Deep Transfer Learning

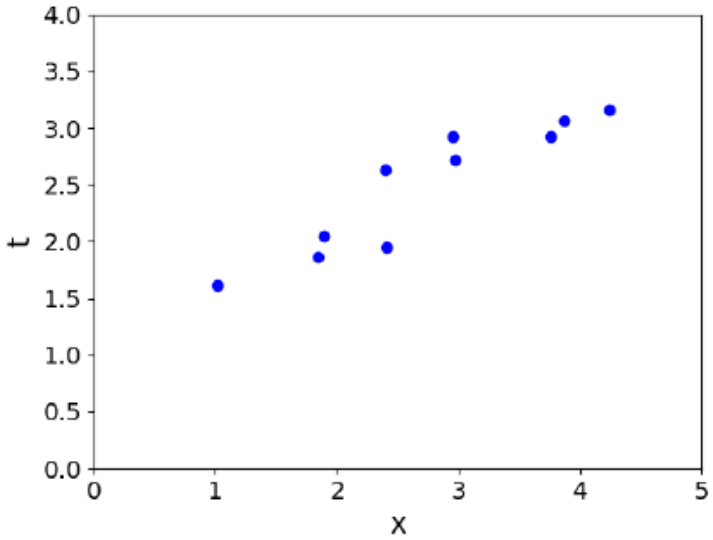
Linear Regression & Optimization

Ashley Gao

William & Mary

Feburary 01, 2024

Supervised Learning Setup



- In supervised learning:
  - There is input $x \in \mathcal{X}$, typically a vector of features (or covariates)
  - There is target $t \in \mathcal{T}$ (also called response, outcome, output, class)
  - Objective is to learn a function $f : \mathcal{X} \to \mathcal{T}$ such that $t \approx y = f(x)$ based on the dataset $\mathcal{D} = \{(x^{(i)}, t^{(i)})\}$ for $i = 1, 2, \ldots, N$.

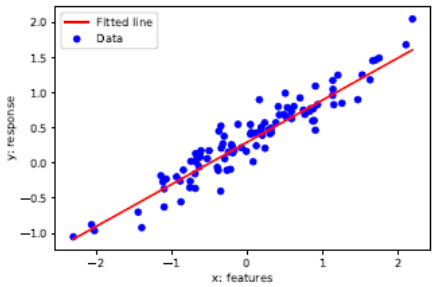
Linear Regression - Model

- Model: In linear regression, we use a linear function of the features $x = (x_1, ..., x_D) \in \mathbb{R}^D$ to make prediction y of the target value $t \in \mathbb{R}$:

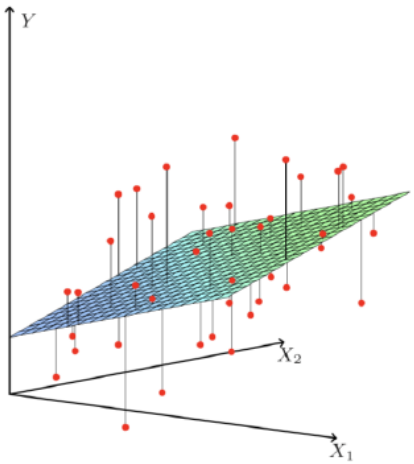
$$y = f(x) = \sum_j w_j x_j + b \tag{1}$$

  - y is the prediction
  - w is the weights
  - b is the bias (or intercept)
- 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

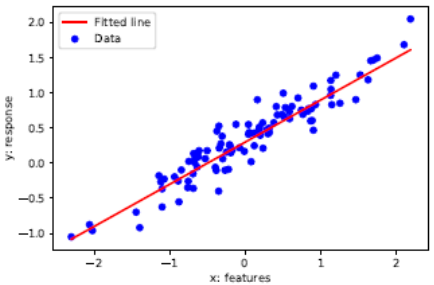


- If we have only D feature:
  $y = \boldsymbol{w}^\top \boldsymbol{x} + b$ where $\boldsymbol{w}, \boldsymbol{x} \in \mathbb{R}^D$ and $b \in \mathbb{R}$
- y is linear in x

- Relation between the prediction y and inputs x is linear in both cases.

Linear Regression

- We have a dataset $\mathcal{D} = \{(x^{(i)}, t^{(i)})\}$ for $i = 1, 2, ..., N$, where
  - $x^{(i)} = (x_1^{(i)}, x_2^{(i)}, ..., x_D^{(i)})^\top \in \mathbb{R}^D$ are the inputs (i.e. age, height)
  - $t^{(i)} \in \mathbb{R}$ is the target or response (i.e. income)
  - Predict $t^{(i)}$ with a linear function of $x^{(i)}$



  - $t^{(i)} \approx y^{(i)} = \boldsymbol{w}^\top \boldsymbol{x} + b$
  - Different $(\boldsymbol{w}, b)$ combinations define different lines
  - We want the best line $(\boldsymbol{w}, b)$
  - How to quantify “best"?
- Relation between the prediction y and inputs x is linear in both cases.

Linear Regression - Loss Function

- A loss function $\mathcal{L}(y, t)$ defines how bad it is if, for some example $\boldsymbol{x}$, the algorithm predicts y, but the target is actually t.
- Squared error loss function:

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

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

$$\mathcal{J}(\boldsymbol{w}, b) = \frac{1}{2N}\sum_{i=1}^{N}(y^{(i)} - t^{(i)})^2 = \frac{1}{2N}\sum_{i=1}^{N}(\boldsymbol{w}^\top \boldsymbol{x}^{(i)} + b - t^{(i)})^2 \tag{3}$$

- Terminology varies. Some call "cost" empirical or average loss.

Vectorization

- Notion-wise, $\frac{1}{2N}\sum_{i=1}^{N}(y^{(i)} - t^{(i)})^2$ gets messy if we expand $y^{(i)}$:

$$\frac{1}{2N}\sum_{i=1}^{N}(\sum_{j=1}^{D}(w_j x_j^{(i)} + b) - t^{(i)})^2 \tag{4}$$

- The code equivalent is to compute the prediction using a for loop:

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

- Excessive super/sub scripts are hard to work with, and Python loops are slow, so we vectorize algorithms by expressing them in terms of vectors and matrices.

$$\boldsymbol{w} = (w_1, ..., w_D)^\top; \boldsymbol{x} = (x_1, ..., x_D); y = \boldsymbol{w}^\top \boldsymbol{x} + b \tag{5}$$

- This is simpler and executes much faster:

$$y = np.dot(\boldsymbol{w}, \boldsymbol{x}) + b \tag{6}$$

Vectorization

- Why vectorize?
  - The equations, and the code, will be simpler and more readable. Gets rid of dummy variables and indices!
  - Vectorized code is much faster
    - Cut down on Python interpreter overhead
    - Use highly optimized linear algebra libraries (hardware support)
    - Matrix multiplication very fast on GPU (Graphics Processing Unit)
- Switching in and out of vectorized form is a skill you gain with practice
  - Some algorithms are easier to write/understand using for-loops and vectorize later for performance

Vectorization

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

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:

$$\boldsymbol{y} = \boldsymbol{X}\boldsymbol{w} + b\mathbf{1}; \mathcal{J} = \frac{1}{2N}||y - t||^2 \tag{7}$$

- Sometimes we use $\mathcal{J} = \frac{1}{2}||y - t||^2$ without a normalizer. This would correspond to the sum of losses, and not the averaged loss. The minimizer does not depend on N (but optimization might!).
- We can also add a column of 1's to design matrix, combine the bias and the weights, and conveniently write

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

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

Vectorization

- We have defined a cost function. This is what we'd like to minimize.
- Two commonly applied mathematical approaches:
  - Algebraic, e.g., using inequalities:
    - To show that z^* minimizes $f(z)$, show that $\forall z, f(z) \geq f(z^*)$
  - Calculus: 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).
- Solutions may be direct or iterative
  - Sometimes we can directly find provably optimal parameters (e.g. set the gradient to zero and solve in closed form). We call this a 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: Calculus

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

$$\frac{\partial}{\partial x_1} f(x_1, x_2) = \lim_{h \to 0} \frac{f(x_1 + h, x_2) - f(x_1, x_2)}{h} \tag{8}$$

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

$$\frac{\partial y}{\partial w_j} = \frac{\partial}{\partial w_j} \left(\sum_{j'} w_{j'} x_{j'} + b \right) = x_j \tag{9}$$

$$\frac{\partial y}{\partial b} = \frac{\partial}{\partial b} \left(\sum_{j'} w_{j'} x_{j'} + b \right) = 1 \tag{10}$$

Direct Solution: Calculus

- For loss derivatives, apply the chain rule:

$$\frac{\partial(L)}{\partial w_j} = \frac{d(L)}{dy}\frac{\partial(y)}{\partial w_j} = \frac{d}{dy}\left(\frac{1}{2}(y-t)^2\right)x_j = (y-t)x^j \tag{11}$$

$$\frac{\partial(L)}{\partial b} = \frac{d(L)}{dy}\frac{\partial(y)}{\partial b} = y - t \tag{12}$$

- For cost derivatives, use linearity and average over data points.
- Minimum must occur at a point where partial derivatives are zero.

$$\frac{\partial(J)}{\partial w_j} = \frac{1}{N}\sum_{i=1}^{N}(y^{(i)} - t^{(i)})x_j^{(i)} = 0 \tag{13}$$

$$\frac{\partial(J)}{\partial b} = \frac{1}{N}\sum_{i=1}^{N} y^{(i)} - t^{(i)} = 0 \tag{14}$$

- if $\frac{\partial(J)}{\partial w_j} \neq 0$, you could reduce the cost by changing w_j

Direct Solution: Calculus

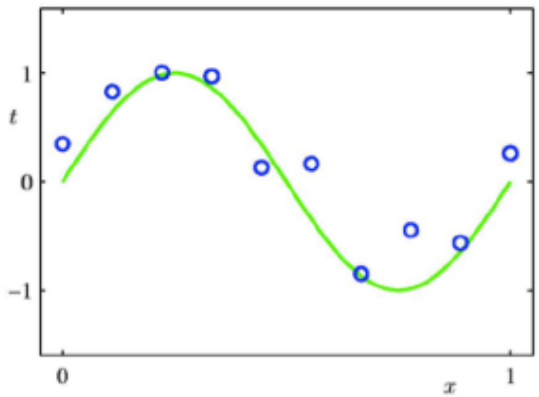
- The derivation on the previous slide gives a system of linear equations, which we can solve efficiently.
- As is often the case for models and code, however, the solution is easier to characterize if we vectorize our calculus.
- We call the vector of partial derivatives the gradient
- Thus, the gradient of $f : \mathbb{R}^D \rightarrow \mathbb{R}$, denoted $\nabla f(\boldsymbol{w})$, is:

$$\left(\frac{\partial}{\partial w_1} f(\boldsymbol{w}), ..., \frac{\partial}{\partial w_D} f(\boldsymbol{w}) \right)^\top \tag{15}$$

- The gradient points in the direction of the greatest rate of increase.
- Analogue of the second derivative (the Hessian matrix): $\nabla^2 f(\boldsymbol{w}) \in \mathbb{R}^{D \times D}$ is a matrix with $[\nabla^2 f(\boldsymbol{w})]_{i,j} = \frac{\partial^2}{\partial w_i \partial w_j} f(\boldsymbol{w})$

Feature Mapping (Basic Expansion)

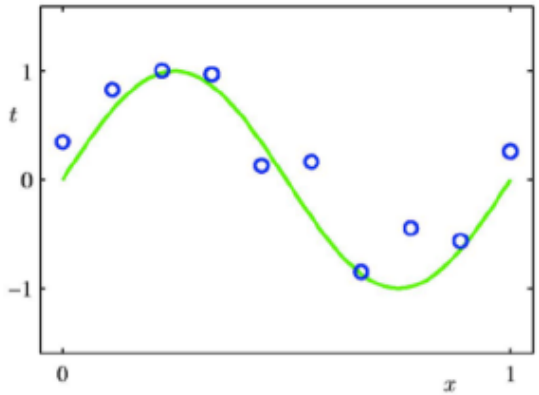
- The relation between the input and output may not be linear.



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

Feature Mapping (Basic Expansion)

- If the relationship doesn't look linear, we can fit a polynomial.



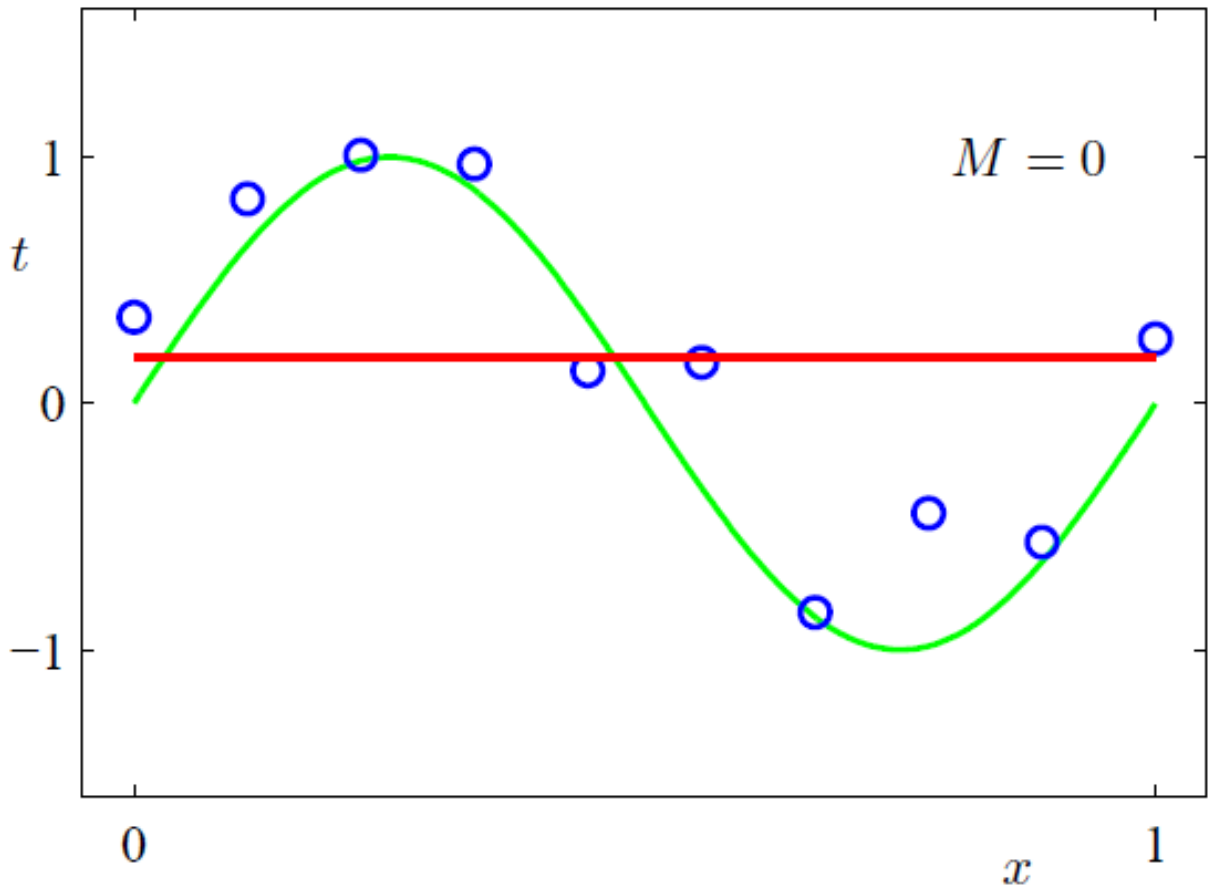
- Fit the data using a degree-M polynomial function of the form:

$$y = w_0 + w_1 x + w_2 x^2 + \ldots + w_M x^M = \sum_{i=0}^{M} w_i x^i \tag{16}$$

- Here the feature mapping is $\psi(x) = [1, x, x^2, \ldots, x^M]^\top$
- We can still use linear regression to find $\boldsymbol{w}$ since $y = \psi(x)^\top$ is linear in $w_0, w_1, \ldots$, because the coefficients are still **linear**!

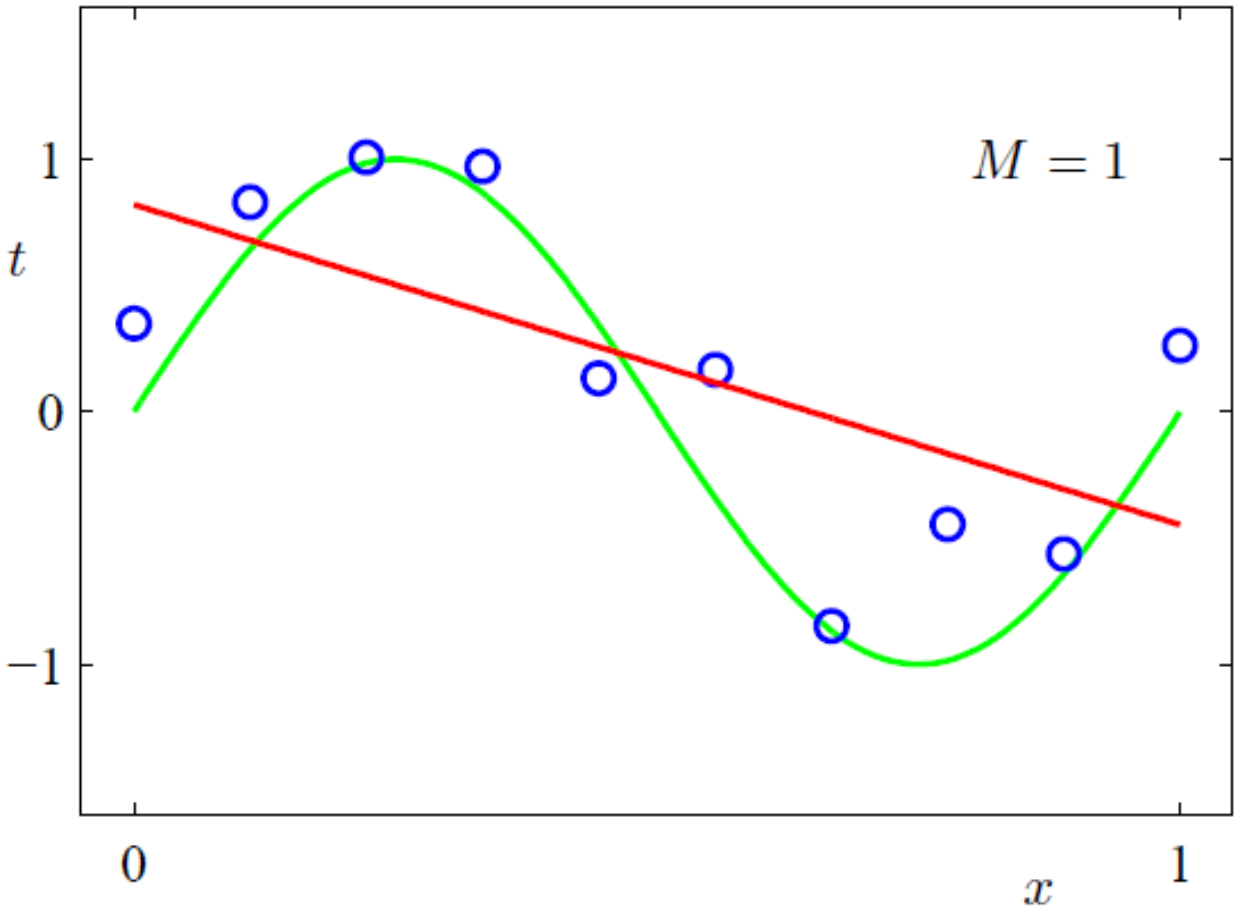
Polynomial Feature Mapping with M = 0

$$y = w_0$$



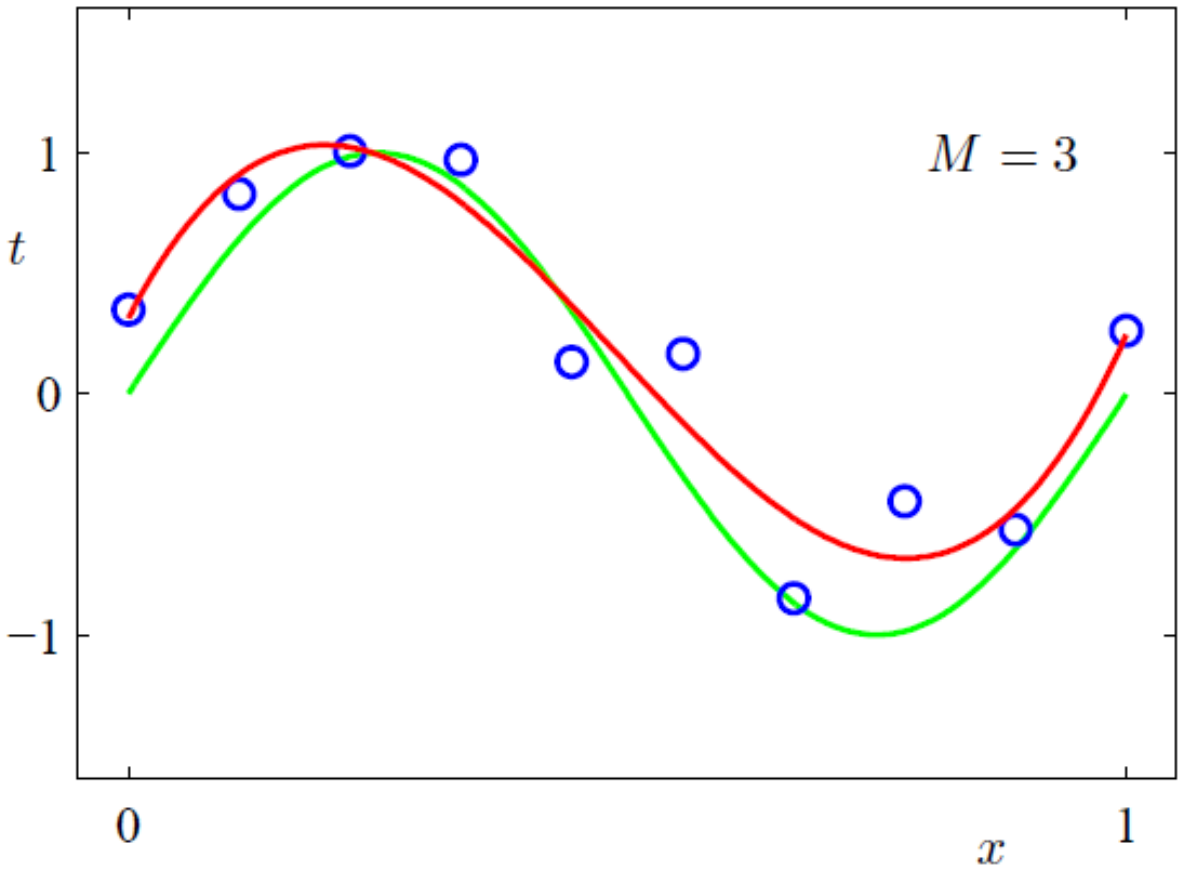
Polynomial Feature Mapping with M = 1

$$y = w_0 + w_1 x$$



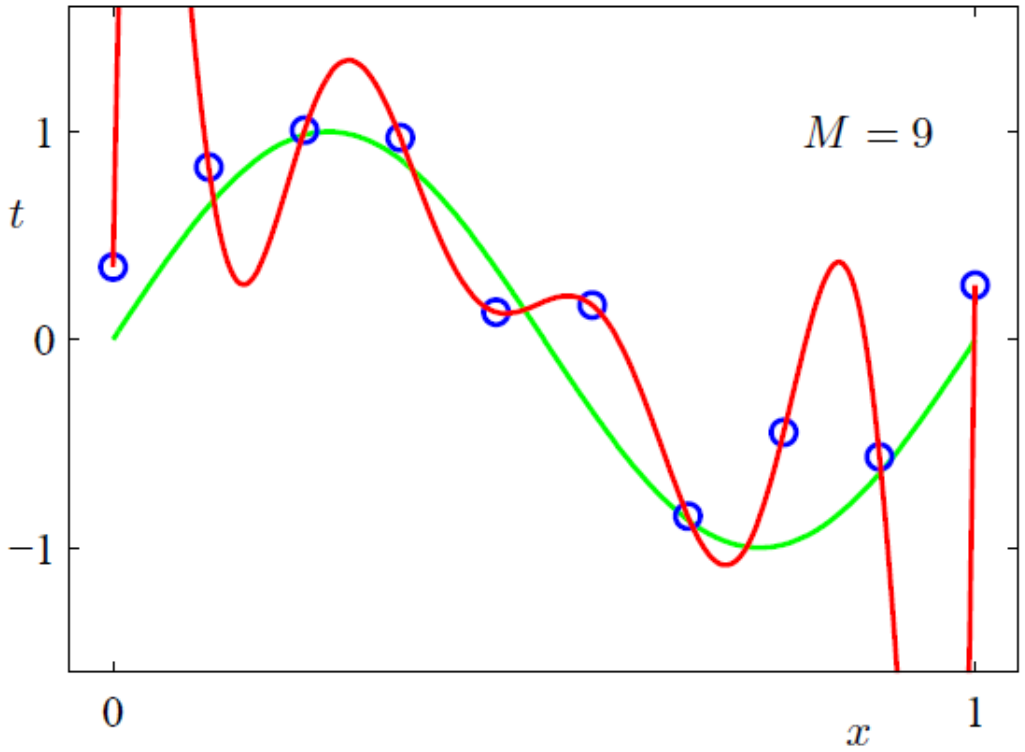
Polynomial Feature Mapping with M = 3

$$y = w_0 + w_1 x + w_2 x^2 + w_3 x^3$$



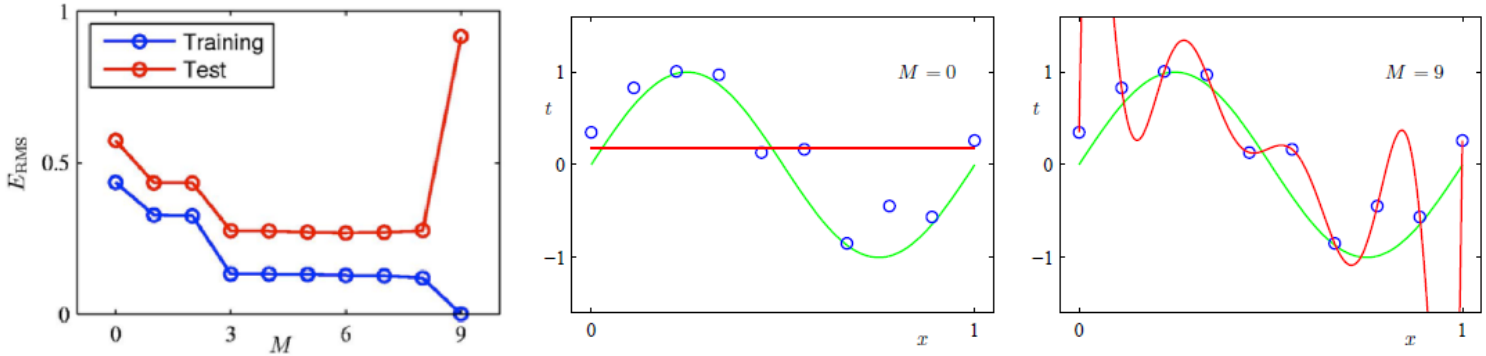
Polynomial Feature Mapping with M = 9

$$y = w_0 + w_1 x + w_2 x^2 + w_3 x^3 + \ldots + w_9 x^9$$

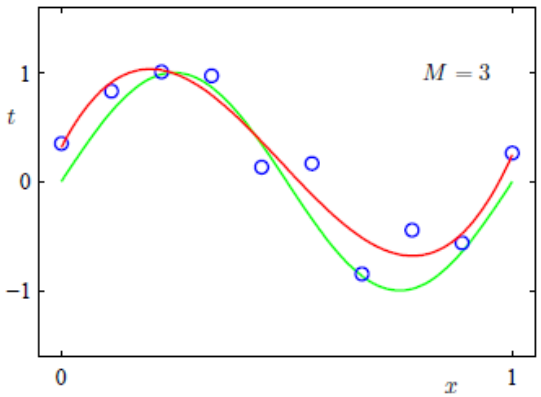


Model Complexity and Generalization

- Underfitting (M=0): model is too simple → does not fit the data.
- Overfitting (M=9): model is too complex → fits perfectly.

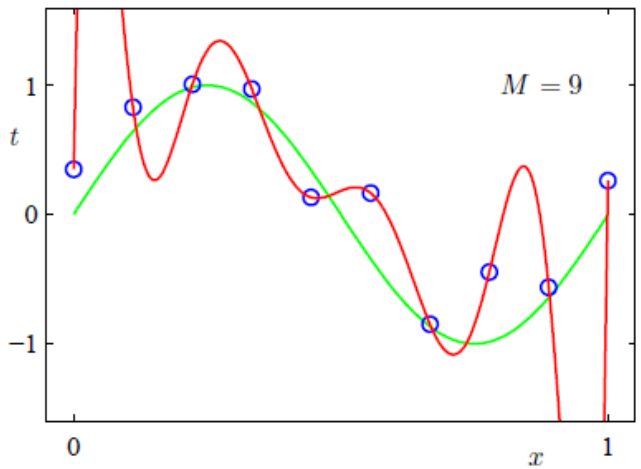


- Good model (M=3): Achieves small test error (generalizes well).



Model Complexity and Generalization

	$M=0$	$M=1$	$M=3$	$M=9$
$w_0^\star$	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



- 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

- The degree M of the polynomial controls the model's complexity.
- 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 is a crude approach to controlling the model complexity.
- Another approach: keep the model large, but regularize it
  - Regularizer: a function that quantifies how much we prefer one hypothesis vs. another

L^2 or l_2 Regularization

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

$$\mathcal{R}(\boldsymbol{w}) = \frac{1}{2}||\boldsymbol{w}||_2^2 = \frac{1}{2}\sum_j w_j^2 \tag{17}$$

  - Note: To be precise, the L^2 norm is Euclidean distance, so we're regularizing the squared L^2 norm.
- The regularized cost function makes a tradeoff between fit to the data and the norm of the weights.

$$\mathcal{J}_{reg}(\boldsymbol{w}) = \mathcal{J}(\boldsymbol{w}) + \lambda\mathcal{R}(\boldsymbol{w}) = \mathcal{J}(\boldsymbol{w}) + \frac{1}{2}\sum_j w_j^2 \tag{18}$$

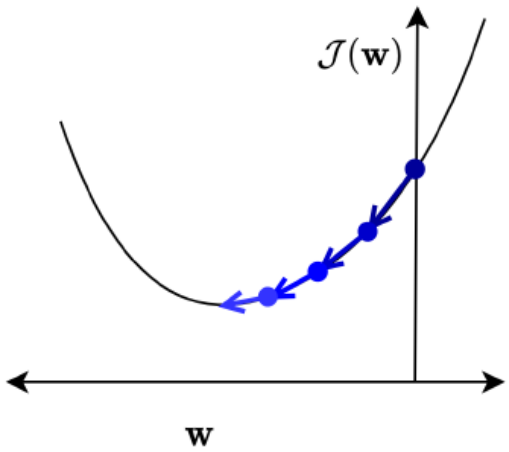
- If you fit training data poorly, $\mathcal{J}$ is large. If your optimal weights have high values, $\mathcal{R}$ is large. Large λ penalizes weight values more.
- Like M, λ is a hyperparameter we can tune with a validation set.

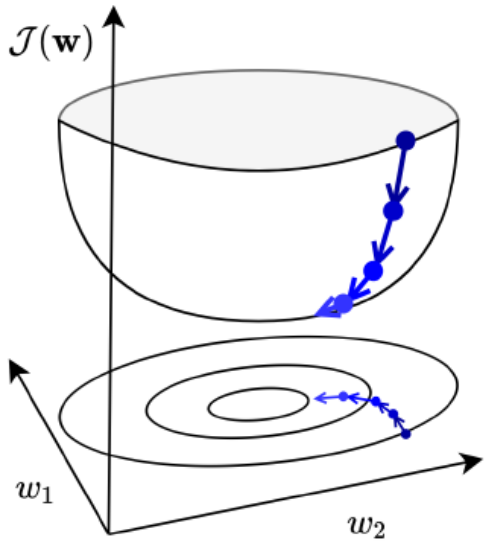
Conclusion So Far

- 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 (next topic)
  - 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

Gradient Descent

- Now let's see a second way to minimize the cost function which is more broadly applicable: gradient descent.
- Many times, we do not have a direct solution: Taking derivatives of $\mathcal{J}$ w.r.t w and setting them to 0 doesn't have an explicit solution.
- 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 the steepest descent.





Gradient Descent

- Observe:
  - if $\frac{\partial \mathcal{J}}{\partial w_j} > 0$, then increasing w_j increases $\mathcal{J}$
  - if $\frac{\partial \mathcal{J}}{\partial w_j} < 0$, then increasing w_j decreases $\mathcal{J}$
- The following update always decreases the cost function for small enough α unless $\frac{\partial \mathcal{J}}{\partial w_j} = 0$:

$$w_j \leftarrow w_j - \alpha \frac{\partial \mathcal{J}}{\partial w_j} \tag{19}$$

- $\alpha > 0$ is a learning rate (or step size). The larger it is, the faster 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:

$$\nabla_{\boldsymbol{w}} \mathcal{J} = \frac{\partial \mathcal{J}}{\partial \boldsymbol{w}} = \left(\frac{\partial \mathcal{J}}{\partial w_1}, \ldots, \frac{\partial \mathcal{J}}{\partial w_D} \right) \tag{20}$$

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

$$\boldsymbol{w} \leftarrow \boldsymbol{w} - \alpha \frac{\partial \mathcal{J}}{\partial \boldsymbol{w}} \tag{21}$$

- And for linear regression we have:

$$\boldsymbol{w} \leftarrow \boldsymbol{w} - \frac{\alpha}{N} \sum_{i=0}^{N} (y^{(i)} - t^{(i)}) \boldsymbol{x}^{(i)} \tag{22}$$

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

Gradient Descent

- The squared error loss of linear regression is a convex function.
- 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 space, GD is more efficient than direct solution
    - Linear regression solution: $(\boldsymbol{X}^\top \boldsymbol{X})^{-1}\boldsymbol{X}^\top \boldsymbol{t}$
    - Matrix inversion: $\mathcal{O}(D^3)$
    - Each GD update costs $\mathcal{O}(ND)$
    - Or less with stochastic GD (SGD, in a few slides)
    - Huge difference if $D \gg 1$

Gradient Descent Under the L^2 Regularization

- Gradient descent update to minimize $\mathcal{J}$:

$$\boldsymbol{w} \leftarrow \boldsymbol{w} - \alpha \frac{\partial \mathcal{J}}{\partial \boldsymbol{w}} \tag{23}$$

- The gradient descent update to minimize the L^2 regularized cost $\mathcal{J} + \lambda \mathcal{R}$ results in weight decay:

$$\boldsymbol{w} \leftarrow \boldsymbol{w} - \alpha \frac{\partial}{\partial \boldsymbol{w}} (\mathcal{J} + \lambda \mathcal{R}) \tag{24}$$

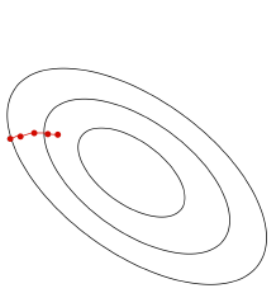
$$\boldsymbol{w} - \alpha \frac{\partial}{\partial \boldsymbol{w}} (\mathcal{J} + \lambda \mathcal{R}) = \boldsymbol{w} - \alpha \left(\frac{\partial \mathcal{J}}{\partial \boldsymbol{w}} + \lambda \frac{\partial \mathcal{R}}{\partial \boldsymbol{w}} \right) \tag{25}$$

$$\boldsymbol{w} - \alpha \left(\frac{\partial \mathcal{J}}{\partial \boldsymbol{w}} + \lambda \frac{\partial \mathcal{R}}{\partial \boldsymbol{w}} \right) = \boldsymbol{w} - \alpha \left(\frac{\partial \mathcal{J}}{\partial \boldsymbol{w}} + \lambda \boldsymbol{w} \right) \tag{26}$$

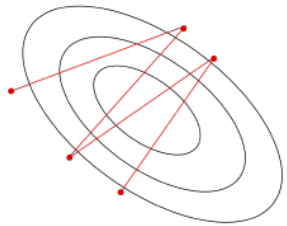
$$\boldsymbol{w} \leftarrow (1 - \alpha\lambda)\boldsymbol{w} - \alpha \frac{\partial \mathcal{J}}{\partial \boldsymbol{w}} \tag{27}$$

Learning Rate (Step Size)

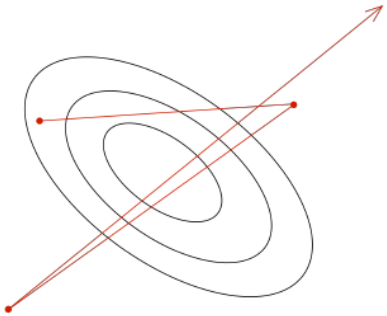
- In gradient descent, the learning rate α is a hyperparameter we need to tune. Here are some things that can go wrong:



α too small:
slow progress



α too large:
oscillations

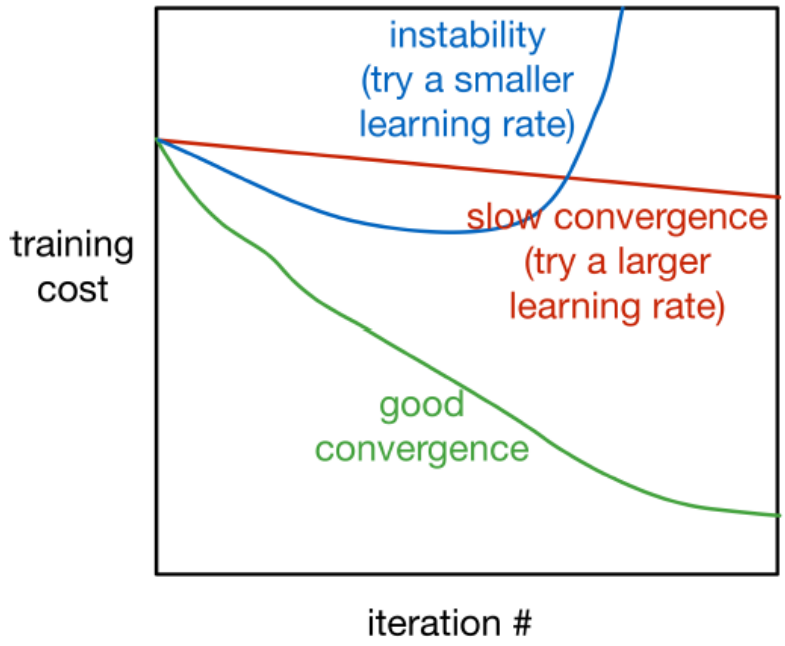


α much too large:
instability

- Good values are typically between 0.001 and 0.1. You should do a grid search if you want good performance.

Training Curve

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



- Warning: in general, 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.

Stochastic Gradient Descent

- So far, the cost function $\mathcal{J}$ has been the average loss over the training examples:

$$\mathcal{J}(\boldsymbol{\theta}) = \frac{1}{N}\sum_{i=1}^{N}\mathcal{L}^{(i)} = \frac{1}{N}\sum_{i=1}^{N}\mathcal{L}(y(\boldsymbol{x}^{(i)}, \boldsymbol{\theta}), \boldsymbol{t}^{(i)}) \tag{28}$$

- $\boldsymbol{\theta}$ denotes the parameters; e.g., in linear regression, $\boldsymbol{\theta} = (\boldsymbol{w}, b)$
- By linearity,

$$\frac{\partial \mathcal{J}}{\partial \boldsymbol{\theta}} = \frac{1}{N}\sum_{i=1}^{N}\frac{\partial \mathcal{L}^{(i)}}{\partial \boldsymbol{\theta}} \tag{29}$$

- Computing the gradient requires summing over all of the training examples. This is known as batch training.
- Batch training is impractical if you have a large dataset $N \gg 1$ (e.g. millions of training examples)!

Stochastic Gradient Descent

- Stochastic gradient descent (SGD): update the parameters based on the gradient for a single training example,
  - Choose i uniformly at random;
  - $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \alpha \frac{\partial \mathcal{L}^{(i)}}{\partial \boldsymbol{\theta}}$
- Cost of each SGD update is independent of N!
- SGD can make significant progress before even seeing all the data!
- Mathematical justification: if you sample a training example uniformly at random, the stochastic gradient is an unbiased estimate of the batch gradient:

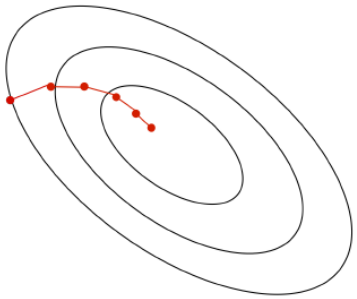
$$\mathbb{E}\left[\frac{\partial \mathcal{L}^{(i)}}{\partial \boldsymbol{\theta}}\right] = \frac{1}{N}\sum_{i=1}^{N}\frac{\partial \mathcal{L}^{(i)}}{\partial \boldsymbol{\theta}} = \frac{\partial \mathcal{J}}{\partial \boldsymbol{\theta}} \tag{30}$$

Stochastic Gradient Descent

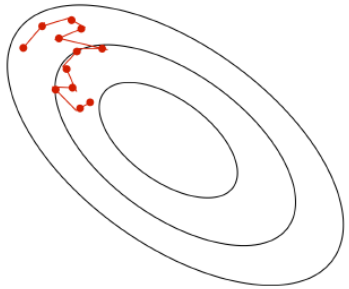
- Problems with using a single training example to estimate gradient:
  - Variance in the estimate may be high
  - We can't exploit efficient vectorized operations
- Compromise approach:
  - Compute the gradients on a randomly chosen medium-sized set of training $\mathcal{M} \subset \{1, ..., N\}$ examples, called a mini-batch.
- Stochastic gradients computed on larger mini-batches have smaller variances.
- The mini-batch size $|\mathcal{M}|$ is a hyperparameter that needs to be set.
  - Too large: requires more compute; e.g., it takes more memory to store the activations, and longer to compute each gradient update
  - Too small: can't exploit vectorization, has high variance
  - reasonable value might be $|\mathcal{M}| = 100$.

Stochastic Gradient Descent

- Batch gradient descent moves directly downhill (locally speaking).
- SGD takes steps in a noisy direction, but moves downhill on average.



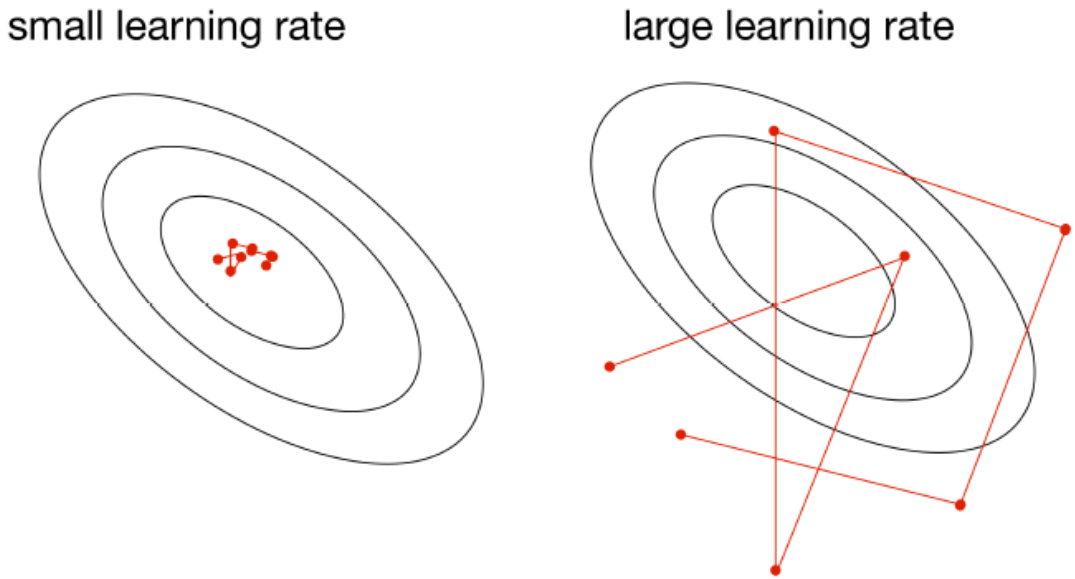
batch gradient descent



stochastic gradient descent

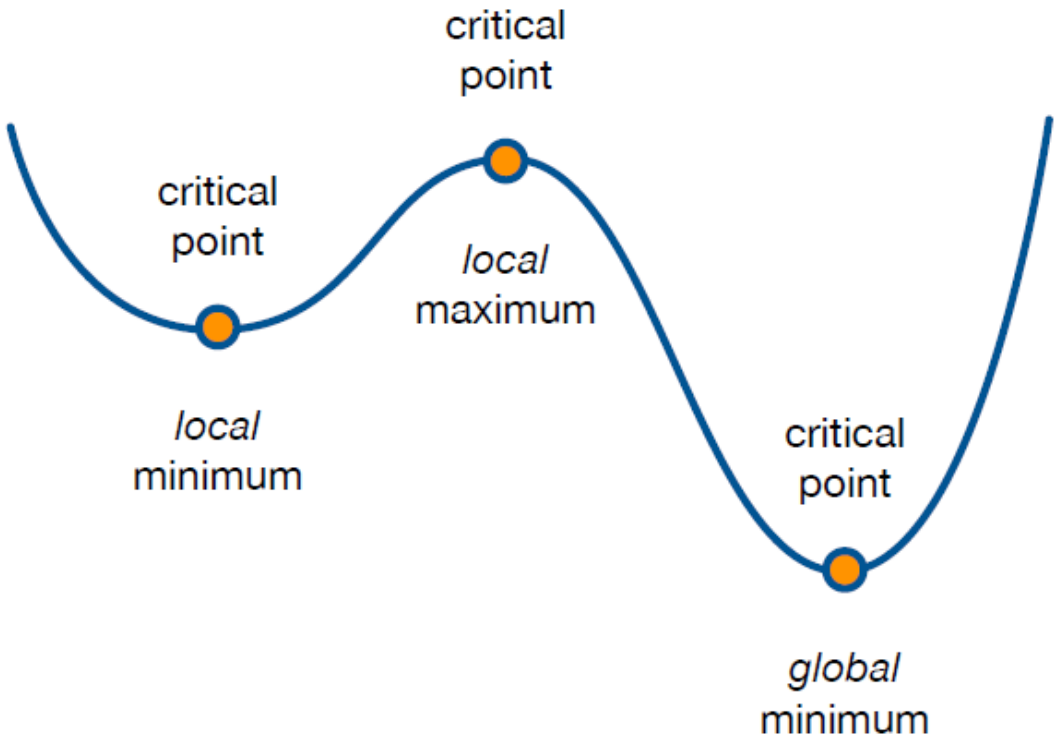
SDG Learning Rate

- In stochastic training, the learning rate also influences the fluctuations due to the stochasticity of the gradients.
- Stochasticity, in the context of machine learning, refers to the introduction of randomness or probabilistic elements into the learning process.



- Typical strategy:
  - Use a large learning rate early in training so you can get close to the optimum
  - Gradually decay the learning rate to reduce the fluctuations

When Are Critical Points Optimal?



- Gradient descent finds a critical point, but it may be a local optima.
- Convexity is a property that guarantees that all critical points are global minima.

Conclusion

- In this lecture, we looked at linear regression, which exemplifies a modular approach that will be used throughout this course:
  - choose a model describing the relationships between variables of interest (linear)
  - define a loss function quantifying how bad the fit to the data is (squared error)
  - choose a regularizer to control the model complexity/overfitting (L^2, L^p regularization)
  - fit/optimize the model (gradient descent, stochastic gradient descent, convexity)
- By mixing and matching these modular components, we can obtain new ML methods.