Announcements

- Homework 1 is posted! Deadline Sept 30, 23:59.

- Instructor hours are announced on the course website. (TA OH TBA)

- No ProctorU!
Overview

- Second learning algorithm of the course: linear regression.
  - Task: predict scalar-valued targets (e.g. stock prices)
  - Architecture: linear function of the inputs

- While KNN was a complete algorithm, linear regression exemplifies a modular approach that will be used throughout this course:
  - 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 a model that minimizes the loss function and satisfies the constraint/penalty imposed by the regularizer, possibly using an optimization algorithm

- Mixing and matching these modular components give us a lot of new ML methods.
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 $f : \mathcal{X} \rightarrow \mathcal{T}$ such that $t \approx y = f(x)$ based on some data $D = \{(x^{(i)}, t^{(i)})$ for $i = 1, 2, ..., N\}$. 

![Graph showing supervised learning data points]
Model: In linear regression, we use a \textit{linear} function of the features 
\( x = (x_1, \ldots, x_D) \in \mathbb{R}^D \) to make predictions \( y \) of the target value \( t \in \mathbb{R}: \)

\[
y = f(x) = \sum_j w_j x_j + b
\]

- \( y \) 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:
  \[ y = wx + b \text{ where } w, x, b \in \mathbb{R}. \]
  - \( y \) is linear in \( x \).

- If we have \( D \) features:
  \[ y = \mathbf{w}^\top \mathbf{x} + b \text{ where } \mathbf{w}, \mathbf{x} \in \mathbb{R}^D, \]
  - \( y \) 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 $\mathcal{D} = \{(\mathbf{x}^{(i)}, t^{(i)}) \mid i = 1, 2, \ldots, N\}$ where,

- $\mathbf{x}^{(i)} = (x_1^{(i)}, x_2^{(i)}, \ldots, x_D^{(i)})^\top \in \mathbb{R}^D$ 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 $\mathbf{x}^{(i)}$:

$$t^{(i)} \approx y^{(i)} = \mathbf{w}^\top \mathbf{x}^{(i)} + b$$

- Different $(\mathbf{w}, b)$ define different lines.
- We want the “best” line $(\mathbf{w}, b)$.
- How to quantify “best”? 
A loss function $\mathcal{L}(y, t)$ defines how bad it is if, for some example $x$, 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 this small in magnitude.

The $\frac{1}{2}$ factor is just to make the calculations convenient.

Cost function: loss function averaged over all training examples

$$\mathcal{J}(w, b) = \frac{1}{2N} \sum_{i=1}^{N} \left( y^{(i)} - t^{(i)} \right)^2$$

$$= \frac{1}{2N} \sum_{i=1}^{N} \left( w^\top x^{(i)} + b - t^{(i)} \right)^2$$

Terminology varies. Some call “cost” empirical or average loss.
Vectorization

- Notation-wise, \( \frac{1}{2N} \sum_{i=1}^{N} (y^{(i)} - t^{(i)})^2 \) gets messy if we expand \( y^{(i)} \):
  \[
  \frac{1}{2N} \sum_{i=1}^{N} \left( \sum_{j=1}^{D} (w_j x_{j}^{(i)} + b) - t^{(i)} \right)^2
  \]

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

  \[
  \mathbf{w} = (w_1, \ldots, w_D)^	op \quad \mathbf{x} = (x_1, \ldots, x_D)^	op
  \]

  \[
  y = \mathbf{w}^\top \mathbf{x} + b
  \]

- This is simpler and executes much faster:
  ```python
  y = np.dot(w, x) + b
  ```
Vectorization

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 (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 derivations are easier to do element-wise
- 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 $X$ with one row per training example, and all the targets into the target vector $t$.

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

- Computing the predictions for the whole dataset:

\[
Xw + b \mathbf{1} = \begin{pmatrix}
    w^T x^{(1)} + b \\
    \vdots \\
    w^T x^{(N)} + b
\end{pmatrix} = \begin{pmatrix}
    y^{(1)} \\
    \vdots \\
    y^{(N)}
\end{pmatrix} = y
\]
Vectorization

- Computing the squared error cost across the whole dataset:

\[ y = Xw + b1 \]

\[ J = \frac{1}{2N} \| y - t \|^2 \]

- Sometimes we may use \( J = \frac{1}{2} \| y - t \|^2 \), 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

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

Then, our predictions reduce to \( y = Xw \).
Solving the Minimization Problem

We defined a cost function. This is what we’d like to minimize.

Two commonly applied mathematical approaches:

- **Algebraic**, e.g., using inequalities:
  - to show $z^*$ minimizes $f(z)$, show that $\forall z, f(z) \geq f(z^*)$
  - to show that $a = b$, show that $a \geq b$ and $b \geq a$

- **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 I: Linear Algebra

- We seek $w$ to minimize $\|Xw - t\|^2$, or equivalently $\|Xw - t\|$
- $\text{range}(X) = \{Xw \mid w \in \mathbb{R}^D\}$ is a $D$-dimensional subspace of $\mathbb{R}^N$.
- Recall that the closest point $y^* = Xw^*$ in subspace $\text{range}(X)$ of $\mathbb{R}^N$ to arbitrary point $t \in \mathbb{R}^N$ is found by orthogonal projection.

- We have $(y^* - t) \perp Xw$, $\forall w \in \mathbb{R}^D$
- Why is $y^*$ the closest point to $t$?
  - Consider any $z = Xw$
  - By Pythagorean theorem and the trivial inequality ($x^2 \geq 0$):
    \[
    \|z - t\|^2 = \|y^* - t\|^2 + \|y^* - z\|^2 \\
    \geq \|y^* - t\|^2
    \]
From the previous slide, we have \((y^* - t) \perp Xw, \forall w \in \mathbb{R}^D\)

Equivalently, the columns of the design matrix \(X\) are all orthogonal to \((y^* - t)\), and we have that:

\[
X^\top (y^* - t) = 0
\]

\[
X^\top Xw^* - X^\top t = 0
\]

\[
X^\top Xw^* = X^\top t
\]

\[
w^* = (X^\top X)^{-1}X^\top t
\]

While this solution is clean and the derivation easy to remember, like many algebraic solutions, it is somewhat ad hoc.

On the hand, the tools of calculus are broadly applicable to differentiable loss functions...
Partial derivative: derivative of a multivariate function with respect to one of its arguments.

\[
\frac{\partial}{\partial x_1} f(x_1, x_2) = \lim_{h \to 0} \frac{f(x_1 + h, x_2) - f(x_1, x_2)}{h}
\]

To compute, take the single variable derivative, pretending the other arguments are constant.

Example: partial derivatives of the prediction \( y \)

\[
\frac{\partial y}{\partial w_j} = \frac{\partial}{\partial w_j} \left[ \sum_{j'} w_{j'} x_{j'} + b \right] = x_j
\]

\[
\frac{\partial y}{\partial b} = \frac{\partial}{\partial b} \left[ \sum_{j'} w_{j'} x_{j'} + b \right] = 1
\]
For loss derivatives, apply the chain rule:

\[
\frac{\partial L}{\partial w_j} = \frac{dL}{dy} \frac{\partial y}{\partial w_j} = \frac{d}{dy} \left[ \frac{1}{2} (y - t)^2 \right] \cdot x_j = (y - t)x_j
\]

For cost derivatives, use linearity and average over data points:

\[
\frac{\partial J}{\partial w_j} = \frac{1}{N} \sum_{i=1}^{N} (y^{(i)} - t^{(i)}) x_j^{(i)} \quad \frac{\partial J}{\partial b} = \frac{1}{N} \sum_{i=1}^{N} y^{(i)} - t^{(i)}
\]

Minimum must occur at a point where partial derivatives are zero.

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

(if \( \partial J / \partial w_j \neq 0 \), you could reduce the cost by changing \( w_j \))
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 $\nabla f(\mathbf{w})$, is:

$$
\left( \frac{\partial}{\partial w_1} f(\mathbf{w}), \ldots, \frac{\partial}{\partial w_D} f(\mathbf{w}) \right)^\top
$$

The gradient points in the direction of the greatest rate of increase.
Analogue of second derivative (the “Hessian” matrix):
$\nabla^2 f(\mathbf{w}) \in \mathbb{R}^{D \times D}$ is a matrix with $[\nabla^2 f(\mathbf{w})]_{ij} = \frac{\partial^2}{\partial w_i \partial w_j} f(\mathbf{w})$. 
Aside: The Hessian Matrix

- Analogue of second derivative (the Hessian): $\nabla^2 f(w) \in \mathbb{R}^{D \times D}$ is a matrix with $[\nabla^2 f(w)]_{ij} = \frac{\partial^2 f}{\partial w_i \partial w_j}(w)$.
  - Recall from multivariable calculus that for continuously differentiable $f$, $\frac{\partial^2 f}{\partial w_i \partial w_j} = \frac{\partial^2 f}{\partial w_j \partial w_i}$, so the Hessian is symmetric.

- The second derivative test in single variable calculus: a critical point is a local minimum if the second derivative is positive.

- The multivariate analogue involves the eigenvalues of the Hessian.
  - Recall from linear algebra that the eigenvalues of a symmetric matrix (and therefore the Hessian) are real-valued.
  - If all of the eigenvalues are positive, we say the Hessian is positive definite.
  - A critical point ($\nabla f(w) = 0$) of a continuously differentiable function $f$ is a local minimum if the Hessian is positive definite.
Aside: The Hessian Matrix

Visualization:\(^1\)

\[ x^2 + y^2 \]
(definite)

\[ x^2 \]
(semidefinite)

\[ x^2 - y^2 \]
(indefinite)

\(^1\)Image source: mkwiki.org
Direct Solution II: Calculus

- We seek $\mathbf{w}$ to minimize $J(\mathbf{w}) = \frac{1}{2} \| \mathbf{Xw} - \mathbf{t} \|^2$

- Taking the gradient with respect to $\mathbf{w}$ (see course notes for additional details) we get:

$$\nabla_{\mathbf{w}} J(\mathbf{w}) = \mathbf{X}^\top \mathbf{Xw} - \mathbf{X}^\top \mathbf{t} = 0$$

- We get the same optimal weights as before:

$$\mathbf{w}^* = (\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top \mathbf{t}$$

- 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 features to another space using feature mapping (or basis expansion).\( \psi(x) : \mathbb{R}^D \rightarrow \mathbb{R}^d \) and treat the mapped feature (in \( \mathbb{R}^d \)) as the input of a linear regression procedure.

- Let us see how it works when \( x \in \mathbb{R} \) and we use a polynomial feature mapping.
Polynomial Feature Mapping

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

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

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

Here the feature mapping is $\psi(x) = [1, x, x^2, \ldots, x^M]^\top$.

We can still use linear regression to find $\mathbf{w}$ since $y = \psi(x)^\top \mathbf{w}$ is linear in $w_0, w_1, \ldots$.

In general, $\psi$ can be any function. Another example: $\psi(x) = [1, \sin(2\pi x), \cos(2\pi x), \sin(4\pi x), \ldots]^\top$. 
Polynomial Feature Mapping with $M = 0$

\[ y = w_0 \]

- Pattern Recognition and Machine Learning, Christopher Bishop.
Polynomial Feature Mapping with $M = 1$

$y = w_0 + w_1 x$

- Pattern Recognition and Machine Learning, Christopher Bishop.
Polynomial Feature Mapping with $M = 3$

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

-Pattern Recognition and Machine Learning, Christopher Bishop.
Polynomial Feature Mapping with $M = 9$

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

- 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).
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.
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 / basis functions ($M$) 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
**L² (or ℓ²) Regularization**

- We can encourage the weights to be small by choosing as our regularizer the $L^2$ penalty.
  
  $$\mathcal{R}(w) = \frac{1}{2} \| w \|^2_2 = \frac{1}{2} \sum_j w_j^2.$$  

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

- If you fit training data poorly, $J$ is large. If your optimal weights have high values, $\mathcal{R}$ is large.

- Large $\lambda$ penalizes weight values more.

- Like $M$, $\lambda$ is a hyperparameter we can tune with a validation set.
$L^2$ (or $\ell_2$) Regularization

- The geometric picture:
For the least squares problem, we have $J(w) = \frac{1}{2N}\|Xw - t\|^2$.

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

$$w_{\lambda}^{\text{Ridge}} = \arg\min_w J_{\text{reg}}(w) = \arg\min_w \frac{1}{2N}\|Xw - t\|^2_2 + \frac{\lambda}{2}\|w\|^2_2$$

$$= (X^\top X + \lambdaNI)^{-1}X^\top t$$

- The case $\lambda = 0$ (no regularization) reduces to least squares solution!

- Note that it is also common to formulate this problem as

$$\arg\min_w \frac{1}{2}\|Xw - t\|^2_2 + \frac{\lambda}{2}\|w\|^2_2$$

in which case the solution is

$$w_{\lambda}^{\text{Ridge}} = (X^\top X + \lambda I)^{-1}X^\top t.$$
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 $\mathbf{w}$ 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 steepest descent.
Gradient Descent

- Observe:
  - if \( \frac{\partial J}{\partial w_j} > 0 \), then increasing \( w_j \) increases \( J \).
  - if \( \frac{\partial J}{\partial w_j} < 0 \), then increasing \( w_j \) decreases \( J \).

- The following update always decreases the cost function for small enough \( \alpha \) (unless \( \frac{\partial J}{\partial w_j} = 0 \)):

\[
w_j \leftarrow w_j - \alpha \frac{\partial J}{\partial w_j}
\]

- \( \alpha > 0 \) 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.
  - If cost is the sum of \( N \) individual losses rather than their average, smaller learning rate will be needed (\( \alpha' = \alpha/N \)).
Gradient Descent

- This gets its name from the gradient:

\[
\nabla_w J = \frac{\partial J}{\partial w} = \left( \frac{\partial J}{\partial w_1} \right) \ldots \left( \frac{\partial J}{\partial w_D} \right)
\]

- This is the direction of fastest increase in \( J \).

- Update rule in vector form:

\[
\mathbf{w} \leftarrow \mathbf{w} - \alpha \frac{\partial J}{\partial \mathbf{w}}
\]

And for linear regression we have:

\[
\mathbf{w} \leftarrow \mathbf{w} - \frac{\alpha}{N} \sum_{i=1}^{N} (y^{(i)} - t^{(i)}) \mathbf{x}^{(i)}
\]

- So gradient descent updates \( \mathbf{w} \) in the direction of fastest decrease.

- Observe that once it converges, we get a critical point, i.e. \( \frac{\partial J}{\partial \mathbf{w}} = 0 \).
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: $(\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top \mathbf{t}$
  - Matrix inversion is an $\mathcal{O}(D^3)$ algorithm
  - 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 $\mathcal{J}$:
  \[
  w \leftarrow w - \alpha \frac{\partial}{\partial w} \mathcal{J}
  \]

- The gradient descent update to minimize the $L^2$ regularized cost $\mathcal{J} + \lambda R$ results in weight decay:
  \[
  w \leftarrow w - \alpha \frac{\partial}{\partial w} (\mathcal{J} + \lambda R) = w - \alpha \left( \frac{\partial \mathcal{J}}{\partial w} + \lambda \frac{\partial R}{\partial w} \right)
  \]
  \[
  = w - \alpha \left( \frac{\partial \mathcal{J}}{\partial w} + \lambda w \right)
  \]
  \[
  = (1 - \alpha \lambda)w - \alpha \frac{\partial \mathcal{J}}{\partial w}
  \]
In gradient descent, the learning rate $\alpha$ is a hyperparameter we need to tune. Here are some things that can go wrong:

- $\alpha$ too small: slow progress
- $\alpha$ too large: oscillations
- $\alpha$ 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: 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 $J$ has been the average loss over the training examples:

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

($\theta$ denotes the parameters; e.g., in linear regression, $\theta = (w, b)$)

- By linearity,

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

- 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,

  1. Choose $i$ uniformly at random,
  2. $\theta \leftarrow \theta - \alpha \frac{\partial L^{(i)}}{\partial \theta}$

- 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:

  $$\mathbb{E} \left[ \frac{\partial L^{(i)}}{\partial \theta} \right] = \frac{1}{N} \sum_{i=1}^{N} \frac{\partial L^{(i)}}{\partial \theta} = \frac{\partial J}{\partial \theta}.$$
Stochastic Gradient Descent

- Problems with using 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 examples \( \mathcal{M} \subset \{1, \ldots, N\} \), called a mini-batch.
- Stochastic gradients computed on larger mini-batches have smaller variance.
- 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
  - A 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.
In stochastic training, the learning rate also influences the fluctuations due to the stochasticity of the gradients.

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.
Convex Sets

- A set $S$ is **convex** if any line segment connecting points in $S$ lies entirely within $S$. Mathematically,

  $$x_1, x_2 \in S \implies \lambda x_1 + (1 - \lambda)x_2 \in S \text{ for } 0 \leq \lambda \leq 1.$$

- A simple inductive argument shows that for $x_1, \ldots, x_N \in S$, weighted averages, or **convex combinations**, lie within the set:

  $$\lambda_1 x_1 + \cdots + \lambda_N x_N \in S \text{ for } \lambda_i > 0, \lambda_1 + \cdots + \lambda_N = 1.$$
Convex Functions

- A function $f$ is convex if for any $x_0, x_1$ in the domain of $f$,
  
  \[
  f((1 - \lambda)x_0 + \lambda x_1) \leq (1 - \lambda)f(x_0) + \lambda f(x_1)
  \]

- Equivalently, the set of points lying above the graph of $f$ is convex.

- Intuitively: the function is bowl-shaped.
We just saw that the least-squares loss 
\[ \frac{1}{2} (y - t)^2 \] is convex as a function of \( y \).

For a linear model, 
\[ z = w^\top x + b \] is a linear function of \( w \) and \( b \). If the loss function is convex as a function of \( z \), then it is convex as a function of \( w \) and \( b \).
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*
Linear Regression with $L^p$ Regularization

Which sets are convex?

Solution of linear regression with $L^p$ regularization:

- $p = 2$: Has a closed form solution.
- $p \geq 1, p \neq 2$:
  - The objective is convex.
  - The true solution can be found using gradient descent.
- $p < 1$:
  - The objective is non-convex.
  - Can only find approximate solution (e.g. the best in its neighborhood) using gradient descent.
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.

Next lecture: apply this framework to classification