

CSC 311: Introduction to Machine Learning

Lecture 3 - Linear Regression & Gradient Descent for Optimization

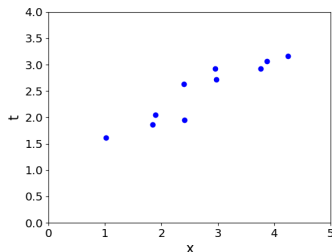
Amir-massoud Farahmand & Emad A.M. Andrews

University of Toronto

Modular Approach to ML Algorithm Design

- So far, we have talked about *procedures* for learning.
 - ▶ KNN and decision trees.
- For the remainder of this course, we will take a more modular approach:
 - ▶ choose a **model** describing the relationships between variables of interest
 - ▶ define a **loss function** quantifying how bad the fit to the data is
 - ▶ choose a **regularizer** saying how much we prefer different candidate models (or explanations of data)
 - ▶ fit the model that minimizes the loss function and satisfy the constraint/penalty imposed by the regularizer, possibly using an **optimization algorithm**
- Mixing and matching these modular components gives us a lot of new ML methods.

The Supervised Learning Setup



Recall that in supervised learning:

- There is target $t \in \mathcal{T}$ (also called response, outcome, output, class)
- There are features $\mathbf{x} \in \mathcal{X}$ (also called inputs and covariates)
- Objective is to learn a function $f : \mathcal{X} \rightarrow \mathcal{T}$ such that

$$t \approx y = f(x)$$

based on some data $\mathcal{D} = \{(\mathbf{x}^{(i)}, t^{(i)}) \text{ for } i = 1, 2, \dots, N\}$.

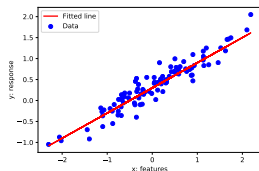
Linear Regression – Model

- **Model:** In linear regression, we use linear functions of the inputs $\mathbf{x} = (x_1, \dots, x_D)$ to make predictions y of the target value t :

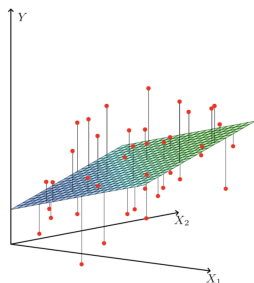
$$y = f(\mathbf{x}) = \sum_j w_j x_j + b$$

- ▶ y is the **prediction**
 - ▶ \mathbf{w} is the **weights**
 - ▶ b is the **bias** (or **intercept**) (do not confuse with the bias-variance tradeoff in the next lecture)
- \mathbf{w} and b together are the **parameters**
- We hope that our prediction is close to the target: $y \approx t$.

What is Linear? 1 Feature vs. D Features



- If we have only 1 feature:
 $y = wx + b$ where $w, x, b \in \mathbb{R}$.
- y is linear in x .



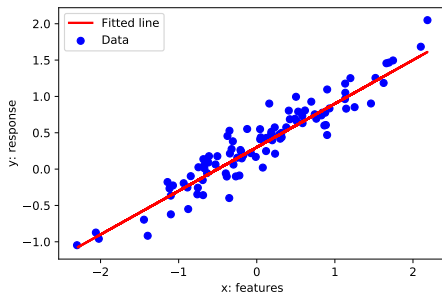
- If we have D features:
 $y = \mathbf{w}^\top \mathbf{x} + b$ where $\mathbf{w}, \mathbf{x} \in \mathbb{R}^D$,
 $b \in \mathbb{R}$
- y is linear in \mathbf{x} .

Relation between the prediction y and inputs \mathbf{x} is linear in both cases.

Linear Regression

We have a dataset $\mathcal{D} = \{(\mathbf{x}^{(i)}, t^{(i)}) \text{ for } i = 1, 2, \dots, N\}$ where,

- $\mathbf{x}^{(i)} = (x_1^{(i)}, x_2^{(i)}, \dots, x_D^{(i)})^\top \in \mathbb{R}^D$ are the inputs, e.g., age, height.
- $t^{(i)} \in \mathbb{R}$ is the target or response (e.g. income),
- predict $t^{(i)}$ with a linear function of $\mathbf{x}^{(i)}$:



- $t^{(i)} \approx y^{(i)} = \mathbf{w}^\top \mathbf{x}^{(i)} + b$
- Find the “best” line (\mathbf{w}, b) .
- minimize $\sum_{i=1}^N \mathcal{L}(y^{(i)}, t^{(i)})$
 (\mathbf{w}, b)

Linear Regression – Loss Function

- How to quantify the quality of the fit to data?
- A **loss function** $\mathcal{L}(y, t)$ defines how bad it is if, for some example \mathbf{x} , the algorithm predicts y , but the target is actually t .
- **Squared error loss function**:

$$\mathcal{L}(y, t) = \frac{1}{2}(y - t)^2$$

- $y - t$ is the **residual**, and we want to make its magnitude small
- The $\frac{1}{2}$ factor is just to make the calculations convenient.
- **Cost function**: loss function averaged over all training examples

$$\begin{aligned}\mathcal{J}(\mathbf{w}, b) &= \frac{1}{2N} \sum_{i=1}^N \left(y^{(i)} - t^{(i)} \right)^2 \\ &= \frac{1}{2N} \sum_{i=1}^N \left(\mathbf{w}^\top \mathbf{x}^{(i)} + b - t^{(i)} \right)^2\end{aligned}$$

- The terminology is not universal. Some might call “loss” *pointwise loss* and the “cost function” the *empirical loss* or *average loss*.

Vector Notation

- We can organize all the training examples into a **design matrix** \mathbf{X} with one row per training example, and all the targets into the **target vector** \mathbf{t} .

one feature across
all training examples

$$\mathbf{X} = \begin{pmatrix} \mathbf{x}^{(1)\top} \\ \mathbf{x}^{(2)\top} \\ \mathbf{x}^{(3)\top} \end{pmatrix} = \begin{pmatrix} 8 & 0 & 3 & 0 \\ 6 & -1 & 5 & 3 \\ 2 & 5 & -2 & 8 \end{pmatrix}$$

one training
example (vector)

- Computing the predictions for the whole dataset:

$$\mathbf{X}\mathbf{w} + b\mathbf{1} = \begin{pmatrix} \mathbf{w}^T \mathbf{x}^{(1)} + b \\ \vdots \\ \mathbf{w}^T \mathbf{x}^{(N)} + b \end{pmatrix} = \begin{pmatrix} y^{(1)} \\ \vdots \\ y^{(N)} \end{pmatrix} = \mathbf{y}$$

Vectorization

- Computing the squared error cost across the whole dataset:

$$\mathbf{y} = \mathbf{X}\mathbf{w} + b\mathbf{1}$$

$$\mathcal{J} = \frac{1}{2N} \|\mathbf{y} - \mathbf{t}\|^2$$

- Note that sometimes we may use $\mathcal{J} = \frac{1}{2} \|\mathbf{y} - \mathbf{t}\|^2$, without normalizer. That would correspond to the sum of losses, and not the average loss. The minimizer does not depend on N .
- We can also add a column of 1s to the design matrix, combine the bias and the weights, and conveniently write

$$\mathbf{X} = \begin{bmatrix} 1 & [\mathbf{x}^{(1)}]^\top \\ 1 & [\mathbf{x}^{(2)}]^\top \\ \vdots & \vdots \end{bmatrix} \in \mathbb{R}^{N \times D+1} \quad \text{and} \quad \mathbf{w} = \begin{bmatrix} b \\ w_1 \\ w_2 \\ \vdots \end{bmatrix} \in \mathbb{R}^{D+1}$$

Then, our predictions reduce to $\mathbf{y} = \mathbf{X}\mathbf{w}$.

Solving the Minimization Problem

- We defined a cost function. This is what we would like to minimize.
- Recall from your calculus class: minimum of a smooth function (if it exists) occurs at a **critical point**, i.e., point where the derivative is zero.
- Multivariate generalization: set the partial derivatives to zero (or equivalently the gradient).
- We would like to find a point where the gradient is (close to) zero. How can we do it?
 - ▶ Sometimes it is possible to directly find the parameters that make the gradient zero in a closed-form. We call this the **direct solution**.
 - ▶ We may also use optimization techniques that iteratively get us closer to the solution. We will get back to this soon.

Direct Solution

- **Partial derivatives:** derivatives of a multivariate function with respect to one of its arguments.

$$\frac{\partial}{\partial x_1} f(x_1, x_2) = \lim_{h \rightarrow 0} \frac{f(x_1 + h, x_2) - f(x_1, x_2)}{h}$$

- To compute, take the single variable derivatives, pretending the other arguments are constant.
- Example: partial derivatives of the prediction y

$$\begin{aligned} \frac{\partial y}{\partial w_j} &= \frac{\partial}{\partial w_j} \left[\sum_{j'} w_{j'} x_{j'} + b \right] \\ &= x_j \\ \frac{\partial y}{\partial b} &= \frac{\partial}{\partial b} \left[\sum_{j'} w_{j'} x_{j'} + b \right] \\ &= 1 \end{aligned}$$

Direct Solution

- Chain rule for derivatives:

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial w_j} &= \frac{d\mathcal{L}}{dy} \frac{\partial y}{\partial w_j} \\ &= \frac{d}{dy} \left[\frac{1}{2}(y - t)^2 \right] \cdot x_j \\ &= (y - t)x_j \\ \frac{\partial \mathcal{L}}{\partial b} &= y - t\end{aligned}$$

- Cost derivatives (average over data points):

$$\begin{aligned}\frac{\partial \mathcal{J}}{\partial w_j} &= \frac{1}{N} \sum_{i=1}^N (y^{(i)} - t^{(i)}) x_j^{(i)} \\ \frac{\partial \mathcal{J}}{\partial b} &= \frac{1}{N} \sum_{i=1}^N y^{(i)} - t^{(i)}\end{aligned}$$

Direct Solution

- The minimum must occur at a point where the partial derivatives are zero, i.e.,

$$\frac{\partial \mathcal{J}}{\partial w_j} = 0 \quad (\forall j), \quad \frac{\partial \mathcal{J}}{\partial b} = 0.$$

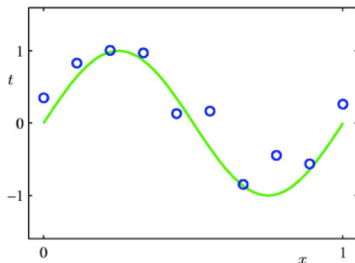
- If $\partial \mathcal{J} / \partial w_j \neq 0$, you could reduce the cost by changing w_j .
- This turns out to give a system of linear equations, which we can solve efficiently. **Full derivation in the preliminaries.pdf.**
- Optimal weights:

$$\mathbf{w}^{\text{LS}} = (\mathbf{X}^T \mathbf{X})^{-1} \mathbf{X}^T \mathbf{t}$$

- Linear regression is one of only a handful of models in this course that permit direct solution.

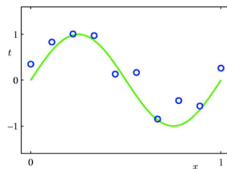
Feature Mapping (Basis Expansion)

- The relation between the input and output may not be linear.



- We can still use linear regression by mapping the input feature to another space using **feature mapping** (or **basis expansion**) $\psi(\mathbf{x}) : \mathbb{R}^D \rightarrow \mathbb{R}^d$ and treat the mapped feature (in \mathbb{R}^d) as the input of a linear regression procedure.
- Let us see how it works when $\mathbf{x} \in \mathbb{R}$ and we use polynomial feature mapping.

Polynomial Feature Mapping

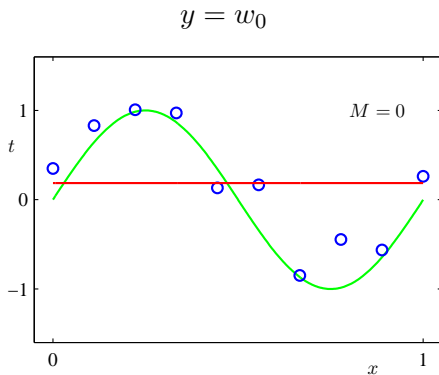


Fit the data using a degree- M polynomial function of the form:

$$y = w_0 + w_1x + w_2x^2 + \dots + w_Mx^M = \sum_{i=0}^M w_i x^i$$

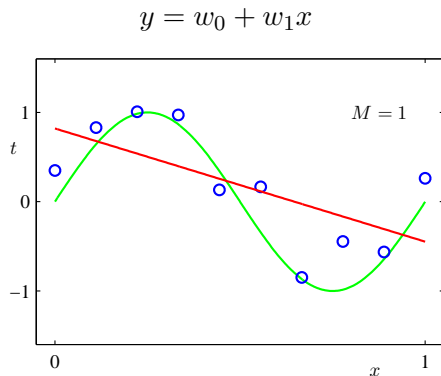
- Here the feature mapping is $\psi(x) = [1, x, x^2, \dots]^\top$.
- We can still use least squares to find \mathbf{w} since $y = \psi(x)^\top \mathbf{w}$ is linear in w_0, w_1, \dots
- In general, ψ can be any function. Another example: $\psi = [1, \sin(2\pi x), \cos(2\pi x), \sin(4\pi x), \cos(4\pi x), \sin(6\pi x), \cos(6\pi x), \dots]^\top$.

Polynomial Feature Mapping with $M = 0$



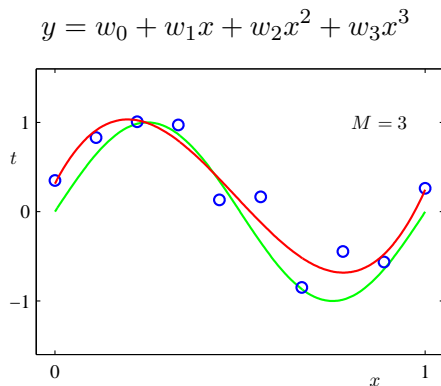
-Pattern Recognition and Machine Learning, Christopher Bishop.

Polynomial Feature Mapping with $M = 1$



-Pattern Recognition and Machine Learning, Christopher Bishop.

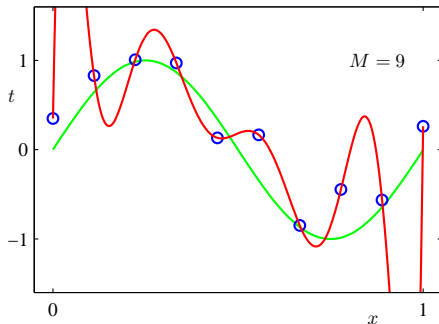
Polynomial Feature Mapping with $M = 3$



-Pattern Recognition and Machine Learning, Christopher Bishop.

Polynomial Feature Mapping with $M = 9$

$$y = w_0 + w_1x + w_2x^2 + w_3x^3 + \dots + w_9x^9$$

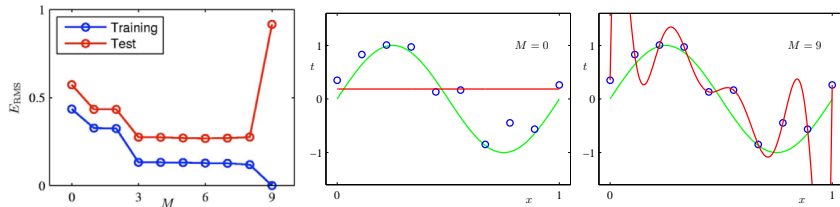


-Pattern Recognition and Machine Learning, Christopher Bishop.

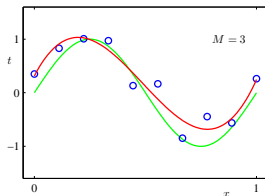
Model Complexity and Generalization

Underfitting ($M=0$): model is too simple — does not fit the data.

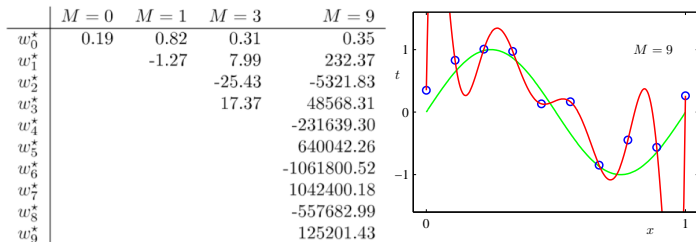
Overfitting ($M=9$): model is too complex — fits perfectly.



Good model ($M=3$): Achieves small test error (generalizes well).



Model Complexity and Generalization



- As M increases, the magnitude of coefficients gets larger.
- For $M = 9$, the coefficients have become finely tuned to the data.
- Between data points, the function exhibits large oscillations.

Regularization for Controlling the Model Complexity

- The degree of the polynomial M controls the complexity of the model.
- The value of M is a hyperparameter for polynomial expansion, just like k in KNN. We can tune it using a validation set.
- Restricting the number of parameters of a model (M here) is a crude approach to control the complexity of the model.
- A better solution: keep the number of parameters of the model large, but enforce “simpler” solutions within the same space of parameters.
- This is done through [regularization](#) or [penalization](#).
 - ▶ [Regularizer](#) (or [penalty](#)): a function that quantifies how much we prefer one hypothesis vs. another
- Q: How?!

ℓ_2 (or L^2) Regularization

- We can encourage the weights to be small by choosing as our regularizer the ℓ_2 (or L^2) penalty.

$$\mathcal{R}(\mathbf{w}) = \frac{1}{2} \|\mathbf{w}\|_2^2 = \frac{1}{2} \sum_j w_j^2.$$

- ▶ Note: To be precise, we are regularizing the *squared* ℓ_2 norm.
- The regularized cost function makes a tradeoff between fit to the data and the norm of the weights:

$$\mathcal{J}_{\text{reg}}(\mathbf{w}) = \mathcal{J}(\mathbf{w}) + \lambda \mathcal{R}(\mathbf{w}) = \mathcal{J}(\mathbf{w}) + \frac{\lambda}{2} \sum_j w_j^2.$$

- The basic idea is that “simpler” functions have smaller ℓ_2 -norm of their weights \mathbf{w} , and we prefer them to functions with larger ℓ_2 -norms.
- If you fit training data poorly, \mathcal{J} is large. If your optimal weights have high values, \mathcal{R} is large.
- Large λ penalizes weight values more.
- Here, λ is a hyperparameter that we can tune with a validation set.

ℓ_2 Regularized Least Squares: Ridge Regression

For the least squares problem, we have $\mathcal{J}(\mathbf{w}) = \frac{1}{2N} \|\mathbf{X}\mathbf{w} - \mathbf{t}\|^2$.

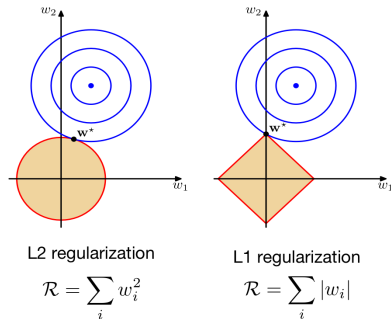
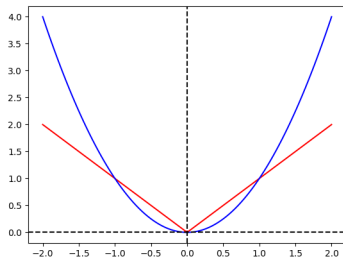
- When $\lambda > 0$ (with regularization), regularized cost gives

$$\begin{aligned}\mathbf{w}_\lambda^{\text{Ridge}} &= \underset{\mathbf{w}}{\operatorname{argmin}} \mathcal{J}_{\text{reg}}(\mathbf{w}) = \underset{\mathbf{w}}{\operatorname{argmin}} \frac{1}{2N} \|\mathbf{X}\mathbf{w} - \mathbf{t}\|_2^2 + \frac{\lambda}{2} \|\mathbf{w}\|_2^2 \\ &= (\mathbf{X}^T \mathbf{X} + \lambda N \mathbf{I})^{-1} \mathbf{X}^T \mathbf{t}\end{aligned}$$

- The case $\lambda = 0$ (no regularization) reduces to least squares solution!
- Q: What happens when $\lambda \rightarrow \infty$?
- Note that it is also common to formulate this problem as $\underset{\mathbf{w}}{\operatorname{argmin}} \|\mathbf{X}\mathbf{w} - \mathbf{t}\|_2^2 + \frac{\lambda}{2} \|\mathbf{w}\|_2^2$ in which case the solution is $\mathbf{w}_\lambda^{\text{Ridge}} = (\mathbf{X}^T \mathbf{X} + \lambda \mathbf{I})^{-1} \mathbf{X}^T \mathbf{t}$.

ℓ_1 vs. ℓ_2 Regularization

- The ℓ_1 norm, or sum of absolute values, is another regularizer that encourages weights to be exactly zero. (How can you tell?)
- We can design regularizers based on whatever property we'd like to encourage.

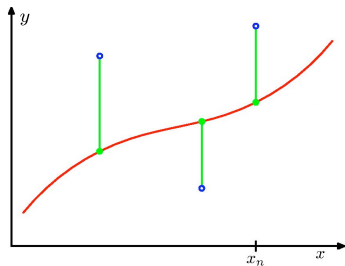


— Bishop, *Pattern Recognition and Machine Learning*

Probabilistic Interpretation of the Squared Error

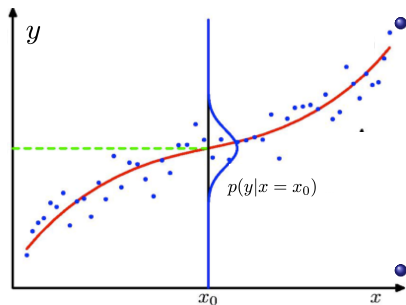
For the least squares: we minimize the sum of the squares of the errors between the predictions for each data point $\mathbf{x}^{(i)}$ and the corresponding target values $t^{(i)}$, i.e.,

$$\underset{(\mathbf{w}, \mathbf{w}_0)}{\text{minimize}} \sum_{i=1}^n (\mathbf{w}^\top \mathbf{x}^{(i)} + b - t^{(i)})^2$$



- $t \approx \mathbf{x}^\top \mathbf{w} + b$, $(\mathbf{w}, b) \in \mathbb{R}^D \times \mathbb{R}$
- We measure the quality of the fit using the squared error loss. Why?
- Even though the squared error loss looks natural, we did not really justify it.
- We provide a probabilistic perspective here.
- There are other justifications too; we get to them in the next lecture.

Probabilistic Interpretation of the Squared Error



- Suppose that our model arose from a statistical model ($b=0$ for simplicity):

$$y^{(i)} = \mathbf{w}^\top \mathbf{x}^{(i)} + \epsilon^{(i)}$$

where $\epsilon^{(i)} \sim \mathcal{N}(0, \sigma^2)$ is independent of anything else.

- Thus, $y^{(i)} | \mathbf{x}^{(i)} \sim p(y | \mathbf{x}^{(i)}, \mathbf{w}) = \mathcal{N}(\mathbf{w}^\top \mathbf{x}^{(i)}, \sigma^2)$.

Probabilistic Interpretation of the Squared Error:

Maximum Likelihood Estimation

- Suppose that the input data $\{\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \dots, \mathbf{x}^{(N)}\}$ are given and the outputs are independently drawn from

$$y^{(i)} \sim p(y|x^{(i)}, \mathbf{w})$$

with an unknown parameter \mathbf{w} . So the dataset is $\mathcal{D} = \{(\mathbf{x}^{(1)}, y^{(1)}), (\mathbf{x}^{(2)}, y^{(2)}), \dots, (\mathbf{x}^{(N)}, y^{(N)})\}$.

- The likelihood function is $\Pr(\mathcal{D}|\mathbf{w})$.
- The **maximum likelihood estimation (MLE)** is the “principle” that suggests we have to find a parameter $\hat{\mathbf{w}}$ that maximizes the likelihood, i.e.,

$$\hat{\mathbf{w}} \leftarrow \underset{\mathbf{w}}{\operatorname{argmax}} \Pr(\mathcal{D}|\mathbf{w}).$$

Maximum likelihood estimation: after observing the data samples $(\mathbf{x}^{(i)}, y^{(i)})$ for $i = 1, 2, \dots, N$, we should choose \mathbf{w} that maximizes the likelihood.

Probabilistic Interpretation of the Squared Error: Maximum Likelihood Estimation

- For independent samples, the likelihood function of samples \mathcal{D} is the product of their likelihoods

$$p(y^{(1)}, y^{(2)}, \dots, y^{(n)} | x^{(1)}, x^{(2)}, \dots, x^{(N)}, \mathbf{w}) = \prod_{i=1}^N p(y^{(i)} | x^{(i)}, \mathbf{w}) = L(\mathbf{w}).$$

- Product of N terms is not easy to minimize. Taking log reduces it to a sum! Two objectives are equivalent since log is strictly increasing.
- Maximizing the likelihood is equivalent to minimizing the negative log-likelihood:

$$\ell(\mathbf{w}) = -\log L(\mathbf{w}) = -\log \prod_{i=1}^N p(z^{(i)} | \mathbf{w}) = -\sum_{i=1}^n \log p(z^{(i)} | \mathbf{w})$$

Maximum Likelihood Estimator (MLE)

After observing $z^{(i)} = (\mathbf{x}^{(i)}, y^{(i)})$ for $i = 1, \dots, n$ i.i.d. samples from $p(z|\mathbf{w})$, MLE is

$$\mathbf{w}^{\text{MLE}} = \underset{\mathbf{w}}{\operatorname{argmin}} \quad l(\mathbf{w}) = -\sum_{i=1}^n \log p(z^{(i)} | \mathbf{w}).$$

Probabilistic Interpretation of the Squared Error: From MLE to Squared Error

- Suppose that our model arose from a statistical model:

$$y^{(i)} = \mathbf{w}^\top x^{(i)} + \epsilon^{(i)}$$

where $\epsilon^{(i)} \sim \mathcal{N}(0, \sigma^2)$ is independent of anything else.

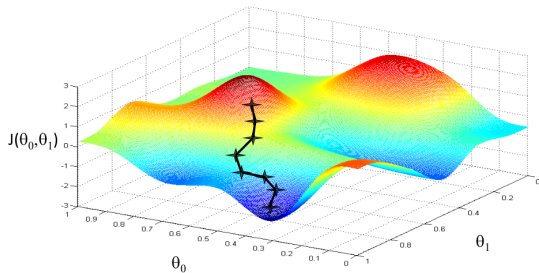
- $p(y^{(i)}|x^{(i)}, \mathbf{w}) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left\{-\frac{1}{2\sigma^2}(y^{(i)} - \mathbf{w}^\top x^{(i)})^2\right\}$
- $\log p(y^{(i)}|x^{(i)}, \mathbf{w}) = -\frac{1}{2\sigma^2}(y^{(i)} - \mathbf{w}^\top x^{(i)})^2 - \log(\sqrt{2\pi\sigma^2})$
- The MLE solution is

$$\mathbf{w}^{\text{MLE}} = \underset{\mathbf{w}}{\text{argmin}} \mathcal{L}(\mathbf{w}) = \frac{1}{2\sigma^2} \sum_{i=1}^N (y^{(i)} - \mathbf{w}^\top x^{(i)})^2 + C.$$

- As C and σ do not depend on \mathbf{w} , they do not contribute to the minimization.

$\mathbf{w}^{\text{MLE}} = \mathbf{w}^{\text{LS}}$ when we work with Gaussian densities.

Gradient Descent



Gradient Descent

- Now let's see a second way to minimize the cost function which is more broadly applicable: **gradient descent**.
- Gradient descent is an **iterative algorithm**, which means we apply an update repeatedly until some criterion is met.
- We **initialize** the weights to something reasonable (e.g. all zeros) and repeatedly adjust them in the **direction of steepest descent**.

Gradient Descent

- Observe:
 - ▶ if $\partial \mathcal{J} / \partial w_j > 0$, then increasing w_j increases \mathcal{J} .
 - ▶ if $\partial \mathcal{J} / \partial w_j < 0$, then increasing w_j decreases \mathcal{J} .
- The following update decreases the cost function:

$$\begin{aligned}w_j &\leftarrow w_j - \alpha \frac{\partial \mathcal{J}}{\partial w_j} \\&= w_j - \frac{\alpha}{N} \sum_{i=1}^N (y^{(i)} - t^{(i)}) x_j^{(i)}\end{aligned}$$

- α is a **learning rate**. The larger it is, the faster \mathbf{w} changes.
 - ▶ We'll see later how to tune the learning rate, but values are typically small, e.g. 0.01 or 0.0001

Gradient Descent

- This gets its name from the **gradient**:

$$\frac{\partial \mathcal{J}}{\partial \mathbf{w}} = \begin{pmatrix} \frac{\partial \mathcal{J}}{\partial w_1} \\ \vdots \\ \frac{\partial \mathcal{J}}{\partial w_D} \end{pmatrix}$$

- ▶ This is the direction of fastest increase in \mathcal{J} .
- Update rule in vector form:

$$\begin{aligned} \mathbf{w} &\leftarrow \mathbf{w} - \alpha \frac{\partial \mathcal{J}}{\partial \mathbf{w}} \\ &= \mathbf{w} - \frac{\alpha}{N} \sum_{i=1}^N (y^{(i)} - t^{(i)}) \mathbf{x}^{(i)} \end{aligned}$$

- Hence, gradient descent updates the weights in the direction of fastest *decrease*.
- Observe that once it converges, we get a critical point, i.e. $\frac{\partial \mathcal{J}}{\partial \mathbf{w}} = 0$.

Gradient Descent for Linear regression

- Even for linear regression, where there is a direct solution, we sometimes need to use GD.
- Why gradient descent, if we can find the optimum directly?
 - ▶ GD can be applied to a much broader set of models
 - ▶ GD can be easier to implement than direct solutions
 - ▶ For regression in high-dimensional spaces, GD is more efficient than direct solution
 - ▶ Linear regression solution: $(\mathbf{X}^T \mathbf{X})^{-1} \mathbf{X}^T \mathbf{t}$
 - ▶ matrix inversion is an $\mathcal{O}(D^3)$ algorithm
 - ▶ each GD update costs $\mathcal{O}(ND)$
 - ▶ Huge difference if $D \gg 1$

Gradient Descent under the ℓ_2 Regularization

- Recall the gradient descent update:

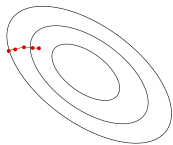
$$\mathbf{w} \leftarrow \mathbf{w} - \alpha \frac{\partial \mathcal{J}}{\partial \mathbf{w}}$$

- The gradient descent update of the regularized cost $\mathcal{J} + \lambda \mathcal{R}$ has an interesting interpretation as **weight decay**:

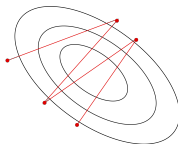
$$\begin{aligned}\mathbf{w} &\leftarrow \mathbf{w} - \alpha \left(\frac{\partial \mathcal{J}}{\partial \mathbf{w}} + \lambda \frac{\partial \mathcal{R}}{\partial \mathbf{w}} \right) \\ &= \mathbf{w} - \alpha \left(\frac{\partial \mathcal{J}}{\partial \mathbf{w}} + \lambda \mathbf{w} \right) \\ &= (1 - \alpha \lambda) \mathbf{w} - \alpha \frac{\partial \mathcal{J}}{\partial \mathbf{w}}\end{aligned}$$

Learning Rate (Step Size)

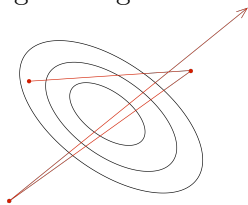
- In gradient descent, the learning rate α is a hyperparameter we need to tune. Here are some things that can go wrong:



α too small:
slow progress



α too large:
oscillations

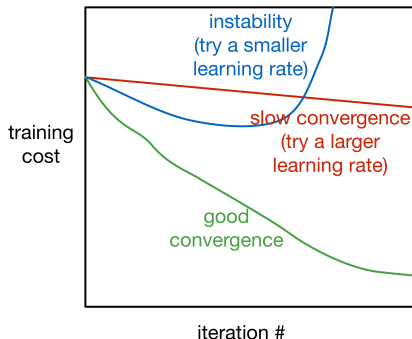


α much too large:
instability

- Good values are typically between 0.001 and 0.1. You should do a grid search if you want good performance (i.e. try 0.1, 0.03, 0.01, ...).

Training Curves

- To diagnose optimization problems, it's useful to look at **training curves**: plot the training cost as a function of iteration.



- Warning: it's very hard to tell from the training curves whether an optimizer has converged. They can reveal major problems, but they can't guarantee convergence.

Brief Matrix and Vector Calculus

- For a function $f : \mathbb{R}^p \rightarrow \mathbb{R}$, $\nabla f(z)$ denotes the gradient at z which points in the direction of the greatest rate of increase.
- $\nabla f(x) \in \mathbb{R}^p$ is a vector with $[\nabla f(x)]_i = \frac{\partial}{\partial x_i} f(x)$.
- $\nabla^2 f(x) \in \mathbb{R}^{p \times p}$ is a matrix with $[\nabla^2 f(x)]_{ij} = \frac{\partial^2}{\partial x_i \partial x_j} f(x)$
- At any minimum of a function f , we have $\nabla f(\mathbf{w}) = 0$, $\nabla^2 f(\mathbf{w}) \succeq 0$.
- Consider the problem minimize $\ell(\mathbf{w}) = \frac{1}{2} \|y - X\mathbf{w}\|_2^2$,
- $\nabla \ell(\mathbf{w}) = X^\top (X\mathbf{w} - y) = 0 \implies \hat{\mathbf{w}} = (X^\top X)^{-1} X^\top y$ (assuming $X^\top X$ is invertible)

At an arbitrary point x (old/new observation), our prediction is $y = \hat{\mathbf{w}}^\top x$.

Vectorization

- Computing the prediction using a for loop:

```
y = b
for j in range(M):
    y += w[j] * x[j]
```

- For-loops in Python are slow, so we **vectorize** algorithms by expressing them in terms of vectors and matrices.

$$\mathbf{w} = (w_1, \dots, w_D)^T \quad \mathbf{x} = (x_1, \dots, x_D)$$

$$y = \mathbf{w}^T \mathbf{x} + b$$

- This is simpler and much faster:

```
y = np.dot(w, x) + b
```


Why vectorize?

- The equations, and the code, will be simpler and more readable. Gets rid of dummy variables/indices!
- Vectorized code is much faster
 - ▶ Cut down on Python interpreter overhead
 - ▶ Use highly optimized linear algebra libraries
 - ▶ Matrix multiplication is very fast on a Graphics Processing Unit (GPU)

Conclusion

Linear regression exemplifies recurring themes of this course:

- choose a **model** and a **loss function**
- formulate an **optimization problem**
- solve the minimization problem using one of two strategies
 - ▶ **direct solution** (set derivatives to zero)
 - ▶ **gradient descent** (see appendix)
- **vectorize** the algorithm, i.e. represent in terms of linear algebra
- make a linear model more powerful using **features**
- improve the generalization by adding a **regularizer**
- Probabilistic Interpretation as MLE with Gaussian noise model