Lecture 1, Jan 9, 2024

Types of Learning

- *Supervised learning*: given a training inputs and training targets, we want to learn a relationship for prediction purposes
  - Input data is specified as pairs: $\left\{ (\boldsymbol{x}^{(i)}, y^{(i)}) \right\}_{i=1}^{N}$
  - e.g. regression, classification, time series forecasting, even learning governing equations
  - Main goal is to make predictions
- *Unsupervised learning*: given training data without labels (targets), we want to learn meaningful patterns in the data
  - e.g. clustering, probability density estimation, dimensionality reduction, generative AI models
- *Semi-supervised learning*: targets are only known for a small subset of the training inputs
- *Reinforcement learning*: an agent continuously interacting with the environment learning to maximize a reward function
  - This is an approach for sequential decision making

Parameter Estimation

- Frequentist approach: estimate the model parameters by minimizing a loss function, resulting in a single point in parameter space $\boldsymbol{w}$
  - e.g. regression using least squares
- Bayesian approach: estimate the posterior distribution of the parameters using Bayes' theorem
  - This allows us to also estimate the uncertainty in the model
  - Frequentist estimation is more efficient, but carries no information about the uncertainty

Lecture 2, Jan 12, 2024

Supervised Learning

- Denote the input (aka features) $\boldsymbol{x} = \{ x_1, x_2, \ldots, x_D \}^T \in \mathcal{X} \subseteq \mathbb{R}^D$
  - Extraction of relevant features from data is often necessary (*feature engineering*)
- Denote the target (aka labels) $y \in \mathcal{Y}$
  - Regression: $\mathcal{Y} \subseteq \mathbb{R}$ or $\mathbb{R}^K$
  - Binary classification: $\mathcal{Y} = \{ -1, +1 \}$
  - Multi-class classification: $\mathcal{Y} = \{ 1, 2, \ldots, K \}$
    - This can be decomposed into a sequence of binary classification problems
    - We usually use a one-hot encoding (a K-dimensional target with a 1 in the desired class and 0s elsewhere)
- The probability distribution that the targets and inputs are sampled from is denoted $\mathcal{P}(X, Y), \Pr(X, Y)$ or $p(\boldsymbol{x}, y) = p(x_1, \ldots, x_d, y)$ (joint density of features and targets)
- The expectation is denoted $\mathbb{E}_{\boldsymbol{x}}[g(\boldsymbol{x})] = \int p(\boldsymbol{x}) g(\boldsymbol{x}) \, \mathrm{d}x, \mathbb{E}_y[g(y)|\boldsymbol{x}] = \int p(y|\boldsymbol{x}) g(y) \, \mathrm{d}y$

Definition

Supervised learning: Given the training dataset

$$\mathcal{D} := \{ (\boldsymbol{x}^{(1)}, y^{(1)}), (\boldsymbol{x}^{(2)}, y^{(2)}), \ldots, (\boldsymbol{x}^{(N)}, y^{(N)}) \}$$

with

$$y^{(i)} = f(\boldsymbol{x}^{(i)}) + \epsilon$$

where f is the function we wish to learn and ϵ is some measurement noise; the goal is to find a function $\hat{f} \colon \mathcal{X} \mapsto \mathcal{Y}$ such that

$$\hat{f}(\boldsymbol{x}^{(i)}) \approx f^{(i)} \forall (\boldsymbol{x}^{(i)}, y^{(i)})$$

for both points in $\mathcal{D}$ and outside.

- To make this problem tractable, we need additional assumptions
  - This is due to the No Free Lunch theorem: no single model is best for all problems!
  - We restrict $\hat{f}$ to a set of possible functions that we refer to the *hypothesis class* $\mathcal{H}$ (*model structure specification*)
  - We attempt to solve $\hat{f} = \underset{h \in \mathcal{H}}{\operatorname{argmin}} \mathcal{L}(h)$ where $\mathcal{L} \colon \mathcal{H} \mapsto \mathbb{R}$
- If the hypothesis class is parameterized by $\boldsymbol{w}$, we are essentially solving $\hat{\boldsymbol{w}} = \underset{\boldsymbol{w}}{\operatorname{argmin}} \mathcal{L}(h_{\boldsymbol{w}})$
  - Hypothesis classes can be e.g. neural networks, polynomials, etc
  - e.g. $\mathcal{H}_{\boldsymbol{w}} := \left\{ w_0 + \sum_{i=1}^{D} w_i x_i, \boldsymbol{w} \in \mathbb{R}^{D+1} \right\}$
- The loss function is an expectation with respect to the joint distribution of inputs and outputs: $\mathcal{L}(h) = \mathbb{E}[l(h(\boldsymbol{x}), y)]$
  - Examples:
    * Squared loss: $l(y, y') = (y - y')^2$
    * Absolute loss: $l(y, y') = |y - y'|$
      - This is less affected by outliers compared to squared loss
    * Huber loss: $l(y, y') = \begin{cases} \frac{1}{2}(y - y')^2 & |y - y'| \leq \delta \\ \delta(|y - y'| - \frac{1}{2}\delta) & \text{otherwise} \end{cases}$
    * ϵ-insensitive loss: $l(y, y') = \begin{cases} 0 & |y - y'| \leq \epsilon \\ |y - y'| - \epsilon & \text{otherwise} \end{cases}$
      - Useful in regression tasks where some degree in error tolerance is acceptable
  - For classification, y' is the raw output of the classifier (not the output label, so this can include confidence):
    * Zero-one loss: $l(y, y') = \begin{cases} 1 & y \neq y' \\ 0 & \text{otherwise} \end{cases}$
      - Correctly classified points that are far from the decision boundary (i.e. very confident) are not penalized
      - i.e. this doesn't care how confident we are
    * Hinge loss: $l(y, y') = \max(0, 1 - yy')$
      - Correctly classified points that are close to the decision boundary are penalized
- The actual loss function we want to minimize is the generalized $\mathcal{L}(h) = \mathbb{E}_{(\boldsymbol{x},y) \sim \Pr(\boldsymbol{x},y)}[l(h(\boldsymbol{x}), y)] = \mathcal{L}_{\text{gen}}(h)$
  - Consider the squared error loss; it can be rewritten as $\mathbb{E}_{\boldsymbol{x}}[\mathbb{E}_y[h(\boldsymbol{x})^2 + y^2 - 2h(\boldsymbol{x})y|\boldsymbol{x}]]$
  - We can rearrange the inner expectation as $(h(\boldsymbol{x}) - \mathbb{E}[y|\boldsymbol{x}])^2 + \text{Var}(y|\boldsymbol{x})$
  - The first term can be minimized by choosing $h(\boldsymbol{x}) = \mathbb{E}[y|\boldsymbol{x}]$
  - The second term does not depend on $h(\boldsymbol{x})$; it cannot be minimized because it is the intrinsic variance of the outputs
    * This is the *Bayes error*

  - An algorithm that achieves the Bayes error is *Bayes optimal*; this is not something we can do in practice (if we had all the information about the distribution, we wouldn't need to learn in the first place)
- Let the optimal predictor be $h_*(\boldsymbol{x}) = \mathbb{E}[y|\boldsymbol{x}]$; for any dataset we can run our learning algorithm to get a particular $h(\boldsymbol{x}; \mathcal{D})$
  - Let's take the expectation of the error over all choices of datasets
  - We can rewrite $\mathbb{E}_{\mathcal{D}}[(h(\mathcal{D}) - h_*)^2 + \text{Var}(y)] = (\mathbb{E}_{\mathcal{D}}[h] - h_*)^2 + \text{Var}(h) + \text{Var}(y)$
  - So $\mathbb{E}_{\mathcal{D}}[\mathbb{E}_y[(h(\boldsymbol{x}; \mathcal{D}) - y)^2|\boldsymbol{x}]] = \underbrace{(\mathbb{E}_{\mathcal{D}}[h] - h_*)^2}_{\text{bias}} + \underbrace{\text{Var}(h)}_{\text{variance}} + \underbrace{\text{Var}(y)}_{\text{Bayes error}}$
  - The loss is now decomposed into 3 terms:
    * The *bias* indicates how the average prediction over all datasets differs from the optimal predictor
    * The *variance* indicates how sensitive $h(\boldsymbol{x})$ is to the choice of a particular dataset
    * The *Bayes error* is irreducible noise that is intrinsic to the data generation process
  - There is often a tradeoff between bias and variance; lower bias usually result in high variance and vice-versa
    * Often high bias and high variance are used as synonyms for underfitting and overfitting (even though this technically only applies for squared loss)
- However we can't actually compute $\mathcal{L}_{\text{gen}}(h)$ since we don't know the underlying distribution; we can approximate it using the *empirical loss*: $\mathcal{L}(h) \approx \frac{1}{N} \sum_{i=1}^{N} l(h(\boldsymbol{x}^{(i)}), y^{(i)}) = \mathcal{L}_{\text{emp}}(h)$
  - Minimization of the empirical loss is known as *empirical risk minimization*
  - Convergence in the limit $N \to \infty$ holds due to the weak law of large numbers
- The empirical loss measures performance only on the training set $\mathcal{D}$; this means that the training error can be reduced to zero simply by memorizing the entire training dataset
  - Such a "memorizer" model is useless for predicting on new data, so it's undesirable
  - To guarantee that the out-of-sample error (i.e. the error for data not in $\mathcal{D}$) is low, we need to minimize the generalized loss
  - But we can't actually compute the generalized loss, so much of the focus of theoretical research is on bounding the generalized loss in terms of the empirical loss
- An overly flexible model will memorize irrelevant details of the training set (*overfitting*) whereas simpler models don't have enough degrees of freedom to approximate the underlying function (*underfitting*)
  - Generally if two models fit the data equally well, the simpler model probably generalizes better
- To prevent overfitting, standard practice splits $\mathcal{D}$ into training, validation, and testing sets
  - This works when we have a large amount of data so we can afford to reduce the training dataset
  - The best way to partition is dependent on the problem and size of the dataset available
  - The training set is used to fit the model parameters
  - The validation set is used to select model complexity (i.e. hyperparameters)
  - The testing set is used to estimate the generalization performance
- For smaller datasets, another popular approach is ν-fold cross-validation
  - $\mathcal{D}$ is spit into ν equal partitions/folds
  - For $i = 1, \ldots, \nu$, train models on data in all folds except the i-th, and test on the i-th fold
  - For each model the average loss over all ν folds is calculated
  - Large values of ν typically lead to a large increase in computational cost, but this can be parallelized
  - The choice of ν is dictated by the bias-variance tradeoff; typical values are 5 or 10
    * Increasing ν leads to less bias (since we are averaging over more terms)
    * For $\nu = N$ this is called *leave-one-out error estimation*
- Other approaches to improving generalization:
  - Prior knowledge (feature engineering)
  - Getting more data
  - Fake more data (e.g. adding noise)
  - Ensembles (e.g. bootstrap)
  - Bayesian approaches

Ensembles: Bootstrap Aggregation (Bagging)

- Take the dataset and generate M new datasets by sampling N training examples from $\mathcal{D}$ with replacement
- Train the model separately on each of the M datasets to get models h_i, $i = 1, \ldots, M$
- Average the predictions of models trained on each of the M datasets: $h_{\text{bootstrap}}(\boldsymbol{x}) = \frac{1}{M}\sum_{i=1}^{M} h_i(\boldsymbol{x})$
- The bias remains unchanged: $E_{\mathcal{D}}\left[\frac{1}{M}\sum_{i=1}^{M} h_i\right] = \frac{1}{M}\sum_{i=1}^{M}\mathbb{E}_{\mathcal{D}}[h_i] = E_{\mathcal{D}}[h]$
- The variance can be shown to be $\text{Var}(h) = \frac{1}{M}(1-\rho)\sigma^2 + \rho\sigma^2$ where ρ is the pairwise correlation coefficient between models and σ^2 is the variance
  - If we can reduce the correlation between models, i.e. $\rho \to 0$, we can approach a variance reduction of $\frac{1}{M}$
  - This is difficult to do but even if we can't reduce ρ to 0, bootstrapping generally still reduces the variance

Lecture 3, Jan 16, 2024

k-Nearest Neighbours

Definition

k-Nearest Neighbours: The prediction for a test point $\boldsymbol{x}^*$ is computed as the average output across the k nearest neighbours in the training dataset:

$$\hat{f}(\boldsymbol{x}^*) = \frac{1}{k}\sum_{i \in \mathcal{N}_k(\boldsymbol{x}^*)} y^{(i)}$$

where $\mathcal{N}_k(\boldsymbol{x}^*)$ is the set of k training cases with inputs closest to $\boldsymbol{x}^*$. Alternatively, we can use a weighted average with each weight being inversely proportional to the neighbour's distance from $\boldsymbol{x}^*$.

- k-NN assumes that similar inputs have similar outputs – we're making an assumption on the smoothness of the underlying function
  - This is a memory-based method and does not require any model to be fit
  - k-NN is able to achieve Bayes optimality
- This can also be used for classification, in which case the most common label amongst the k nearest neighbours is used
  - To avoid ties, we can use an odd value of k for binary classification problems
  - For multi-class classification, we can decrease k until there is no longer a tie, reducing to $k = 1$ in the worst case
- With increasing values of k, the model becomes *less* complex; the resulting output becomes smoother, exhibits more bias but less variance
  - For $k = 1$, this is essentially equivalent to constructing a Voronoi diagram of the input data
  - Smaller values of k give more complex decision boundaries but risk overfitting as with any complex model
    - Overfitting makes us more susceptible to outliers
  - Rule of thumb: choose $k < \sqrt{N}$
    - We can also plot loss as shown in the figure and find the minimum
- k-NN requires a similarity/distance metric to find the nearest neighbours
  - Minkowski distance: $\text{dist}(\boldsymbol{x}, \boldsymbol{z}) = \left(\sum_{i=1}^{D}|x_i - z_i|^p\right)^{\frac{1}{p}}$

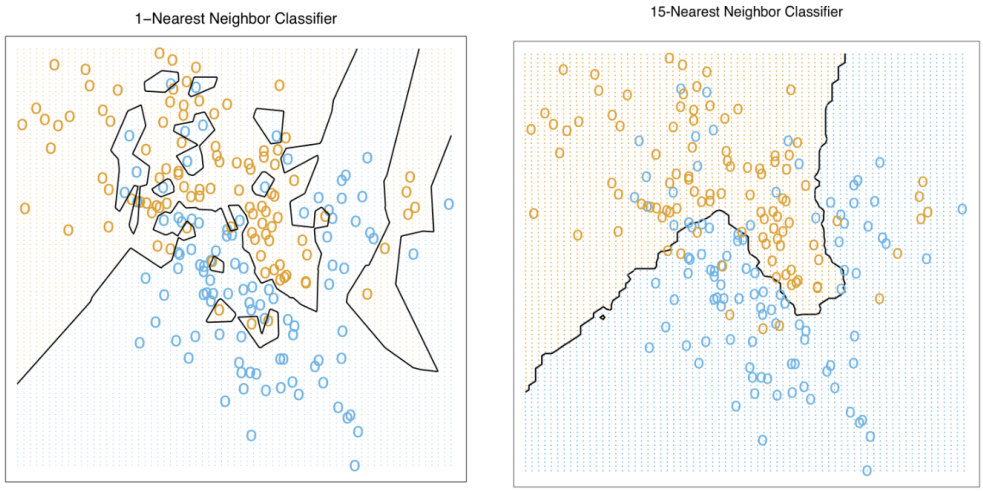

Figure 1: k-NN classifiers for different values of k.

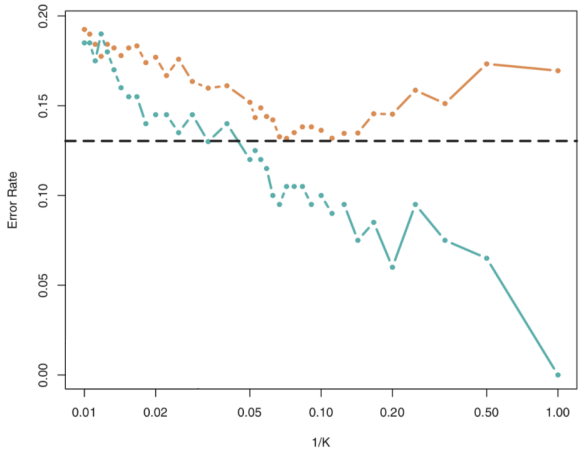

Figure 2: Comparison of training loss (green) and test loss (orange) for different values of k.

        * For $p = 1$ this is Manhattan distance, for $p = 2$ this is Euclidean distance, for $p = \infty$ this is is the max of $|x_i - z_i|$
    - Mahalanobis distance: $\text{dist}(\boldsymbol{x}, \boldsymbol{z}) = \sqrt{(\boldsymbol{x} - \boldsymbol{z})^T \boldsymbol{\Sigma}^{-1} (\boldsymbol{x} - \boldsymbol{z})}$ where $\boldsymbol{\Sigma}$ is the covariance matrix of $\boldsymbol{x}$
- The choice of distance metric plays a key role in performance
    - Algorithms exist to choose the metric automatically
- k-NN can be sensitive to the scale of features, so if scale is unimportant, we should *normalize* each feature to be zero-mean and unit variance
    - Since the distance metric considers each dimension to be equal, the variance in each dimension can have large effects on the nearest neighbour calculations
    - Normalize as $x_i \leftarrow \dfrac{x_i - \mu_i}{\sigma_i}, i = 1, 2, \ldots, D$
    - μ_i, σ_i are the mean and standard deviations of the i-th feature
    - Note we should not normalize in a problem where the units/scale of the axes matter
- Each prediction has a runtime complexity of $\mathcal{O}(ND + N \log N)$ where N is the number of training samples, D is the number of features (dimensionality)
    - This includes both distance calculations and sorting
        * Distance calculations can be parallelized
    - Lots of research exists on efficient implementation of this algorithm
        * Using k-d trees reduces the cost to $\mathcal{O}(D \log N)$ but only if $D \ll N$
        * Randomized approximate NN calculations are more appropriate for sparse, high-dimensional problems
- All training points are required to be stored in order to make predictions, since the model doesn't learn
    - Can use automatic clustering and pick only the center of each cluster
    - Dimensionality reduction as a preprocessing step can reduce memory and time usage

The Curse of Dimensionality

- This is the main problem associated with k-NN; as the number of dimensions increases, the number of training samples we need increases exponentially
- Consider a D-dimensional hypercube $[0, 1]^D$ where all training points are distributed uniformly
- Consider a test point $\boldsymbol{x}^*$; what is the length l of the smallest hypercube within the unit cube that contains the k-nearest neighbours of $\boldsymbol{x}^*$?
    - Due to the uniform distribution, the proportion of points in the cube is equal to the volume of the cube divided by the volume of the unit cube
    - Therefore $l^D \approx \dfrac{k}{N}$ which gives $l \approx \left(\dfrac{k}{N}\right)^{\frac{1}{D}}$
- The value of l increases very quickly with increasing D; with larger values of D, we have $l \approx 1$, so we will have to search almost the entire space
    - But if we are searching the entire space, this means the points might be far apart, so the algorithm will perform very poorly
- How many training points do we need to keep l small?
    - If we want $l = 0.01$, then we can solve to get $N = 100^D k$
    - This exponential growth in the amount of data needed is one of the main problems with k-NN
- Dimensionality reduction can be very important/helpful for this algorithm

Probabilistic k-NN

- How can we make the algorithm probabilistic?
- For the case of binary classification, we can calculate the distribution of labels in the neighbourhood of a point
- If the data is parse, we might have zero probabilities for some classes
    - To overcome this, we can add pseudo-counts to the data and then normalize
    - Add 1 to the count of every category and then renormalise so the distribution still sums to 1
- e.g. binary classification problem with $k = 3$, we have 2 neighbours in class 1 and 1 neighbour in class

2, which gives us $P = [2/3, 1/3]$
  - To counteract the sparse data problem we will instead have $P = [2+1, 1+1]/5$

Summary

k-nearest neighbours algorithm benefits:
- Simple and easy to implement
- Easily parallelized

Drawbacks:
- Choice of simliarity metric has significant impact on performance
- Since the entire training set has to be stored, memory usage can be prohibitive for large datasets
- Sensitive to noise in the labels (outliers/overfitting)
- Susceptible to the curse of dimensionality – high dimensions require exponential amounts of data

Lecture 4, Jan 23, 2024

Linear Regression

- A linear model in general is represented by $\hat{f}(\boldsymbol{x}, \boldsymbol{w}) = w_0 + \sum_{j=1}^{D} w_j x_d$
  - $\boldsymbol{w} = \{ w_0, \dots, w_D \}^T \in \mathbb{R}^{D+1}$ are undetermined weights of the model
  - This is a parametric supervised learning technique
- Using least squares loss gives the optimization problem: $\hat{\boldsymbol{w}} = \underset{\boldsymbol{w} \in \mathbb{R}^{D+1}}{\operatorname{argmin}} \sum_{i=1}^{N} \left(y^{(i)} - w_0 - \sum_{j=1}^{D} w_j x_j^{(i)} \right)^2$
  - Let the dummy feature $x_0 = 1$, then we have $\boldsymbol{x} = \{ x_0, \dots, x_D \}^T \in \mathbb{R}^{D+1}$ so $\hat{f}(\boldsymbol{x}, \boldsymbol{w}) = \boldsymbol{w}^T \boldsymbol{x}$
  - Let $\boldsymbol{X} \in \mathbb{R}^{N \times (D+1)}$ such that the ith row contains $\boldsymbol{x}^{(i)}$, i.e. $\boldsymbol{X}_{ij} = x_j^{(i)}$; this allows us to write the vector of predictions as $\hat{\boldsymbol{y}} = \boldsymbol{X}\boldsymbol{w} \in \mathbb{R}^N$
  - Let $\boldsymbol{y} = \{ y^{(1)}, \dots, y^{(N)} \}^T \in \mathbb{R}^N$
- The problem is then $\hat{\boldsymbol{w}} = \underset{\boldsymbol{w} \in \mathbb{R}^{D+1}}{\operatorname{argmin}} \|\boldsymbol{y} - \boldsymbol{X}\boldsymbol{w}\|_2^2$
  - The loss function is $\mathcal{L}(\boldsymbol{w}) = \|\boldsymbol{y} - \boldsymbol{X}\boldsymbol{w}\|_2^2 = (\boldsymbol{y} - \boldsymbol{X}\boldsymbol{w})^T (\boldsymbol{y} - \boldsymbol{X}\boldsymbol{w})$
  - $$\begin{aligned} \frac{\partial \mathcal{L}}{\partial \boldsymbol{W}} &= \frac{\partial}{\partial \boldsymbol{w}} (\boldsymbol{y}^T \boldsymbol{y} + \boldsymbol{w}^T \boldsymbol{X}^T \boldsymbol{X} \boldsymbol{w} - 2\boldsymbol{y}^T \boldsymbol{X} \boldsymbol{w}) \\ &= 2\boldsymbol{X}^T \boldsymbol{X} \boldsymbol{w} - 2\boldsymbol{X}^T \boldsymbol{y} \\ &= 2\boldsymbol{X}^T (\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y}) = \boldsymbol{0} \end{aligned}$$
    - Note: $\frac{\partial}{\partial z}(\boldsymbol{z}^T \boldsymbol{A} \boldsymbol{z}) = (\boldsymbol{A} + \boldsymbol{A}^T)\boldsymbol{z}$, $\frac{\partial}{\partial \boldsymbol{z}}(\boldsymbol{A}\boldsymbol{z}) = \boldsymbol{A}^T$
- Therefore we need to solve $\boldsymbol{X}^T \boldsymbol{X} \boldsymbol{w} = \boldsymbol{X}^T \boldsymbol{y}$
  - $\boldsymbol{X}^T \boldsymbol{X}$ is invertible if $\boldsymbol{X}$ is full rank, i.e. if the features are linearly independent
    - Note equations of this form are known as *normal equations*
  - Can be interpreted as a projection scheme since we are enforcing that $(\boldsymbol{y} - \hat{\boldsymbol{y}}) \perp \boldsymbol{X}_i$ for all columns $\boldsymbol{X}_i$ of $\boldsymbol{X}$ (the residual should be orthogonal to the column space)
- We're essentially trying to solve $\boldsymbol{X}\boldsymbol{w} = \boldsymbol{y}$ where $\boldsymbol{X} \in \mathbb{R}^{N \times (D+1)}, \boldsymbol{w} \in \mathbb{R}^{D+1}, \boldsymbol{y} \in \mathbb{R}^N$
  - The problem is *overdetermined* if $N > D+1$ (i.e. we have more data points than feature dimensions, so $\boldsymbol{X}$ is tall and skinny)
    - We therefore cannot find $\boldsymbol{w}$ to solve this equation, so we can only minimize the residual
  - The problem is *undetermined* if $N < D+1$ (i.e. we have more dimensions than data points, so $\boldsymbol{X}$ is short and fat)
    - This would have an infinite number of solutions, so we need to impose additional constraints

Solving for the Weights

- Cholesky decomposition: $\boldsymbol{X}^T\boldsymbol{X} = \boldsymbol{R}^T\boldsymbol{R}$ where $\boldsymbol{R} \in \mathbb{R}^{(D+1)\times(D+1)}$ is upper triangular
  - Note this is only possible since $\boldsymbol{X}^T\boldsymbol{X}$ is symmetric positive definite if it is full rank
  - Then we have $\hat{\boldsymbol{w}} = (\boldsymbol{X}^T\boldsymbol{X})^{-1}\boldsymbol{X}^T\boldsymbol{y} = \boldsymbol{R}^{-1}\boldsymbol{R}^{-T}\boldsymbol{X}^T\boldsymbol{y}$
    - Note $\boldsymbol{R}^{-T} = (\boldsymbol{R}^{-1})^T = (\boldsymbol{R}^T)^{-1}$
  - Computationally this involves a forward and backward substitution to invert the upper and lower triangular matrices
    - First solve for $\boldsymbol{z} = \boldsymbol{R}^{-T}\boldsymbol{x}^T\boldsymbol{y}$, then $\boldsymbol{w} = \boldsymbol{R}^{-1}\boldsymbol{z}$
    - Both inverses are easy to compute due to them being triangular
  - Note it is common to add a small perturbation, replacing $\boldsymbol{X}^T\boldsymbol{X}$ with $\boldsymbol{X}^T\boldsymbol{X} + \lambda\boldsymbol{I}$ to prevent ill-conditioning; this is equivalent to l_2 regularization
  - Cost: $\mathcal{O}(N(D+1)^2 + \frac{1}{3}(D+1)^3)$
    - Computing $\boldsymbol{R}$ takes $mn^2 + \dfrac{1}{3}n^3$ flops
- Economic QR (aka. reduced or thin QR): $\boldsymbol{X} = \boldsymbol{Q}\boldsymbol{R}$ where $\boldsymbol{Q} \in \mathbb{R}^{N\times(D+1)}$ is orthonormal, $\boldsymbol{R} \in \mathbb{R}^{(D+1)\times(D+1)}$ is upper-triangular
  - $\boldsymbol{X}^T\boldsymbol{X}\boldsymbol{w} = \boldsymbol{X}^T\boldsymbol{y} \implies \boldsymbol{R}^T\boldsymbol{Q}^T\boldsymbol{Q}^T\boldsymbol{R}\boldsymbol{w} = \boldsymbol{R}^T\boldsymbol{Q}^T\boldsymbol{y} \implies \boldsymbol{R}^T\boldsymbol{R}\boldsymbol{w} = \boldsymbol{R}^T\boldsymbol{Q}^T\boldsymbol{y}$
  - Then we have $\hat{\boldsymbol{w}} = \boldsymbol{R}^{-1}\boldsymbol{Q}^T\boldsymbol{y}$
  - Note instead of directly inverting R, we again use a backward substitution
  - This method can fail when $\boldsymbol{X}$ is nearly rank-deficient (i.e. two data points being close together); in this case, SVD is a more robust option
  - Cost: $\mathcal{O}(2N(D+1)^2 + \frac{2}{3}(D+1)^3)$ (approximate)
    - QR factorization costs about $2mn^2$ flops; for $m \gg n$ Cholesky is faster, but only a factor of 2 at most
- Singular value decomposition: $\boldsymbol{X} = \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^T$ where $\boldsymbol{U} \in \mathbb{R}^{N\times N}, \boldsymbol{V} \in \mathbb{R}^{(D+1)\times(D+1)}$ are orthogonal and $\boldsymbol{\Sigma} \in \mathbb{R}^{N\times(D+1)}$ is rectangular diagonal
  - Note we can write this as $\boldsymbol{X} = \begin{bmatrix}\boldsymbol{U}_1 & \boldsymbol{U}_2\end{bmatrix}\begin{bmatrix}\boldsymbol{\Sigma}_1 \\ \boldsymbol{0}\end{bmatrix}\boldsymbol{V}^T$ or $\boldsymbol{X} = \boldsymbol{U}_1\boldsymbol{\Sigma}_1\boldsymbol{V}^T$
  - $\|\boldsymbol{y} - \boldsymbol{X}\boldsymbol{w}\|_2^2 = \|\boldsymbol{U}^T(\boldsymbol{y} - \boldsymbol{X}\boldsymbol{w})\|_2^2 = \left\|\begin{bmatrix}\boldsymbol{U}_1^T\boldsymbol{y} \\ \boldsymbol{U}_2\boldsymbol{y}\end{bmatrix} - \begin{bmatrix}\boldsymbol{\Sigma}_1\boldsymbol{V}^T\boldsymbol{w} \\ \boldsymbol{0}\end{bmatrix}\right\|_2^2 = \|\boldsymbol{U}_1^T\boldsymbol{y} - \boldsymbol{\Sigma}_1\boldsymbol{V}^T\boldsymbol{w}\|_2^2 + \|\boldsymbol{U}_2^T\boldsymbol{y}\|_2^2$
    - Since $\boldsymbol{U}$ is orthogonal, we can multiply any vector by it and not change the norm
  - We can now minimize this by choosing $\hat{\boldsymbol{w}} = \boldsymbol{V}\boldsymbol{\Sigma}_1^{-1}\boldsymbol{U}_1^T\boldsymbol{y} = \displaystyle\sum_{i=1}^{D+1}\boldsymbol{v}_i\frac{\boldsymbol{u}_i^T\boldsymbol{y}}{\sigma_i}$
    - The summation format is more efficient since $\boldsymbol{\Sigma}$ is diagonal
    - If some singular values are very small, we can truncate this summation for better numerical stability
  - Alternatively the same result can be obtained by simply substituting the SVD into the original expression
  - Cost: $\mathcal{O}(2N(D+1)^2 + 11N(D+1)^3)$ (approximate)
- Moore-Penrose pseudoinverse: $\boldsymbol{X}^\dagger = (\boldsymbol{X}^T\boldsymbol{X})^{-1}\boldsymbol{X}^T = \boldsymbol{V}\boldsymbol{\Sigma}^\dagger\boldsymbol{U}^T$
  - Then $\hat{\boldsymbol{w}} = \boldsymbol{X}^\dagger\boldsymbol{y}$ when $\boldsymbol{X}$ is full rank (so $\boldsymbol{X}^T\boldsymbol{X}$ is symmetric positive definite)
  - Using QR and SVD we can also write $\boldsymbol{X}^\dagger = \boldsymbol{R}^{-1}\boldsymbol{Q}^T$ or $\boldsymbol{X}^\dagger = \boldsymbol{V}\boldsymbol{\Sigma}_1^{-1}\boldsymbol{U}_1^T$
  - If $\boldsymbol{X}$ is rank deficient, then we can take $\hat{\boldsymbol{w}} = \boldsymbol{V}\boldsymbol{\Sigma}_1^\dagger\boldsymbol{U}_1^T\boldsymbol{y}$
    - $\boldsymbol{\Sigma}_1^\dagger = \operatorname{diag}\{\sigma_1^\dagger, \ldots, \sigma_{D+1}^\dagger\}$ and $\sigma_i^\dagger = \begin{cases}\frac{1}{\sigma_i} & \sigma_i > 0 \\ 0 & \text{otherwise}\end{cases}$
- The *condition number* for linear least-squares is defined as $\kappa(\boldsymbol{X}) = \|\boldsymbol{X}\|\|\boldsymbol{X}^\dagger\| = \dfrac{\sigma_{\max}}{\sigma_{\min}}$
  - This is a measure of the sensitivity of the weights to perturbations in the training data
  - High condition numbers can occur in learning problems where the features are strongly correlated
  - Rule of thumb: one digit of precision is lost for every power of 10 in the condition number
    - e.g. IEEE doubles have 16 digits of accuracy, so if a matrix has a condition number of 10^{10} we will only get 6 digits of accuracy

  - Note $\kappa(\boldsymbol{X}^T\boldsymbol{X}) = (\kappa(\boldsymbol{X}))^2$, i.e. when solving normal equations we square the condition number!
    * $\kappa(\boldsymbol{X}^T\boldsymbol{X} + \lambda\boldsymbol{I}) \leq \kappa(\boldsymbol{X}^T\boldsymbol{X})$ for all positive λ
  - On the other hand, performing QR and SVD decomposition keeps the same condition number, so using these methods are a lot more stable
- In general SVD is more expensive than QR and Cholesky, but more numerically stable and can handle rank deficiencies
  - Which one to use is problem dependent
  - From Cholesky to QR to SVD we have increasing stability but also computational cost
- Storing $\boldsymbol{X}, \boldsymbol{y}$ use $\mathcal{O}(ND) + \mathcal{O}(N)$ memory
  - Using economy QR and SVD will require additional $\mathcal{O}(ND)$ memory
    * Full QR and SVD is never practical for large datasets!
  - If the problem is too large to fit into memory, we will need iterative methods that compute the result term-by-term
- Another alternative is to use gradient descent
  - $\boldsymbol{w} \leftarrow \boldsymbol{w} - \dfrac{\alpha}{2N}\vec{\nabla}_{\boldsymbol{w}}\mathcal{L} = \boldsymbol{w} - \dfrac{\alpha}{N}\boldsymbol{X}^T(\hat{\boldsymbol{y}} - \boldsymbol{y}) = \boldsymbol{w} - \dfrac{\alpha}{N}\sum_{i=1}^{N}(\hat{y}^{(i)} - y^{(i)})\boldsymbol{x}^{(i)}$
  - Each iteration requires an additional $\mathcal{O}(ND)$ cost

Underdetermined Least Squares

- Assume that $\text{rank}(\boldsymbol{X}) = N$
- We need to impose additional constraints to get a unique solution
- Heavily underdetermined equations routinely arise in the field of *compressive sensing* and bioinformatics
- One approach is to use $\hat{\boldsymbol{w}} = \underset{\boldsymbol{w}\in\mathbb{R}^{D+1}}{\text{argmin}}\|\boldsymbol{w}\|_2^2$ such that $\boldsymbol{X}\boldsymbol{w} = \boldsymbol{y}$
  - This gives the *minimum norm solution* to the least squares-problem
  - Let $\boldsymbol{\lambda} \in \mathbb{R}^N$ be the Lagrange multipliers
  - The Lagrangian is $L(\boldsymbol{w}, \boldsymbol{\lambda}) = \boldsymbol{w}^T\boldsymbol{w} + \boldsymbol{\lambda}^T(\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y})$
  - The optimality condition is $\vec{\nabla}_{\boldsymbol{w}}L = 2\boldsymbol{w} + \boldsymbol{X}^T\boldsymbol{\lambda} = 0$ and $\vec{\nabla}_{\boldsymbol{\lambda}}L = \boldsymbol{X}\boldsymbol{w} - \boldsymbol{y} = 0$
    * Solve: $\boldsymbol{w} = -\dfrac{1}{2}\boldsymbol{X}^T\boldsymbol{\lambda}$ and $\boldsymbol{\lambda} = -2(\boldsymbol{X}\boldsymbol{X}^T)^{-1}\boldsymbol{y}$
  - Therefore $\hat{\boldsymbol{w}} = \boldsymbol{X}^T(\boldsymbol{X}\boldsymbol{X}^T)^{-1}\boldsymbol{y}$
    * Note in practice we do not calculate this inverse explicitly but instead use a factorization scheme for better stability
  - Other options for constraints also exist such as minimizing the 1-norm
- Using QR factorization: factorize $\boldsymbol{X}^T = \boldsymbol{Q}\boldsymbol{R}$, then $\hat{\boldsymbol{w}} = \boldsymbol{Q}\boldsymbol{R}^{-T}\boldsymbol{y}$
- Using economy SVD: $\boldsymbol{X}^T = \boldsymbol{U}_1\boldsymbol{\Sigma}_1\boldsymbol{V}^T$, then $\hat{\boldsymbol{w}} = \boldsymbol{U}_1\boldsymbol{\Sigma}_1^{-1}\boldsymbol{V}^T\boldsymbol{y}$
  - If $\boldsymbol{X}^T$ is not full rank we can use the same thresholding technique as overdetermined least squares (ignoring nearly zero singular values)

Regression Models for Classification

- For a binary classification problem ($y = +1, -1$), consider a model that minimizes the l_2 loss and makes predictions as $\text{sgn}\,\hat{f}(\boldsymbol{x})$
- Notice that in the example above, the model became much worse after including the additional data
  - This is because we used a loss function that is inappropriate for classification!
- Make the labels instead $y \in \{0, 1\}$, and normalize the predictions using the logistic (aka sigmoid) function: $\sigma(z) = \dfrac{1}{1 + e^{-z}}$ and make predictions as $\hat{y} = \sigma(\boldsymbol{w}^T\boldsymbol{x})$ and threshold at 0.5
- We can interpret $\hat{y}$ as the estimated probability of $y = 1$, and use a loss function that captures the idea that more confidence on a wrong prediction should incur a higher penalty
  - For this, use cross-entropy loss: $\mathcal{L}_{CE}(\boldsymbol{w}) = \dfrac{1}{N}\sum_{i=1}^{N}\left[-y^{(i)}\log\hat{y}^{(i)} - (1 - y^{(i)})\log(1 - \hat{y}^{(i)})\right]$

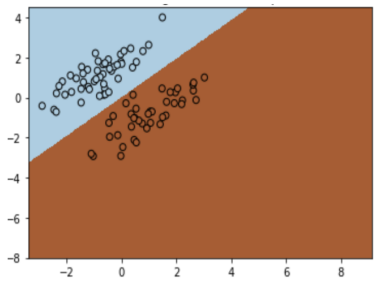

Figure 3: The case of two linearly separable classes that are in clusters.

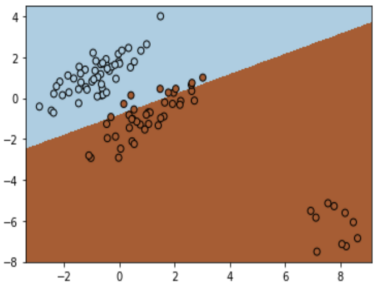

Figure 4: The model after including additional training points.

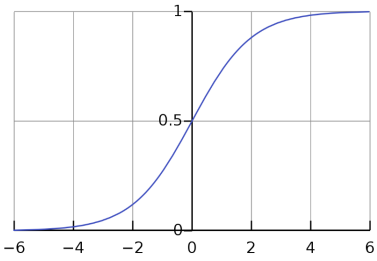

Figure 5: Plot of the logistic function.

- Note for multi-class classification, we can have successive classifiers that pick out one class at a time (Lecture 2), or formulate as a multi-output regression problem and use one-hot encodings
- Note the gradient: $\vec{\nabla}_{\boldsymbol{w}} \mathcal{L}_{CE}(\boldsymbol{w}) = \frac{1}{N} \sum_{i=1}^{N} (\hat{y}^{(i)} - y^{(i)}) \boldsymbol{x}^{(i)}$
  - The gradient is the same as the gradient of the least squares loss; this is not a coincidence

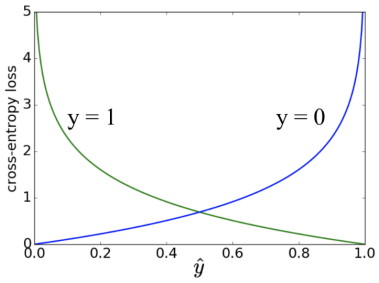

Figure 6: Plot of the cross-entropy loss.

Lecture 5, Jan 26, 2024

Principal Component Analysis (PCA)

Definition

Dimensionality Reduction: Given a dataset $\mathcal{D} = \{\boldsymbol{x}^{(i)}\}_{i=1}^{N}$ where $\boldsymbol{x}^{(i)} \in \mathbb{R}^D$, find a mapping $f : \mathbb{R}^D \mapsto \mathbb{R}^d$ where $d < D$ is a lower dimensional space.

- Dimensionality reduction is a type of unsupervised learning
  - PCA is a dimensionality reduction technique
  - Other techniques can include autoencoders, etc
- Dimensionality reduction can be used for a number of purposes:
  - Saving computational time/memory (helps with the curse of dimensionality)
  - Reduces overfitting
  - Visualize high-dimensional datasets
- We're essentially trying to create a summary of the data
- PCA is one of the only dimensionality reduction techniques with a closed-form solution
- PCA uses a linear model with the form $\boldsymbol{z} = \boldsymbol{U}^T(\boldsymbol{x} - \boldsymbol{b})$ where $\boldsymbol{U} \in \mathbb{R}^{D \times d}$ is an orthonormal matrix and $\boldsymbol{b} \in \mathbb{R}^D$
  - These orthonormal columns form a basis for a subspace $\mathcal{S}$
  - The projection of $\boldsymbol{x}$ onto $\mathcal{S}$ is the point $\tilde{\boldsymbol{x}} \in \mathcal{S}$ closes to $\boldsymbol{x}$ (this is known as the *reproduction* of $\boldsymbol{x}$
  - $\boldsymbol{z}$ is the *representation* or *code* of $\boldsymbol{x}$
- Choose $\boldsymbol{b} = \frac{1}{N} \sum_{i=1}^{N} \boldsymbol{x}^{(i)}$
- Finding a general matrix $\boldsymbol{U}$ is challenging, so we will start with a single column vector $\boldsymbol{u}$
  - We aim to minimize the reconstruction error: $\mathcal{L}(\boldsymbol{u}, \boldsymbol{b}) = \frac{1}{N} \sum_{i=1}^{N} \|\boldsymbol{x}^{(i)} - (\boldsymbol{u}\boldsymbol{u}^T(\boldsymbol{x}^{(i)} - \boldsymbol{b}) + \boldsymbol{b})\|_2^2$
    - $\hat{\boldsymbol{x}}^{(i)} = \boldsymbol{u}\boldsymbol{z} + \boldsymbol{b} = \boldsymbol{u}\boldsymbol{u}^T(\boldsymbol{x} - \boldsymbol{b}) + \boldsymbol{b}$
  - If the data is centered then $\boldsymbol{b} = \boldsymbol{0}$, so $\mathcal{L}(\boldsymbol{u}) = \frac{1}{N} \sum_{i=1}^{N} \|\boldsymbol{x}^{(i)} - \boldsymbol{u}\boldsymbol{u}^T\boldsymbol{x}^{(i)}\|_2^2$
- Expanding the reconstruction error:

- $\mathcal{L}(\boldsymbol{u}) = \frac{1}{N}\sum_{i=1}^{N}(\boldsymbol{x}^{(i)} - \boldsymbol{u}\boldsymbol{u}^T\boldsymbol{x}^{(i)})^T(\boldsymbol{x}^{(i)} - \boldsymbol{u}\boldsymbol{u}^T\boldsymbol{x}^{(i)})$
$= \frac{1}{N}\sum_{i=1}^{n} -2{\boldsymbol{x}^{(i)}}^T\boldsymbol{u}\boldsymbol{u}^T\boldsymbol{x}^{(i)} + {\boldsymbol{x}^{(i)}}^T\boldsymbol{u}\boldsymbol{u}^T\boldsymbol{u}\boldsymbol{u}^T\boldsymbol{x}^{(i)} + \text{const}$
$= \frac{1}{N}\sum_{i=1}^{N} -{\boldsymbol{x}^{(i)}}^T\boldsymbol{u}\boldsymbol{u}^T\boldsymbol{x}^{(i)} + \text{const}$
- So we can formate the problem as minimizing $-\frac{1}{N}\sum_{i=1}^{N}{\boldsymbol{x}^{(i)}}^T\boldsymbol{u}\boldsymbol{u}^T\boldsymbol{x}^{(i)}$ subject to $\boldsymbol{u}^T\boldsymbol{u} = 1$
- Equivalently, maximize $\frac{1}{N}\sum_{i=1}^{N}{\boldsymbol{x}^{(i)}}^T\boldsymbol{u}\boldsymbol{u}^T\boldsymbol{x}^{(i)} = \frac{1}{N}\sum_{i=1}^{N}\|\boldsymbol{z}^{(i)}\|_2^2$ subject to $\boldsymbol{u}^T\boldsymbol{u} = 1$

- Note the mean of $\boldsymbol{z}$ is zero since we centered $\boldsymbol{x}$ so the objective function is equivalent to $\frac{1}{N}\sum_{i=1}^{N}\|\boldsymbol{z}^{(i)} - \bar{\boldsymbol{z}}\|_2^2$
  - Minimizing the reconstruction error is equivalent to maximizing the variance of the code vectors
- $\frac{1}{N}\sum_{i=1}^{N}\|\boldsymbol{z}^{(i)}\|_2^2 = \frac{1}{N}\sum_{i=1}^{N}\boldsymbol{u}^T(\boldsymbol{x}^{(i)} - \boldsymbol{\mu})(\boldsymbol{x}^{(i)} - \boldsymbol{\mu})^T\boldsymbol{u}$
$= \boldsymbol{u}^T\left[\frac{1}{N}\sum_{i=1}^{N}(\boldsymbol{x}^{(i)} - \boldsymbol{\mu})(\boldsymbol{x}^{(i)} - \boldsymbol{\mu})^T\right]\boldsymbol{u}$
$= \boldsymbol{u}^T\boldsymbol{\Sigma}\boldsymbol{u}$
$= \boldsymbol{u}^T\boldsymbol{Q}\boldsymbol{\Lambda}\boldsymbol{Q}^T\boldsymbol{u}$
$= \boldsymbol{a}^T\boldsymbol{\Lambda}\boldsymbol{a}$
$= \sum_{j=1}^{D}\lambda_j\boldsymbol{a}_j^2$
  - We can decompose $\boldsymbol{\Sigma}$ since it is symmetric positive definite, as it is the empirical covariance matrix
  - $\boldsymbol{a} = \boldsymbol{Q}^T\boldsymbol{u}$ is a change of basis to the eigenbasis of $\boldsymbol{\Sigma}$
- Assuming all λ_i are sorted and distinct, we can choose $a_1 = \pm 1$ and $a_j = 0$ (since the first eigenvalue is the largest eigenvalue) in order to maximize the objective
  - Therefore $\boldsymbol{u} = \boldsymbol{Q}\boldsymbol{a} = \boldsymbol{q}_1$ which is just the top eigenvector
  - More generally, we can show that the kth principal component is given by the kth eigenvector of $\boldsymbol{\Sigma}$ (Courant-Fischer Theorem)
- Alternative derivation: we want to maximize a
  - The Lagrangian is $\mathcal{L}(\boldsymbol{u}, \gamma) = \boldsymbol{u}^T\boldsymbol{\Sigma}\boldsymbol{u} + \gamma(1 - \boldsymbol{u}^T\boldsymbol{u})$
  - $\vec{\nabla}_{\boldsymbol{u}}\mathcal{L} = (\boldsymbol{\Sigma} + \boldsymbol{\Sigma}^T)\boldsymbol{u} - 2\gamma\mathbf{1}\boldsymbol{u} = 0 \implies 2\boldsymbol{\Sigma}\boldsymbol{u} = 2\gamma\boldsymbol{u} \implies \boldsymbol{\Sigma}\boldsymbol{u} = \gamma\boldsymbol{u}$
- We can also perform PCA with SVD:
  - If $\boldsymbol{X}$ is a data matrix written in centered form, then the covariance matrix is $\boldsymbol{\Sigma} = \frac{1}{N}\boldsymbol{X}^T\boldsymbol{X}$
  - Using an SVD, we can write $\boldsymbol{\Sigma} = \boldsymbol{V}\boldsymbol{S}_1\boldsymbol{U}_1^T\boldsymbol{U}_1\boldsymbol{S}\boldsymbol{V}^T = \frac{1}{N}\boldsymbol{V}\boldsymbol{S}_1^2\boldsymbol{V}^T$
  - Since this is equal to $\boldsymbol{Q}\boldsymbol{\Lambda}\boldsymbol{Q}^T$ and spectral decompositions are unique, we must have that the columns of $\boldsymbol{V}$ are the principal components and $\frac{\boldsymbol{S}_1^2}{N} = \boldsymbol{\Lambda}$
  - So to construct the PCA we can just take the first d columns
  - Using SVD is faster and more stable
- Note the code vectors given by PCA are de-correlated (i.e. their covariance matrix is diagonal)

Lecture 6, Feb 2, 2024

Generalized Linear Models (GLMs)

Definition

Generalized Linear Models: A GLM is given by

$$\hat{f}(\boldsymbol{x}, \boldsymbol{w}) = w_0 + \sum_{i=1}^{M-1} w_i \phi_i(\boldsymbol{x})$$

where $\boldsymbol{w}$ is a set of undetermined weigths and $\phi_i \colon \mathbb{R}^D \mapsto \mathbb{R}$ are a set of known basis functions.

- The models may be nonlinear in the inputs $\boldsymbol{x}$, but still linear in the weights $\boldsymbol{w}$, which makes is still possible to use linear techniques
- To construct a GLM we need to select the appropriate basis functions, and formulate a strategy to estimate the weights
- Let $\phi_0(\boldsymbol{x}) = 1$ (the bias term) and $\boldsymbol{\phi}(\boldsymbol{x}) = \begin{bmatrix} \phi_0(\boldsymbol{x}) \\ \phi_1(\boldsymbol{x}) \\ \vdots \\ \phi_{M-1}(\boldsymbol{x}) \end{bmatrix} \in \mathbb{R}^M$
  - Then if we define the weight vector $\boldsymbol{w} = \begin{bmatrix} w_0 & \dots & w_m \end{bmatrix}^T$, we can write $\hat{f}(\boldsymbol{x}, \boldsymbol{w}) = \boldsymbol{w}^T \boldsymbol{\phi}(\boldsymbol{x})$
  - We are using ϕ to map from the input space $\mathcal{X}$ to the *feature space* $\mathcal{F}$, and performing linear regression in the feature space
- Let the vector of training targets $\boldsymbol{y} = \begin{bmatrix} y^{(1)} & \dots & y^{(N)} \end{bmatrix}^T \in \mathbb{R}^N$ and $\boldsymbol{\Phi} \in \mathbb{R}^{N \times M}$ where the ith row contains $\boldsymbol{\phi}(\boldsymbol{x}^{(i)}) \in \mathbb{R}^M$
  - Then $\hat{\boldsymbol{y}} = \boldsymbol{\Phi}\boldsymbol{w}$
- Use the l_2 loss function $\hat{\boldsymbol{w}} = \underset{\boldsymbol{w} \in \mathbb{R}^M}{\operatorname{argmin}} \|\boldsymbol{y} - \hat{\boldsymbol{y}}\|_2^2$
  - Again the loss function can be written as $(\boldsymbol{y} - \boldsymbol{\Phi}\boldsymbol{w})^T (\boldsymbol{y} - \boldsymbol{\Phi}\boldsymbol{w})$
  - We can use the same techniques for the linear model, but instead of $D + 1$ weights we have M weights
  - We essentially replace $\boldsymbol{X} \in \mathbb{R}^{N \times (D+1)}$ with $\boldsymbol{\Phi} \in \mathbb{R}^{N \times M}$
- Derivation:
  - $\mathcal{L}(\boldsymbol{w}) = \boldsymbol{y}^T \boldsymbol{y} + \boldsymbol{w}^T \boldsymbol{\Phi}^T \boldsymbol{\Phi} \boldsymbol{w} - 2\boldsymbol{y}^T \Phi \boldsymbol{w}$
  - $\dfrac{\partial \mathcal{L}}{\partial \boldsymbol{w}} = (\boldsymbol{\Phi}^T \boldsymbol{\Phi} + (\boldsymbol{\Phi}^T \boldsymbol{\Phi})^T)\boldsymbol{w} - 2\boldsymbol{\Phi}^T \boldsymbol{y} = 2\boldsymbol{\Phi}^T \boldsymbol{\Phi} \boldsymbol{w} - 2\boldsymbol{\Phi}^T \boldsymbol{y} = 0$
  - Therefore $\boldsymbol{\Phi}^T \boldsymbol{\Phi} \boldsymbol{w} = \boldsymbol{\Phi}^T \boldsymbol{y}$
- We can use the same techniques as linear models to solve the normal equations:
  - Cholesky factorization
    - Avoid this because the condition number is squared
  - (Economy) QR factorization
    - Use this only if $\boldsymbol{\Phi}$ is not rank-deficient
  - (Economy) SVD, or truncated SVD if $\boldsymbol{\Phi}$ is rank-deficient
    - Slowest, but the most stable

Polynomial Regression

- The basis functions are the univariate polynomials $\{ 1, x_i, x_i^2, \dots, x_i^p \}$ up to order p
- If $D = 1$, then the basis functions are $\phi_i = x^i$ so $\hat{f}(x) = w_0 + w_1 x + \cdots + w_p x^p$
  - Note taking tensor products of higher-order univariate polynomials is not a good idea since we will generate p^D basis functions

– For arbitrary D we would have $\boldsymbol{\phi}(\boldsymbol{x}) = \begin{bmatrix} 1 \\ x_1 \\ \vdots \\ x_D \\ x_1 x_2 \\ \vdots \\ x_{D-1} x_D \\ \vdots \\ x_1 x_2 \ldots x_D \end{bmatrix}$

– We can circumvent this with the *kernel trick* covered later

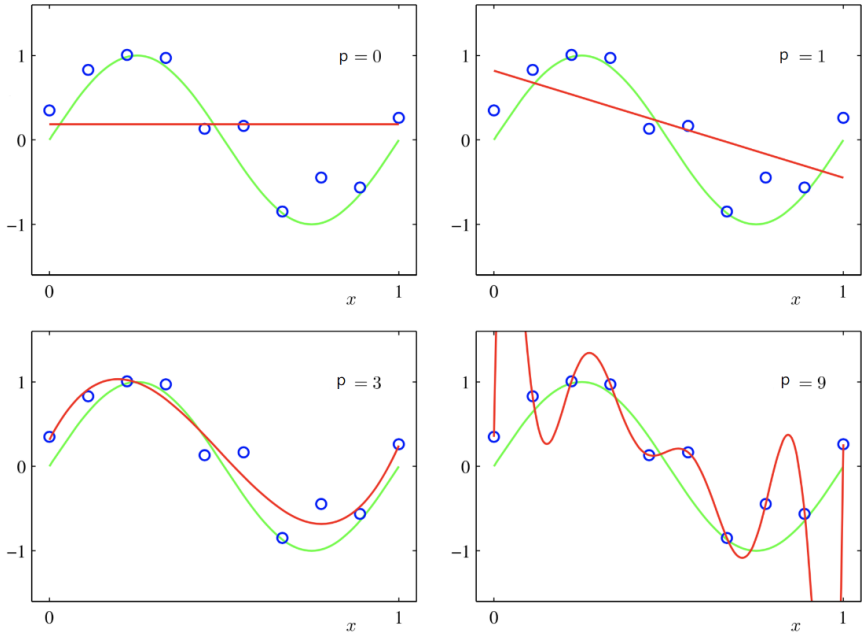

Figure 7: 1D polynomial regression for various values of p. (Note $M = p + 1$.)

- Results for various values of p are shown above
  - Notice that for smaller values of p the model doesn't fit well since it doesn't have enough complexity (underfitting)
  - But for large values of p, the polynomial matches the training points perfectly but approximates the underlying function poorly
- To prevent overfitting, we need to restrict the number of features M (which in this case restricts the degree of the polynomial
  - Increasing the number of data points also helps but we often can't just get more data
  - What if we know the underlying model is complex but we don't have enough data points?
  - How do we deal with noise?

Regularization

- One pattern we may notice is that when the model is overfitting (M too large), the weights start becoming very big in magnitude
- *Regularization* tries to keep the magnitudes of the weights reasonably small, as a way to prevent overfitting
- To keep the weight small, we can introduce the norm of the weights to the loss function, so the model is penalized for having weights that are too large

Definition

Ridge Regression Method: Choose the weights as

$$\hat{\boldsymbol{w}} = \underset{\boldsymbol{w}\in\mathbb{R}^M}{\operatorname{argmin}} |\boldsymbol{y} - \boldsymbol{\Phi}\boldsymbol{w}|_2^2 + \lambda \|\boldsymbol{w}\|_2^2$$

where λ is the *regularization parameter*.

- Note w_0 is often excluded from the regularization term
- The regularized loss function is also quadratic in $\boldsymbol{w}$, so we can use the same steps as before
  - Expanded loss: $\boldsymbol{y}^T\boldsymbol{y} + \boldsymbol{w}^T\boldsymbol{\Phi}^T\boldsymbol{\Phi}\boldsymbol{w} - 2\boldsymbol{w}^T\boldsymbol{\Phi}^T\boldsymbol{y} + \lambda\boldsymbol{w}^T\boldsymbol{w}$
  - $\dfrac{\partial \mathcal{L}}{\partial \boldsymbol{w}} = 2\boldsymbol{\Phi}^T\boldsymbol{\Phi}\boldsymbol{w} - 2\boldsymbol{\Phi}^T\boldsymbol{y} + 2\lambda\boldsymbol{w} = 0$
  - Rearrange: $\boldsymbol{\Phi}^T\boldsymbol{\Phi}\boldsymbol{w} + \lambda\boldsymbol{w} = \boldsymbol{\Phi}^T\boldsymbol{y} \implies (\boldsymbol{\Phi}^T\boldsymbol{\Phi} + \lambda\mathbf{1})\boldsymbol{w} = \boldsymbol{\Phi}^T\boldsymbol{y}$
    - Therefore l_2 regularization is equivalent to adding a small positive perturbation to the diagonal of $\boldsymbol{\Phi}^T\boldsymbol{\Phi}$
    - We saw this in a previous lecture – this also helps with ill-conditioning
    - If λ is sufficiently large we can avoid ill-conditioning completely
- Using SVD: $\boldsymbol{\Phi} = \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^T$
  - $((\boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^T)^T\boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^T + \lambda\mathbf{1})\boldsymbol{w} = (\boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^T)^T\boldsymbol{y}$
  - Simply to get $\boldsymbol{V}(\boldsymbol{\Sigma}^T\boldsymbol{\Sigma} + \lambda\mathbf{1})\boldsymbol{V}^T\boldsymbol{w} = \boldsymbol{V}\boldsymbol{\Sigma}^T\boldsymbol{U}^T\boldsymbol{y}$
  - Multiply each side by $\boldsymbol{V}^T$ to get $(\boldsymbol{\Sigma}^T\boldsymbol{\Sigma} + \lambda\mathbf{1})\boldsymbol{V}^T\boldsymbol{w} = \boldsymbol{\Sigma}^T\boldsymbol{U}^T\boldsymbol{y}$
  - Therefore $\boldsymbol{w} = \boldsymbol{V}(\boldsymbol{\Sigma}^T\boldsymbol{\Sigma} + \lambda\mathbf{1})^{-1}\boldsymbol{\Sigma}^T\boldsymbol{U}^T\boldsymbol{y}$
  - This can be rewritten as $\hat{\boldsymbol{w}}(\lambda) = \sum_{i=1}^{M} \boldsymbol{v}_i \dfrac{\sigma_i \boldsymbol{u}_i^T\boldsymbol{y}}{\sigma_i^2 + \lambda}$
    - $\boldsymbol{v}_i, \boldsymbol{u}_i$ are the ith columns of $\boldsymbol{V}$ and $\boldsymbol{U}$
    - If $0 \approx \sigma_i \ll \lambda$ this goes to 0
    - If $\sigma_i \gg \lambda$ this goes to the original unregularized solution
- The regularization has almost no impact on the contributions of large singular values but zeros out the contribution of smaller singular values

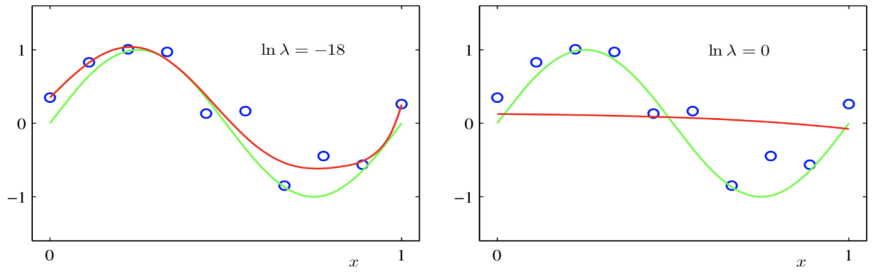

Figure 8: The same polynomial regression from above for $p = 9$, with different values of λ.

- λ is an important hyperparameter
  - Notice that with a reasonable value of λ we have a pretty good model even at $p = 9$
  - However if the regularization is too extreme, the model will underfit as the loss is too focused on minimizing $\|\boldsymbol{w}\|_2^2$
- To estimate λ we can again use ν-fold cross-validation just like we chose k for k-NN
  - If the training dataset is small we can use leave-one-out cross-validation (i.e. $\nu = 1$)
  - There are fast algorithms for calculating this
  - Use cross-validation to select the best value of λ, then retrain the model on all the data using this new value

Lecture 7, Feb 6, 2024

Dual Representations of GLMs

- Consider the loss function: $\mathcal{L}(\boldsymbol{w}) = \|\boldsymbol{y} - \boldsymbol{\phi w}\|_2^2 = \sum_{i=1}^{N} (\boldsymbol{w}^T \boldsymbol{\phi}(\boldsymbol{x}^{(i)}) - y^{(i)})^2 + \lambda \boldsymbol{w}^T \boldsymbol{w}$
  - Setting $\dfrac{\partial \mathcal{L}}{\partial \boldsymbol{w}} = 0 \implies 2 \sum_{i=1}^{N} \left(\boldsymbol{w}^T \boldsymbol{\phi}(\boldsymbol{x}^{(i)}) - y^{(i)} a \right) \boldsymbol{\phi}(\boldsymbol{x}^{(i)}) + 2\lambda \boldsymbol{w} = 0$
  - Then $\boldsymbol{w} = -\dfrac{1}{\lambda} \sum_{i=1}^{N} \left(\boldsymbol{w}^T \boldsymbol{\phi}(\boldsymbol{x}^{(i)}) - y^{(i)} a \right) \boldsymbol{\phi}(\boldsymbol{x}^{(i)}) = \sum_{i=1}^{N} \alpha_i \boldsymbol{\phi}(\boldsymbol{x}^{(i)}) = \boldsymbol{\Phi}^T \boldsymbol{\alpha}$
    - $\alpha_i = -\dfrac{1}{\lambda} (\boldsymbol{w}^T \boldsymbol{\phi}(\boldsymbol{x}^{(i)}) - y^{(i)})$ are the *dual variables* while $\boldsymbol{w}$ are the *primal variables*
- Substitute $\boldsymbol{w} = \boldsymbol{\Phi}^T \boldsymbol{\alpha}$ into the loss function: $\mathcal{L}(\boldsymbol{\alpha}) = \boldsymbol{\alpha}^T \boldsymbol{\Phi} \boldsymbol{\Phi}^T \boldsymbol{\Phi} \boldsymbol{\Phi}^T \boldsymbol{\alpha} - 2\boldsymbol{\alpha}^T \boldsymbol{\Phi} \boldsymbol{\Phi}^T \boldsymbol{y} + \boldsymbol{y}^T \boldsymbol{y} + \lambda \boldsymbol{\alpha}^T \boldsymbol{\Phi} \boldsymbol{\Phi}^T \boldsymbol{\alpha} = \boldsymbol{\alpha}^T \boldsymbol{K} \boldsymbol{K} \boldsymbol{\alpha} - 2\boldsymbol{\alpha}^T \boldsymbol{K} \boldsymbol{y} + \boldsymbol{y}^T \boldsymbol{y} + \lambda \boldsymbol{\alpha}^T \boldsymbol{K} \boldsymbol{\alpha}$
  - $\boldsymbol{K} = \boldsymbol{\Phi} \boldsymbol{\Phi}^T \in \mathbb{R}^{N \times N}$ is the *Gram matrix*, which is real and symmetric
  - The (i, j)th entry of $\boldsymbol{K}$ is given by $K_{ij} = \boldsymbol{\phi}(\boldsymbol{x}^{(i)})^T \boldsymbol{\phi}(\boldsymbol{x}^{(j)}) = k(\boldsymbol{x}^{(i)}, \boldsymbol{x}^{(j)})$
  - $k \colon \mathcal{X} \times \mathcal{X} \mapsto \mathbb{R}$ is the *kernel*
- Using the loss in terms of $\boldsymbol{K}$, we take $\nabla_{\boldsymbol{\alpha}} J = 0$ leads to $\boldsymbol{\alpha} = (\boldsymbol{K} + \lambda \boldsymbol{1})^{-1} \boldsymbol{y}$
  - With this solution for $\boldsymbol{\alpha}$ we have $\hat{f}(\boldsymbol{x}, \boldsymbol{w}) = \boldsymbol{\phi}(\boldsymbol{x})^T \boldsymbol{w} = \boldsymbol{\phi}(\boldsymbol{x})^T \boldsymbol{\Phi}^T (\boldsymbol{K} + \lambda \boldsymbol{1})^{-1} \boldsymbol{y}$
  - Note the ith entry of $\boldsymbol{\Phi} \boldsymbol{\phi}(\boldsymbol{x})$ is $\boldsymbol{\phi}(\boldsymbol{x}^{(i)}) \boldsymbol{\phi}(\boldsymbol{x}) = k(\boldsymbol{x}^{(i)}, \boldsymbol{x})$
  - Let $\boldsymbol{k}(\boldsymbol{x}) = \{ k(\boldsymbol{x}^{(1)}, \boldsymbol{x}), \ldots, k(\boldsymbol{x}^{(N)}, \boldsymbol{x}) \}^T \in \mathbb{R}^N$
- The model can then be rewritten as $\hat{f}(\boldsymbol{x}, \boldsymbol{w}) = \boldsymbol{k}(\boldsymbol{x})^T (\boldsymbol{K} + \lambda \boldsymbol{1})^{-1} \boldsymbol{y} = \sum_{i=1}^{N} \alpha_i k(\boldsymbol{x}, \boldsymbol{x}^{(i)})$
  - This is known as the *dual representation*
  - We've defined our model entirely in terms of the kernel; we don't actually need to evaluate the basis functions themselves, and only the inner products between the bases are needed
  - The choice of a kernel implicitly characterizes the feature space mapping ϕ
- Using the kernel is often much more efficient than using the basis functions explicitly
  - e.g. for the polynomial features, $k(\boldsymbol{x}, \boldsymbol{z}) = \boldsymbol{\phi}(\boldsymbol{x})^T \boldsymbol{\phi}(\boldsymbol{z}) = 1 + x_1 z_1 + x_2 z_2 + x_1 x_2 z_1 z_2 + \cdots + x_1 \ldots x_D z_1 \ldots z_D = \prod_{i=1}^{D} (1 + x_i z_i)$
  - The original features would need $\mathcal{O}(2^D)$ computation time, but using the kernel this is reduced to $\mathcal{O}(D)$ for the simple product
- The kernel can also be interpreted as a similarity metric, since it takes two points from $\mathcal{X}$ and returns a real scalar

Definition

The kernel trick: Any linear method that can be written in terms of dot products ${\boldsymbol{x}^{(i)}}^T \boldsymbol{x}^{(j)}$ can be *kernelized* by replacing ${\boldsymbol{x}^{(i)}}^T \boldsymbol{x}^{(j)} \to k(\boldsymbol{x}^{(i)}, \boldsymbol{x}^{(j)})$, which results in a nonlinear generalization of the linear method.

- This allows us to do kernel PCA, kernel SVM, etc
  - e.g. kernel k-NN
    - Distance computation in feature space is $\|\boldsymbol{\phi}(\boldsymbol{x}) - \boldsymbol{\phi}(\boldsymbol{z})\|_2^2 = \boldsymbol{\phi}(\boldsymbol{x})^T \boldsymbol{\phi}(\boldsymbol{x}) + \boldsymbol{\phi}(\boldsymbol{z})^T \boldsymbol{\phi}(\boldsymbol{z}) - 2\boldsymbol{\phi}(\boldsymbol{x})^T \boldsymbol{\phi}(\boldsymbol{z})$
    - Replace this by $k(\boldsymbol{x}, \boldsymbol{x}) + k(\boldsymbol{z}, \boldsymbol{z}) - 2k(\boldsymbol{x}, \boldsymbol{z})$ to kernelize it
- Even though we derived this result for the squared loss specifically, the *representer theorem* states that this kernel form of the model will always be able to minimize the loss

Kernel Selection

- The kernel function must define a dot product for some Hilbert space $\mathcal{F}$, which means it must be symmetric and positive semi-definite
  - Symmetry means $k(\boldsymbol{x}, \boldsymbol{z}) = k(\boldsymbol{z}, \boldsymbol{x})$
  - PSD means $\iint u(\boldsymbol{x})k(\boldsymbol{x}, \boldsymbol{z})u(\boldsymbol{z})\,\mathrm{d}\boldsymbol{x}\,\mathrm{d}\boldsymbol{z} \geq 0$ for all square integrable functions u
  - By extension this means:
    * $\boldsymbol{K}$ is positive semi-definite
    * Cauchy-Schwartz inequality: $k(\boldsymbol{z}, \boldsymbol{z}) \leq \sqrt{k(\boldsymbol{x}, \boldsymbol{x})k(\boldsymbol{z}, \boldsymbol{z})}$
    * Definiteness: $k(\boldsymbol{x}, \boldsymbol{x}) \geq 0$
  - This all makes sense intuitively if the kernel is interpreted as a distance metric
- Example kernels:
  - Linear: $k(\boldsymbol{x}, \boldsymbol{z}) = \boldsymbol{x}^T\boldsymbol{z}$
  - Polynomial: $k(\boldsymbol{x}, \boldsymbol{z}) = (1 + \boldsymbol{x}^T\boldsymbol{z})^n$
  - Isotropic Gaussian: $k(\boldsymbol{x}, \boldsymbol{z}) = \exp\left(-\frac{1}{\theta}\|\boldsymbol{x} - \boldsymbol{z}\|_2^2\right)$
    * $\theta > 0$ is a hyperparameter
  - Anisotropic Gaussian: $k(\boldsymbol{x}, \boldsymbol{z}) = \exp(-(\boldsymbol{x} - \boldsymbol{z})^T\boldsymbol{\Theta}^{-1}(\boldsymbol{x} - \boldsymbol{z}))$
    * $\boldsymbol{\Theta} \in \mathbb{R}^{D\times D}$ is symmetric positive definite and a hyperparameter
- We can go from kernels back to features, e.g. for the polynomial kernel:
  - $k(\boldsymbol{x}, \boldsymbol{z}) = (1 + x_1z_1 + x_2z_2 + \cdots + x_Dz_D)^n$
  - For $D = 2$ and $n = 2$, $k(\boldsymbol{x}, \boldsymbol{z}) = 1 + x_1^2z_1^2 + x_2^2z_2^2 + 2x_1z_1 + 2x_2z_2 + 2x_1z_2x_2z_2$
  - Therefore $\boldsymbol{\phi}(\boldsymbol{x}) = \begin{bmatrix}1 & x_1^2 & x_2^2 & \sqrt{2}x_1 & \sqrt{2}x_2 & \sqrt{2}x_1x_2\end{bmatrix}$
- The feature vector can even be infinite dimensional, e.g. for the Gaussian kernel:
  - For $D = 1, \theta = 1$, $k(x, z) = \exp(-(x - z)^2)$
    $$= \exp(-x^2)\exp(-z^2)\exp(2xz)$$
    $$= \exp(-x^2)\exp(-z^2)\sum_{k=0}^{\infty}\frac{2^kx^kz^k}{k!}$$
  - Therefore $\boldsymbol{\phi}(x) = \left[\exp(-x^2) \quad \sqrt{\frac{2^1}{1!}}x^1\exp(-x^2) \quad \sqrt{\frac{2^2}{2!}}x^2\exp(-x^2) \quad \ldots\right]$
- To select the kernel, we can use prior knowledge of the target function
  - If the target function is known to be smooth (i.e. differentiable k times) then we can use a kernel that also has the same degree of smoothness
  - If the function is finitely smooth, use the Gaussian or another C^∞ kernel
  - If the function is periodic we can use a periodic kernel
  - Plenty of literate exists in this area
- *Radial basis functions* (RBFs) are kernels that are translation invariant, i.e. their value only depends on the distance between the features
  - $k(\boldsymbol{x}^{(i)}, \boldsymbol{x}^{(j)}) = k(\|\boldsymbol{x}^{(i)} - \boldsymbol{x}^{(j)}\|) = k(r)$
  - Examples of RBF kernels:
    * Gaussian: $k(r) = e^{-\frac{r^2}{\theta}}$
    * Multiquadratic: $k(r) = \sqrt{1 + \frac{r^2}{\theta}}$
    * Inverse multiquadratic: $k(r) = \frac{1}{\sqrt{1 + \frac{r^2}{\theta}}}$
    * Matern kernels: a family including
      - C^0: $\exp\left(-\frac{r}{\theta}\right)$
      - C^2: $\frac{1}{1 + \frac{r}{\theta}}\exp\left(-\frac{r}{\theta}\right)$
      - C^4: $\left(3 + 3\frac{r}{\theta} + \left(\frac{r}{\theta}\right)^2\right)\exp\left(-\frac{r}{\theta}\right)$

  - All the above kernels have θ has a hyperparameter; this is the shape parameter, where larger values spread out the function and gives a higher value for larger values of r

Sparsity

- The regression model is $\hat{f}(\boldsymbol{x}, \boldsymbol{\alpha}) = \sum_{i=1}^{N} \alpha_i k(\boldsymbol{x}, \boldsymbol{x}^{(i)})$ where $\boldsymbol{\alpha} = (\boldsymbol{K} + \lambda \boldsymbol{1})^{-1} \boldsymbol{y}$
- This can be interpreted as a GLM constructed using the N basis functions $k(\boldsymbol{x}, \boldsymbol{x}^{(1)}), \ldots, k(\boldsymbol{x}, \boldsymbol{x}^{(N)})$
  - We have one basis function per data point, so this is a *dense* regression model
- Note that when $\lambda = 0$, since we have N basis functions, we will match our N training points exactly
  - This can be useful if we know that there is no noise in the training data
  - When $\lambda = 0$, $\boldsymbol{K}$ is guaranteed to be non-singular if and only if the training data points are unique
- When $\lambda > 0$, $\boldsymbol{K} + \lambda \boldsymbol{1}$ is symmetric positive definite, so we can compute the Cholesky factorization without worrying about singularities
  - Since we never formed normal equations, we never squared the condition number, so this is stable
- Computing this will take $\mathcal{O}(N^2)$ memory and $\mathcal{O}(N^3)$ time, which makes it very difficult to scale up
  - We can improve this by choosing only a subset of the basis functions, which gives us a *sparse* regression model
  - Alternatively, we can use k-means clustering to extract a set of representative points
    * Then the model is $\hat{f}(\boldsymbol{x}, \boldsymbol{\alpha}) = \sum_{i=1}^{M} \alpha_i k(\boldsymbol{x}, \boldsymbol{z}^{(i)})$ and $\boldsymbol{\alpha}$ is computed with the $\boldsymbol{z}$ vectors
    * This also reduces inference cost
- Sparsity is generally a good idea because:
  - Reduction in computational and inference cost
  - Reduction in memory usage
  - Makes models more interpretable
  - Prevents overfitting
- *Orthogonal Marching Pursuit*: a greedy algorithm for sparse regression
  - Procedure:
    * Set $k = 0$ and let $\mathcal{D}_\phi = \{ \phi_1, \ldots, \phi_M \}$ be a dictionary of basis functions
    * Initialize $\mathcal{I}_s^{(k)}$, the set of selected basis functions, and $\mathcal{I}_c^{(k)}$, the set of candidate basis functions
    * Initialize $\boldsymbol{r}^{(0)} = \boldsymbol{y}$ as the residual, or training error vector
    * While $\|\boldsymbol{r}^{(k)}\|_2 > \epsilon$, do:
      - $k \leftarrow k + 1$
      - Pick $i_k = \underset{i \in \mathcal{I}_c^{(k-1)}}{\operatorname{argmax}} J(\phi_i)$
        - The metric is $J(\phi_i) = \dfrac{(\boldsymbol{\Phi}_i^T \boldsymbol{r}^{(k)})^2}{\boldsymbol{\Phi}_i^T \boldsymbol{\Phi}_i}$ where $\boldsymbol{\Phi}_i$ is the ith column of $\boldsymbol{\Phi}$
        - This is an approximation of the reduction in training error as a result of choosing the ith basis function
        - Think of this as checking how much the ith basis function is in the direction of the residual error
      - Add selected basis function index to $\mathcal{I}_s^{(k)}$ and remove it from $\mathcal{I}_c^{(k)}$
      - Solve $\boldsymbol{\phi}^{(k)} \boldsymbol{w}^{(k)} \approx \boldsymbol{y}$ for the weights
        - Note $\boldsymbol{w}^{(k)} \in \mathbb{R}^k$ since in this iteration we have k basis functions
        - $\boldsymbol{\Phi}^{(k)}$ has k columns corresponding to the basis functions
      - Update the residual by $\boldsymbol{r}^{(k)} = \boldsymbol{y} - \boldsymbol{\Phi}^{(k)} \boldsymbol{w}^{(k)}$
    * The final sparse model is $\sum_{i \in \mathcal{I}_s^{(k)}} w_i \phi_i(\boldsymbol{x})$
  - Updating the weights in each iteration can be done using incremental QR factorization to save time
  - The parameter ϵ can be chosen via cross-validation, or other model selection criteria
- For GLMs, if minimizing the least squares error with l_2 regularization, we can find a more efficient

method to calculate the leave-one-out error
  - Let $\boldsymbol{A} = \boldsymbol{K}(\boldsymbol{K} + \lambda \mathbf{1})^{-1} = \boldsymbol{\Phi}(\boldsymbol{\Phi}^T\boldsymbol{\Phi})^{-1}\boldsymbol{\Phi}^T$
  - Let $\hat{f}^{\setminus i}$ denote the model constructed by leaving out the ith training point
  - Then $y^{(i)} - \hat{f}^{\setminus i}(\boldsymbol{x}^{(i)}) = \dfrac{y^{(i)} - \hat{f}(\boldsymbol{x}^{(i)})}{1 - A_{ii}}$
  - Therefore the total leave-one-out error is $\dfrac{1}{N}\displaystyle\sum_{i=1}^{N}\left(\frac{y^{(i)} - \hat{f}(\boldsymbol{x}^{(i)})}{1 - A_{ii}}\right)^2$
    * This is a function of λ, the regularization parameter; using this we can estimate the optimal value of λ
  - This means we don't have to train the model N times for each data point we leave out, making this much more efficient
- Using l_1 regularization can also give models that are more sparse and easy to interpret
  - However with l_1 regularization we can no longer use linear algebra to obtain a closed form solution
  - Optimization algorithms need to be used in this case
- In summary:
  - If M is high or possibly infinite, use kernel methods
  - If N is high, use explicit basis functions
  - When both are high, options include greedy algorithms for sparsity, clustering, scholastic algorithms, etc

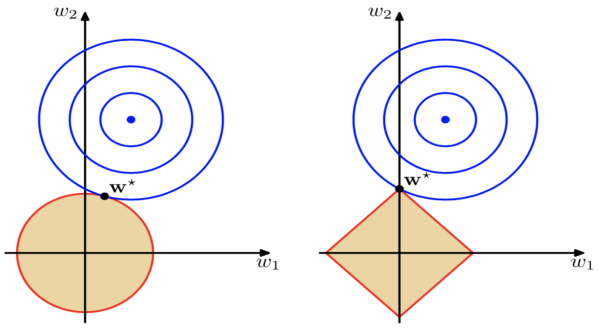

Figure 9: Comparison of l_1 vs l_2 regularization.

Lecture 8, Feb 13, 2024

Probability Density Estimation

- Previously we've considered learning problems using a loss function perspective; now we would like to consider a statistical perspective
- We begin by looking at density estimation problems
- Given a dataset $\mathcal{D} = \{\, \boldsymbol{x}^{(i)} \,\}_{i=1}^{N}$, we would like to determine the distribution generating this data
  - Assume θ is a hypothesis class that parametrizes the density function
  - $\mathcal{P}_{\boldsymbol{\theta}} = \{\, p(\boldsymbol{x} \mid \boldsymbol{\theta}) \mid \boldsymbol{\theta} \in \Gamma \,\}$

Maximum Likelihood Estimation

- In ML we aim to find the parameter value $\hat{\boldsymbol{\theta}}$ for which the observed data has the highest probability/density of occurring
- $\hat{\boldsymbol{\theta}}_{ML} = \underset{\boldsymbol{\theta}\in\Gamma}{\operatorname{argmax}}\, p(\boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}, \ldots, \boldsymbol{x}^{(N)} | \boldsymbol{\theta})$
  - The term $p(\boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}, \ldots, \boldsymbol{x}^{(N)} | \boldsymbol{\theta})$ is known as the *likelihood function*
- We often assume that the data is *independently and identically distributed* (IID), which allows us to decompose the likelihood into a product

- Assuming IID, $p(\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(N)}|\boldsymbol{\theta}) = \prod_{i=1}^{N} p(\boldsymbol{x}^{(i)}|\boldsymbol{\theta})$
- Maximizing the likelihood is the same as maximizing the log of the likelihood function; this is referred to as *log-likelihood*
  - $\log p(\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(N)}|\boldsymbol{\theta}) = \sum_{i=1}^{N} \log\left(p(\boldsymbol{x}^{(i)}|\boldsymbol{\theta})\right)$
  - Practically, using log-likelihood prevents instability due to underflow (multiplying many very small numbers)
- To solve for the ML estimator we simply differentiate and set the derivative to zero
  - $\sum_{i=1}^{N} \frac{\nabla_{\boldsymbol{\theta}} p(\boldsymbol{x}^{(i)}|\boldsymbol{\theta})}{p(\boldsymbol{x}^{(i)}|\boldsymbol{\theta})} = 0$
  - In special cases we may obtain analytical solutions using linear algebra, but in general we may have to use nonlinear optimization methods
- MLE can also be used to perform regression
  - Consider observations as $y(\boldsymbol{x}) = \hat{f}(\boldsymbol{x}, \boldsymbol{w}) + \epsilon$ where $\epsilon \in \mathcal{N}(0, \sigma^2)$
  - $\hat{f}(\boldsymbol{x}, \boldsymbol{w})$ is the underlying function; we add some noise ϵ to get the measurement
  - $p(y|\boldsymbol{x}, \boldsymbol{w}, \sigma^2) = \mathcal{N}(y|\hat{f}(\boldsymbol{x}, \boldsymbol{w}), \sigma^2) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(y - \hat{f}(\boldsymbol{x}, \boldsymbol{w}))^2}{2\sigma^2}\right)$
  - The goal is to estimate the parameters $\boldsymbol{w}$
  - $p(\boldsymbol{y}|\boldsymbol{X}, \boldsymbol{w}, \sigma^2) = \prod_{i=1}^{N} \mathcal{N}(y^{(i)}|\hat{f}(\boldsymbol{x}^{(i)}, \boldsymbol{w}), \sigma^2)$
    $= \left(\frac{1}{2\pi\sigma^2}\right)^{\frac{N}{2}} \exp\left(-\frac{1}{2\sigma^2} \sum_{i=1}^{N} (y^{(i)} - \hat{f}(\boldsymbol{x}^{(i)}, \boldsymbol{w}))^2\right)$
    * $\boldsymbol{y}$ is a column vector of all the observations while $\boldsymbol{X}$ has each of the $\boldsymbol{x}^{(i)}$ vectors as its rows
  - The negative log-likelihood is $\frac{1}{2\sigma^2} \sum_{i=1}^{N} (y^{(i)} - \hat{f}(\boldsymbol{x}^{(i)}, \boldsymbol{w}))^2 + N \log \sigma + \frac{N}{2} \log 2\pi$
    * Notice that the first term is just the l_2 loss function
    * The other two terms are constant in $\boldsymbol{w}$, so we see that MLE is equivalent to using a l_2 loss function when the data is IID Gaussian
  - This also lets us estimate the variance of the noise by differentiating the log-likelihood wrt σ^2 and solve for zero
    * $-\frac{1}{2\sigma^3} \sum_{i=1}^{N} (y^{(i)} - \hat{f}(\boldsymbol{x}^{(i)}, \boldsymbol{w}))^2 + \frac{N}{\sigma} = 0$
    * $\sigma^2 = \frac{1}{N} \sum_{i=1}^{N} (y^{(i)} - \hat{f}(\boldsymbol{x}^{(i)}, \boldsymbol{w}))^2$
- For regression, we get a constant variance, so the error bars are constant size throughout the data
  - This is not reasonable since we expect the error bars to be smaller where we have more data points
  - Near the middle where we have more data, we should get smaller error while near the edges we should expect more
- Example exercise: assume IID Laplacian noise, formulate an optimization problem and solve for $\hat{\boldsymbol{w}}_{ML}$
  - The Laplace distribution is given by $\text{Lap}(\epsilon|\mu, b) = \frac{1}{2b} e^{-\frac{|\epsilon - \mu|}{b}}$
    * Mean, variance of $2b^2$
  - Get the joint likelihood: $p(\boldsymbol{y}|\boldsymbol{X}, \boldsymbol{w}, 2b^2) = \prod_{i=1}^{N} \text{Lap}(y^{(i)}|\hat{f}(\boldsymbol{x}, \boldsymbol{w}), 2b^2)$
    $= \left(\frac{1}{2b}\right)^N \exp\left(-\sum_{i=1}^{N} \frac{|y^{(i)} - \hat{f}(\boldsymbol{x}^{(i)}, \boldsymbol{w})|}{b}\right)$

- Negative log likelihood: $-\log(p(\boldsymbol{y}|\boldsymbol{X}, \boldsymbol{w}, 2b^2)) = N\log 2b - \frac{1}{b}\sum_{i=1}^{N}|y^{(i)} - \hat{f}(\boldsymbol{x}^{(i)}, \boldsymbol{w})|$
- Optimization problem: $\hat{\boldsymbol{w}}_{ML} = \underset{\boldsymbol{w}}{\operatorname{argmin}}\, N\log 2b - \frac{1}{b}\sum_{i=1}^{N}|y^{(i)} - \hat{f}(\boldsymbol{x}^{(i)}, \boldsymbol{w})|$

$$= \underset{\boldsymbol{w}}{\operatorname{argmin}} \sum_{i=1}^{N}|y^{(i)} - \hat{f}(\boldsymbol{x}^{(i)}, \boldsymbol{w})|$$

  * Notice that this is akin to minimizing an l_1 loss function
  * This is no longer solvable analytically

- Example: Given measurements $x^{(1)} = 1, x^{(2)} = 2, x^{(3)} = 3, x^{(4)} = 3, x^{(5)} = 4$ distributed according to an exponential distribution $\rho e^{-\rho x}$, find the MLE of ρ
  - $p(x^{(1)}, \ldots, x^{(5)}|\rho) = \prod_{i=1}^{5} \rho e^{-\rho x^{(i)}} = \rho^5 e^{-13\rho}$
  - NLL: $-\log(p(x^{(1)}, \ldots, x^{(5)}|\rho)) = -5\log\rho + 13\rho$
  - Differentiate: $-\frac{5}{\rho} + 13 = 0 \implies \hat{\rho}_{\text{ML}} = \frac{5}{13}$

Maximum a Posteriori (MAP) Estimation

- In MAP estimation, we aim to find the parameter value that is most likely to occur given the data and a prior distribution of the parameter value
- $p(\boldsymbol{\theta}|\mathcal{D}) = \frac{p(\mathcal{D}|\boldsymbol{\theta})p(\boldsymbol{\theta})}{p(\mathcal{D})}$
  - The evidence/marginal likelihood in the denominator is often hard to compute
  - However for MAP we don't need to compute it since it does not depend on $\boldsymbol{\theta}$
- $\hat{\boldsymbol{\theta}}_{MAP} = \underset{\boldsymbol{\theta}}{\operatorname{argmax}}\, p(\mathcal{D}|\boldsymbol{\theta})p(\boldsymbol{\theta})$
  - When the prior is uniform, this is equivalent to MLE
- Consider regression with a Gaussian prior and noise:
  - $p(\boldsymbol{w}|\alpha) = \mathcal{N}(\boldsymbol{w}|\boldsymbol{0}, \alpha\boldsymbol{1}) = \left(\frac{1}{\sqrt{2\pi\alpha}}\right)^M \exp\left(-\frac{\boldsymbol{w}^T\boldsymbol{w}}{2\alpha}\right)$
  - $p(\boldsymbol{y}|\boldsymbol{x}, \boldsymbol{w}, \sigma^2) = \prod_{i=1}^{N} \mathcal{N}(y^{(i)}|\hat{f}(\boldsymbol{x}^{(i)}, \boldsymbol{w}), \sigma^2)$
  - The posterior is proportional to the product of the two
  - $\hat{\boldsymbol{w}}_{MAP} = \underset{\boldsymbol{w}}{\operatorname{argmin}} \frac{1}{2\sigma^2}\sum_{i=1}^{N}(\hat{f}(\boldsymbol{x}^{(i)}, \boldsymbol{w}) - y^{(i)})^2 + \frac{1}{2\alpha}\boldsymbol{w}^T\boldsymbol{w}$

$$= \underset{\boldsymbol{w}}{\operatorname{argmin}} \frac{1}{2}\sum_{i=1}^{N}(\hat{f}(\boldsymbol{x}^{(i)}, \boldsymbol{w}) - y^{(i)})^2 + \frac{\sigma^2}{2\alpha}\boldsymbol{w}^T\boldsymbol{w}$$

    * Notice that the first term is the l_2 loss function while the second is l_2 regularization
    * MAP estimation is equivalent to using an l_2 loss function with l_2 regularization, assuming a zero-mean Gaussian prior and IID Gaussian noise distribution
    * In the statistical perspective we are saying that we believe the weights are small prior to seeing the data; in the loss function perspective we are forcing the weights to be small

- Now consider a Laplace prior: $p(\boldsymbol{w}|\alpha) = \prod_{i=1}^{M} \text{Lap}(w_i|0, \alpha) = \left(\frac{1}{2\alpha}\right)^M \exp\left(-\frac{1}{\alpha}\sum_{i=1}^{M}|w_i|\right)$
  - Likelihood: $p(\boldsymbol{y}|\boldsymbol{x}, \boldsymbol{w}, \sigma^2) = \prod_{i=1}^{N} \mathcal{N}(y^{(i)}|\hat{f}(\boldsymbol{x}^{(i)}, \boldsymbol{w}), \sigma^2) = \left(\frac{1}{2\pi\sigma^2}\right)^{\frac{N}{2}}$
  - Negative log likelihood of posterior: $-\log(p(\boldsymbol{y}|\boldsymbol{X}, \boldsymbol{w}, \sigma^2)) - \log(p(\boldsymbol{w}|\alpha))$

    * $\frac{1}{2\sigma^2}\sum_{i=1}^{N}(y^{(i)} - \hat{f}(\boldsymbol{x}^{(i)}, \boldsymbol{w}))^2 - \frac{1}{\alpha}\sum_{i=1}^{M}|w_i|$
  - We see again that this is equivalent to using l_2 loss with l_1 regularization with $\lambda = \frac{2\sigma^2}{\alpha}$

Frequentist vs. Bayesian Estimation

- In the frequentist approach, we assume that there exists a true fixed parameter value θ^*
  - We can get error bars by considering the distribution of possible datasets given this parameter value
  - However the error bars are not very good because they are independent of the inputs
  - Both MLE and MAP are frequentist methods since they give point estimates
- In the Bayesian approach, we use a single observation dataset to estimate the entire posterior distribution
  - This gives us both the mean as an estimate and a measure of uncertainty
  - Enables leveraging of priors

Lecture 9, Feb 16, 2024

Unconstrained Optimization

- Given $f(\boldsymbol{\theta})$ where $\boldsymbol{\theta} \in \mathbb{R}^n$ and $f\colon \mathbb{R}^n \mapsto \mathbb{R}$, we want to find $\boldsymbol{\theta}^* = \underset{\boldsymbol{\theta}}{\operatorname{argmin}}\, f(\boldsymbol{\theta})$
- Let $\boldsymbol{\theta}^*$ be a local minimum of f, then $f(\boldsymbol{\theta}^* + \epsilon\boldsymbol{p}) \geq f(\boldsymbol{\theta}^*)$ for any arbitrary $\boldsymbol{p}$ and small ϵ
  - $f(\boldsymbol{\theta}^* + \epsilon\boldsymbol{p}) = f(\boldsymbol{\theta}^*) + \epsilon\boldsymbol{p}^T\boldsymbol{g}(\boldsymbol{\theta}^*) + \frac{1}{2}\epsilon^2\boldsymbol{p}^T\boldsymbol{H}(\boldsymbol{\theta}^*)\boldsymbol{p} + \mathcal{O}(\epsilon^3)$ where $\boldsymbol{g}(\boldsymbol{\theta})$ is the gradient and $\boldsymbol{H}(\boldsymbol{\theta})$ is the Hessian
  - The middle two terms must be greater than or equal to zero due to the local minimum condition
  - This means that $\boldsymbol{g}(\boldsymbol{\theta}^*) = \boldsymbol{0}$, and $\boldsymbol{H}(\boldsymbol{\theta}^*)$ is symmetric positive definite, since $\boldsymbol{p}$ is arbitrary
- KKT conditions:
  - The first order necessary optimality condition states that the gradient must be zero at the minimum
  - The second order necessary optimality condition states that the Hessian must be positive semi-definite
  - If both the gradient is zero and the Hessian is positive definite, then we have sufficient conditions for a minimum

Gradient-Based Unconstrained Numerical Optimization

- Gradient-based algorithms are based on the following template:
  - Start with $k = 0$ and an initial guess $\boldsymbol{\theta}_0$
  - In each iteration:
    * Test for convergence; if we have converged, stop and take $\boldsymbol{\theta}_k$ as the solution; if not, continue
    * Compute the search direction $\boldsymbol{p}_k$
    * Compute the step length $\alpha_k > 0$ s.t. $f(\boldsymbol{\theta}_k + \alpha_k\boldsymbol{p}_k) < f(\boldsymbol{\theta}_k)$
    * Take $\theta_{k=1} \leftarrow \boldsymbol{\theta}_k + \alpha_k\boldsymbol{p}_k$ and $k \leftarrow k + 1$
- To obtain a valid search direction, we need $\boldsymbol{p}_k^T\boldsymbol{g}_k < 0$, i.e. the step and the gradient have to point in opposite directions
  - Take $\boldsymbol{p}_k = -\boldsymbol{B}\boldsymbol{g}_k$ for some symmetric positive definite $\boldsymbol{B}$
  - Possible choices for $\boldsymbol{B}$:
    * Steepest descent: $\boldsymbol{B} = \boldsymbol{1}$
      - The search direction is directly opposite to the gradient
    * Newton's method: $\boldsymbol{B} = \boldsymbol{H}_k^{-1}$
    * Quasi-Newton methods: $\boldsymbol{B} \approx \boldsymbol{H}_k^{-1}$
      - For computational reasons, these methods take an approximation of the inverse Hessian
- Computing the appropriate α_k is a tradeoff between reducing function evaluations (i.e. getting to the goal with fewer steps) and computational cost at each step

  - One technique is to take a number of α_ks and stop at the first one that meets some condition
  - Armijo sufficient decrease condition: $f(\boldsymbol{\theta}_k + \alpha_k \boldsymbol{p}_k) \leq f(\boldsymbol{\theta}_k) + \mu_1 \alpha_k \boldsymbol{g}_k^T \boldsymbol{p}_k$
    * The constant μ_1 is typically chosen in the range of 10^{-4}
  - Backtracking line search:
    * Choose starting step length (between 0 and 1)
    * Check if Armijo condition is satisfied, and if so use the current step length
    * If not, $\alpha \leftarrow \rho\alpha$ (typically $\rho \in [0.1, 0.5]$) and check again
- Steepest descent algorithm:
  - Select initial guess $\boldsymbol{\theta}_0$, gradient tolerance ϵ_g, absolute tolerance ϵ_a, relative tolerance ϵ_r
  - If $\|\boldsymbol{g}(\boldsymbol{\theta}_k)\|_2 \leq \epsilon_g$ then stop
  - Set $\boldsymbol{p}_k = -\dfrac{\boldsymbol{g}(\boldsymbol{\theta}_k)}{\|\boldsymbol{g}(\boldsymbol{\theta}_k)\|_2}$
  - Find α_k such that $f(\boldsymbol{\theta}_k + \alpha \boldsymbol{p}_k)$ satisfies the sufficient decrease conditions
  - Update as $\boldsymbol{\theta}_{k+1} \leftarrow \boldsymbol{\theta}_k + \alpha_k \boldsymbol{p}_k$
  - Evaluate $f(\boldsymbol{\theta}_{k+1})$; if $|f(\boldsymbol{\theta}_{k+1}) - f(\boldsymbol{\theta}_k)| \leq \epsilon_a + \epsilon_r |f(\boldsymbol{\theta}_k)|$ for two successive iterations, stop
    * In this case our algorithm has gotten stuck
- For steepest descent, for all k, we can show that $\boldsymbol{p}_{k+1}$ is orthogonal to $\boldsymbol{p}_k$
  - $\dfrac{\partial f(\boldsymbol{\theta}_{k+1})}{\partial \alpha} = 0 \implies \vec{\nabla}^T f(\boldsymbol{\theta}_{k+1}) \boldsymbol{p}_k = 0 \implies \boldsymbol{g}(\boldsymbol{\theta}_{k-1})^T \boldsymbol{g}(\boldsymbol{\theta}_k) = 0$
  - This means we're always zig-zagging at each iteration
  - This is inefficient (high number of iterations needed), but it is guaranteed to converge
  - Convergence rate is linear: $\lim\limits_{k \to \infty} \dfrac{|f(\boldsymbol{\theta}_{k-1}) - f(\boldsymbol{\theta}^*)|}{|f(\boldsymbol{\theta}_k) - f(\boldsymbol{\theta}^*)|} = K$
- Conjugate gradient method (nonlinear, first order, i.e. only uses the gradient):
  - $\boldsymbol{p}_0 = -\dfrac{\boldsymbol{g}(\boldsymbol{\theta}_0)}{\|\boldsymbol{g}(\boldsymbol{\theta}_0)\|_2}$ for the first step
  - For all other steps $\boldsymbol{p}_k = -\boldsymbol{g}_k + \beta_k \boldsymbol{p}_{k-1}$
    * Fletcher-Reeves: $\beta_k = \dfrac{\boldsymbol{g}_k^T \boldsymbol{g}_k}{\boldsymbol{g}_{k-1}^T \boldsymbol{g}_{k-1}}$
    * Polak-Ribieve: $\beta_k = \dfrac{\boldsymbol{g}_k^T (\boldsymbol{g}_k - \boldsymbol{g}_{k-1})}{\boldsymbol{g}_{k-1}^T \boldsymbol{g}_{k-1}}$
  - Since we're using knowledge from previous gradients, this does not zig zag as much as steepest descent
- Newton's method (second order, i.e. also uses the Hessian):
  - Quadratic convergence: $\lim\limits_{k \to \infty} \dfrac{|f(\boldsymbol{\theta}_{k-1}) - f(\boldsymbol{\theta}^*)|}{|f(\boldsymbol{\theta}_k) - f(\boldsymbol{\theta}^*)|^2} = K > 0$
  - Use $\boldsymbol{p}_k = \boldsymbol{H}_k^{-1} \boldsymbol{g}_k$
    * Note that this step direction is only valid if $\boldsymbol{H}_k$ is positive definite
    * We can check the dot product of $\boldsymbol{p}_k$ with $\boldsymbol{g}_k$ at each step, and if this is invalid (i.e. greater than 0) we simply go in the opposite direction
- Newton's method requires computation of the inverse Hessian, which can be very inefficient; for computational reasons, we use quasi-Newton methods which approximate the inverse Hessian
  - These don't make use of the Hessian but can still get better than linear convergence
  - Iteratively update $\boldsymbol{B}_{k+1}^{-1} = \boldsymbol{B}_k^{-1} + \Delta \hat{\boldsymbol{B}}_k$
  - $\Delta \hat{\boldsymbol{B}}_k$ depends on the specific method

Lecture 10, Feb 27, 2024

Quasi-Newton Methods (Symmetric Rank 1 (SR1))

- Recall that for quasi-Newton methods, since computing the inverse Hessian is expensive, we use approximations to speed up computation
- Idea: use an iterative update routine to approximate the inverse Hessian

- Start with a quadratic approximation of the objective function $f(\boldsymbol{\theta})$ at the current iteration, $m_k(\boldsymbol{\theta})$
- $m_k(\boldsymbol{\theta}) = f(\boldsymbol{\theta}_k) + \vec{\nabla}^T f(\boldsymbol{\theta}_k)(\boldsymbol{\theta} - \boldsymbol{\theta}_k) + \frac{1}{2}(\boldsymbol{\theta} - \boldsymbol{\theta}_k)^T \boldsymbol{B}_k(\boldsymbol{\theta} - \boldsymbol{\theta}_k)$
  - $\boldsymbol{B}_k \in \mathbb{R}^{n \times n}$ is an approximation of the inverse Hessian
  - When $\boldsymbol{\theta} = \boldsymbol{\theta}_k$ we have $m_k = f$ and $\vec{\nabla} m_k = \vec{\nabla} f$
  - These conditions are known as the *zero* and *first-order consistency conditions*
- The minimum of the quadratic model can be obtained by differentiating and setting to zero as usual
  - $\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - \boldsymbol{B}_k^{-1} \vec{\nabla} f(\boldsymbol{\theta}_k)$
- Using the new minimum, we construct a new quadratic approximation $m_{k+1}(\boldsymbol{\theta})$ using the same formula and the new approximate Hessian $\boldsymbol{B}_{k+1}$
- To update the approximate Hessian, we impose the constraint that $m_{k+1}(\boldsymbol{\theta})$ matches the gradient of $f(\boldsymbol{\theta})$ at both $\boldsymbol{\theta}_k$ and $\boldsymbol{\theta}_{k+1}$
  - We want $\vec{\nabla} m_{k+1}(\boldsymbol{\theta}_{k+1}) = \vec{\nabla} f(\boldsymbol{\theta}_{k+1}) + \boldsymbol{B}_{k+1}(\boldsymbol{\theta}_k - \boldsymbol{\theta}_{k+1}) = \vec{\nabla} f(\boldsymbol{\theta}_k)$
  - Rearrange to get $\boldsymbol{B}_{k+1}(\boldsymbol{\theta}_k - \boldsymbol{\theta}_{k+1}) = \vec{\nabla} f(\boldsymbol{\theta}_k) - \vec{\nabla} f(\boldsymbol{\theta}_{k+1})$
    - This is known as the *secant equation*
- Since the Hessian is SPD, we want $\boldsymbol{B}_{k+1}$ to also be SPD
  - Symmetry gives us $\frac{1}{2}n(n+1)$ independent entries, but the secant equation only gives a system of n equations
  - To obtain a unique solution, we impose the constraint that $\boldsymbol{B}_{k+1}$ should be closest to $\boldsymbol{B}_k$
  - Therefore we use the update formula: $\boldsymbol{B}_{k+1} = \boldsymbol{B}_k + \boldsymbol{u}\boldsymbol{u}^T$
    - $\boldsymbol{u}\boldsymbol{u}^T$ is the "symmetric rank 1" matrix
    - This guarantees that $\boldsymbol{B}_{k+1}$ is close to $\boldsymbol{B}_k$ in terms of rank
- Let $\boldsymbol{s}_k = \boldsymbol{\theta}_{k+1} - \boldsymbol{\theta}_k, \boldsymbol{y}_k = \vec{\nabla} f(\boldsymbol{\theta}_{k+1}) - \vec{\nabla} f(\boldsymbol{\theta}_k)$
  - Rewrite the secant equation as $\boldsymbol{B}_{k+1}\boldsymbol{s}_k = \boldsymbol{y}_k$
  - Plugging in the SR1 update, $\boldsymbol{B}_k\boldsymbol{s}_k + \boldsymbol{u}\boldsymbol{u}^T\boldsymbol{s}_k = \boldsymbol{y}_k \implies \boldsymbol{u}(\boldsymbol{u}^T\boldsymbol{s}_k) = \boldsymbol{y}_k - \boldsymbol{B}_k\boldsymbol{s}_k$
    - This means $\boldsymbol{u} = \gamma(\boldsymbol{y}_k - \boldsymbol{B}_k\boldsymbol{s}_k)$ where $\gamma = \boldsymbol{u}^T\boldsymbol{s}_k$ is a scalar
  - Plug this back in: $\gamma^2(\boldsymbol{y}_k - \boldsymbol{B}_k\boldsymbol{s}_k)(\boldsymbol{s}_k^T(\boldsymbol{y}_k - \boldsymbol{B}_k\boldsymbol{s}_k)) = \boldsymbol{y}_k - \boldsymbol{B}_k\boldsymbol{s}_k$
  - $\gamma^2 = \dfrac{1}{\boldsymbol{s}_k^T(\boldsymbol{y}_k - \boldsymbol{B}_k\boldsymbol{s}_k)}$
- The final update formula is $\boldsymbol{B}_{k+1} = \boldsymbol{B}_k + \dfrac{(\boldsymbol{y}_k - \boldsymbol{B}_k\boldsymbol{s}_k)(\boldsymbol{y}_k - \boldsymbol{B}_k\boldsymbol{s}_k)^T}{(\boldsymbol{y}_k - \boldsymbol{B}_k\boldsymbol{s}_k)^T\boldsymbol{s}_k}$
  - However, this only gives the approximate Hessian and not its inverse (inverting at each iteration would be too expensive)

Theorem

Sherman-Morrison-Woodbury (SMW) Formula: Given $\boldsymbol{A} \in \mathbb{R}^{n \times n}, \boldsymbol{u}, \boldsymbol{v} \in \mathbb{R}^{n \times p}$, then

$$(\boldsymbol{A} + \boldsymbol{u}\boldsymbol{v}^T)^{-1} = \boldsymbol{A}^{-1} - \boldsymbol{A}^{-1}\boldsymbol{u}(\boldsymbol{1} + \boldsymbol{v}^T\boldsymbol{A}^{-1}\boldsymbol{u})^{-1}\boldsymbol{v}^T\boldsymbol{A}^{-1}$$

- Using the SMW formula: $\boldsymbol{B}_{k+1}^{-1} = \boldsymbol{B}_k^{-1} + \dfrac{(\boldsymbol{s}_k - \boldsymbol{B}_k^{-1}\boldsymbol{y}_k)(\boldsymbol{s}_k - \boldsymbol{B}_k^{-1}\boldsymbol{y}_k)^T}{(\boldsymbol{s}_k - \boldsymbol{B}_k^{-1}\boldsymbol{y}_k)^T\boldsymbol{y}_k}$
  - This gives a cost of $\mathcal{O}(n^2)$ (compared to $\mathcal{O}(n^3)$ for matrix inversion)
- An alternative approach to compute $\boldsymbol{B}_{k+1}$ is to formulate it as a constrained optimization problem
  - $\boldsymbol{B}_{k+1} = \min_{\boldsymbol{B}} \|\boldsymbol{B} - \boldsymbol{B}_k\|$ subject to $\boldsymbol{B} = \boldsymbol{B}^T, \boldsymbol{B}(\boldsymbol{\theta}_k - \boldsymbol{\theta}_{k+1}) = \vec{\nabla} f(\boldsymbol{\theta}_k) - \vec{\nabla} f(\boldsymbol{\theta}_{k+1})$
  - The choice of matrix norm to use leads to different variations of the method:
    - Davidon-Fletcher-Powell (DFP): $\boldsymbol{B}_{k+1}^{-1} = \boldsymbol{B}_k^{-1} - \boldsymbol{A}_k + \boldsymbol{C}_k$
      - $\boldsymbol{A}_k = \dfrac{\boldsymbol{B}_k^{-1}\boldsymbol{y}_k\boldsymbol{y}_k^T\boldsymbol{B}_k^{-1}}{\boldsymbol{y}_k^T\boldsymbol{B}_k^{-1}\boldsymbol{y}_k}$
      - $\boldsymbol{C}_k = \dfrac{\boldsymbol{s}_k\boldsymbol{s}_k^T}{\boldsymbol{s}_k^T\boldsymbol{y}_k}$
      - This is a rank-2 method
    - Broyden–Fletcher–Goldfarb–Shanno (BFGS):

- $\boldsymbol{B}_{k+1}^{-1} = \left(\mathbf{1} - \frac{\boldsymbol{s}_k \boldsymbol{y}_k^T}{\boldsymbol{s}_k^T \boldsymbol{y}_k}\right) \boldsymbol{B}_k^{-1} \left(\mathbf{1} - \frac{\boldsymbol{s}_k \boldsymbol{y}_k^T}{\boldsymbol{s}_k^T \boldsymbol{y}_k}\right) + \frac{\boldsymbol{s}_k \boldsymbol{s}_k^T}{\boldsymbol{s}_k^T \boldsymbol{y}_k}$

- Quasi-Newton methods generally have between linear and quadratic convergence; we calls this *superlinear*
- In problems where n is very large such that $\mathcal{O}(n^2)$ is impractical, limiting-memory quasi-Newton methods compute the search step directly

Constrained Optimization – Penalty Methods

- Consider the problem of minimizing $f(\boldsymbol{\theta})$ subject to constraints $g_i(\boldsymbol{\theta}) \geq 0, h_j(\boldsymbol{\theta}) = 0$ where $i = 1, 2, \ldots, m$ and $j = 1, 2, \ldots, q$ and $\boldsymbol{\theta}_l \leq \boldsymbol{\theta} \leq \boldsymbol{\theta}_u$
- Penalty methods minimize $\pi(\boldsymbol{\theta}, \rho_k) = f(\boldsymbol{\theta}) + \rho_k \phi(\boldsymbol{\theta})$
  - $\phi(\boldsymbol{\theta})$ is the *penalty function* and ρ_k is the *penalty parameter*
  - We want $\boldsymbol{\phi}(\theta)$ equal to zero when no constraints are violated and positive when constraints are violated
  - We need to ensure that the objective and the penalty function are appropriately scaled, so one doesn't dominate the other
- Quadratic penalty function: $\phi(\boldsymbol{\theta}_k) = \sum_{i=1}^{m} (\max(0, -g_i(\boldsymbol{\theta})))^2 + \sum_{i=1}^{q} (h_i(\boldsymbol{\theta}))^2$
- Penalty methods template:
  1. Check termination conditions
  2. Minimize $\pi(\boldsymbol{\theta}, \rho_k)$ to find $\boldsymbol{\theta}_{k+1}$
  3. Increment the penalty parameter, $\rho_{k+1} > \rho_k$
     - Typically we multiply by a factor of 1.4 to 10, but this is problem dependent

Nonlinear Least Squares

- $\min_{\boldsymbol{\theta} \in \mathbb{R}^n} = \frac{1}{2} \sum_{i=1}^{N} r_i(\boldsymbol{\theta})^2$ where $r_i(\boldsymbol{\theta}) = \hat{f}(\boldsymbol{x}^{(i)}, \boldsymbol{\theta}) - y^{(i)}$
  - Assume $N > n$, i.e. we have more data points than dimensions
- Let $\boldsymbol{r} = \begin{bmatrix} r_1 \\ \vdots \\ r_n \end{bmatrix} \in \mathbb{R}^N$ and $f(\boldsymbol{\theta}) = \frac{1}{2} \|\boldsymbol{r}(\boldsymbol{\theta})\|_2^2$
- $\vec{\nabla} f(\boldsymbol{\theta}) = \sum_{j=1}^{N} r_j(\boldsymbol{\theta}) \vec{\nabla} r_j(\boldsymbol{\theta}) = \boldsymbol{J}(\boldsymbol{\theta})^T \boldsymbol{r}(\boldsymbol{\theta})$
  - $\boldsymbol{J}(\boldsymbol{\theta}) = \left[\frac{\partial r_j}{\partial r_i}\right] \in \mathbb{R}^{N \times n}$ is the Jacobian
- $\vec{\nabla}^2 f(\boldsymbol{\theta}) = \sum_{j=1}^{N} \vec{\nabla} r_j(\boldsymbol{\theta}) \vec{\nabla} r_j(\boldsymbol{\theta})^T + \sum_{j=1}^{N} r_j(\boldsymbol{\theta}) \vec{\nabla}^2 \boldsymbol{r}_j(\boldsymbol{\theta})$

  $= \boldsymbol{J}(\boldsymbol{\theta})^T \boldsymbol{J}(\boldsymbol{\theta}) + \sum_{j=1}^{N} r_j(\boldsymbol{\theta}) \vec{\nabla}^2 r_j(\boldsymbol{\theta})$
  - The Jacobian is easy to compute, which gets us most of the way to the Hessian
  - Often the second term is small so we can ignore it altogether and use the Jacobian to approximate the Hessian
    * This happens when the initial residual is small

Gauss-Newton Method

- This is similar to a modified Newton's method with line search
- Use $\vec{\nabla}^2 f(\boldsymbol{\theta}) \approx \boldsymbol{J}(\boldsymbol{\theta})^T \boldsymbol{J}(\boldsymbol{\theta})$ as an approximation of the Hessian
- Solve $\boldsymbol{J}(\boldsymbol{\theta})^T \boldsymbol{J}(\boldsymbol{\theta}) \boldsymbol{p}_k = -\boldsymbol{J}(\boldsymbol{\theta})^T \boldsymbol{r}(\boldsymbol{\theta}_k)$ for the search direction
  - Update $\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k + \alpha_k \boldsymbol{p}_k$ where α_k is chosen via line search

- In the case where the initial residual is small or approximately linear in $\boldsymbol{\theta}$, the Gauss-Newton method can perform similar to the full Newton's method, despite only computing first-order derivatives
- If $\boldsymbol{J}(\boldsymbol{\theta}_k)$ is full-rank and $\vec{\nabla} f(\boldsymbol{\theta}_k) \neq 0$, the search direction is always a valid direction

Stochastic Gradient Descent (SGD)

- In general we have loss function $\mathcal{L}(\boldsymbol{\theta}; \mathcal{D}) = \dfrac{1}{N} \displaystyle\sum_{i=1}^{N} l(\boldsymbol{\theta}; \boldsymbol{x}^{(i)}, y^{(i)}) + \lambda R(\boldsymbol{\theta})$ given data $\mathcal{D} = \{ (\boldsymbol{x}^{(i)}, y^{(i)}) \}_{i=1}^{N}$
  - This consists of the empirical loss and a regularization term
- $\vec{\nabla} \mathcal{L} = \dfrac{1}{N} \displaystyle\sum_{i=1}^{N} \vec{\nabla} l(\boldsymbol{\theta}; \boldsymbol{x}^{(i)}, y^{(i)}) + \lambda \vec{\nabla} R(\boldsymbol{\theta})$
- Applying the steepest descent method, we get the update $\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - \eta_k \vec{\nabla} \mathcal{L}(\boldsymbol{\theta}_k; \mathcal{D})$
  - η_k is the *learning rate*
    - In classical methods we use backtracking line search to find this, but in machine learning we typically choose this heuristically, as a constant
    - e.g. start with a sensible value like $\eta = 0.1$; take smaller steps if objective gets worse or we see oscillation; take larger steps if objective reduces too slowly
  - Since we are computing the gradient over the full dataset $\mathcal{D}$, this is known as *full-batch gradient descent*
- Full-batch gradient descent is typically very expensive since we need to compute the gradient over the entire dataset
- Procedure of SGD:
  1. Shuffle training indices $\{ 1, \ldots, N \}$
  2. Initialize $\boldsymbol{\theta}_0$
  3. Repeat until we reach some convergence criteria:
     - For i from 1 to N, $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \vec{\nabla} l(\boldsymbol{\theta}; \boldsymbol{x}^{(i)}, y^{(i)})$
- Each iteration of the outer loop is an *epoch*, where we loop over the full dataset
- SGD essentially uses only one datapoint at a time
  - This works because the gradient using one datapoint is an unbiased estimator of the full gradient
  - Let $\boldsymbol{g}_t = l(\boldsymbol{\theta}_k; \boldsymbol{x}^{(t)}, y^{(t)})$, we have that $\mathbb{E}[\boldsymbol{g}_t] = \mathcal{L}(\boldsymbol{\theta}_k; \mathcal{D})$
- Consider gradient descent over a GLM with M terms
  - The cost of full-batch gradient descent is $\mathcal{O}(NM)$, and converges in $\mathcal{O}\left(\log \dfrac{1}{\rho}\right)$
  - The cost of stochastic gradient descent is only $\mathcal{O}(M)$, and converges in $\mathcal{O}\left(\dfrac{1}{\rho}\right)$ iterations
    - Even though SGD takes more iterations to converge (sub-linearly), it's cheaper overall when factoring in the cost per iteration
    - Sometimes it's not practical to do full-batch gradient descent due to the size of the dataset
- In *mini-batch gradient descent* we compute the gradient over a mini-batch that is smaller than the full dataset, but more than 1 sample, in each iteration
  - This is a compromise between full-batch gradient descent and SGD
  - The larger the batch size, the closer we get to full-batch and the faster we converge (in iterations)

Lecture 11, Mar 5, 2024

Lecture 12, Mar 12, 2024

Neural Networks

- $w_{jk}^{(i)}$ denotes the weight for input k, from neuron j in layer i
- $\boldsymbol{W}^{(i)}, \boldsymbol{b}^{(i)}$ are the weight matrix and bias vector for layer i
- Each layer's input is denoted by $\boldsymbol{X} \in \mathbb{R}^{N \times D}$ and output is denoted by $\hat{\boldsymbol{F}}$

- Each layer's output is computed as $\phi^{(i)}\left(\boldsymbol{X}\boldsymbol{W}^{(i)^T} + \boldsymbol{b}^{(i)^T}\right)$
- A linear activation function would see no benefit from stacking layers, so it is typically not used
- Other common activation functions are ReLU, soft/smooth/leaky ReLU, threshold (perceptron), logistic (sigmoid), and tanh

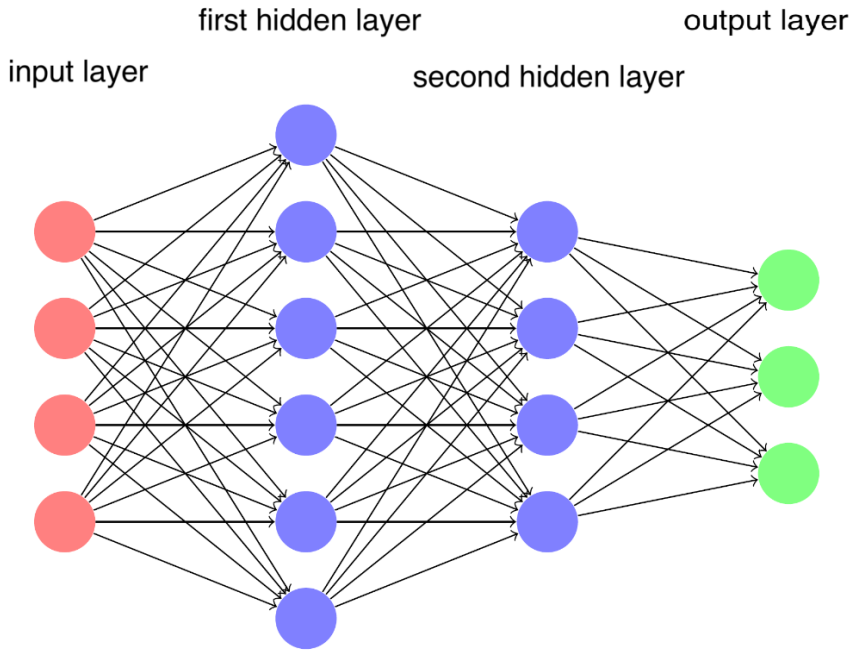

Figure 10: Fully connected feedforward neural network.

- To find weights, we either minimize a loss (e.g. least-squares) or maximize a likelihood (e.g. Gaussian)
- A logistic sigmoid activation $\sigma(z) = \dfrac{1}{1+\exp(-z)}$ is used in classification to restrict the output range to $(0, 1)$, allowing us to interpret it as a probability
- For multi-class classification we assume an output distribution of a categorical (multinomial) distribution
  - This leads us to the softmax function, $\dfrac{e^{z_j}}{\sum_k e^{z_k}}$
  - For $k = 2$ we get back the sigmoid (up to a scaling)
- For numerical stability, use the LogSumExp function, i.e. taking the log of the softmax

Backpropagation

- A recursive procedure to find the gradient
- Consider the simple example $f(x, y, z) = (x + y)z$ where $x = -2, y = 5, z = -4$
  - We have $q = x + y$, so $\dfrac{\partial q}{\partial x} = 1$, $\dfrac{\partial q}{\partial y} = 1$, and $f = qz$ so $\dfrac{\partial f}{\partial q} = z$, $\dfrac{\partial f}{\partial z} = q$
  - To perform backpropagation, we start from the end of the computational graph and work backwards
  - Use the chain rule to get successively deeper in the graph

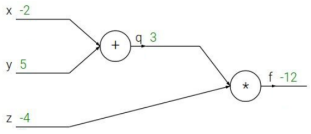

Figure 11: Computational graph for the example.

- In a computational graph, each node is aware of only its surroundings: its local inputs x, y and its output z, and some operation f that is applied
  - We can compute a local gradient $\frac{\partial z}{\partial x}$ and $\frac{\partial z}{\partial y}$
- We also have the upstream gradient of the final output L with respect to the current node output, $\frac{\partial L}{\partial z}$
- Now when we pass downstream, we pass $\frac{\partial L}{\partial x} = \frac{\partial L}{\partial z}\frac{\partial z}{\partial x}$ and $\frac{\partial L}{\partial y} = \frac{\partial L}{\partial z}\frac{\partial z}{\partial y}$

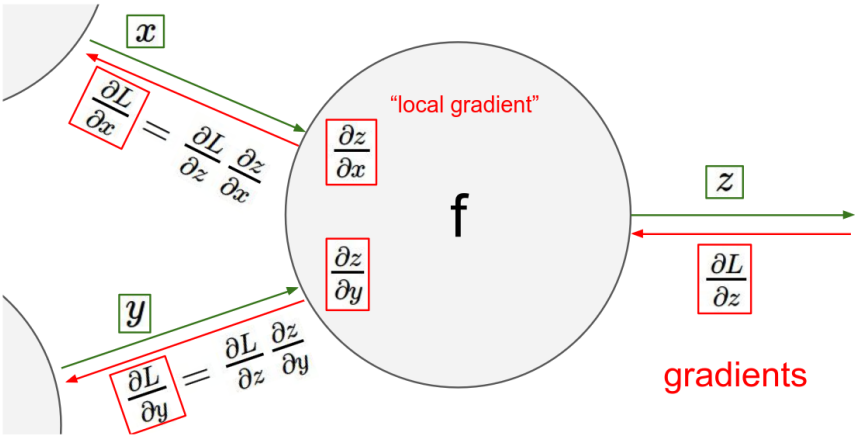

Figure 12: Illustration of the backpropagation algorithm.

- Given any function, first find its computational graph, then apply the algorithm recursively starting from the output, until we reach the inputs we want
- The computational graph can be broken down into any level of granularity; e.g. instead of breaking a sigmoid into a negation, exponentiation, addition, etc, we can treat the entire thing as a sigmoid gate
- We can observe some patterns in how common operations affect gradient flow:
  - Add gates are gradient distributors: the upstream gradient is propagated as-is to both inputs
  - Max and min gates are gradient routers: the upstream gradient is propagated as-is to only a single input, while the other input(s) get zero (since they do not affect the output)
  - Multiplication gates are gradient switchers: the gradient of one input is the upstream gradient multiplied by the value of the other inputs

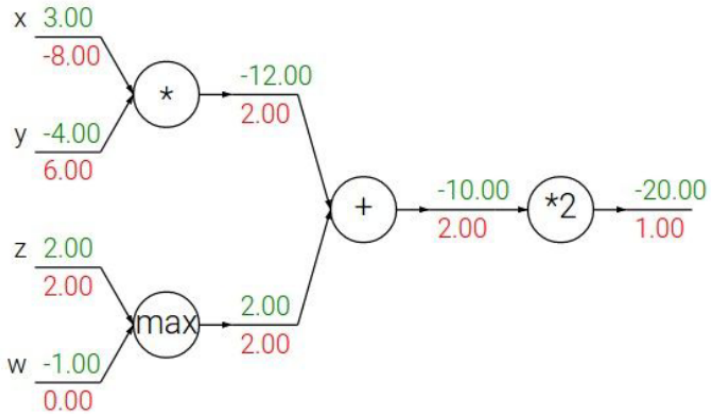

Figure 13: Common operations in a computational graph.

- One node can connect to two other nodes, in which case the upstream gradient is the sum of the upstream gradients of all the nodes it connects to

  - $\frac{\partial f}{\partial x} = \sum_i \frac{\partial f}{\partial q_i} \frac{\partial q_i}{\partial x}$ where q_i are upstream nodes connected to this node
- The whole operation can be vectorized, replacing gradients with Jacobian matrices

Lecture 13, Mar 15, 2024

Neural Networks – Training Considerations

Weight Initialization

- Deep neural networks suffer from the problem of *vanishing* or *exploding* gradients
  - Since we have to multiply together a large number of gradients, if the gradients are each individually small, the overall gradient goes to zero; if the gradients are individually large, the overall gradient goes to infinity
  - If our gradient is too small, then we might be moving very slowly or detect false convergence
  - If the gradients are too large, we suffer from instability
- Initializing the weights to the same value, e.g. zero, is not a good choice
  - With all weights being zero, each neuron receives an input of zero, so the gradients are the same and the network cannot learn complex patterns
  - We also have the symmetry issue; if the network architecture and weights are symmetric, we get symmetric gradients and the entire network just becomes symmetric
- What about small random numbers?
  - Since the weights are small, all activations will tend to zero, and all gradients will become the same
- What about large random numbers?
  - The opposite happens – the activations saturate and all gradients become zero (assuming sigmoid or tanh activation)
- If our input data is normalized to zero-mean and unit variance, we can expect that we'd also want our weights to be distributed the same
- *Xavier initialization* scales the weights by the square root of the number of inputs
  - Derived by looking at what weights will avoid vanishing or exploding gradients
  - The distribution is a unit Gaussian scaled down by the root of the number of inputs
  - This is the default for PyTorch, Tensorflow, etc
- For other activation functions, we have other strategies
  - For ReLU, we instead divide by the root of half the number of inputs
- Symmetries in the weight space lead to equivalent networks with different sets of weights (*model identifiability problem*)
  - Having different ways to connect weights that lead to the same result will give many local optima
  - However, if we overparameterize our network, it will turn the minima into saddle points (at the cost of overfitting)

Overfitting Prevention

- Regularization can be used to prevent overfitting in neural networks as well
  - Penalize the magnitude of the weights
  - Using l_2 regularization is commonly called weight decay
  - l_1 regularization introduces sparsity
- *Early stopping* is the idea of stopping training after a certain number of iterations, instead of checking for convergence in the gradient
  - The number of iterations is treated as a hyperparameter
  - Once the validation loss starts increasing, we stop, backtrack a bit till the point before it started increasing, and take that as the final model
- Another method is to use more data
  - If it's not possible to collect more data, we can use *data augmentation* techniques, such as rotation, blurring, cropping etc, to generate new training samples that the model should still recognize

  - We can also intentionally add adversarial examples by adding noise
- Bagging/bootstrap aggregation can also be used to reduce the variance in the estimates
- *Dropout* is another technique where some hidden units are "dropped" during training with probability $1 - \pi$, where π is a hyperparameter)
  - During testing/inference, the weights are scaled back up by π
  - Typical values are $\pi \in [0.5, 0.8]$
  - Statistically, this is an approximate Bayesian inference scheme
- *Weight sharing* is a technique that uses prior knowledge to identify weights that should be close to each other, and force the weights to be the same or penalize their difference
  - CNNs are an example of this, since convolutions share the same weights in the kernels

Convolutional Neural Networks (CNNs)

- In a fully-connected network each layer is fully connected to the one before it – all neurons are connected to all the neuron in the previous layer
  - In such a network, if we have m neurons in the previous layer and n in the current, we'd need nm weights
- In convolutional neural networks, we convolve a filter with the image
  - This keeps the spacial structure of the input image
  - Note filters always extend the full depth of the input volume, e.g. for an RGB image the filters have a depth of 3
  - The filter slides over the input, taking a dot product at each position, resulting in an activation/feature map
- Multiple filters can be stacked together to get more output channels, for variety in feature spaces
- A CNN has a sequence of convolutional layers, interspersed with activation functions

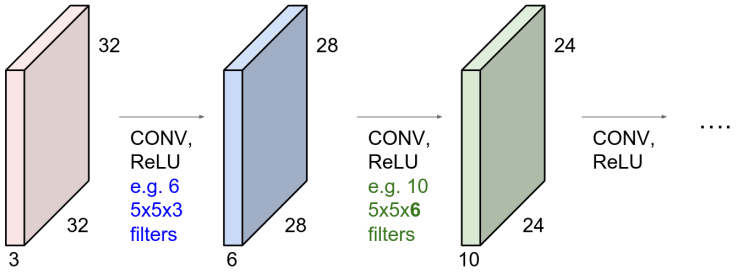

Figure 14: CNN structure.

- Earlier layers in the stack will learn low-level features, and layers deeper in the network learn more abstract features
  - Eventually the data is transformed into a linearly separable form, which can be passed to fully connected layer(s) to be processed into the final output
- Between fully connected layers, we can use pooling layers to reduce the size of the feature map to make it more manageable
  - Pooling layers are essentially downsampling the network spatially
  - The depth of the map remains the same since we only pool spatially
  - Pooling methods include max pooling and average pooling

Autoencoders

- Autoencoders are a type of model for unsupervised learning, which can be used for dimensionality reduction
- Autoencoders consist of an encoder, mapping from input to feature space, and a decoder, mapping from feature to output space
- Data is passed through the encoder and mapped into the feature space, and then mapped by the decoder back into a reconstruction of the input

  - Due to the reduction in dimension of the feature space, the reconstruction of the input will only have the "important parts"
  - Now we apply a loss function between the input and output data (usually l_2)
- After training we can discard the decoder, and use the encoder as a dimensionality reducer
  - We can use the encoder to initialize a supervised model – the output from the encoder can be fed to a classier
  - This is important for semi-supervised learning where we only have a small amount of labelled data
  - Improves performance since the input is lower in dimension and already processed to only contain the "important parts"

Lecture 14, Mar 26, 2024

Bayesian Estimation

- Bayesian approaches allow us to quantify the uncertainty in predictions, whereas MLE and MAP are frequentist approaches that only give a point estimate
- Additional benefits include:
  - Allows us to use a prior toe encode our beliefs about the parameters before seeing any data
  - Prevents overfitting so long as the prior and likelihood are accurate
  - Allows us to construct models in the low-data regime
- Frequentist methods assume that there exists a true, fixed parameter value θ^*
  - Error bars on the estimate of θ are obtained by considering the distribution of all possible datasets
- In the Bayesian approach, we have a single observational dataset, and we estimate the posterior distribution of the parameters given the data
  - Error bars are obtained from this posterior distribution
- Bayesian methods use two things:
  - The likelihood $p(y^{(1)}, \dots, y^{(N)}|\theta) = p(\mathcal{D}|\theta)$
    - This is not a probability distribution but rather a function of the parameters θ
  - The prior $p(\theta)$
    - Encodes prior beliefs about the parameters before looking at the data
    - Often we use a form that makes the computation easy rather than some rigorous statistical assumption
- We are interested in computing two distributions:
  - The *posterior distribution* $p(\theta|\mathcal{D})$
    - This encodes our beliefs about the parameters after observing the data
    - Comes from Bayes rule, $p(\theta|\mathcal{D}) = \dfrac{p(\mathcal{D}|\theta)p(\theta)}{\int p(\mathcal{D}|\theta')p(\theta')\,\mathrm{d}\theta'}$
  - The *posterior predictive distribution* $p(y'|\mathcal{D})$
    - This is the distribution of a future observation given the data
    - Used to estimate what unseen values in the data are
    - Marginalize out θ to get $p(y'|\mathcal{D}) = \displaystyle\int p(y'|\theta)p(\theta|\mathcal{D})\,\mathrm{d}\theta$
- Process:
  1. Write down the likelihood $p(\mathcal{D}|\theta)$
  2. Write down the prior $p(\theta)$
  3. Compute the posterior $p(\theta|\mathcal{D}) = \dfrac{p(\mathcal{D}|\theta)p(\theta)}{p(\mathcal{D})}$
  4. Compute the posterior predictive distribution $p(y'|\mathcal{D})$
- The two last steps are often challenging to do
- Example: suppose we have a coin where θ is the probability of heads; we have a dataset of N flips
  - Likelihood: Bernoulli $p(\mathcal{D}|\theta) = \theta^{N_H}(1-\theta)^{N_T}$
  - Prior: beta prior $p(\theta, a, b) = \dfrac{\Gamma(a+b)}{\Gamma(a)\Gamma(b)}\theta^{(a-1)}(1-\theta)^{(b-1)}$
    - This distribution encodes possible beliefs about the prior

    * Expectation at $\frac{a}{a+b}$
  - Posterior $p(\theta|D) \propto \theta^{a+N_H-1}(1-\theta)^{b+N_T-1}$
    * This is another beta distribution with parameters $a+N_H$ and $b+N_T$
  - Posterior predictive distribution $p(y'=H|\mathcal{D}) = \int p(y'=H|\theta)p(\theta|\mathcal{D})\,\mathrm{d}\theta = \frac{N_H+a}{N_H+N_T+a+b}$
- In the above example we chose the beta prior because it has the same form as the Bernoulli distribution
  - This is known as a *conjugate prior*, which makes the computation convenient
  - Any distribution in the exponential family has a corresponding conjugate prior
  - e.g. for Bernoulli we have beta; for Gaussian we have Gaussian again
- As we increase the amount of data, we rely on the prior less and the distribution approaches the MLE estimate
- If the prior and likelihood assumptions are incorrect, the Bayesian approach can still overfit
- In practice, it also involves evaluating a high-dimensional integral which is not practical
  - Bayesian linear regression can allow us to approximate these integrals

Bayesian Linear Regression

- Assume a dataset $\mathcal{D}$ where each output is assumed to be IID from a normal distribution with mean $\boldsymbol{w}^T\boldsymbol{\phi}(\boldsymbol{x})$ and variance σ^2
  - This led us to the normal GLM with MLE, and the regularized GLM with MAP
- Likelihood: $\log p(\boldsymbol{y}|\boldsymbol{w},\boldsymbol{X},\sigma^2) = \sum_{i=1}^{N} \log \mathcal{N}(y^{(i)}|\boldsymbol{w}^T\boldsymbol{\phi}(\boldsymbol{x}^{(i)}),\sigma^2)$
- Prior: $p(\boldsymbol{w}|\alpha) = \mathcal{N}(\boldsymbol{w}|\boldsymbol{0},\alpha\boldsymbol{1})$
  - This is the conjugate prior
- Posterior: $\log p(\boldsymbol{w}|\mathcal{D}) = \log p(\boldsymbol{w}) + \log p(\mathcal{D}|\boldsymbol{w}) + \text{const}$
$$= -\frac{1}{2\alpha}\boldsymbol{w}^T\boldsymbol{w} - \frac{1}{2\sigma^2}(\boldsymbol{w}^T\boldsymbol{\Phi}^T\boldsymbol{\Phi}\boldsymbol{w} - 2\boldsymbol{w}^T\boldsymbol{\Phi}^T\boldsymbol{y} + \boldsymbol{y}^T\boldsymbol{y})$$
$$= -\frac{1}{2}(\boldsymbol{w}-\boldsymbol{\mu})^T\boldsymbol{\Sigma}^{-1}(\boldsymbol{w}-\boldsymbol{\mu}) + \text{const}$$
  - Exponentiate this and we get a Gaussian, so $p(\boldsymbol{w}|\mathcal{D},\alpha,\sigma) = \mathcal{N}(\boldsymbol{w}|\boldsymbol{\mu},\boldsymbol{\Sigma})$
  - $\boldsymbol{\mu} = \frac{1}{\sigma^2}\boldsymbol{\Sigma}\boldsymbol{\phi}^T y$
  - $\boldsymbol{\Sigma}^1 = \frac{1}{\sigma^2}\boldsymbol{\Phi}^T\boldsymbol{\Phi} + \frac{1}{\alpha}\boldsymbol{1}$
- Since the posterior is a Gaussian, its mean is the MAP estimate $\boldsymbol{\mu} = \left(\boldsymbol{\Phi}^T\boldsymbol{\Phi} + \frac{\sigma^2}{\alpha}\boldsymbol{1}\right)^{-1}\boldsymbol{\Phi}^T\boldsymbol{y}$
- Posterior predictive: $p(y'|\boldsymbol{x}',\mathcal{D}) = \int p(y'|\boldsymbol{x}',\boldsymbol{w})p(\boldsymbol{w}|\mathcal{D})\,\mathrm{d}\boldsymbol{w}$
$$= \int \mathcal{N}(y'|\boldsymbol{w}^T\boldsymbol{\phi}(\boldsymbol{x}'),\sigma^2)(\boldsymbol{w}|\boldsymbol{\mu},\boldsymbol{\Sigma})\,\mathrm{d}\boldsymbol{w}$$
$$= \mathcal{N}(y'|\boldsymbol{\mu}^T\boldsymbol{\phi}(\boldsymbol{x}'),\boldsymbol{\phi}(\boldsymbol{x}')^T\boldsymbol{\Sigma}\boldsymbol{\phi}(\boldsymbol{x}') + \sigma^2)$$
  - The last line is obtained because we have a convolution of two Gaussians
- Bayesian linear regression considers all possible explanations of how the data was generated, and predicts using all possible regression weights, weighted by the posterior probability
- Between each row of the figure, we add data points; the new posterior is obtained by taking the prior and multiplying by the likelihood, and the data space shows lines indicating the distribution of possible parameters
- We need to choose a good α and σ to get a good result
  - If we know about the noise (e.g. via a sensor model), we can use this to specify σ^2
  - If we previously estimated the posterior and we would like to update it (i.e. sequential inference), we can use the previous posterior as the prior, like in the first figure
  - If we don't have enough information for either, we could specify priors over α and σ^2

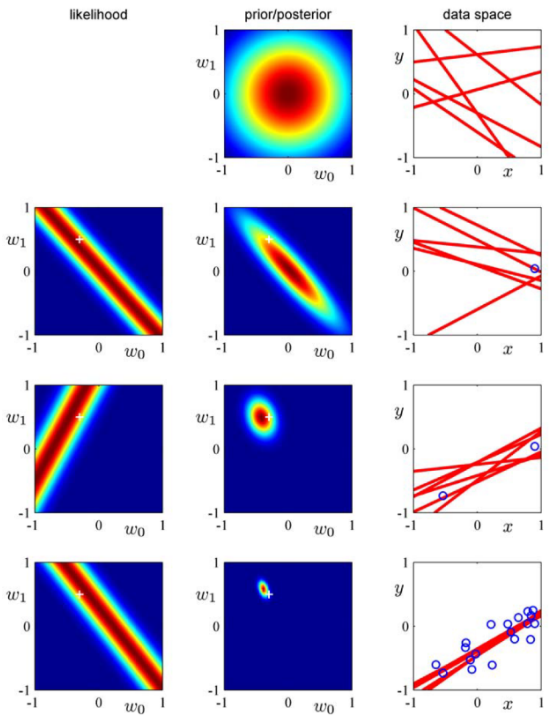

Figure 15: Illustration of the interpretation of Bayesian linear regression.

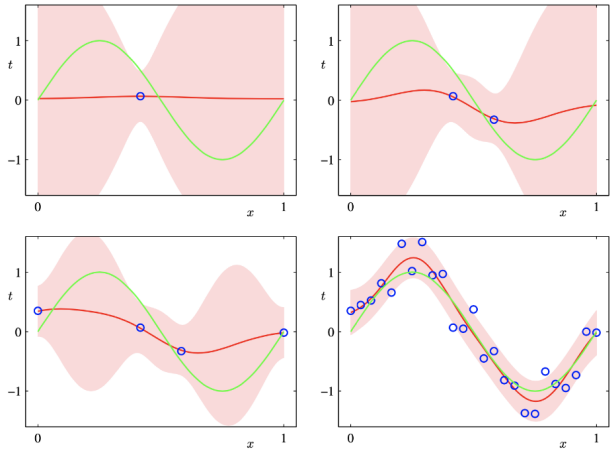

Figure 16: Illustration of Bayesian linear regression with a GLM with radial basis functions.

    * This is the full Bayesian method
    * No analytic solution exists for the inference
- In *type-II inference*, we numerically optimize α and σ^2 to maximize $\log p(\boldsymbol{y}|\boldsymbol{X}, \alpha, \sigma^2)$, known as the *evidence* (i.e. the likelihood of the observations), to find good values for α and σ^2
  - Computationally cheap
  - Only two parameters, so not as prone to overfitting
  - Tends to underestimate the uncertainty (which is not good for engineering), because we're using point estimates for the parameters α and σ^2

Lecture 15, Apr 5, 2024

Gaussian Processes – Regression in Function-Space

- Gaussian processes are a kernelized version of Bayesian linear regression
  - Allows scaling to infinitely many basis functions
  - Priors over functions instead of parameters, which is a lot more powerful (e.g. allows specifying smoothness, periodicity, etc)
- We want to compute the posterior predictive distribution $p(y'|\boldsymbol{y}) = \dfrac{p(y', \boldsymbol{y})}{\int p(y', \boldsymbol{y})\,\mathrm{d}y'}$
  - $\boldsymbol{y}$ is the data we have, and y' is the prediction we make about the future samples
- Derivation:
  - Since we assume both Gaussian weights and noise, the distribution of targets will also be Gaussian
  - $y = \boldsymbol{w}^T\boldsymbol{\phi}(\boldsymbol{x}) + \varepsilon \implies p\left(\begin{bmatrix} y' \\ \boldsymbol{y} \end{bmatrix}\right) = \mathcal{N}\left(\boldsymbol{0}, \alpha \begin{bmatrix} \boldsymbol{\phi}^T(\boldsymbol{x}') \\ \boldsymbol{\Phi} \end{bmatrix} \begin{bmatrix} \boldsymbol{\phi}(\boldsymbol{x}') & \boldsymbol{\Phi}^T \end{bmatrix} + \sigma^2\boldsymbol{1}\right)$
    $$= \mathcal{N}\left(\boldsymbol{0}, \alpha \begin{bmatrix} \boldsymbol{\phi}^T(\boldsymbol{x}')\boldsymbol{\phi}(\boldsymbol{x}') & \boldsymbol{\phi}^T(\boldsymbol{x}')\boldsymbol{\Phi}^T \\ \boldsymbol{\Phi}\boldsymbol{\phi}(\boldsymbol{x}') & \boldsymbol{\Phi}\boldsymbol{\Phi}^T \end{bmatrix} + \sigma^2\boldsymbol{1}\right)$$
    * $\boldsymbol{x}'$ is the test point and y' is our prediction for it
    * Note $\boldsymbol{w}\mathcal{N}(\boldsymbol{0}, \alpha\boldsymbol{1})$ is our prior (regularization) and $\varepsilon \sim \mathcal{N}(0, \sigma^2)$ is the noise
  - Let the Gram matrix $\boldsymbol{K}_{\boldsymbol{X},\boldsymbol{X}} = \boldsymbol{\Phi}\boldsymbol{\Phi}^T \in \mathbb{R}^{N\times N}$, where entry ij is $\alpha\boldsymbol{\phi}^T(\boldsymbol{x}^{(i)})\boldsymbol{\phi}(\boldsymbol{x}^{(j)}) = k(\boldsymbol{x}^{(i)}, \boldsymbol{x}^{(j)})$, where $k\colon \mathcal{X}\times\mathcal{X} \mapsto \mathbb{R}$ is the kernel
  - Let $\boldsymbol{k}_{\boldsymbol{X},\boldsymbol{x}'} = \begin{bmatrix} k(\boldsymbol{x}^{(1)}, \boldsymbol{x}') & k(\boldsymbol{x}^{(2)}, \boldsymbol{x}') & \dots & k(\boldsymbol{x}^{(N)}, \boldsymbol{x}') \end{bmatrix}^T$
  - Then $p\left(\begin{bmatrix} y' \\ \boldsymbol{y} \end{bmatrix}\right) = \mathcal{N}\left(\boldsymbol{0}, \begin{bmatrix} k_{\boldsymbol{x}',\boldsymbol{x}'} & \boldsymbol{k}_{\boldsymbol{x}',\boldsymbol{X}} \\ \boldsymbol{k}_{\boldsymbol{X},\boldsymbol{x}'} & \boldsymbol{K}_{\boldsymbol{X},\boldsymbol{X}} \end{bmatrix} + \sigma^2\boldsymbol{1}\right)$
    * The Gram matrix is a covariance matrix
    * Here we have implicitly marginalized out $\boldsymbol{w}$
  - Therefore $p(y'|\boldsymbol{y}) = \mathcal{N}(\mu_p, \sigma_p^2)$ where:
    * $\mu_p = \boldsymbol{k}_{\boldsymbol{x}',\boldsymbol{X}}(\boldsymbol{K}_{\boldsymbol{X},\boldsymbol{X}} + \sigma^2\boldsymbol{1})^{-1}\boldsymbol{y}$
    * $\sigma_p^2 = k_{\boldsymbol{x}',\boldsymbol{x}'} - \boldsymbol{k}_{\boldsymbol{x}',\boldsymbol{X}}(\boldsymbol{K}_{\boldsymbol{X},\boldsymbol{X}} + \sigma^2\boldsymbol{1})^{-1}\boldsymbol{k}_{\boldsymbol{X},\boldsymbol{x}'} + \sigma^2$
    * We have written the posterior predictive distribution entirely in terms of the kernel
    * Note this is equivalent to what we derived for a GLM, with squared error, l_2 regularization and $\lambda = \dfrac{\sigma^2}{\alpha}$
  - This is known as *Gaussian process regression*
- We have developed a kernelized version of Gaussian linear regression, similar to kernelized GLMs
  - The kernels we can use for this are the same as the ones for kernelized GLMs
- Compare the time and memory requirements:
  - With normal Bayesian linear regression, i.e. GP regression in weight-space, we need an expensive matrix inversion for $\boldsymbol{\Phi}^T\boldsymbol{\Phi}$ and also to store these matrices
    * $\mathcal{O}(NM^2 + M^3)$ time
    * $\mathcal{O}(NM + M^2)$ memory
  - With the kernelized , i.e. GP regression in function-space, the cost is independent of M
    * $\mathcal{O}(N^3)$ time
    * $\mathcal{O}(N^2)$ memory

- Similar to kernelized GLMs, using GP in function space is much more efficient when we have $M \gg N$

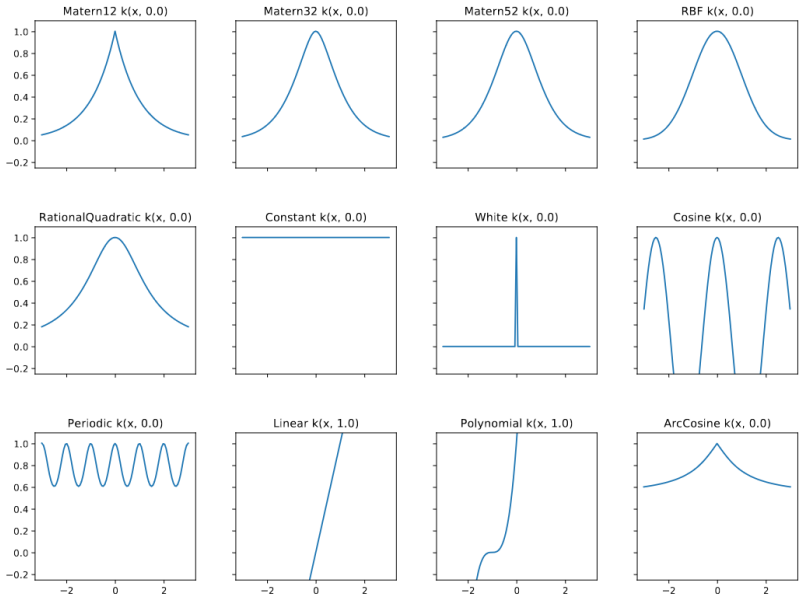

Figure 17: Visualization of some kernels in 1 dimension.

- Kernel selection is very important; changing the kernel drastically impacts the model, since it changes our assumptions about what possible models look like, including smoothness, periodicity, etc
  - As always kernels need to be positive definite
- We can compose new kernels from multiple kernels, by adding them together, multiplying them together, or by composing with a function as $k(x, y) = k_1(f(x), f(y))$; all these will preserve positive definiteness
  - e.g. if the data has both long-term trends and short-term trends (e.g. Mauna Loa dataset), we can add together a kernel with a large lengthscale and a kernel with a small one, to produce a better kernel overall

- Kernels also have hyperparameters, e.g. in Gaussian kernel $k(x, y) = \sigma^2 e^{-\frac{(x-y)^2}{2\theta}}$ the output variance σ^2 and lengthscale $l = 1/\theta$ are important hyperparameters
- These hyperparameters can be selected through a number of means, like with Bayesian linear regression, e.g. prior knowledge, cross validation, full Bayesian inference and type-II maximum likelihood
  - Recall that in type-II maximum likelihood we try to maximize $p(\boldsymbol{y}|\boldsymbol{X})$ as a function of hyperparameters
  - $\log p(\boldsymbol{y}|\boldsymbol{X}) = -\frac{N}{2}\log\alpha - \frac{N}{2}\log(\sigma^2) - \frac{1}{2\sigma^2}\boldsymbol{y}^T\boldsymbol{y} + \frac{1}{2}\boldsymbol{\mu}^T\boldsymbol{\Sigma}^{-1}\boldsymbol{\mu} + \frac{1}{2}\log\det(\boldsymbol{\Sigma}) - \frac{N}{2}\log(2\pi)$
    * Used for weight space
  - $\log(\boldsymbol{y}|\boldsymbol{X}) = -\frac{N}{2}\log(2\pi) - \frac{1}{2}\log\det(\boldsymbol{K}_{\boldsymbol{X},\boldsymbol{X}} + \sigma^2\boldsymbol{1}) - \frac{1}{2}\boldsymbol{y}^T(\boldsymbol{K}_{\boldsymbol{X},\boldsymbol{X}} + \sigma^2\boldsymbol{1})^{-1}\boldsymbol{y}$
    * Used for function space

Approximate Bayesian Methods

- Generally, given a set of observed evidence, X_E and a set of unobserved variables that we want to infer, X_F, a general class of problems is computing $p(X_F|X_E) = \frac{p(X_E, X_F)}{p(X_E)}$
  - Often we know the joint distribution, but not the conditional distribution, because finding $p(X_E)$ is difficult or impractical
  - This is is a generalization of Bayesian inference estimation of $p(\boldsymbol{w}|\mathcal{D}) = \frac{p(\mathcal{D}|\boldsymbol{w})p(\boldsymbol{w})}{p(\mathcal{D})}$

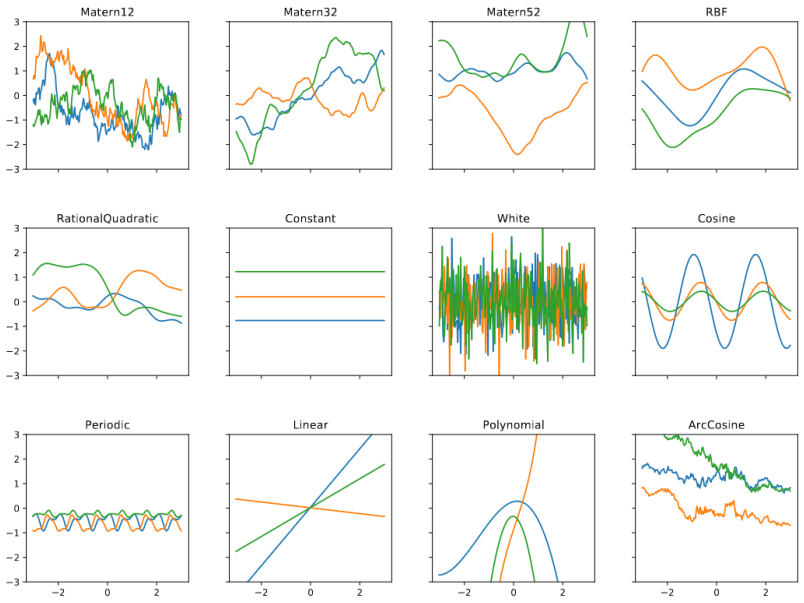

Figure 18: Visualization of the priors encoded by the kernels in the previous figure. These are different possibilities of $\hat{\boldsymbol{f}}$ sampled from the prior.

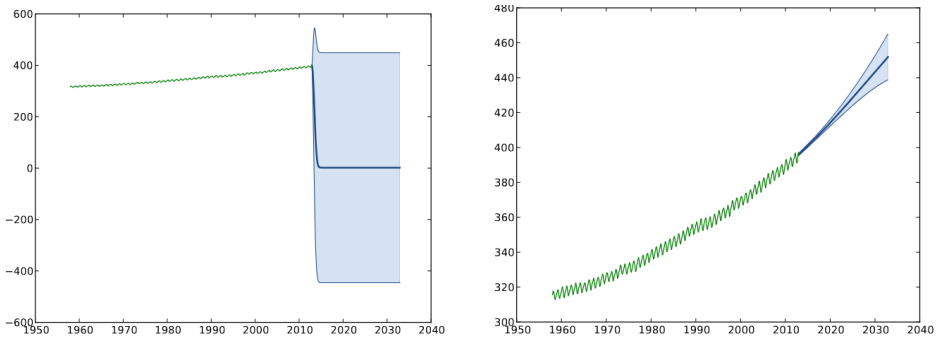

Figure 19: Examples of predictions using only a large lengthscale kernel and a small lengthscale one.

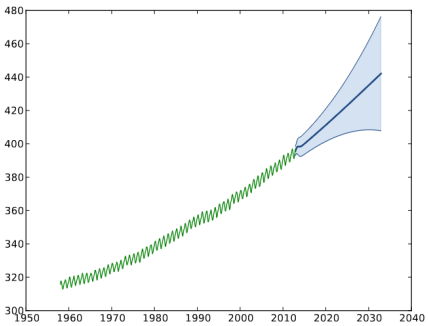

Figure 20: Predictions using the sum of both kernels.

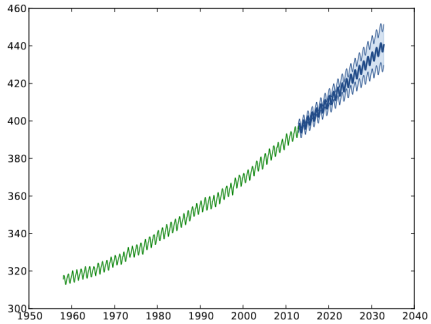

Figure 21: Predictions using the sum of two kernels with different lengthscales, plus a periodic kernel, and a degree 2 polyomial kernel.

    * In this case we know $p(\mathcal{D}|\boldsymbol{w})$ from our model setup + noise, and $p(\boldsymbol{w})$ from our prior on the parameters
- Since we often have $p(X_E, X_F)$, we know $p(X_F|X_E)$ up to a normalization constant, which is intractable to compute due to having to integrate $p(X_E) = \int p(X_E, X_F)\,\mathrm{d}X_F$
- We can try to estimate the $p(X_E)$ integral through quadrature numerical integration, but the number of points we need to sample increases exponentially with the dimensionality of X_F, making this impractical in most cases
- The *Laplace approximation* finds a Gaussian approximation of the posterior, based on a second-order Taylor approximation at the MAP
  - Let $X_F = \boldsymbol{z}$, then $p(\boldsymbol{z}|X_E) = \frac{1}{Z}p(X_E, \boldsymbol{z}) = \frac{1}{Z}\tilde{p}(\boldsymbol{z})$
  - Consider the MAP, $\hat{\boldsymbol{z}}_{\mathrm{MAP}} = \underset{\boldsymbol{z}}{\operatorname{argmax}} \log \tilde{p}(\boldsymbol{z})$; this must be a critical point of $\log \tilde{p}(\boldsymbol{z})$, so the gradient is zero
  - The second-order Taylor expansion is then $\log p(\boldsymbol{z}|X_E) \approx \log \tilde{p}(\hat{\boldsymbol{z}}_{\mathrm{MAP}}) - \frac{1}{2}(\boldsymbol{z} - \hat{\boldsymbol{z}}_{\mathrm{MAP}})^T \boldsymbol{A} (\boldsymbol{z} - \hat{\boldsymbol{z}}_{\mathrm{MAP}})$
    * $\boldsymbol{A} = -\vec{\nabla}^2 \log \tilde{p}(\boldsymbol{z})$ is the (negative) Hessian, evaluated at $\hat{\boldsymbol{z}}_{\mathrm{MAP}}$
      - Note we define $\boldsymbol{A}$ with a negative sign, since the Hessian at a maximum is negative-definite, but we need a positive-definite matrix later to be the covariance
    * The first-order term is zero here because the gradient is zero at a critical point
  - Exponentiate the approximation, then $p(\boldsymbol{z}|X_E) \approx \mathcal{N}(\boldsymbol{z}|\hat{\boldsymbol{z}}_{\mathrm{MAP}}, \boldsymbol{A}^{-1})$
- The Laplace approximation is often used due to its simplicity; we only need to estimate the MAP, then approximate and invert the Hessian at the MAP
  - However, it often does a poor job
  - The main limitation is that it only approximates the posterior around the MAP and doesn't account for global properties
- We will introduce another method, based on Monte Carlo expectation approximation
  - $\mathbb{E}_{x\sim p(x)}[f(x)] \approx \frac{1}{M}\sum_{i=1}^{M} f(\boldsymbol{x}^{(i)})$ is the Monte Carlo approximation for the expectation of $f(x)$, given a distribution $p(x)$, for M samples chosen independently from $p(x)$
  - It is an unbiased estimator and has variance proportional to $\frac{1}{\sqrt{M}}$
  - Important, the accuracy of the Monte Carlo estimate is independent of the dimensionality of x, making it much more useful in high-dimension contexts

Lecture 16, Apr 9, 2024

Stochastic Variational Inference (SVI)

- *Stochastic variational inference* is the technique of approximating the true conditional $p(\boldsymbol{z}|\boldsymbol{x}) = \dfrac{p(\boldsymbol{x},\boldsymbol{z})}{p(\boldsymbol{x})}$ by a simpler distribution, $q(\boldsymbol{z}|\boldsymbol{\theta})$
  - We want $q(\boldsymbol{z}|\boldsymbol{\theta})$ to be "close to" $p(\boldsymbol{z}|\boldsymbol{x})$; to do this we need to define "closeness" of distributions
  - We can choose $q(\boldsymbol{z}|\boldsymbol{\theta})$ to come from a known family of distributions, e.g. Gaussians

Definition

The *Kullback-Leibler (KL) divergence* of two distributions $p(\boldsymbol{z})$ and $q(\boldsymbol{z})$ is

$$KL(q \parallel p) = \mathbb{E}_{\boldsymbol{z}\sim q(\boldsymbol{z})}\left[\log \frac{q(\boldsymbol{z})}{p(\boldsymbol{z})}\right]$$

with the following properties:

- $KL(q \parallel p) \geq 0$
- $KL(q \parallel p) = 0 \iff q = p$
- $KL(q \parallel p) \neq KL(p \parallel q)$

- KL divergence is always positive and zero when distributions are equal, however it is not symmetric!
  - For *reverse-KL* (aka *information projection*), we take $KL(q \parallel p)$, which penalizes q having mass where p has none
    - When p is large where q is small, the KL divergence is small
    - When p is small where q is large, the KL divergence is large
    - This will compress q so it fits to one of the peaks of p
  - For *forward-KL* (aka *moment projection*), we take $KL(p \parallel q)$, which penalizes q missing mass where p has some
    - When p is large where q is small, the KL divergence is large
    - When p is small where q is large, the KL divergence is small
    - This will stretch out q to cover all the peaks of p
  - The choice of which KL divergence to optimize leads to different fits
    - In practice however we normally use reverse KL for computational reasons

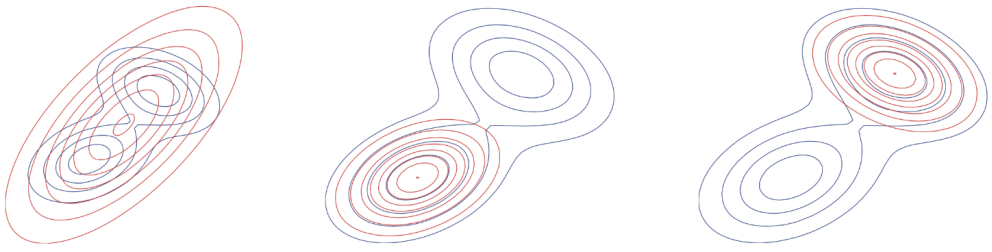

Figure 22: Approximating a bimodal distribution by a unimodal distribution; (a) minimizes forward KL, (b) and (c) minimize reverse KL.

- SVI tries to minimize the KL divergence of p and q

- $KL(q(\boldsymbol{z}|\boldsymbol{\theta}) \parallel p(\boldsymbol{z}|\boldsymbol{x})) = \mathbb{E}_{\boldsymbol{z}\sim q}\left[\log \frac{q(\boldsymbol{z}|\boldsymbol{\theta})}{p(\boldsymbol{z}|\boldsymbol{x})}\right]$
$= \mathbb{E}_{\boldsymbol{z}\sim q}\left[\log\left(q(\boldsymbol{z}|\boldsymbol{\theta})\frac{p(\boldsymbol{x})}{p(\boldsymbol{z},\boldsymbol{x})}\right)\right]$
$= \mathbb{E}_{\boldsymbol{z}\sim q}\left[\log\left(\frac{q(\boldsymbol{z}|\boldsymbol{\theta})}{p(\boldsymbol{z},\boldsymbol{x})}\right]\right) + \log p(\boldsymbol{x})$
$= -\mathcal{L}(\boldsymbol{\theta},\boldsymbol{x}) + \log p(\boldsymbol{x})$
- $\mathcal{L}(\boldsymbol{\theta},\boldsymbol{x}) = -\mathbb{E}_{\boldsymbol{z}\sim q}\left[\log \frac{q(\boldsymbol{z}|\boldsymbol{\theta})}{p(\boldsymbol{z},\boldsymbol{x})}\right]$ is the *evidence lower bound* (ELBO)
- Since $-\mathcal{L}(\boldsymbol{\theta},\boldsymbol{x}) + \log p(\boldsymbol{x}) \geq 0$ (since KL is positive), the ELBO is a lower bound for $\log p(\boldsymbol{x})$
- As $\log p(\boldsymbol{x})$ is constant, to minimize the KL divergence we have to maximize the ELBO; therefore we do not have to compute the normalization, which is infeasible to do

- The ELBO gradient is $\vec{\nabla}_{\boldsymbol{\theta}} = \vec{\nabla}_{\boldsymbol{\theta}} \int q(\boldsymbol{z}|\boldsymbol{\theta}) \log \frac{p(\boldsymbol{x},\boldsymbol{z})}{q(\boldsymbol{z}|\boldsymbol{\theta})}\,\mathrm{d}\boldsymbol{z}$, which must be estimated since we cannot compute this high-dimension integral
  - The *score function* (aka *REINFORCE*) gradient estimator
    - $\vec{\nabla}_{\boldsymbol{\theta}}\mathcal{L}(\boldsymbol{\theta},\boldsymbol{x}) = \mathbb{E}_{\boldsymbol{z}\sim q}\left[\vec{\nabla}_{\boldsymbol{\theta}} \log q(\boldsymbol{z}|\boldsymbol{\theta}) \log \frac{p(\boldsymbol{x},\boldsymbol{z})}{q(\boldsymbol{z}|\boldsymbol{\theta})}\right]$
    - Using Monte Carlo, $\vec{\nabla}_{\boldsymbol{\theta}}\mathcal{L}(\boldsymbol{\theta},\boldsymbol{x}) \approx \frac{1}{B}\sum_{i=1}^{B} \vec{\nabla}_{\boldsymbol{\theta}} \log q(\boldsymbol{z}^{(i)}|\boldsymbol{\theta}) \log \frac{p(\boldsymbol{x},\boldsymbol{z}^{(i)})}{q(\boldsymbol{z}^{(i)}|\boldsymbol{\theta}}$
      - B is the number of samples
      - This is an unbiased estimator and easy to compute
    - In practice, this has higher variance than the pathwise gradient estimator
    - Use in specific domains such as reinforcement learning
  - The *pathwise* (aka *reparametrization*) gradient estimator factors out all the randomness of the distribution into a parameterless fixed source of noise, $p(\boldsymbol{\varepsilon})$
    - Find $T(\boldsymbol{\varepsilon},\boldsymbol{\theta})$ such that for $\boldsymbol{\varepsilon} \sim p(\boldsymbol{\varepsilon})$, then $\boldsymbol{z} = T(\boldsymbol{\varepsilon},\boldsymbol{\theta}) \implies \boldsymbol{z} \sim q(\boldsymbol{z}|\boldsymbol{\theta})$
      - e.g. for a Gaussian, $\boldsymbol{\theta} = \{\mu, \sigma\}$, let $\varepsilon \sim \mathcal{N}(\varepsilon|0,1)$ and $T(\varepsilon,\boldsymbol{\theta}) = \sigma\varepsilon + \mu$, then $z \sim \mathcal{N}(z|\mu,\sigma)$
    - Using the above, $\vec{\nabla}_{\boldsymbol{\theta}}(\boldsymbol{\theta},\boldsymbol{x}) = \mathbb{E}_{\boldsymbol{\varepsilon}\sim p(\boldsymbol{\varepsilon})}\left[\vec{\nabla}_{\boldsymbol{\theta}} \log \frac{p(\boldsymbol{x},T(\boldsymbol{\varepsilon},\boldsymbol{\theta}))}{q(T(\boldsymbol{\varepsilon},\boldsymbol{\theta})|\boldsymbol{\theta})}\right]$
    - This can then be estimated using Monte Carlo
- The main drawback of SVI is the challenge of determining how good the approximation is after the optimization terminates

Monte Carlo and Importance Sampling

- So far we've examined methods of estimating the full distribution $p(\boldsymbol{x})$, but sometimes we're only interested in the expectation of some function $\phi(\boldsymbol{x})$ under the distribution, i.e. $I = \mathbb{E}_{\boldsymbol{x}\sim p(\boldsymbol{x})}[\phi(\boldsymbol{x})]$
- The Monte Carlo approximation is given by $I = \mathbb{E}_{\boldsymbol{x}\sim p(\boldsymbol{x})}[\phi(\boldsymbol{x})] \approx \hat{I} = \frac{1}{R}\sum_{i=1}^{R} \phi(\boldsymbol{x}^{(i)})$
  - This is unbiased, with a standard deviation proportional to $\frac{1}{\sqrt{R}}$, independent of the dimension of $\boldsymbol{x}$
- If we only need the expectation, we only need to be able to sample from the distribution, and apply Monte Carlo to find the expectation
  - However, sampling is hard because we typically only have the unnormalized distribution, $\tilde{p}(\boldsymbol{x}) = Zp(\boldsymbol{x})$; even if we did have the full distribution, sampling from a high-dimension distribution is hard
- *Importance sampling* is a method for approximating the expectation when we only have the unnormalized distribution
  - A notable example is the particle filter for state estimation in robotics
- Let $q(\boldsymbol{x})$ be the *sampler density*, a simpler density function that we can easily sample from

- $I = \int p(\boldsymbol{x})\phi(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}$
  $= \int \phi(\boldsymbol{x})\frac{p(\boldsymbol{x})}{q(\boldsymbol{x})}q(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}$
  $= \frac{\int \frac{\phi(\boldsymbol{x})p(\boldsymbol{x})}{q(\boldsymbol{x})}q(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}}{\int \frac{p(\boldsymbol{x})}{q(\boldsymbol{x})}q(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}}$
  $= \frac{\int \frac{\phi(\boldsymbol{x})\frac{1}{Z}\tilde{p}(\boldsymbol{x})}{q(\boldsymbol{x})}q(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}}{\int \frac{\frac{1}{Z}\tilde{p}(\boldsymbol{x})}{q(\boldsymbol{x})}q(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}}$
  $= \frac{\int \frac{\phi(\boldsymbol{x})p(\boldsymbol{x})}{q(\boldsymbol{x})}q(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}}{\int \frac{\tilde{p}(\boldsymbol{x})}{q(\boldsymbol{x})}q(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}}$
  $= \frac{\mathbb{E}_{\boldsymbol{x}\sim q(\boldsymbol{x})}\left[\frac{\phi(\boldsymbol{x})p(\boldsymbol{x})}{q(\boldsymbol{x})}\right]}{\mathbb{E}_{\boldsymbol{x}\sim q(\boldsymbol{x})}\left[\frac{\tilde{p}(\boldsymbol{x})}{q(\boldsymbol{x})}\right]}$
  - Now we can use Monte Carlo to approximate the expectations
  - $\hat{I} = \frac{\frac{1}{R}\sum_{r=1}^{R}\frac{\phi(\boldsymbol{x}^{(r)})\tilde{p}(\boldsymbol{x}^{(r)})}{q(\boldsymbol{x}^{(r)})}}{\frac{1}{R}\sum_{r=1}^{R}\frac{\tilde{p}(\boldsymbol{x}^{(r)})}{q(\boldsymbol{x}^{(r)})}} = \frac{\sum_r w_r\phi(\boldsymbol{x}^{(r)})}{\sum_r w_r}$
  - Each $w_r = \frac{\tilde{p}(\boldsymbol{x}^{(r)})}{q(\boldsymbol{x}^{(r)})}$ is referred to as the *importance weight*
    * Intuitively, if at a point $p(\boldsymbol{x}^{(r)}) > q(\boldsymbol{x}^{(r)})$, then sampling from q will under-represent this point; therefore the points are weighted more in the sum, since w_r will be larger
    * Conversely $p(\boldsymbol{x}^{(r)}) < q(\boldsymbol{x}^{(r)})$ means q over-represents the point, so in this case w_r will be small and less weight is applied to it
    * When $p(\boldsymbol{x}^{(r)}) = q(\boldsymbol{x}^{(r)})$ we can show that $\hat{I}$ applies no reweighing to samples
- The sampler density should have heavy tails (e.g. a Cauchy distribution instead of a Gaussian), since we need to compensate for the difference between distribution
- If the sampler is chosen improperly, the variance of the result can be extremely high
- In high dimensions, if the sampler distribution is not a near-perfect approximation of the target, then the entire sum will likely be dominated by a few samples with a huge weight, leading to a very bad estimate