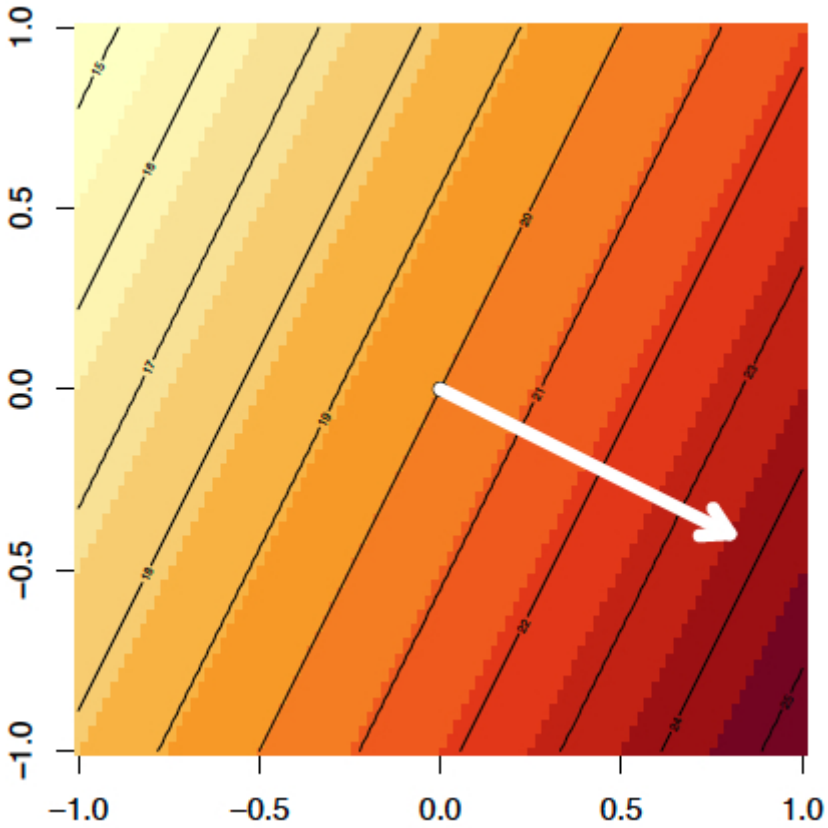
1. Steepest ascent

- Design space, design radius, center points
  Design space: the region inside the factorial points
  center points: origin (0, 0) in coded units. Model's prediction is best at the center point.
  design radius: measure how far away from the center point.
- First-order response surfaces
  Approximate the response surface with a plane.



- Finding direction of steepest ascent
  Compute partial derivatives along each factor. The rate of ascent along each direction is the effect size β_i.
- Standardized step sizes

Consider the general first-order model

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_n x_n$$

1. Find the effect size with the largest **magnitude**. We'll call this β_j and the associated factor x_j.
2. Choose a step size (in coded units) along this dimension, called Δx_j.
3. For all other dimensions $i \neq j$, the step size is

$$\Delta x_i = \frac{\beta_i}{\beta_j} \Delta x_j$$

Example: $y = 20 + 3.6x_1 - 1.8x_2$.

1. $|3.6| > |-1.8|$, so we standardize using x_1 ($j \equiv 1$).
2. Let $\Delta x_1 = 1$.
3. $\Delta x_2 = \frac{\beta_2}{\beta_1} \Delta x_1 = \frac{-1.8}{3.6}(1) = -0.5$

Why standardize: 1) uniform steps give uniform differences in design radii; 2) a standardized step of 1 always defines a point on the design space boundary.

- Testing for pure error and lack of fit

  Pure error: standard deviation of the repeated runs at the design center.

  Lack of fit:

1. $\bar{y}_{\text{center}}$ = mean response of the n_{center} center points
   $\bar{y}_{\text{fact}}$ = mean response of the n_{fact} factorial points
2. $$SS_{\text{curve}} = \frac{n_{\text{fact}} n_{\text{center}} (\bar{y}_{\text{fact}} - \bar{y}_{\text{center}})^2}{n_{\text{fact}} + n_{\text{center}}}, \quad \text{DF}(SS_{\text{curve}}) = 1$$
3. $$SS_{\text{error}} = \sum_{\substack{\text{center} \\ \text{points}}} (y_i - \bar{y}_{\text{center}})^2, \quad \text{DF}(SS_{\text{error}}) = n_{\text{center}} - 1$$
4. $$F_{\text{curve}} = \frac{SS_{\text{curve}}/\text{DF}(SS_{\text{curve}})}{SS_{\text{error}}/\text{DF}(SS_{\text{error}})}$$

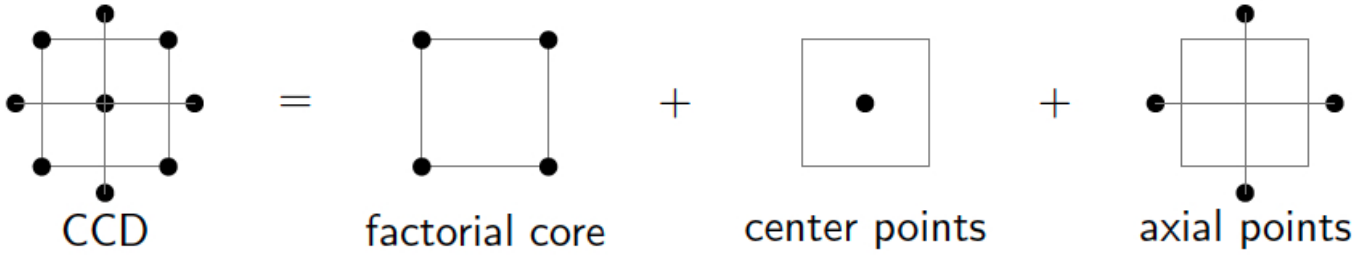
2. RSM

- Quadratic response models

$$y = \beta_0 + \sum_{i=1}^{k} \beta_i x_i + \sum_{i=1}^{k} \beta_{ii} x_i^2 + \sum_{j=1}^{k} \sum_{i=1}^{j-1} \beta_{ij} x_i x_j.$$

This model has $1 + 2k + k(k-1)/2$ parameters, so RSM designs must have at least this many runs.

- Central composite designs (CCDs)



  - Factorial points alone estimates the FO and TWO terms. It core must be Resolution V or higher.
  - Axial points allow estimation of the PQ terms. Without axial points we could only estimate the sum of all PQ terms.
  - CCDs have a pair of axial runs for each factor:
    - One factor is set to $\pm\alpha$
    - All other factors are set to 0.
  - Center points estimate pure error and help (some) with PQ terms.
  - To build a CCD you need to decide:
    1. The size of the FF core
    2. The number of center runs
    3. The value of α

- Uniform precision

  The variance at design radius 1 is the same as the center. Choose correct number of the center points to ensure uniform precision.

- Rotatable designs

  Designs where the variance only depends on the radius.

  The change in precision should be independent of the direction moving away from the center.

  CCD with F factorial points is rotatable when $\alpha = F^{1/4}$

Rotatable, uniform precision CCDs

factors (k)	2	3	4	5	$5-1$	6
factorial points	4	8	16	32	16	64
axial points	4	6	8	10	10	12
center points	5	6	7	10	6	15
axial distance (α)	1.414	1.682	2.000	2.378	2.000	2.828

factors (k)	$6-1$	7	$7-1$	8	$8-1$	$8-2$
factorial points	32	128	64	256	128	64
axial points	12	14	14	16	16	16
center points	9	21	14	28	20	13
axial distance (α)	2.378	3.364	2.828	4.000	3.364	2.828

- Calculating factor levels

$$F = 2^3 = 8 \Rightarrow \alpha = \sqrt[4]{8} = 1.68$$

$$\mathrm{A} = \mathrm{center(A)} + \frac{\mathrm{range(A)}}{2\alpha}\mathrm{[code]}$$

- Finding stationary points

  The argmin, argmax, or inflection point of a saddle.

$$\begin{aligned}\frac{\partial y}{\partial \mathbf{x}} &= \frac{\partial}{\partial \mathbf{x}}\left(b_0 + \mathbf{x}^\mathsf{T}\mathbf{b} + \mathbf{x}^\mathsf{T}\mathbf{B}\mathbf{x}\right) \\ &= \mathbf{b} + 2\mathbf{B}\mathbf{x}\end{aligned}$$

Solving for where the derivative equals zero:

$$\mathbf{b} + 2\mathbf{B}\mathbf{x}_s = \mathbf{0} \Rightarrow \mathbf{x}_s = -\frac{1}{2}\mathbf{B}^{-1}\mathbf{b}$$

$$b_0 = \beta_0, \quad \mathbf{b} = \begin{pmatrix} \beta_1 \\ \vdots \\ \beta_k \end{pmatrix}, \quad \mathbf{B} = \begin{pmatrix} \beta_{11} & \beta_{12}/2 & \cdots & \beta_{1k}/2 \\ & \beta_{22} & \cdots & \beta_{2k}/2 \\ & & \ddots & \vdots \\ \text{sym} & & & \beta kk \end{pmatrix}$$

- Responses at the stationary point

$$\begin{aligned} y_s &= b_0 + \mathbf{x}_s{}^{\mathsf{T}}\mathbf{b} + \mathbf{x}_s{}^{\mathsf{T}}\mathbf{B}\mathbf{x}_s \\ &= b_0 + \mathbf{x}_s{}^{\mathsf{T}}\mathbf{b} + \left(-\frac{1}{2}\mathbf{b}^{\mathsf{T}}\mathbf{B}^{-1}\right)\mathbf{B}\mathbf{x}_s \\ &= b_0 + \frac{1}{2}\mathbf{x}_s{}^{\mathsf{T}}\mathbf{b} \end{aligned}$$

  The response at the stationary point only depends on the intercept and the main effects.

- Testing for maxima/minima/saddle points

  The type of extremum is determined by the eigenvalues (λ_i) of the matrix $\mathbf{B}$.

  - All $\lambda_i < 0 \rightarrow$ maximum
  - All $\lambda_i > 0 \rightarrow$ minimum
  - Indeterminate signs $\rightarrow$ saddle point

- Alternative designs: BBD, Hoke, Koshal, Roquemore Hybrid, SCD, DSD

  1. The **CCD** is the best overall choice for RSM.
  2. A **BBD** is a close second, but only preferable to a CCD when 3-level factors are more convenient than 5-level factors.
  3. **Hoke** or **Hybrid** designs are the preferred designs when your run budget is too small for a CCD or BBD.
  4. The **SCD** should only be used when a tight budget demands immediate follow-up from steepest ascent. In this case, you need to use a Resolution III* screening design for steepest ascent.
  5. **Koshal** designs are obsolete; we include them only for a historical perspective.

Box-Behnken design:

- 3-level design with performance close to a CCD.
- Similar number of runs to a CCD.
- Built from 2^2 factorials for each pair of factors.
- Nearly rotatable (rotatable for $k = 4$ or 7).
- 3–5 center runs are recommended. (At least one center run is required for $k = 4$ or 7).

x_1	x_2	x_3
-1	-1	0
-1	1	0
1	-1	0
1	1	0
-1	0	-1
-1	0	1
1	0	-1
1	0	1
0	-1	-1
0	-1	1
0	1	-1
0	1	1
$\mathbf{0}$	$\mathbf{0}$	$\mathbf{0}$

Note that in the bottom row $\mathbf{0}$ is a vector, i.e. a set of repeated center points.

All points are on the edges and $\sqrt{2}$ away from the design center when k=3.

It's not good at predicting response near the corners.

Hoke Design

D2 (10 runs) and D6 (13 runs) are most popular designs.

$\mathbf{D_2} =$

x_1	x_2	x_3
-1	-1	-1
1	1	-1
1	-1	1
-1	1	1
1	-1	-1
-1	1	-1
-1	-1	1
-1	0	0
0	-1	0
0	0	-1

$\mathbf{D_6} =$

x_1	x_2	x_3
-1	-1	-1
1	1	-1
1	-1	1
-1	1	1
1	-1	-1
-1	1	-1
-1	-1	1
-1	0	0
0	-1	0
0	0	-1
1	1	0
1	0	1
0	1	1

Requemore hybrid designs:

near-saturated and near-rotatable

$\mathbf{D_{310}}$ (saturated)

x_1	x_2	x_3
0	0	1.2906
0	0	-0.1360
-1	-1	0.6386
1	-1	0.6386
-1	1	0.6386
1	1	0.6386
1.736	0	-0.9273
-1.736	0	-0.9273
0	1.736	-0.9273
0	-1.736	-0.9273

$\mathbf{D_{311A}}$ (near-saturated)

x_1	x_2	x_3
0	0	$\sqrt{2}$
0	0	$-\sqrt{2}$
-1	-1	$1/\sqrt{2}$
1	-1	$1/\sqrt{2}$
-1	1	$1/\sqrt{2}$
1	1	$1/\sqrt{2}$
$\sqrt{2}$	0	$-1/\sqrt{2}$
$-\sqrt{2}$	0	$-1/\sqrt{2}$
0	$\sqrt{2}$	$-1/\sqrt{2}$
0	$\sqrt{2}$	$-1/\sqrt{2}$
$\mathbf{0}$	$\mathbf{0}$	$\mathbf{0}$

Small composite design (SCD)

Resolution III^* design: resolutoin III with no 4-letter word in the defining relation.

High variance for main effects and TWI terms.

x_1	x_2	x_3
-1	-1	-1
1	1	-1
1	-1	1
-1	1	1
$-\alpha$	0	0
α	0	0
0	$-\alpha$	0
0	α	0
0	0	$-\alpha$
0	0	α
$\mathbf{0}$	$\mathbf{0}$	$\mathbf{0}$

Definitive screening designs (DSD)

- The DSD combines features of FF screening, foldover, PB, and BB designs.
- For k factors a DSD requires only $2k + 1$ runs.
- Continuous factors use 3 levels; some discrete 2-level factors can be added.
- Can estimate FO, TWI, and PQ terms (up to saturation).
  - Main effects are clear of TWI and PQ terms.
  - All PQ terms are estimable.
  - Complex aliasing of TWI and PQ terms.

6-factor, minimum run DSD

A	B	C	D	E	F
0	1	1	1	1	1
0	−1	−1	−1	−1	−1
1	0	−1	1	1	−1
−1	0	1	−1	−1	1
1	−1	0	−1	1	1
−1	1	0	1	−1	−1
1	1	−1	0	−1	1
−1	−1	1	0	1	−1
1	1	1	−1	0	−1
−1	−1	−1	1	0	1
1	−1	1	1	−1	0
−1	1	−1	−1	1	0
0	0	0	0	0	0

3. Mixtures

- Slack variable and Scheffe models

  Mixture: k components, each with proportion x_i, where $0 \leq x_i \leq 1$ and $\sum_{x_i} = 1$

  Let's start with the standard FO model

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_3 + \epsilon$$

  By the summation constraint, $x_3 = 1 - x_1 - x_2$. Substituting, we see that

$$\begin{aligned} y &= \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_3 (1 - x_1 - x_2) + \epsilon \\ &= (\beta_0 + \beta_3) + (\beta_1 - \beta_3) x_1 + (\beta_2 - \beta_3) x_2 + \epsilon \\ &= \beta_0' + \beta_1' x_1 + \beta_2' x_2 + \epsilon \end{aligned}$$

  The effect of factor x_3 is folded into the intercept and the other effects. When using a slack variable model we choose x_3 to be the least important factor, but the effects remain confounded.

  The Scheffé model takes a different approach. Rather than substitute for x_3, we substitute for the intercept (the 1 multiplied by β_0) using the constraint $x_1 + x_2 + x_3 = 1$:

$$\begin{aligned} y &= \beta_0 (1) + \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_3 + \epsilon \\ &= \beta_0 (x_1 + x_2 + x_3) + \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_3 + \epsilon \\ &= (\beta_0 + \beta_1) x_1 + (\beta_0 + \beta_2) x_2 + (\beta_0 + \beta_3) x_3 + \epsilon \\ &= \beta_1^* x_1 + \beta_2^* x_2 + \beta_3^* x_3 + \epsilon \end{aligned}$$

  The effects in a Scheffé model are clean, but they have a special interpretation. The coefficient β_i^* is the expected response for a pure mixture with $x_i = 1.0$ and all other ingredients set to zero.

- SLD vs. SCD

The Simplex-Lattice Design (SLD)

A Simplex-Lattice Design SLD$\{k, m\}$ studies mixtures with k ingredients set at $m + 1$ equally-spaced levels. The design uses all combinations of the ingredient levels.

SLD$\{3, 1\}$: Levels 0, 1

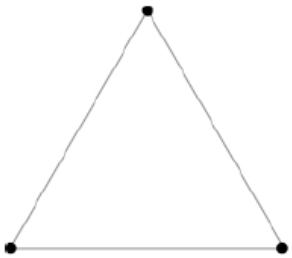
(1,0,0) (0,1,0) (0,0,1)

SLD$\{3, 2\}$: Levels 0, $\frac{1}{2}$, 1

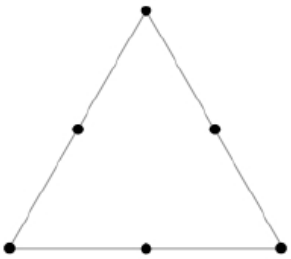
$(\frac{1}{2},\frac{1}{2},0)$ $(\frac{1}{2},0,\frac{1}{2})$ $(0,\frac{1}{2},\frac{1}{2})$ + SLD$\{3, 1\}$

SLD$\{3, 3\}$: Levels 0, $\frac{1}{3}$, $\frac{2}{3}$, 1

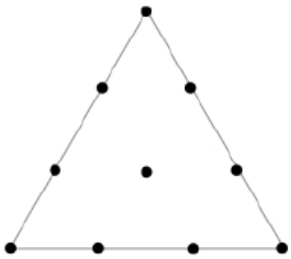
$(\frac{1}{3},\frac{2}{3},0)$ $(\frac{2}{3},\frac{1}{3},0)$ $(\frac{1}{3},0,\frac{2}{3})$ $(\frac{2}{3},0,\frac{1}{3})$ $(0,\frac{1}{3},\frac{2}{3})$ $(0,\frac{2}{3},\frac{1}{3})$ $(\frac{1}{3},\frac{1}{3},\frac{1}{3})$ + SLD$\{3, 1\}$



SLD{3,1}



SLD{3,2}

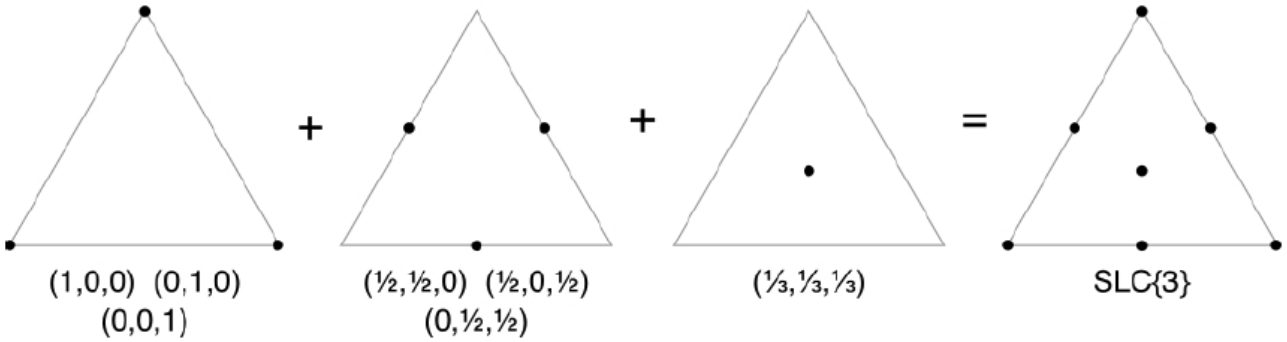


SLD{3,3}

The Simplex-Centroid Design (SCD)

A downside of the SLD$\{k, m\}$ design is there are no interior points until $m \geq 3$. An alternative is the SCD$\{k\}$ design which includes

- All k pure mixtures: (1,0,. . .,0)
- All binary combinations at $\frac{1}{2}$: $(\frac{1}{2},\frac{1}{2},0,\ldots,0)$
- All trinary combinations at $\frac{1}{3}$: $(\frac{1}{3},\frac{1}{3},\frac{1}{3},0,\ldots,0)$
- ⋮
- The single k-nary mixture: $(\frac{1}{k},\frac{1}{k},\ldots,\frac{1}{k})$



SLD allows fitting an m^{th} order model. SCD fit k-order model with up to k-way interaction term.
In SCD, no single ingredient is run at a proportion > 1/2.
SLD has better coverage of the boundary, while SCD has better coverage of the interior.

4. Crossover Designs

- Motivation for crossover designs
  When individual experimental units are rare or expensive. The number of treatment levels and subjects is small.
- Washout, carryover
  Carryover effect: Treatments are sequential, so the effects can persist into the next treatment.
  2 way to deal with: 1) allow a washout period between treatments (easy but costly); 2) include carryover effects in the model (difficult)
- Blocking in crossover designs

How do we model CODs?

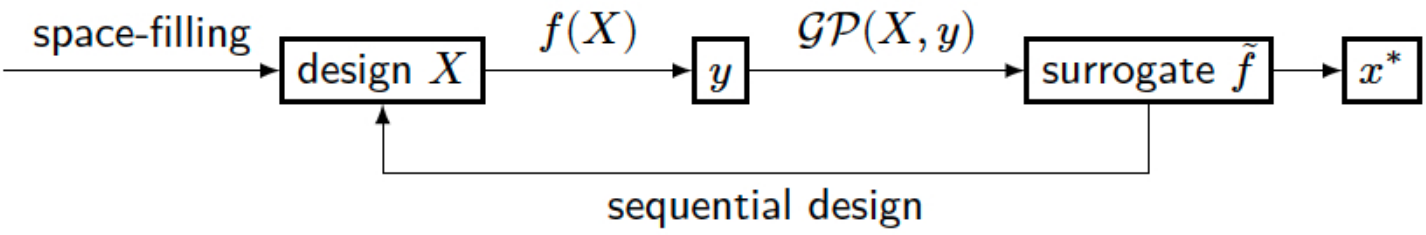
The response of every subject in each period involves multiple factors:

$$\underbrace{y}_{\text{response}} = \underbrace{\mu}_{\text{mean}} + \underbrace{\tau}_{\text{treatment}} + \underbrace{\pi}_{\text{period}} + \underbrace{s}_{\text{subject}}$$

- The mean effect (μ) is the overall response of all subjects to either treatment in either period.
- The treatment effects (τ) are what we're trying to measure.
- The period effects (π) block for changes between the first and second periods, assuming all subjects were run simultaneously.
- The subject effects (s) block for differences between each experimental unit.

Normally we can't include blocking factors for every experimental unit (since there is usually one unit per run). Since CODs make multiple independent measurements on the same unit (period 1 vs. 2), we have enough data to fit a parameter for each individual.

5. Surrogate Optimization



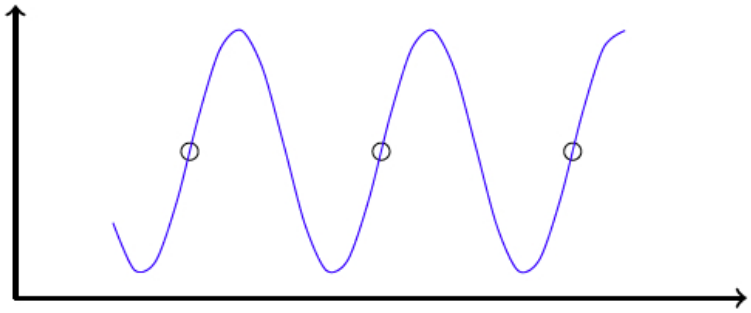
To maximize the response y of an unknown function f using no more than N function evaluations:

1. Create a space-filling design X_n for $n < N$.
2. Measure the responses $y_n(X_n)$ and train $\mathcal{GP}(X_n, y_n)$.
3. Use a nonlinear optimizer (`optim`) to find the argmax x of a metric (mean, SD, EI).
4. Measure $y(x)$ and update $\mathcal{GP}(X_{n+1}, y_{n+1})$.
5. Go to #3 and repeat until all N runs are used.
6. Search $\mathcal{GP}(X_N, y_N)$ for the global maximum $y^*(x^*)$.

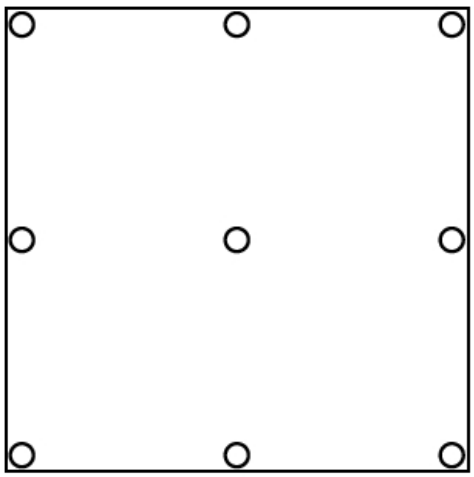
- Global vs. local optimization
  Steepest ascent/RSM finds the optimal operating conditions in a local design space. Global optimization searches the entire design region.
- Latin Hypercube Designs
  Evenly-spaced design:

  Evenly-spaced designs have two big drawbacks:

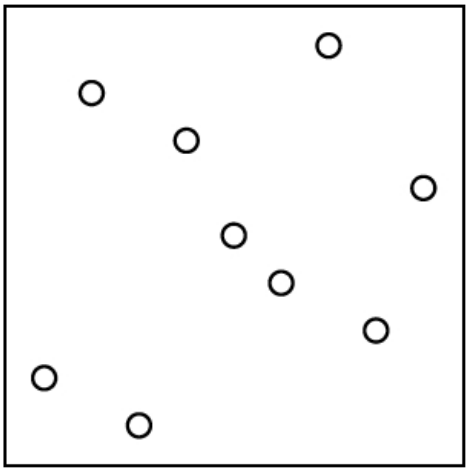
  1. Regular spacing can alias patterns in the response surface.



  2. Regular designs have poor **projection spacing**. This is a problem because of effect sparsity!

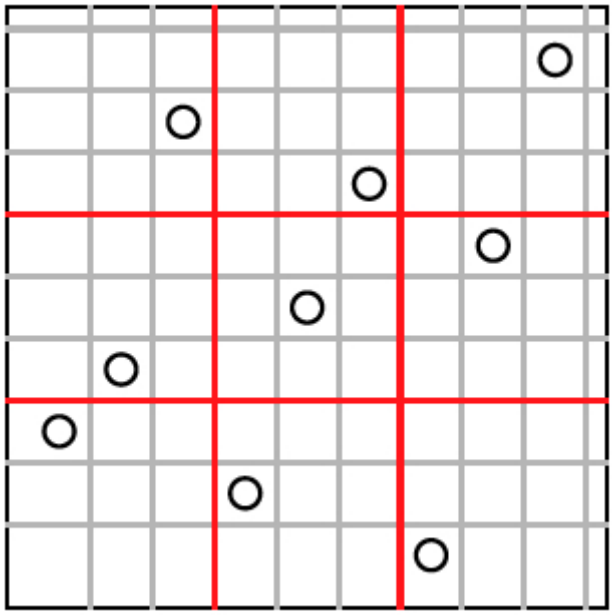


  Random designs are clumpy.
  Latin hypercube design:



A Latin Hypercube Design (LHD) is a semi-random design that guarantees uniform projection.

- Each dimension is divided into n intervals.
- Points are placed randomly, but only one point is allowed in each interval along each dimension.
- Points can be placed in the center or a random position in each "square".



  - placed points semi-randomly to avoid aliasing
  - avoid clumps of points
  - project well onto lower dimensions

    Orthogonal array LHD:

- Maximin Designs

  Maximize the distance between points.

The *Euclidean distance* between any two points x and x' is

$$d(x, x') = \|x - x'\|^2 = \sum_{j=1}^{k} (x_j - x'_j)^2$$

The *maximin* design matrix with n samples, called X_n is

$$\arg\max_{X_n} \min\{d(x, x'), \forall x \neq x'\}$$

- Gaussian Process Regression

  GPR assumes the covariance between the data have a particular shape. The covariance function is called the kernel.

  - Let's start with a space-filling design X_n and assume we measured the responses y_n at each point in the design.
  - Using our kernel function, we can calculate the covariance among the points in the design set
    $$\Sigma_n = \Sigma(X_n, X_n)$$
  - We want to find the response y at a new, unmeasured point x. Using some identities for multivariate normal distributions,
    $$y(x) = \Sigma(x, X_n)\Sigma_n^{-1} y_n.$$
  - GPR assumes that $y(x)$ is itself normally distributed with variance
    $$\sigma^2(x) = \Sigma(x, x) - \Sigma(x, X_n)\Sigma_n^{-1}\Sigma(x, X_n)^\top.$$

  GRP limitations:

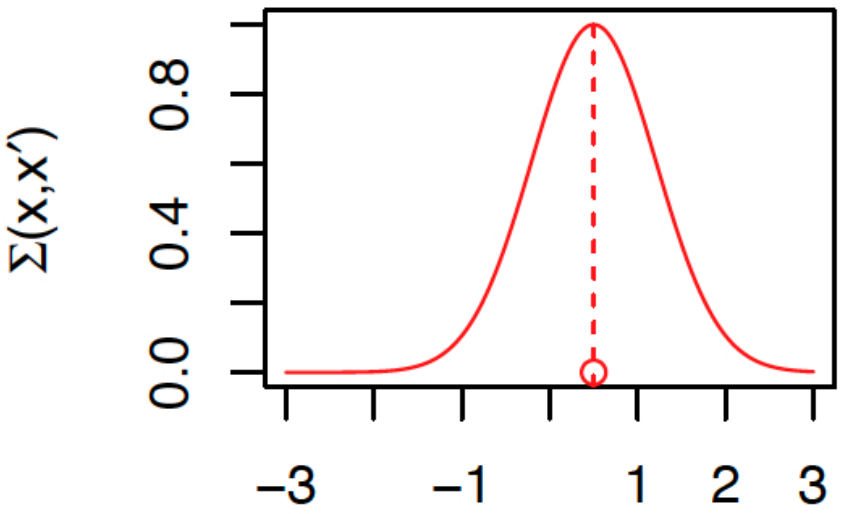
  - **Data intensive.** Since GPR does not use a parametric model, the entire shape of the response surface must come from data. GPR generally requires more data than a linear model.
  - **Computationally intensive.** Training a GPR requires inverting Σ_n, an $n \times n$ dense matrix. Practically, this limits GPR to 1,000's or a few 10,000's of points.
  - **Interpolation only.** GPR has no idea what the response should look like beyond the training data. GPR requires a space-filling design that covers the entire search region.

GPR is bayesian: the kernel (prior) is updated with data (X_n, y_n) to compute the posterior estimates of (x, y). But it's nonparametric since the equation for $y(x)$ and $\sigma^2(x)$ don't contain parameters.

- Kernels
  - We will use the *inverse exponentiated squared Euclidean distance* kernel:

$$\Sigma(x, x') = \exp\{-\|x - x'\|^2\}.$$

  - Note that $\Sigma(x, x) = 1$ and $\Sigma(x, x') < 1$ if $x \neq x'$.



- Hyperparameters: scale, nugget, lengthscale

Scale

  - For our sinusoidal example, the response was in $[-1, 1]$, so we never noticed a problem. But not all problems have nice scaling.
  - Previously, the covariance matrix Σ was defined based on a correlation function

$$C(x, x') = \exp\{-\|x - x'\|^2\}.$$

  - Let's scale the correlation function by a hyperparameter τ^2:

$$\Sigma = \tau^2 C(x, x') = \tau^2 \exp\{-\|x - x'\|^2\}$$

  - Where do we get τ^2? From the data! The maximum likelihood estimate is

$$\hat{\tau}^2 = \frac{y_n^\top C_n^{-1} y_n}{n}, \quad C_n \equiv C(X_n, X_n)$$

Nugget

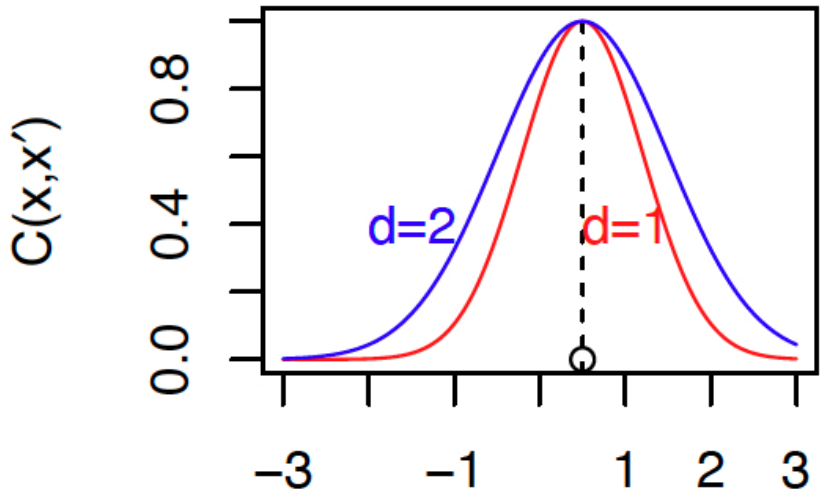
- With real experiments, our measurements of y will be noisy, and we want GPR to smooth over this noise. Also, what if we made repeated measurements at y? What will the prediction be?
- **Solution:** Break the perfect correlation in C by injecting a small amount of white noise.
- **Method:** Add a "nugget" g to the diagonal of Σ_n:

$$\Sigma_n = \tau^2[C(X_n, X_n) + g\mathbb{I}].$$

Lengthscale

- GPR requires the correlation function $C(x, x')$ to quantify how quickly the relationship between points decays.
- The lengthscale of decay can also be tuned by adding a hyperparameter d:

$$C(x, x') = \exp\left\{-\frac{\|x - x'\|^2}{d}\right\}.$$



Dimensions with longer lengthscales require fewer data for prediction.

Putting all 3 factors together. Need to compute inverse of C_n at each iteration. computational expensive.

- Sequential design using mean, variance, and expected improvement
  Use the point where mean/variance/EI is biggest as next input data.
  Sequential design methods are last sample optimal. But sequential design is greedy.If $N-2$ of N runs are finished, two rounds of sequential design may not be optimal. It suffers limited lookahead.
  Expected improvement:

As usual, assume we've measured n responses y_n at locations X_n. Define

$$y_{\max} = \max\{y_1, \ldots, y_n\}.$$

The *improvement* in the objective at a new input x is

$$I(x) = \max\{0, y(x) - y_{\max}\}$$

where the maximization "floors" the improvement at zero.

The *expected improvement* $\mathrm{EI}(x) = \mathbb{E}\{I(x)\}$ quantifies how much we expect the best objective value to increase after measuring at point x.

We can sample $y(x)$ many times, averaging the improvement $I(x)$ for T samples:

$$\mathrm{EI}(x) \approx \frac{1}{T} \sum_{i=1}^{T} \max\{0, y_i(x) - y_{\max}\}.$$

Even better, we can leverage that GPR predictions are multivariate normal with mean $\mu(x)$ and variance $\sigma(x)$. Let $z = (\mu(x) - y_{\max})/\sigma(x)$. Then

$$\mathrm{EI}(x) = (\mu(x) - y_{\max})\mathrm{CDF}(z) + \sigma(x)PDF(z)$$

using the PDF and CDF of a standard Gaussian distribution.

- Exploration vs. exploitation
  Exploration searches areas of high uncertainty to find new regions of interest.
  Exploitation refines existing optima by adding points to known regions of interest.
  Explore early, exploit later.
  Alternate between batches of exploration and exploitation.
  Exploit by maximizing the predicted GPR mean.
  Explore by maximizing the predicted GPR standard deviation

6. Reinforcement Learning

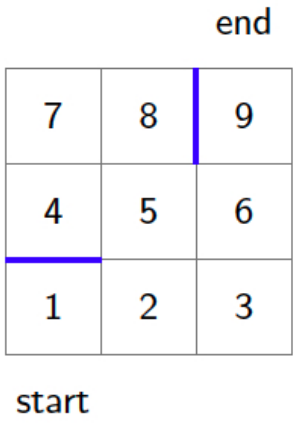
Learning from trial and error. Many RL algorithms rely on random processes to generate data.

- Markov Decision Processes: state, action, policy, reward, trajectory, episode
  Describes how agent interacts with its environment.
  State: the agent and environment at any time
  Action: the agent selects an action to move between states.
  Reward: every action and state produce a reward.
  The agent goal is to maximize the total reward.
  MDP has Markov property: 1) all decisions depend only on the current state. 2) each state includes all of the relevant history.
  Policy: a function that maps states to actions. The value $\pi(s, a)$ is the probability that the agent will select action a in state s.
  MDP can be deterministic or stochastic. Deterministic: actions always determine the next state. Stochastic: action change the probability that any other state will be the next state.
  Trajectory: a single pass through a finite horizon MDP. Finite horizon MDP or episodic MDP stops after a finite number of actions.
- Value functions
  The value of a state is the expected reward from that state to the end of the trajectory. $V(s_i) = \mathbb{E}(\sum r_k) = \mathbb{E}(R_i)$.
  A trajectory is a sequence of states, actions and rewards. Its length can vary for every trajectory. No action in terminal state, but there can be a terminal reward.

From trajectories to value functions

Let's calculate $V(s)$ for a 3×3 Gridworld board.

The MDP is deterministic, so knowing s_i and s_{i+1} tells us a_i. Also, $r_i = -1$ for all $0 \leq i < T$.



Four trajectories beginning at s_1:

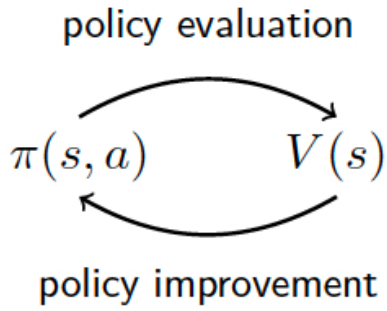
$$\begin{aligned}
\tau_1: &\quad 1,2,5,4,5,6,3,6,9 && R_{\tau_1} = -8 \\
\tau_2: &\quad 1,2,3,6,3,2,5,8,7,8,5,6,9 && R_{\tau_2} = -12 \\
\tau_3: &\quad 1,2,5,2,3,6,9 && R_{\tau_3} = -6 \\
\tau_4: &\quad 1,2,5,4,5,2,3,6,5,8,5,6,3,2,5,6,9 && R_{\tau_4} = -16
\end{aligned}$$

$$V(s_1) \approx \frac{R_{\tau_1} + R_{\tau_2} + R_{\tau_3} + R_{\tau_4}}{4} = \frac{(-8) + (-12) + (-6) + (-16)}{4} = -10.5$$

Every-visit vs last-visit:
Gridworld is deterministic and the agent should never visit the same state twice. The last-visit estimate is closest to optimal.
Stochastic problems can revisit the same state under the optimal policy.
Acting greedy w.r.t a value function is optimal. $V(s_i) = \mathbb{E}(r_i + .. + r_T) \rightarrow maxV(s_i) = max_{a_i}\mathbb{E}(r_i) + V(s_{i+1})$. So the optimal policy at state s_i satisfies $max_{a_i} V(s_{i+1})$
Pure Monte Carlo is inefficient.

One solution is **generalized policy iteration**.

1. Use a random policy π to generate trajectories and estimate $V(s)$.
2. Adjust π to be *greedy* for $V(s)$.
3. Repeat (1–2) until π stops changing.



policy iteration is guaranteed to find optimal policy provided every state is visited an infinite number of times. But tabular methods require visiting every state.

- **Policy evaluation** computes $V(s)$ for a given policy.
- **Policy improvement** makes greedy improvements to a policy.
- If you don't have a good starting policy, behave randomly.
- Tabular methods track $V(s)$ for every state in the MDP. They require visiting every state many times and storing the results.

- Rollout

  V is rarely known.

  - Monte Carlo estimation of V is only approximate even with many simulations.
  - If the state space is very large, we may never visit every state to estimate V by tabular methods.

  Ways to approximate value functions:

  1. **Parametric approximation** trains a model that predicts value from previous visits to states. The model predicts the value for *all* states, even those that have not been visited.
     - Any type of model can be used to predict value: Linear models, Gaussian Process Regression, Artificial Neural Networks ("Deep RL").
     - Parametric methods are often *offline*; the model is trained before the agent uses the model to navigate an MDP.
  2. **Monte Carlo approximation** uses a model of the MDP to simulate the rewards following a state.
     - Monte Carlo methods work well *online* by simulating states just ahead of the agent in the MDP.
     - These methods are *sample efficient* but require a computational model of the MDP.

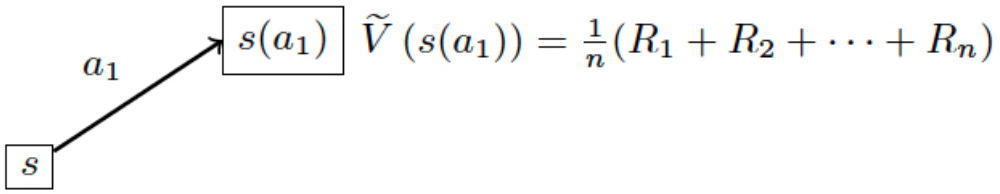
  Rollout is a Monte Carlo method, which looks ahead to estimate the value of states the agent is likely to visit next.

Rollout requires

- A *simulator* that generates sequences

$$s_i, a_i, r_i, \ldots, s_{T-1}, a_{T-1}, r_{T-1}, s_T, r_T$$

  given a policy π.
- A *base policy* π_{base} to use with the simulator. A random base policy works.



policy improvement with rollout:

- The online policy we are finding is called the *rollout policy*.
- The rollout policy has the *policy improvement property* — it will be equal to or better than the base policy.
- We can repeat the process with another trip through the MDP using the rollout policy as the new base policy.
- However, iterating in this way requires us to add exploration to our policies.

Rollout is an online method that reduces simulation by focusing on local starts. Iteration and exploration are required to find optimal policies.

- Discount factors

Let's extend our RL theory to incorporate *discounting* — reducing the present value of rewards from the future.

Discounting applies a horizon to the problem. The agent cares less and less about future states.

$$\begin{aligned} \text{Undiscounted:} \quad & \max_a \mathbb{E}\left\{r_i + V(s_{i+1})\right\} \\ \text{Discounted:} \quad & \max_a \mathbb{E}\left\{r_i + \gamma V(s_{i+1})\right\} \end{aligned}$$

The *discount factor* $\gamma \in [0, 1]$ determines the length of the horizon.

- $\gamma = 0$ makes the algorithms greedy; only the immediate reward r_i influences the agent.
- $\gamma = 1$ equally weights all rewards to the end of the trajectory.

Penalized stepping with $r_i = -1$, $r_T = 0$:

$$\begin{aligned} \text{reward} &= r_0 + r_1 + \cdots + r_{T-1} + r_T \\ &= \sum_{i=0}^{T-1} r_i + 0 \\ &= -T \end{aligned}$$

Discounting with $r_i = 0$, $r_T = 1$, $\gamma < 1$:

$$\begin{aligned} \text{reward} &= r_0 + \gamma(r_1 + \gamma(r_2 + \gamma(\cdots\gamma(r_{T-1} + \gamma(r_T))))) \\ &= r_0 + \gamma r_1 + \gamma^2 r_2 + \cdots \gamma^{T-1} r_{T-1} + \gamma^T r_T \\ &= \gamma^T \end{aligned}$$

In both cases, the maximum reward is achieved by minimizing the number of steps T.

When to use discounting?

- don't want to discount the future rewards, set $\gamma = 1$
- compare with greedy algorithm, set $\gamma = 0$

- want the agent to terminate the process quickly or solve non-episodic problems, set $\gamma < 1$
  Model-free learning: directly learn from experience, no need a model to simulate ahead when estimating value functions. Their only method of sampling is to interact with the environment, maximizing the information extracted from every trajectory.

- TD learning for value functions
  Ideally, update estimate of value function from every trajectory, but a single trajectory is a noisy estimate of value.
  Temporal difference (TD) learning balances new experience with previous results when updating V.

  1. Initialize our value estimates $V(s)$ for all states s.
  2. Experience a new trajectory $s_0, a_0, r_0, s_1, a_1, r_1 \ldots, s_T, r_T$.
  3. For each state s_i in the trajectory, calculate the *TD target*
     $$\hat{V}(s_i) = r_i + \gamma V(s_{i+1})$$
     using the experienced reward r_i and the previous estimate for $V(s_{i+1})$.
  4. Incrementally update the value of state s_i using a learning rate α:
     $$V(s_i) = V(s_i) + \alpha \left[\hat{V}(s_i) - V(s_i)\right].$$
  5. Go to step #2 and repeat.

  TD-learning is a *bootstrap* method since $V(s)$ is updated using $V(s_i)$ and $V(s_{i+1})$ from the previous iteration. New information only enters through r_i when estimating the TD target $\hat{V}(s_i)$.

- Q-factors
  Q-factors: Learn the value of each state/action pair, because we don't know s_{i+1} given s_i and a.

Using Q-factors, the policy problem at state s

$$\max_a \mathbb{E}\left\{r_i + \gamma V(s_{i+1})\right\}$$

becomes

$$\max_a \mathbb{E}\left\{Q(s_i, a)\right\}.$$

- **Pro:** We do not need a model or a way to predict s_{i+1}.
- **Con:** We need to learn a Q-factor for every state/action pair.

We can learn Q-factors using a TD approach given a trajectory $s_0, a_0, r_0, s_1, a_1, r_1 \ldots, s_T, r_T$:

$$\hat{Q}(s_i, a_i) = r_i + \gamma Q(s_{i+1}, a_{i+1}) \qquad \text{target}$$

$$Q(s_i, a_i) = Q(s_i, a_i) + \alpha\left[\hat{Q}(s_i, a_i) - Q(s_i, a_i)\right] \qquad \text{update}$$

This approach is also called *SARSA*.

The number of Q-factors is much greater than the number of states.

- SARSA, Q-learning, and Double Q-learning

SARSA: Learn Q-factors using a TD approach.

The SARSA update equation is

$$Q(s_i, a_i) \leftarrow Q(s_i, a_i) + \alpha\left[\underbrace{r_i + \gamma Q(s_{i+1}, a_{i+1})}_{\text{target}} - Q(s_i, a_i)\right].$$

Our estimate of $Q(s_i, a_i)$ is based on

- The reward r_i experienced by selecting action a_i in state s_i.
- The future reward $Q(s_{i+1}, a_{i+1})$ based on the action a_{i+1} from the trajectory.

The policy that generates the trajectory is not optimal, so a_{i+1} is not the best action.

Q-learning:

The Q-learning algorithm changes the SARSA update

$$Q(s_i, a_i) \leftarrow Q(s_i, a_i) + \alpha \left[r_i + \gamma Q(s_{i+1}, a_{i+1}) - Q(s_i, a_i) \right]$$

to use the optimal action in state s_{i+1}:

$$Q(s_i, a_i) \leftarrow Q(s_i, a_i) + \alpha \left[r_i + \gamma \max_a Q(s_{i+1}, a) - Q(s_i, a_i) \right].$$

Q-learning can converge faster to an optimal policy. However, it has two drawbacks:

1. If the number of available actions is large, the maximization operator can be expensive to evaluate.
2. The maximization operator is biased.

Any algorithm with a max operator will drift upwards over time, even if the mean value remains fixed.

Double Q-learning:

One solution to the $\max$ bias is using two separate Q functions (networks), called Q_1 and Q_2.

Both Q_1 and Q_2 are trained with separate experiences. (Or, one network car *lag* behind the other in experiences.)

When updating, we use one network to select the action, and the other netwo to compute its value.

$$Q_1(s_i, a_i) \leftarrow Q_1(s_i, a_i) + \alpha \left[r_i + \gamma Q_2(s_{i+1}, a_1) - Q_1(s_i, a_i) \right]$$
$$a_1 \equiv \arg\max_a Q_1(s_{i+1}, a)$$

$$Q_2(s_i, a_i) \leftarrow Q_2(s_i, a_i) + \alpha \left[r_i + \gamma Q_1(s_{i+1}, a_2) - Q_2(s_i, a_i) \right]$$
$$a_2 \equiv \arg\max_a Q_2(s_{i+1}, a)$$

Even if a_1 was selected because $Q_1(s_{i+1}, a_1)$ was aberrantly high, the value $Q_2(s_{i+1}, a_1)$ will not share this bias.

- Neural networks: neurons, layers, activation functions, width, depth, loss, stochastic gradient descent

  Neuron: connects inputs to an output. If combined input exceeds a threshold, the output fires. It's a linear classifier.

  Activation functions: nonlinear function. sign/step function, sigmoid function, rectified linear unit activation.

  A stack of m neurons can be written as

  $$\mathbf{z} = \mathbf{W}\mathbf{x}$$
  $$\mathbf{y} = \boldsymbol{\sigma}(\mathbf{z})$$

  or, more succinctly as

  $$\mathbf{y} = \boldsymbol{\sigma}(\mathbf{W}\mathbf{x})$$

  where

  $$\dim(\mathbf{y}) = m \times 1, \quad \dim(\mathbf{z}) = m \times 1$$
  $$\dim(\mathbf{W}) = m \times n, \quad \dim(\mathbf{x}) = n \times 1$$

  Width&Depth:

  $$\mathbf{y}_d = \boldsymbol{\sigma}(\mathbf{W}_d(\boldsymbol{\sigma}(\mathbf{W}_{d-1}(\cdots\boldsymbol{\sigma}(\mathbf{W}_2(\boldsymbol{\sigma}(\mathbf{W}_1\mathbf{x})))))))$$

  - The number of neurons in each layer i is the *width* of the layer.
    - If the $(i-1)$th layer has n outputs and the ith layer has m outputs, the weight matrix $\mathbf{W}_i$ has dimensions $m \times n$.
    - The dimensions of the inputs $\mathbf{x}$ and outputs $\mathbf{y}_d$ are fixed by the problem.
    - Layer 1 is called the *input layer*, and layer d is the *output layer*.
    - We can use as many nodes as we want in the *hidden* layers.

  The number d is the depth of the neural network. Deep learning means $d > 2$.
  The importance of nonlinearity: without nonlinear activation function, the network reduces to a single linear system.

Universal approximation theorem states given enough neurons, a 2-layer perception can learn any reasonable function.

Deep networks learn more efficient than wide ones. It reduces the total number of neurons needed to learn a neuron since each of the d layers needs fewer than 1/d the number of neurons.
Why deeper networks learn better? Each layer only needs to improve the features for the next layer.

Loss: measures how well the output of the final layer compared with known training data.

Gradient descent: update weights by 1) compute the gradient of the total loss $g(W) = \sum \frac{\partial L_i(W)}{\partial W}$; 2) update the weights using $W^{(1)} = W^{(0)} - \alpha g(W^{(0)})$; 3) repeat previous steps.
But it has problems in gradient calculation.

- The gradient has one entry for each parameter, of which there are thousands!
- These computations are repeated over all N points in the dataset for each iteration.

SGD: alternates between forward and backward passes through the model and updates parameters using the loss for a single point.

1. Select a single training point $(\mathbf{x}_i, y_i^{\text{true}})$.
2. Make a **forward pass** through the model to compute the output of the neural network

$$y_{d,i} = \boldsymbol{\sigma}(\mathbf{W}_d(\boldsymbol{\sigma}(\mathbf{W}_{d-1}(\cdots\boldsymbol{\sigma}(\mathbf{W}_2(\boldsymbol{\sigma}(\mathbf{W}_1\mathbf{x}_i)))))))).$$

3. Compute the loss for this single prediction

$$L_i = (y_{d,i} - y_i^{\text{true}})^2.$$

4. Calculate the gradient at this point and the current weights.
5. Update the weights using

$$\mathbf{W}^{(k+1)} = \mathbf{W}^{(k)} - \alpha \mathbf{g}(\mathbf{W}^{(k)}).$$

   This completes the **backward pass**.
6. Repeat for the next training point.

One pass through the entire training set is called an epoch. One epoch is not enough and the order of the training data is randomized between epochs.
Minibatches: average a small number of training samples before updating the weights.
SGD is stochastic which is a form of regularization.
Why neural networks learn so well?

1. Lottery ticket theory: A good network is already somewhere in the randomly-initialized network.
2. There are no true local optima in high-dimensional spaces.
3. The noise in SGD allows training to "escape" local optima.

How to improve nn?

1. **Regularize**. Neural networks can memorize anything, so we need to regularize them aggressively.
2. **Boosting**. Train in several rounds, weighting the loss function toward points that are not predicted correctly.
3. **Bagging**. Split the training data into parts and train a separate model on each part. Average the predictions of all the models.
4. **Experiment**. Change number and size of layers, hyperparameters, optimizers, batching, and regularization.

- Deep Q-learning

  Approximate Q-factors via deep learning with artificial neural networks.

  Define state space: onehot-encoding (16x4 matrix) vs ignore block states (12x1 trinary vector)

  Steps of deep Q-learning:

  1. Construct a neural network $\widetilde{Q}(s, a; \mathbf{W})$ with random weights $\mathbf{W}$.
  2. Generate a trajectory $s_0, a_0, r_0, \ldots, s_T, r_T$.
  3. For each (s_i, a_i) pair, calculate the Q-factor target

  $$\hat{Q}(s_i, a_i) = r_i + \max_a \widetilde{Q}(s_{i+1}, a).$$

  4. Compute the loss

  $$L(\mathbf{W}) = \left(\hat{Q}(s_i, a_i) - \widetilde{Q}(s_i, a_i; \mathbf{W})\right)^2.$$

  5. Backpropagate the gradient through the neural network and update the weights based on the loss:

  $$\mathbf{W} \leftarrow \mathbf{W} - \alpha \mathbf{g}(\mathbf{W}).$$

  6. Go to #2 and repeat.

For deep Q-learning with terminal rewards, in the near-terminal state: $\hat{Q}(s_{T-1}, a_{T-1}) = r_{T-1} + max_a \tilde{Q}(s_T, a) = 0 + \tilde{Q}(s_T, .) = r_T$. The reward is bootstrapped back through the Q-factors.

To speed up learning, set $\tilde{Q}(s_i, a_i) \approx r_T$, to reward any state/action pair with a win and penalize state/action pair in losing games.

- Learning Q-factors is a dominant method for model-free RL.
- Q-factor estimates are updated by SARSA, Q-learning, or other algorithms.
- Q-factors can be tabulated for every state/action pair, or approximated with a parameterized function.
- Deep RL uses deep neural networks for Q-factor approximation.

- Policy-based methods and REINFORCE
  - Recall that a policy $\pi(a, s)$ returns the probability of selecting action a.
  - Formally, the implicit policy on the previous slide was

$$\pi(a_i, s_i) = \begin{cases} 1 & a_i = \arg\max_a Q(s_i, a) \\ 0 & \text{otherwise} \end{cases}$$

  - Other policies return a distribution over all possible actions, and the agent selects an action based on these probabilities.
  - Often policies are *parameterized* by a vector of tunable parameters θ, i.e. $\pi = \pi(a, s|\theta)$.

To learn policy directly from experience,

The *reward* is the profit for each day:

$$\begin{aligned} r &= \text{revenue} - \text{expenses} \\ &= p \min\{\theta, D\} - c\,\theta \end{aligned}$$

We can compute the stochastic gradient of the daily reward:

$$\frac{dr}{d\theta} = \begin{cases} p - c & \text{if } \theta < D \\ -c & \text{if } \theta > D \end{cases}$$

Over time we can learn an optimal policy by stochastic steepest ascent

$$\theta^{(k+1)} = \theta^{(k)} + \alpha \frac{d}{d\theta} r(\theta^{(k)})$$

for some learning rate α.

Reinforce algorithm:

1. Initialize the policy parameters θ, possibly to random values.
2. Generate a trajectory $s_0, a_0, r_0, \ldots, s_{T-1}, a_{T-1}, r_{T-1}, s_T, r_T$.
3. foreach $i \in \{0, \ldots, T\}$:
   - Calculate the return $R_i = r_i + r_{i+1} + \cdots + r_T$.
   - $\theta \leftarrow \theta + R_i \nabla_\theta \log \pi(a_i, s_i | \theta)$
4. Go to step #2 and repeat.

Note that the policy π can be any parameterized function, including a neural network. The details of the parameter update change based on the structure of π, e.g. by using backpropagation.