will see the reason in detail later)

sparse means many entries in $\boldsymbol{w}$ will be zero or near zero. Thus those features will be considered irrelevant by the model and will not influence prediction

- Use non-regularization based approaches
  - Early-stopping (stopping training just when we have a decent val. set accuracy)
  - Dropout (in each iteration, don't update some of the weights)
  - Injecting noise in the inputs

All of these are very popular ways to control overfitting in deep learning models. More on these later when we talk about deep learning

Gradient Descent for Linear/Ridge Regression

- Just use the GD algorithm with the gradient expressions we derived
- Iterative updates for linear regression will be of the form

Also, we usually work with average gradient so the gradient term is divided by N

Note the form of each term in the gradient expression update: Amount of current $\boldsymbol{w}$'s error on the n^{th} training example multiplied by the input $\boldsymbol{x}_n$

$$\boldsymbol{w}^{(t+1)} = \boldsymbol{w}^{(t)} - \eta_t \boldsymbol{g}^{(t)}$$

$$= \boldsymbol{w}^{(t)} + \eta_t \frac{2}{N} \sum_{n=1}^{N} \left(y_n - \boldsymbol{w}^{(t)\top} \boldsymbol{x}_n \right) \boldsymbol{x}_n$$

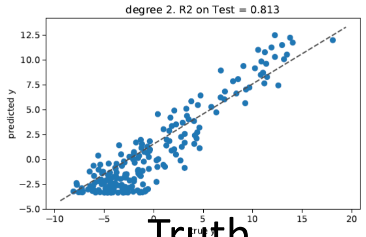
Unlike the closed form solution $(\boldsymbol{X}^\top \boldsymbol{X})^{-1} \boldsymbol{X}^\top \boldsymbol{y}$ of least squares regression, here we have iterative updates but do not require the expensive matrix inversion of the $D \times D$ matrix $\boldsymbol{X}^\top \boldsymbol{X}$ (thus faster)

- Similar updates for ridge regression as well (with the gradient expression being slightly different; left as an exercise)

- More on iterative optimization methods later

Evaluation Measures for Regression Models

Prediction



Truth

- Plotting the prediction $\hat{y}_n$ vs truth y_n for the validation/test set
- Mean Squared Error (MSE) and Mean Absolute Error (MAE) on val./test set

$$MSE = \frac{1}{N}\sum_{n=1}^{N}(y_n - \hat{y}_n)^2 \qquad MAE = \frac{1}{N}\sum_{n=1}^{N}|y_n - \hat{y}_n|$$

- RMSE (Root Mean Squared Error) $\triangleq \sqrt{MSE}$
- Coefficient of determination or R^2

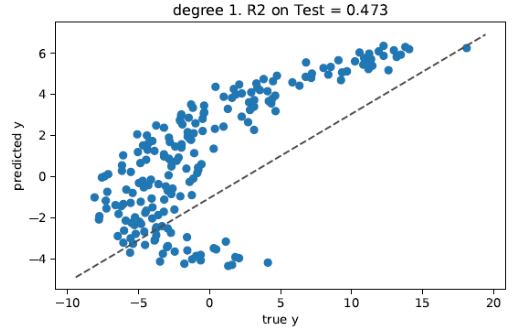
$$R^2 = 1 - \frac{\sum_{n=1}^{N}(y_n - \hat{y}_n)^2}{\sum_{n=1}^{N}(y_n - \bar{y})^2}$$

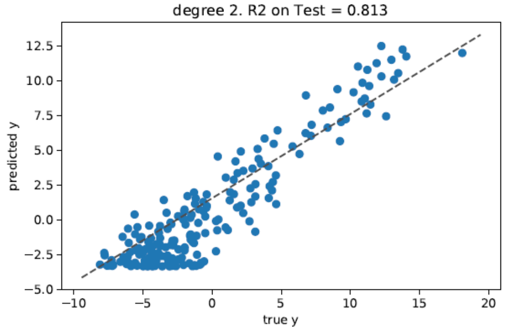
"relative" error w.r.t. a model that makes a constant prediction $\bar{y}$ for all inputs

A "base" model that always predicts the mean $\bar{y}$ will have $R^2 = 0$ and the perfect model will have $R^2 = 1$. Worse than base models can even have negative R^2

$\bar{y}$ is empirical mean of true responses, i.e., $\frac{1}{N}\sum_{n=1}^{N} y_n$

Plots of true vs predicted outputs and R^2 for two regression models





Pic from MLAPP (Murphy)

Linear Regression as Solving System of Linear Eqs

- The form of the lin. reg. model $\boldsymbol{y} \approx \boldsymbol{X}\boldsymbol{w}$ is akin to a system of linear equation
- Assuming N training examples with D features each, we have

First training example: $y_1 = x_{11}w_1 + x_{12}w_2 + \ldots + x_{1D}w_D$

Second training example: $y_2 = x_{21}w_1 + x_{22}w_2 + \ldots + x_{2D}w_D$

N-th training example: $y_N = x_{N1}w_1 + x_{N2}w_2 + \ldots + x_{ND}w_D$

Note: Here x_{nd} denotes the d^{th} feature of the n^{th} training example

N equations and D unknowns here $(w_1, w_2, \ldots, w_D)$

- Usually we will either have $N > D$ or $N < D$
  - Thus we have an underdetermined $(N < D)$ or overdetermined $(N > D)$ system
  - Methods to solve over/underdetermined systems can be used for lin-reg as well
  - Many of these methods don't require expensive matrix inversion

Solving lin-reg as system of lin eq.

$$\boldsymbol{w} = (\boldsymbol{X}^\top\boldsymbol{X})^{-1}\boldsymbol{X}^\top\boldsymbol{y} \quad \Longrightarrow \quad \boxed{\boldsymbol{A}\boldsymbol{w} = \boldsymbol{b}} \text{ where } \boldsymbol{A} = \boldsymbol{X}^\top\boldsymbol{X}, \text{ and } \boldsymbol{b} = \boldsymbol{X}^\top\boldsymbol{y}$$

Now solve this!

System of lin. Eqns with D equations and D unknowns