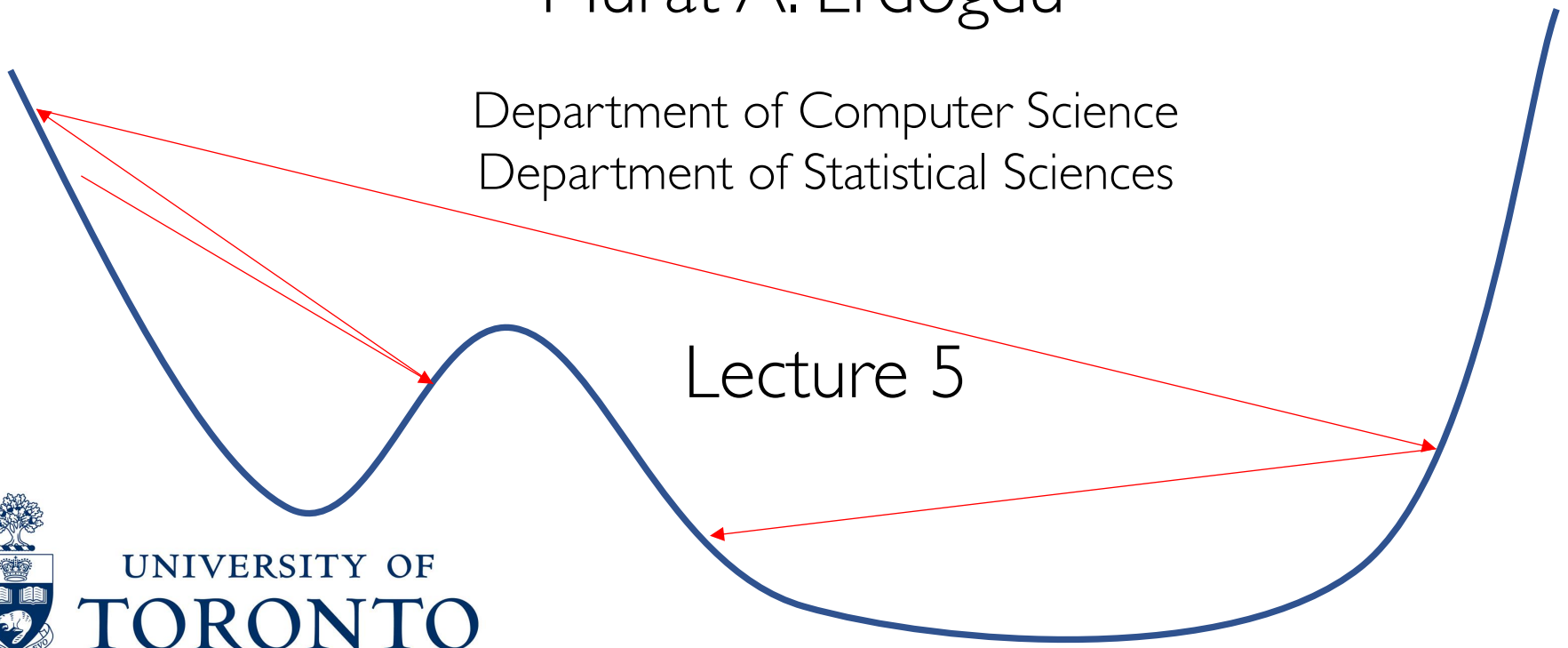
STA414/2104

Statistical Methods for Machine Learning II

Murat A. Erdogdu

Department of Computer Science
Department of Statistical Sciences



Lecture 5

UNIVERSITY OF TORONTO

Announcements

- Homework 2 is released, due on Feb 22.
  - TA Ohs will be announced.
  - Hw is not long! Each function in the starter code needs a few lines of code.
  - Avoid for loops as much as possible.
- No class next week (reading week).

Last time

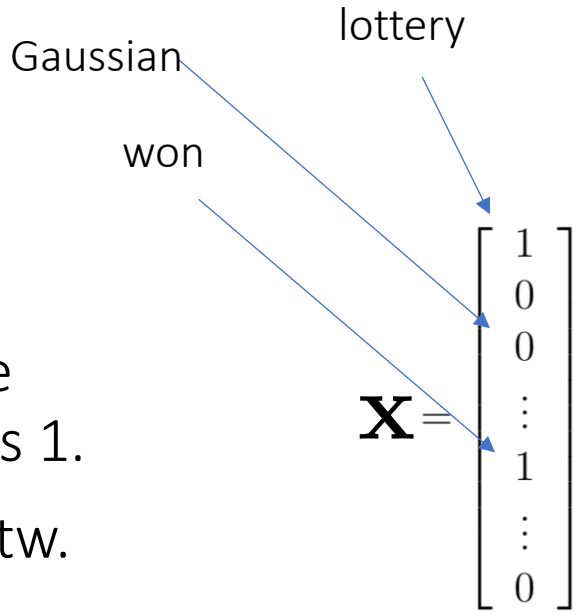
- Linear models for classification
- Least squares
- Logistic regression (2-class and multi-class)
- Fisher's LDA and QDA

Today

- Naïve Bayes classifier
- Statistical decision theory
- Bias-variance decomposition
- Optimization in ML

Back to generative models

Naïve Bayes



- Consider a spam filter example.
  - Goal: Classify if an email is spam or not.
- Features: if a word indexed by j appears in the email, corresponding entry of feature vector is 1.
- Target: 1 if the email is classified as spam, 0 otw.
- x is very high dimensional, say d, depending on the size of vocabulary.
- Modeling $p(\mathbf{x}|y)$ is not easy (2^{d+1} -2 parameters w/o prior).
- **Naïve Bayes assumption:** Conditioned on y, x_i is independent from x_j.
- Then, $p(\mathbf{x}|y) = \prod_{i=1}^{d} p(x_i|y)$ (2d parameters w/o prior)

Naïve Bayes

$$p(\mathbf{x}|y) = \prod_{i=1}^{d} p(x_i|y) \qquad p(y=1) = \pi \ \text{ and } \ p(y=0) = 1-\pi$$

- Now let $p(x_i = 1|y) = \mu_{yi}$, then

$$p(x_i|y=0) = \mu_{0i}^{x_i}(1-\mu_{0i})^{1-x_i}$$
$$p(x_i|y=1) = \mu_{1i}^{x_i}(1-\mu_{1i})^{1-x_i}$$
$$p(y) = \pi^y(1-\pi)^{1-y}$$

- Joint density of y and x, for one sample:

$$p(y,\mathbf{x}) = p(\mathbf{x}|y)p(y) = \prod_{i=1}^{d} p(x_i|y)p(y)$$
$$= \pi^y(1-\pi)^{1-y}\prod_{i=1}^{d}\mu_{yi}^{x_i}(1-\mu_{yi})^{1-x_i}$$

- Now observe m samples (emails), $(y_n, \mathbf{x}_n)$ for n =1,…..,m.
- The joint density of all samples:

$$L(\{\mu_{0i},\mu_{1i}\}_{i=1}^{d},\pi;\{y_n,\mathbf{x}_n\}_{n=1}^{m}) = \prod_{n=1}^{m}\pi^{y_n}(1-\pi)^{1-y_n}\prod_{i=1}^{d}\mu_{y_n i}^{x_{ni}}(1-\mu_{y_n i})^{1-x_{ni}}$$

Naïve Bayes

- Now observe m samples (emails), $(y_n, \mathbf{x}_n)$ for n =1,…..,m.
- We maximize over the parameters and then compute the posterior for prediction.

$$L(\{\mu_{0i}, \mu_{1i}\}_{i=1}^{d}, \pi; \{y_n, \mathbf{x}_n\}_{n=1}^{m}) = \prod_{n=1}^{m} \pi^{y_n}(1-\pi)^{1-y_n} \prod_{i=1}^{d} \mu_{y_n i}^{x_{ni}}(1-\mu_{y_n i})^{1-x_{ni}}$$

- After rearranging and taking logs, and maximizing the log-likelihood, k=0,1

$$\hat{\mu}_{ki} = \frac{\sum_{n=1}^{m} 1\{x_{ni} = 1 \text{ and } y_n = k\}}{\sum_{n=1}^{m} 1\{y_n = k\}} \qquad \hat{\pi} = \frac{\sum_{n=1}^{m} 1\{y_n = 1\}}{m}$$

(Verify!)

- Generalization to multi-class case is straightforward, by simply using multinomial.

Naïve Bayes predictions

- Now that we have the estimators for the parameters, how do we make predictions?
- When making predictions, we use the **posterior**:

$$p(y|\mathbf{x}) \propto \quad p(y,\mathbf{x}) = p(\mathbf{x}|y)p(y) = \prod_{i=1}^{d} p(x_i|y)p(y)$$
$$= \pi^y(1-\pi)^{1-y} \prod_{i=1}^{d} \mu_{yi}^{x_i}(1-\mu_{yi})^{1-x_i}$$

- Assign the input **x** to the class that has the largest posterior.
- Even though relies on a strong assumption, works quite well in practice.

Statistical Decision Theory

• We now develop a small amount of theory that provides a framework for understanding many of the models we consider.

• Suppose we have a real-valued input vector **x** and a corresponding target (output) value t with joint probability distribution: $p(\mathbf{x}, t)$.

• Our goal is to predict target t given a new value for **x**:
- for regression: t is a real-valued continuous target.
- for classification: t is a categorical variable representing class labels.

The joint probability distribution $p(\mathbf{x}, t)$ provides a complete summary of uncertainties associated with these random variables.

Determining $p(\mathbf{x}, t)$ from training data is an example of the inference problem.

Example: Classification

Medical diagnosis: Based on the X-ray image, we would like determine whether the patient has cancer or not.

• The input vector **x** is the set of pixel intensities, and the output variable t will represent the presence of cancer, class C_1, or absence of cancer, class C_2.

C_1: Cancer present

C_2: Cancer absent

x -- set of pixel intensities

• Choose t to be binary: t=0 correspond to class C_1, and t=1 corresponds to C_2.

Inference Problem: Determine the joint distribution $p(\mathbf{x}, \mathcal{C}_k)$ or equivalently $p(\mathbf{x}, t)$. However, in the end, we must make a decision of whether to give treatment to the patient or not.

Example: Classification

Informally: Given a new X-ray image, our goal is to decide which of the two classes that image should be assigned to.

• We could compute conditional probabilities of the two classes, given the input image:

posterior probability of C_k given observed data.

probability of observed data given C_k

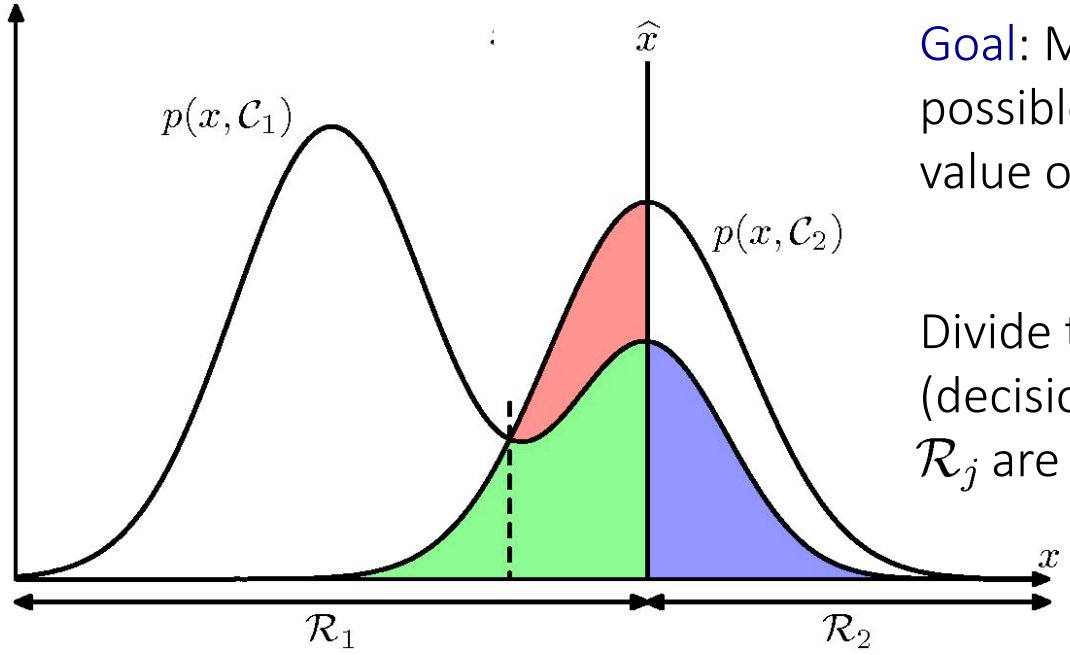
prior probability for class C_k

$$p(\mathcal{C}_k|\mathbf{x}) = \frac{p(\mathbf{x},\mathcal{C}_k)}{\sum_{k=1}^{K} p(\mathbf{x},\mathcal{C}_k)} = \frac{p(\mathbf{x}|\mathcal{C}_k)p(\mathcal{C}_k)}{p(\mathbf{x})}$$

Bayes' Rule

• If our goal is to minimize the probability of assigning **x** to the wrong class, then we should choose the class having the highest posterior probability.

Minimizing Misclassification Rate



Goal: Make as few misclassifications as possible. We need a rule that assigns each value of **x** to one of the available classes.

Divide the input space into regions $\mathcal{R}_j$ (decision regions), such that all points in $\mathcal{R}_j$ are assigned to class $\mathcal{C}_j$.

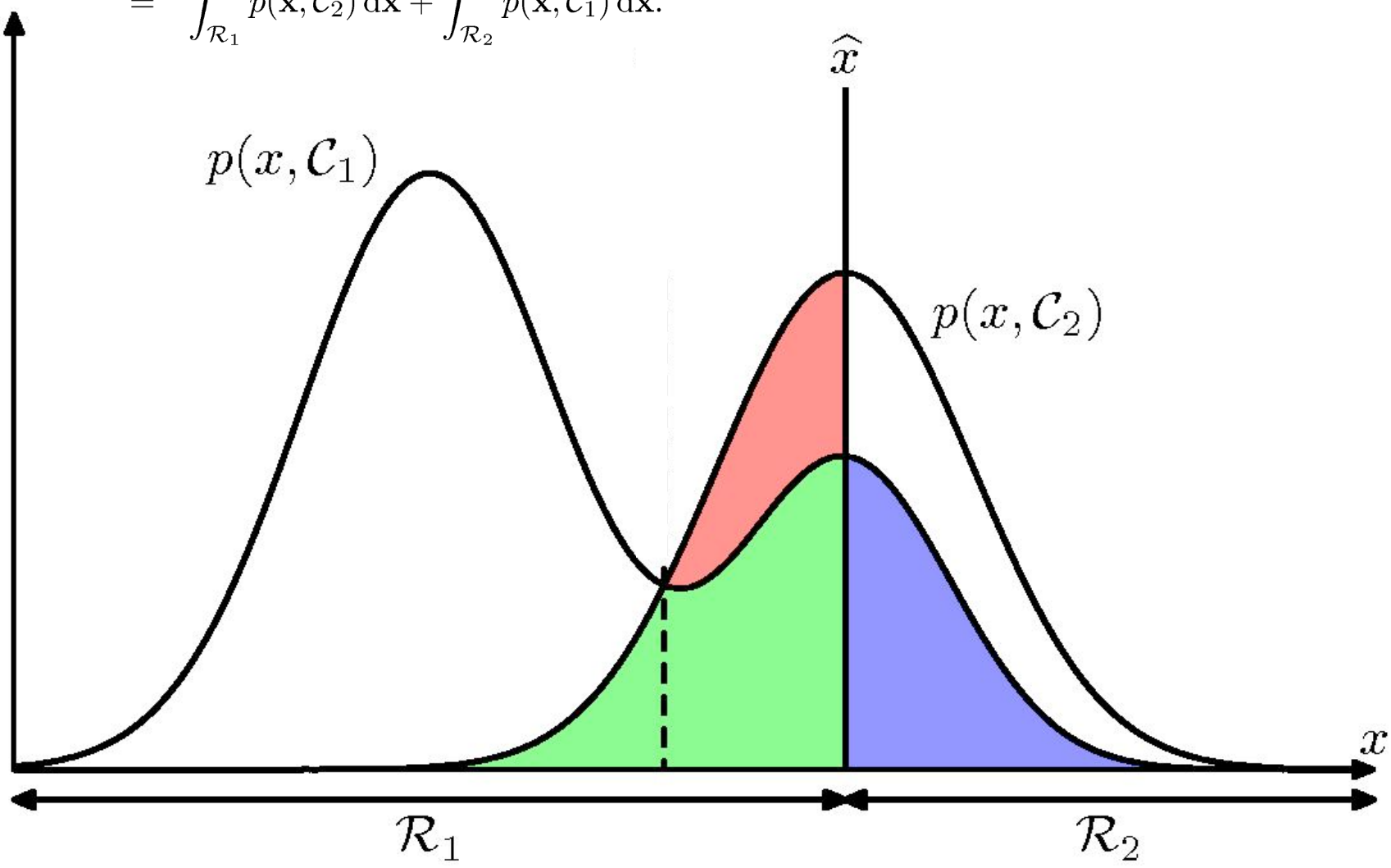
Red + green regions: input belongs to class C_2, but is assigned to C_1

blue region: input belongs to class C_1, but is assigned to C_2

$$
\begin{aligned}
p(\text{mistake}) &= p(\mathbf{x} \in \mathcal{R}_1, \mathcal{C}_2) + p(\mathbf{x} \in \mathcal{R}_2, \mathcal{C}_1) \\
&= \int_{\mathcal{R}_1} p(\mathbf{x}, \mathcal{C}_2)\, \mathrm{d}\mathbf{x} + \int_{\mathcal{R}_2} p(\mathbf{x}, \mathcal{C}_1)\, \mathrm{d}\mathbf{x}.
\end{aligned}
$$

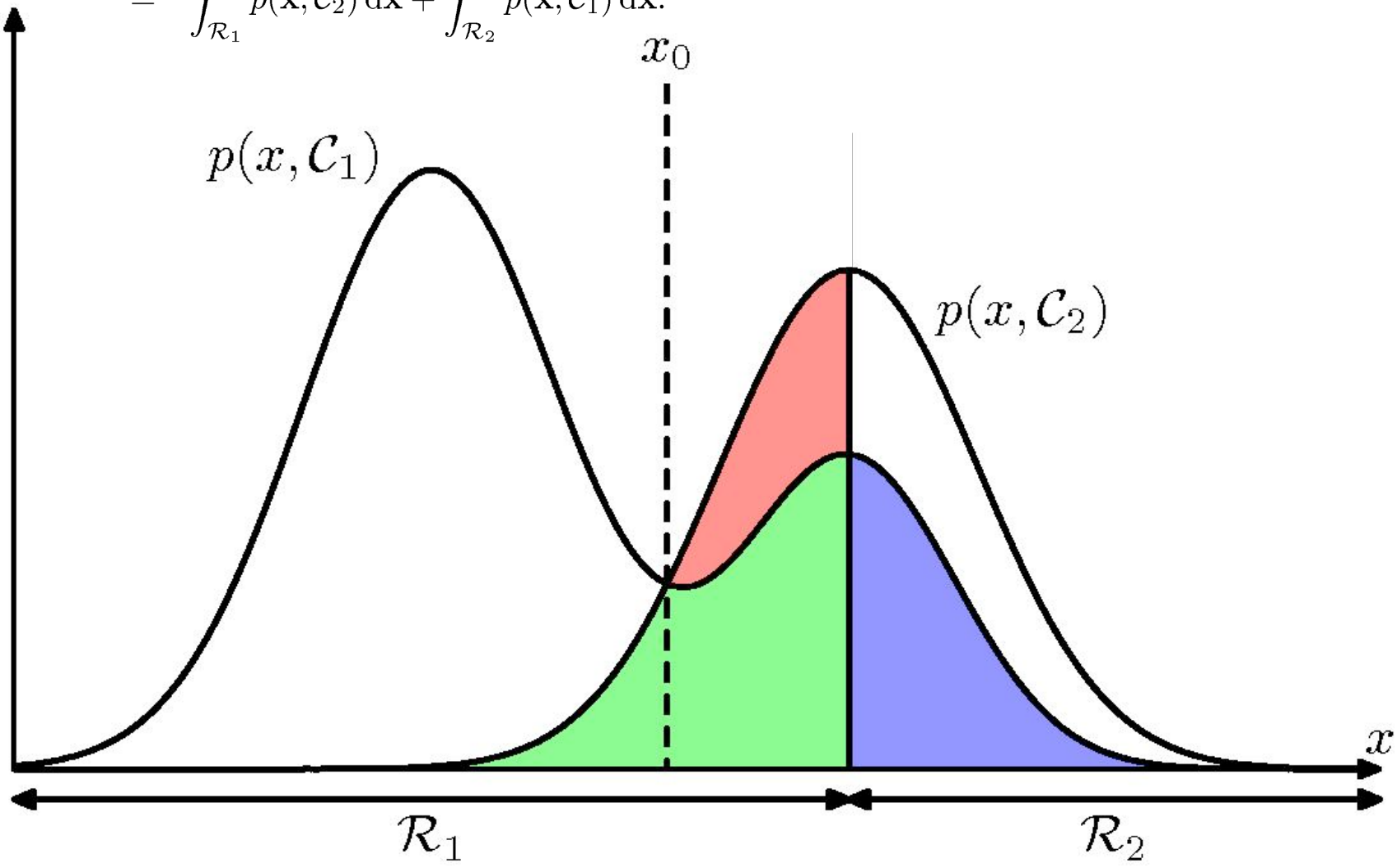
Minimizing Misclassification Rate

$$
\begin{aligned}
p(\text{mistake}) &= p(\mathbf{x} \in \mathcal{R}_1, \mathcal{C}_2) + p(\mathbf{x} \in \mathcal{R}_2, \mathcal{C}_1) \\
&= \int_{\mathcal{R}_1} p(\mathbf{x}, \mathcal{C}_2)\, \mathrm{d}\mathbf{x} + \int_{\mathcal{R}_2} p(\mathbf{x}, \mathcal{C}_1)\, \mathrm{d}\mathbf{x}.
\end{aligned}
$$

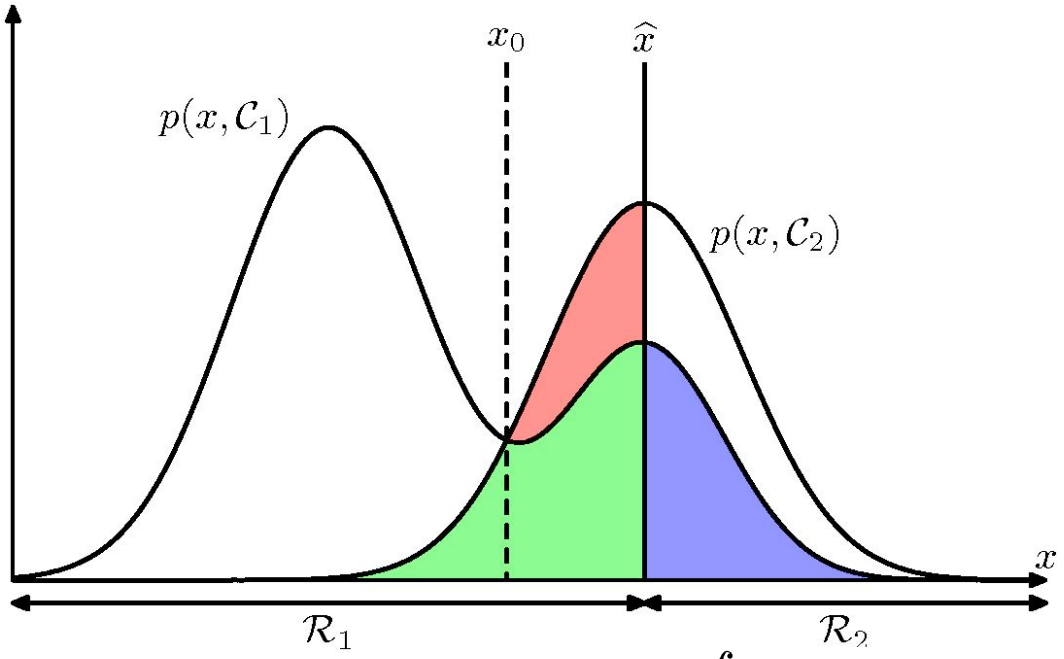


Minimizing Misclassification Rate

$$
\begin{aligned}
p(\text{mistake}) &= p(\mathbf{x} \in \mathcal{R}_1, \mathcal{C}_2) + p(\mathbf{x} \in \mathcal{R}_2, \mathcal{C}_1) \\
&= \int_{\mathcal{R}_1} p(\mathbf{x}, \mathcal{C}_2)\, \mathrm{d}\mathbf{x} + \int_{\mathcal{R}_2} p(\mathbf{x}, \mathcal{C}_1)\, \mathrm{d}\mathbf{x}.
\end{aligned}
$$



Minimizing Misclassification Rate



$$p(\text{mistake}) = p(\mathbf{x} \in \mathcal{R}_1, \mathcal{C}_2) + p(\mathbf{x} \in \mathcal{R}_2, \mathcal{C}_1) = \int_{\mathcal{R}_1} p(\mathbf{x}, \mathcal{C}_2)\mathrm{d}\mathbf{x} + \int_{R_2} p(\mathbf{x}, \mathcal{C}_1)\mathrm{d}\mathbf{x}$$

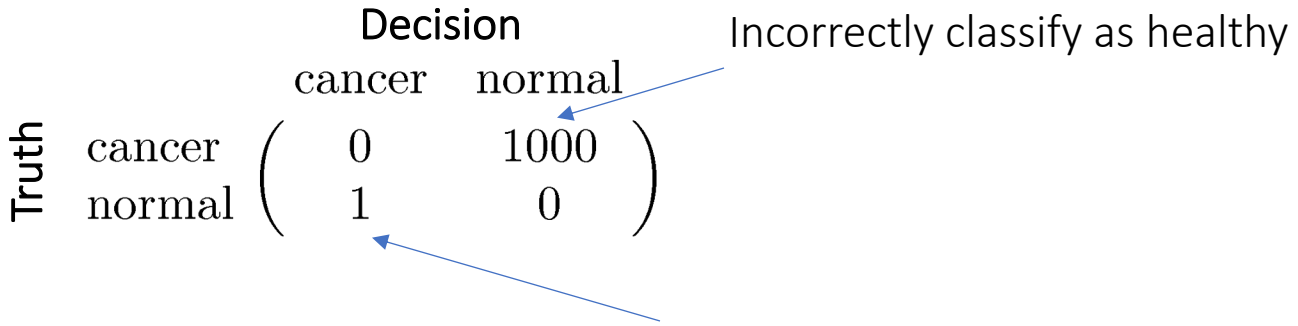
if $p(\mathbf{x}, \mathcal{C}_1) > p(\mathbf{x}, \mathcal{C}_2)$ then we should assign $\mathbf{x}$ to class $\mathcal{C}_1$.

Using $p(\mathbf{x}, \mathcal{C}_k) = p(\mathcal{C}_k|\mathbf{x})p(\mathbf{x})$: To minimize the probability of making mistake, we assign each **x** to the class for which the posterior probability $p(\mathcal{C}_k|\mathbf{x})$ is largest.

Expected Loss

- But misclassification rate may not be what we want to minimize.
- Loss Function: measure of loss incurred by taking any of the available decisions.
- Suppose that for **x**, the true class is C_k, but we assign **x** to class C_j and incur loss of L_{kj} (k,j element of a loss matrix).

Consider medical diagnosis example: example of a loss matrix:



Expected Loss:

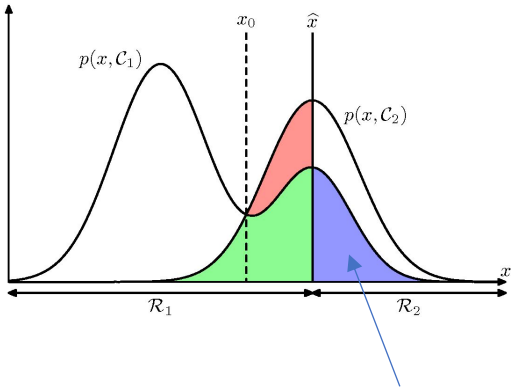
$$\mathbb{E}[L] = \sum_k \sum_j \int_{\mathcal{R}_j} L_{kj} p(\mathbf{x}, \mathcal{C}_k)\, \mathrm{d}\mathbf{x}$$

Goal is to choose regions $\mathcal{R}_j$ as to minimize expected loss.

Expected Loss

Expected Loss:

$$\mathbb{E}[L] = \sum_k \sum_j \int_{\mathcal{R}_j} L_{kj} p(\mathbf{x}, \mathcal{C}_k)\, \mathrm{d}\mathbf{x}$$



L_{12}

Goal is to choose regions $\mathcal{R}_j$ as to minimize expected loss.

$$\mathbb{E}[L] = \sum_j \int_{\mathcal{R}_j} \sum_k L_{kj} p(\mathbf{x}, \mathcal{C}_k) d\mathbf{x}$$

Equivalent to minimizing the following for each **x**: $\sum_k L_{kj} p(\mathbf{x}, \mathcal{C}_k)$

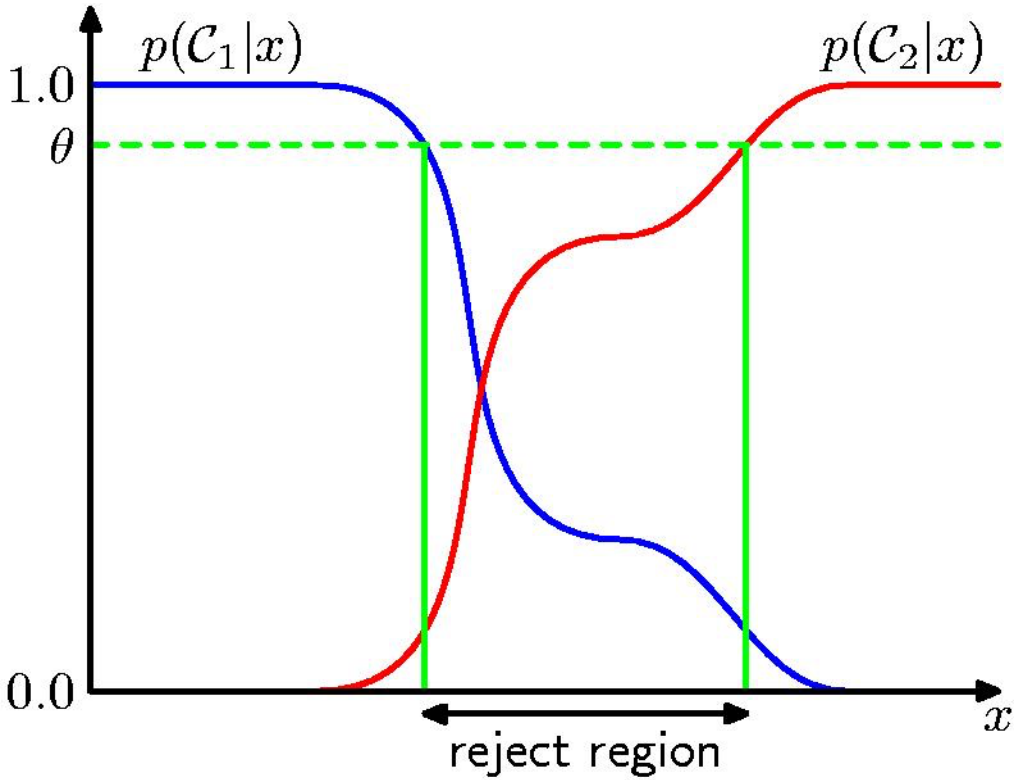
We can also you the product rule $p(\mathcal{C}_k, \mathbf{x}) = p(\mathcal{C}_k|\mathbf{x})p(\mathbf{x})$ and reduce the problem to:
Find regions such that the following is minimized.

$$\sum_k L_{kj} p(\mathcal{C}_k|\mathbf{x})$$

Common in each class

Reject Option

For the regions where we are relatively uncertain about class membership,



you don’t have to make a decision.

Back to Regression

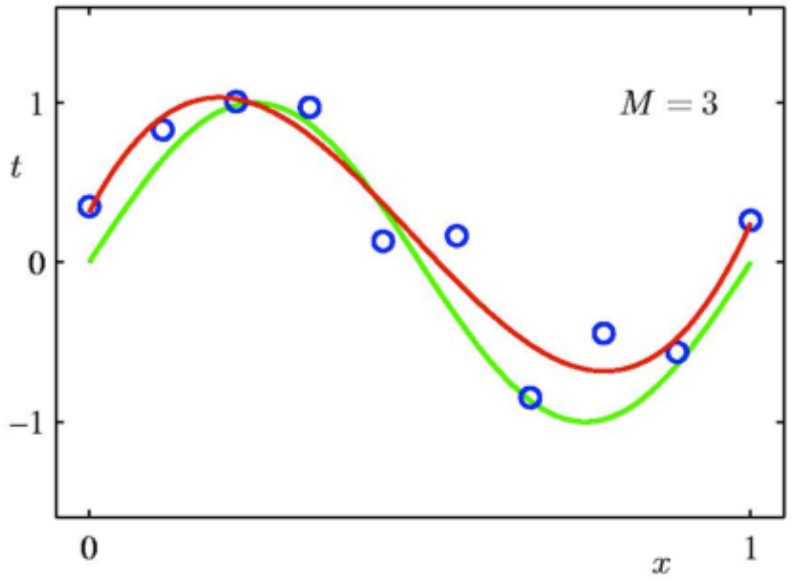
Let **x** in R^d denote a real-valued input vector, and t in R denote a real-valued random target (output) variable with joint the distribution $p(\mathbf{x}, t)$.

- The decision step consists of finding an estimate y(**x**) of t for each input **x**.
- Similar to classification case, to quantify what it means to do well or poorly on a task, we need to define a loss (error) function: $L(t, y(\mathbf{x}))$.
- The average, or expected, loss is given by:

$$\mathbb{E}[L] = \int \int L(t, y(\mathbf{x}))p(\mathbf{x}, t)\mathrm{d}\mathbf{x}\mathrm{d}t.$$

- If we use squared loss, we obtain:

$$\mathbb{E}[L] = \int \int \big(t - y(\mathbf{x})\big)^2 p(\mathbf{x}, t)\mathrm{d}\mathbf{x}\mathrm{d}t.$$



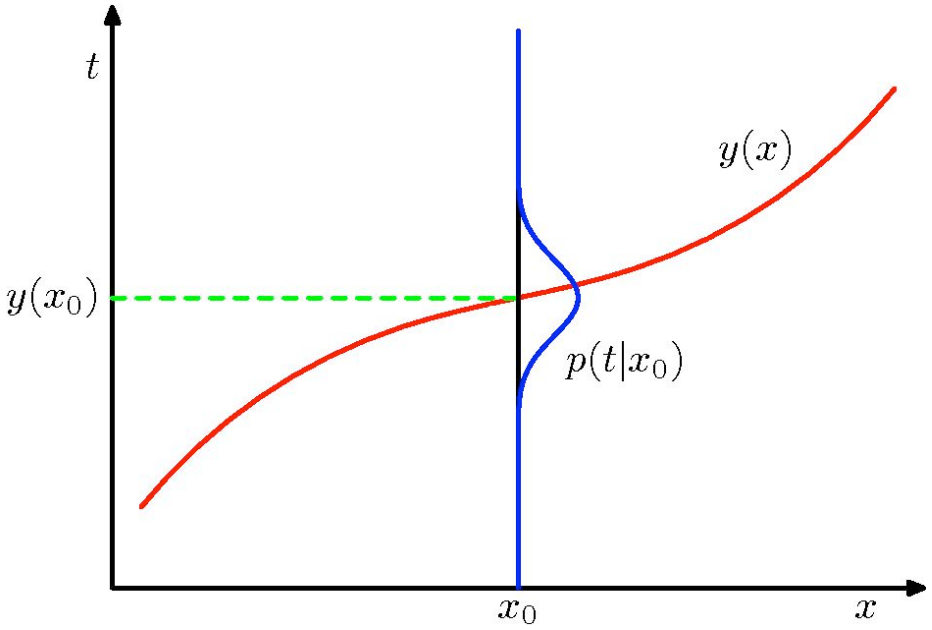
Squared Loss Function

- If we use squared loss, we obtain:

$$\mathbb{E}[L] = \int \int \big(t - y(\mathbf{x})\big)^2 p(\mathbf{x}, t) \mathrm{d}\mathbf{x} \mathrm{d}t.$$

- Our goal is to choose y(x) so as to minimize the expected squared loss.
- The optimal solution is the conditional average (to be proven shortly):

$$y(\mathbf{x}) = \int t p(t|\mathbf{x}) \mathrm{d}t = \mathbb{E}[t|\mathbf{x}].$$



The regression function y(**x**) that minimizes the expected squared loss is given by the mean of the conditional distribution $p(t|\mathbf{x})$.

$$\mathbb{E}[\mathbb{E}[t|\mathbf{x}]] = \mathbb{E}[t]$$

Squared Loss Function

- If we use squared loss, we obtain:

$$\begin{aligned}(y(\mathbf{x}) - t)^2 &= (y(\mathbf{x}) - \mathbb{E}[t|\mathbf{x}] + \mathbb{E}[t|\mathbf{x}] - t)^2 \\ &= \big(y(\mathbf{x}) - \mathbb{E}[t|\mathbf{x}]\big)^2 + 2\big(y(\mathbf{x}) - \mathbb{E}[t|\mathbf{x}]\big)\big(\mathbb{E}[t|\mathbf{x}] - t\big) + \big(\mathbb{E}[t|\mathbf{x}] - t\big)^2.\end{aligned}$$

- Plugging into expected loss:

$$\mathbb{E}[L] = \int \underbrace{\{y(\mathbf{x}) - \mathbb{E}[t|\mathbf{x}]\}^2 p(\mathbf{x})\,\mathrm{d}\mathbf{x}}_{\text{expected loss is minimized when } y(\mathbf{x}) = \mathbb{E}[t|\mathbf{x}].} + \underbrace{\int \mathrm{var}\,[t|\mathbf{x}]\, p(\mathbf{x})\,\mathrm{d}\mathbf{x}}_{\text{intrinsic variability of the target values.}}$$

Because it is independent of the model, it represents an irreducible minimum value of expected loss.

Bias-Variance Decomposition

• Introducing a regularization term can help us control overfitting. But how can we determine a suitable value of the regularization coefficient?

• Let us examine the expected squared loss function. Remember:

$$\mathbb{E}[L] = \int \{y(\mathbf{x}) - h(\mathbf{x})\}^2 p(\mathbf{x})\,\mathrm{d}\mathbf{x} + \iint \{h(\mathbf{x}) - t\}^2 p(\mathbf{x}, t)\,\mathrm{d}\mathbf{x}\,\mathrm{d}t$$

for which the optimal prediction is given by the conditional expectation:

$$h(\mathbf{x}) = \mathbb{E}[t|\mathbf{x}] = \int t p(t|\mathbf{x})\,\mathrm{d}t.$$

intrinsic variability of the target values: The minimum achievable value of expected loss

• Second term is hopeless, focus on the first one.

Bias-Variance Decomposition

• We compute an estimator for w based on the dataset D, and obtain $y(\mathbf{x}; \mathcal{D})$.

• We next interpret the uncertainly of this estimate through the following thought experiment:

- Suppose we had a large number of datasets, each of size N, where each dataset is drawn independently from $p(\mathbf{x}, t)$.
- For each dataset D, we can obtain a prediction function $y(\mathbf{x}; \mathcal{D})$.
- Different datasets will give different w, so different prediction functions.
- The performance of a particular learning algorithm is then assessed by taking the average over the ensemble of these datasets.

• Let us consider the expression:

$$\{y(\mathbf{x}; \mathcal{D}) - h(\mathbf{x})\}^2.$$

• Note that this quantity depends on a particular dataset D.

Bias-Variance Decomposition

- Consider:

$$\{y(\mathbf{x};\mathcal{D}) - h(\mathbf{x})\}^2.$$

Expectation is over data

- Adding and subtracting the term $\mathbb{E}_{\mathcal{D}}[y(\mathbf{x};\mathcal{D})]$, we obtain

$$\begin{aligned}
&\{y(\mathbf{x};\mathcal{D}) - h(\mathbf{x})\}^2 \\
&= \{y(\mathbf{x};\mathcal{D}) - \mathbb{E}_{\mathcal{D}}[y(\mathbf{x};\mathcal{D})] + \mathbb{E}_{\mathcal{D}}[y(\mathbf{x};\mathcal{D})] - h(\mathbf{x})\}^2 \\
&= \{y(\mathbf{x};\mathcal{D}) - \mathbb{E}_{\mathcal{D}}[y(\mathbf{x};\mathcal{D})]\}^2 + \{\mathbb{E}_{\mathcal{D}}[y(\mathbf{x};\mathcal{D})] - h(\mathbf{x})\}^2 \\
&\quad +2\{y(\mathbf{x};\mathcal{D}) - \mathbb{E}_{\mathcal{D}}[y(\mathbf{x};\mathcal{D})]\}\{\mathbb{E}_{\mathcal{D}}[y(\mathbf{x};\mathcal{D})] - h(\mathbf{x})\}.
\end{aligned}$$

Bias-Variance Decomposition

- Consider:

$$\{y(\mathbf{x};\mathcal{D}) - h(\mathbf{x})\}^2.$$

- Adding and subtracting the term $\mathbb{E}_{\mathcal{D}}[y(\mathbf{x};\mathcal{D})]$, we obtain

$$\begin{aligned}
&\{y(\mathbf{x};\mathcal{D}) - h(\mathbf{x})\}^2 \\
&= \{y(\mathbf{x};\mathcal{D}) - \mathbb{E}_{\mathcal{D}}[y(\mathbf{x};\mathcal{D})] + \mathbb{E}_{\mathcal{D}}[y(\mathbf{x};\mathcal{D})] - h(\mathbf{x})\}^2 \\
&= \{y(\mathbf{x};\mathcal{D}) - \mathbb{E}_{\mathcal{D}}[y(\mathbf{x};\mathcal{D})]\}^2 + \{\mathbb{E}_{\mathcal{D}}[y(\mathbf{x};\mathcal{D})] - h(\mathbf{x})\}^2 \\
&\quad +2\{y(\mathbf{x};\mathcal{D}) - \mathbb{E}_{\mathcal{D}}[y(\mathbf{x};\mathcal{D})]\}\{\mathbb{E}_{\mathcal{D}}[y(\mathbf{x};\mathcal{D})] - h(\mathbf{x})\}.
\end{aligned}$$

- Taking the expectation over $\mathcal{D}$, the last term vanishes, so we get:

$$\begin{aligned}
&\mathbb{E}_{\mathcal{D}}\left[\{y(\mathbf{x};\mathcal{D}) - h(\mathbf{x})\}^2\right] \\
&= \underbrace{\{\mathbb{E}_{\mathcal{D}}[y(\mathbf{x};\mathcal{D})] - h(\mathbf{x})\}^2}_{(\text{bias})^2} + \underbrace{\mathbb{E}_{\mathcal{D}}\left[\{y(\mathbf{x};\mathcal{D}) - \mathbb{E}_{\mathcal{D}}[y(\mathbf{x};\mathcal{D})]\}^2\right]}_{\text{variance}}.
\end{aligned}$$

Bias-Variance Trade-off

$$\text{expected loss} = (\text{bias})^2 + \text{variance} + \text{noise}$$

Average predictions over all datasets differ from the optimal regression function.

Solutions for individual datasets vary around their averages -- how sensitive is the function to the particular choice of the dataset.

Intrinsic variability of the target values.

(Irreducible error)

$$(\text{bias})^2 = \int \{\mathbb{E}_{\mathcal{D}}[y(\mathbf{x};\mathcal{D})] - h(\mathbf{x})\}^2 p(\mathbf{x})\,\mathrm{d}\mathbf{x}$$

$$\text{variance} = \int \mathbb{E}_{\mathcal{D}}\left[\{y(\mathbf{x};\mathcal{D}) - \mathbb{E}_{\mathcal{D}}[y(\mathbf{x};\mathcal{D})]\}^2\right] p(\mathbf{x})\,\mathrm{d}\mathbf{x}$$

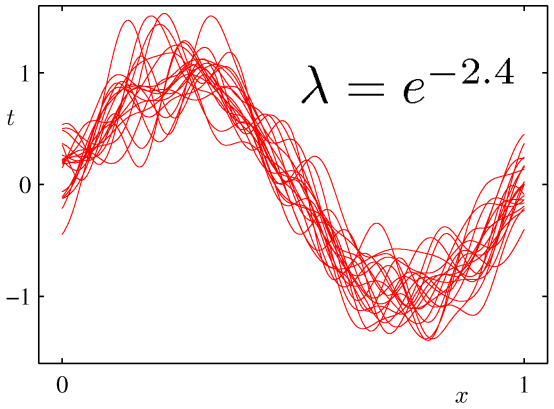
$$\text{noise} = \iint \{h(\mathbf{x}) - t\}^2 p(\mathbf{x},t)\,\mathrm{d}\mathbf{x}\,\mathrm{d}t$$

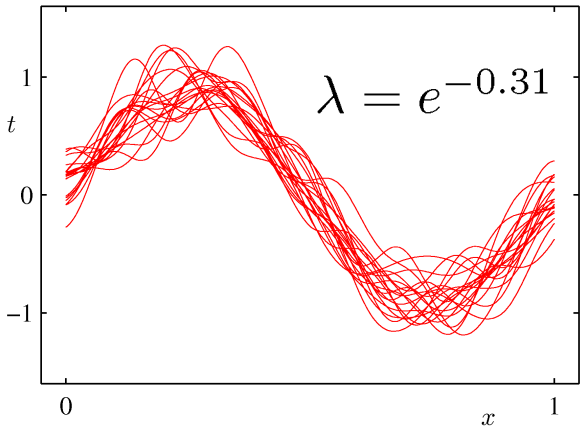
- Trade-off between bias and variance: With very flexible models (high complexity) we have low bias and high variance; With relatively rigid models (low complexity) we have high bias and low variance.
- The model with the optimal predictive capabilities has to balance between bias and variance.

Bias-Variance Trade-off

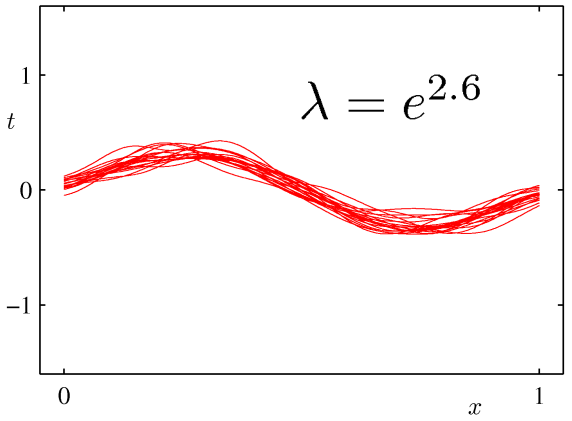
• Consider the sinusoidal dataset. We generate 100 datasets, each containing N=25 points, drawn independently from $h(x) = \sin 2\pi x$.

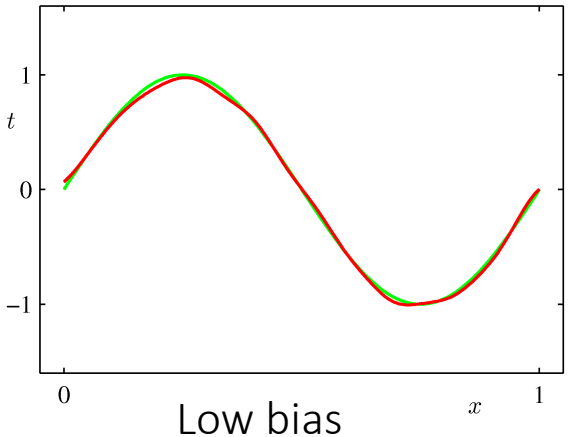
High variance



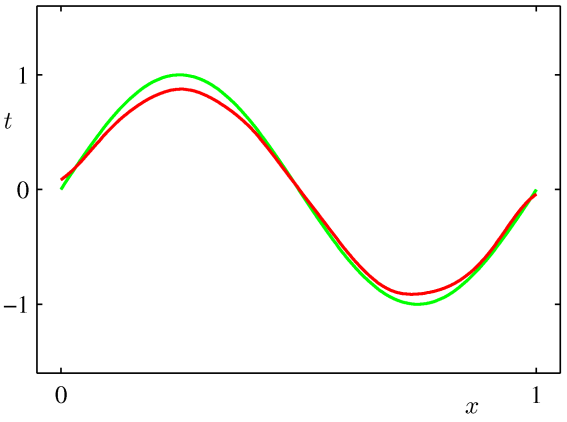


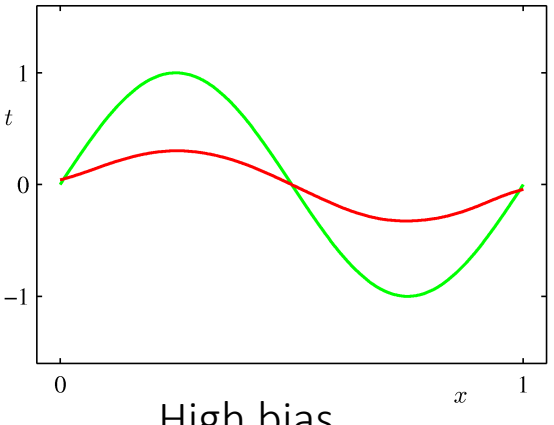
Low variance





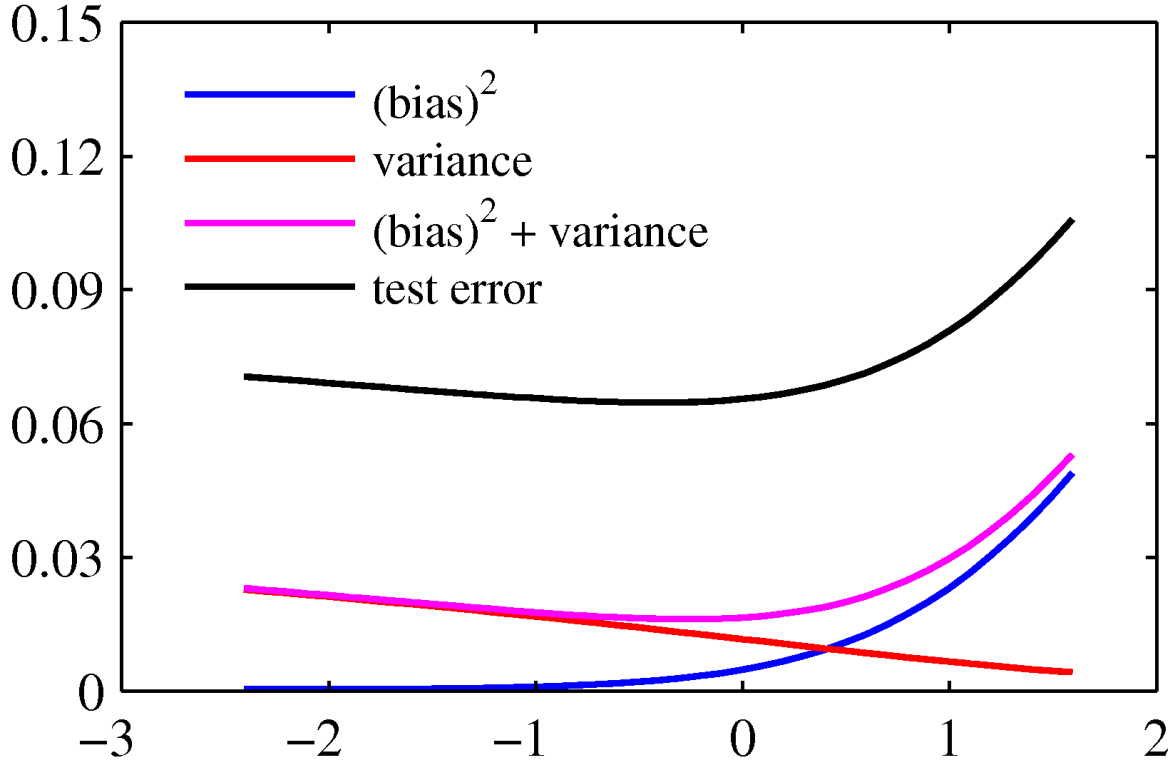
Low bias





High bias

Bias-Variance Trade-off



High model complexity

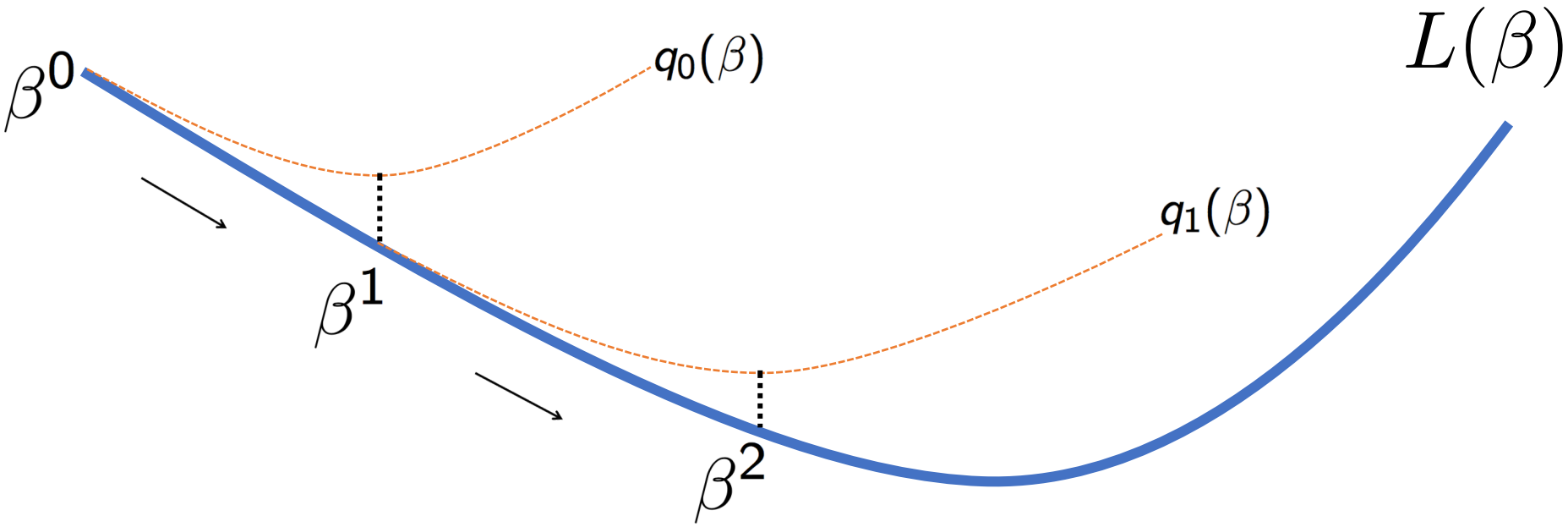
Low model complexity

From these plots note that over-regularized model (large lambda) has high bias, and under-regularized model (low lambda) has high variance.

Beating the Bias-Variance Trade-off

- We can reduce the variance by averaging over many models trained on different datasets:
  - In practice, we only have a single observed dataset. If we had many independent training sets, we would be better off combining them into one large training dataset. With more data, we have less variance.
- Given a standard training set D of size N, we could generate new training sets, N, by sampling examples from D uniformly and with replacement.
  - This is called bagging and it works quite well in practice.

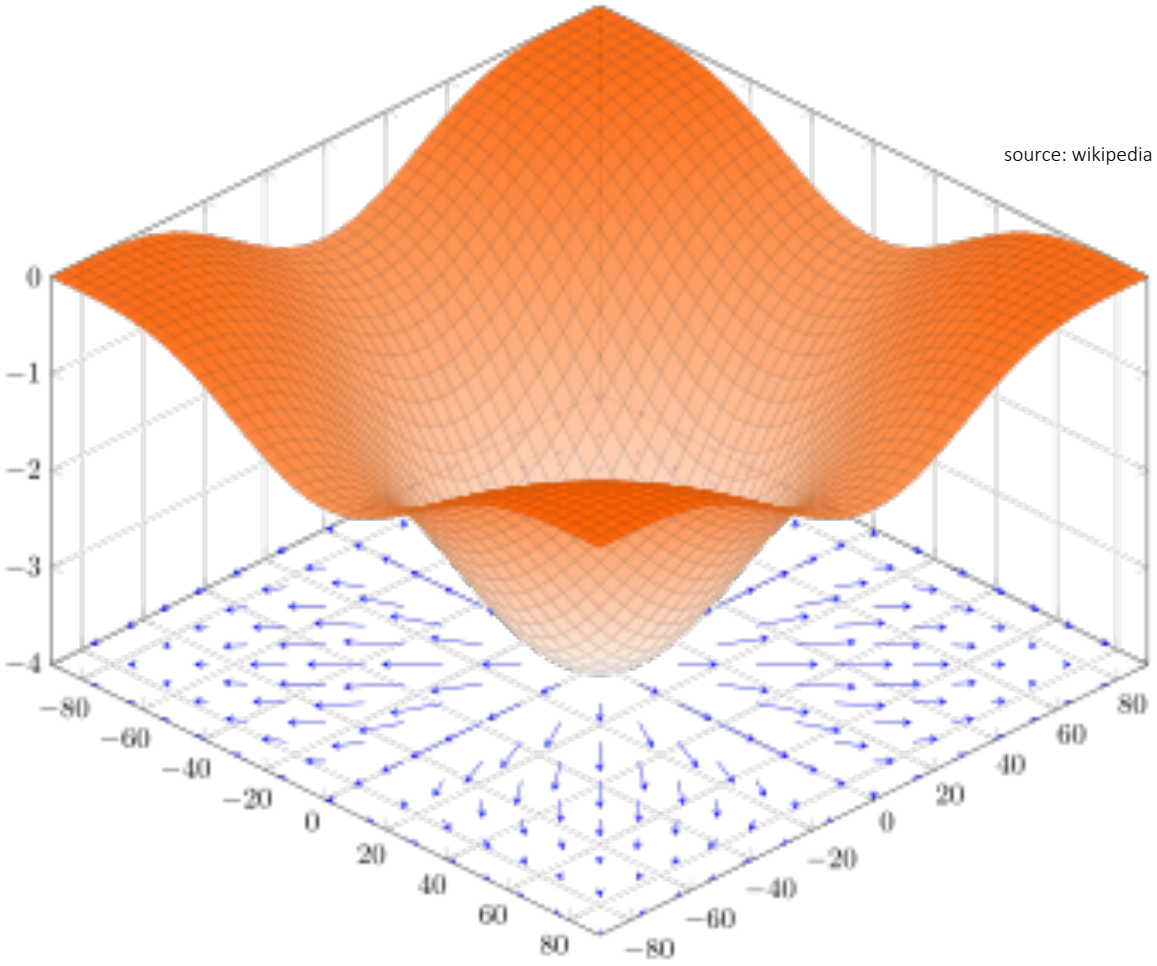
Optimization for Machine Learning



Gradients

$$f(\mathbf{w}) : \mathbb{R}^d \to \mathbb{R}$$

$$\nabla f(\mathbf{w}) = \begin{bmatrix} \partial f(\mathbf{w})/\partial w_1 \\ \partial f(\mathbf{w})/\partial w_2 \\ \vdots \\ \partial f(\mathbf{w})/\partial w_d \end{bmatrix}$$



- Generalization of derivatives in multidimensions.
- It is a vector representing the slope.
- The direction of the gradient points to the greatest rate of increase of the function.
- Its magnitude is the slope of the graph in its direction.

$$f(x, y) = -(\cos(x)^2 + \cos(y)^2)^2$$

$$\nabla f(x, y) = \begin{bmatrix} 4\sin(x)(\cos(x)^2 + \cos(y)^2) \\ 4\sin(y)(\cos(x)^2 + \cos(y)^2) \end{bmatrix}$$

$$= 4(\cos(x)^2 + \cos(y)^2) \begin{bmatrix} \sin(x) \\ \sin(y) \end{bmatrix}$$

Hessian and Jacobian

For a function f: R^n -> R
Hessian is given as

$$\mathbf{H} = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1\,\partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_1\,\partial x_n} \\ \frac{\partial^2 f}{\partial x_2\,\partial x_1} & \frac{\partial^2 f}{\partial x_2^2} & \cdots & \frac{\partial^2 f}{\partial x_2\,\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n\,\partial x_1} & \frac{\partial^2 f}{\partial x_n\,\partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_n^2} \end{bmatrix}.$$

For a function **f**: R^n -> R^m
Jacobian is given as

$$\mathbf{J} = \begin{bmatrix} \frac{\partial \mathbf{f}}{\partial x_1} & \cdots & \frac{\partial \mathbf{f}}{\partial x_n} \end{bmatrix} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \cdots & \frac{\partial f_m}{\partial x_n} \end{bmatrix}$$

Jacobian of the gradient is equal to Hessian, but gradient of vector valued function is equal to its Jacobian transposed. This depends on whether we use column or row vectors to denote derivatives.

What is optimization?

- Typical setup (in machine learning, other areas):
  - Formulate a problem
  - Design a solution (usually a model)
  - Use some quantitative measure to determine how good the solution is.

- E.g., classification:
  - Create a system to classify images
  - Model is some classifier, like logistic regression
  - Quantitative measure is misclassification error (lower is better in this case)

- In almost all cases, you end up with a loss minimization problem of the form

$$\text{minimize}_{\mathbf{w}} E(\mathbf{w})$$

- Ex: least squares

$$\text{minimize} \quad E(\mathbf{w}) \quad = \quad \frac{1}{2}\sum_{n=1}^{N}(\mathbf{x}_n^T\mathbf{w} - t_n)^2$$

Error minimization

- We talked about minimizing sum of squares of errors, or maximizing the log-likelihood.
- The final problem can always be thought of as a error minimization problem, e.g., error in least squares

$$\begin{aligned} E(\mathbf{w}) &= \frac{1}{2}\sum_{n=1}^{N}(\mathbf{x}_n^T\mathbf{w} - t_n)^2 \\ &= \frac{1}{2}(\mathbf{X}\mathbf{w} - \mathbf{t})^{\mathbf{T}}(\mathbf{X}\mathbf{w} - \mathbf{t}). \end{aligned}$$

Error minimization

- Ultimately, training a machine learning model always reduces to solving is an optimization problem

$$\text{minimize}_{\mathbf{w}} E(\mathbf{w})$$

Equivalently, we are interested in finding $\mathbf{w}^* = \text{argmin}_{\mathbf{w}} E(\mathbf{w})$

by using an optimization method.

- Standard approach is **Gradient descent** $\mathbf{w}^{t+1} = \mathbf{w}^t - \eta \nabla E(\mathbf{w}^t)$

where $\eta \in (0, 1]$ is the step size (or learning rate).

- For the least squares, $\text{minimize } E(\mathbf{w}) = \frac{1}{2} \sum_{n=1}^{N} (\mathbf{x}_n^T \mathbf{w} - t_n)^2$
- we have

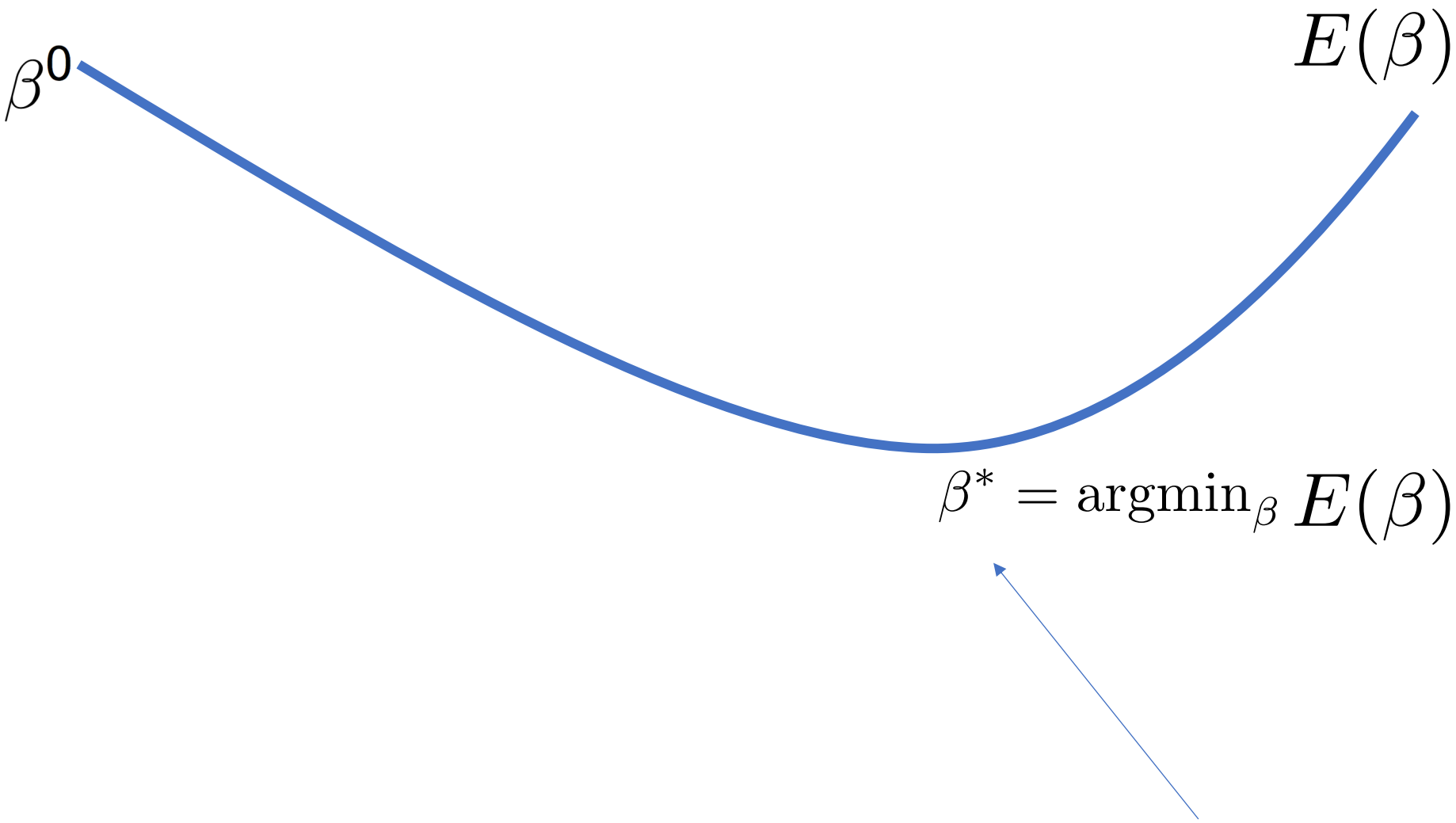
$$\nabla E(\mathbf{w}) = \sum_{n=1}^{N} \mathbf{x}_n (\mathbf{x}_n^T \mathbf{w} - t_n)$$

- We choose an initial point $\mathbf{w}^0$, and do the following iterations

$$\mathbf{w}^{t+1} = \mathbf{w}^t - \eta \nabla E(\mathbf{w}^t)$$

Error minimization

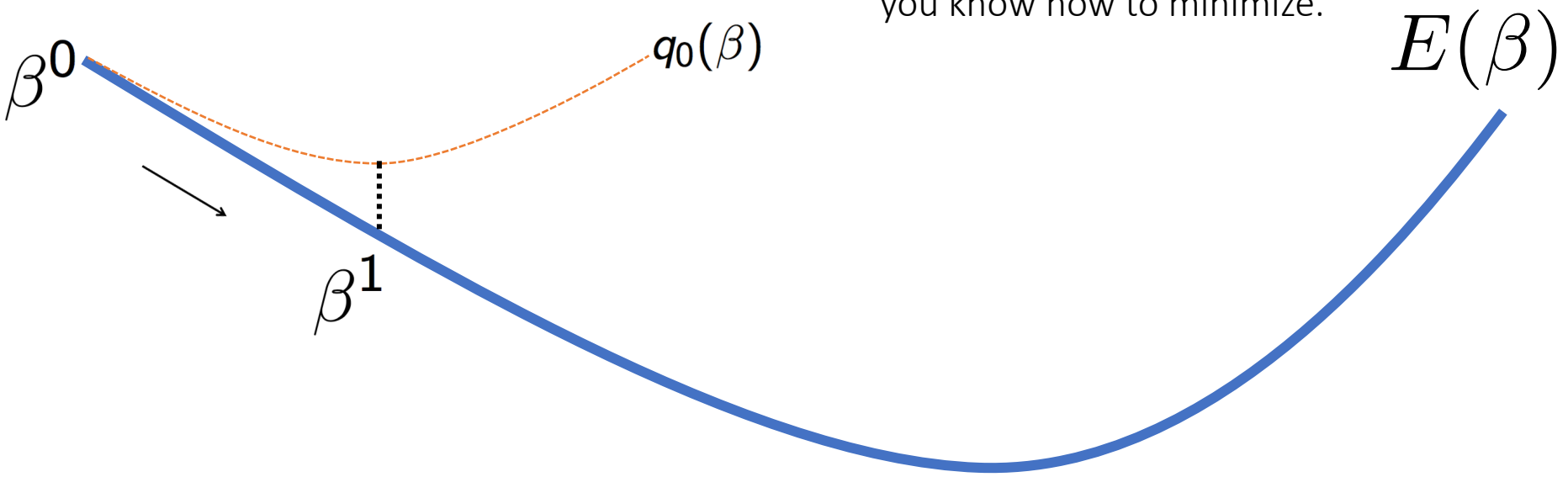
- You start with an initial guess β^0 at t=0



This is the point you want to find, i.e., your estimator, trained weights, coefficients etc.

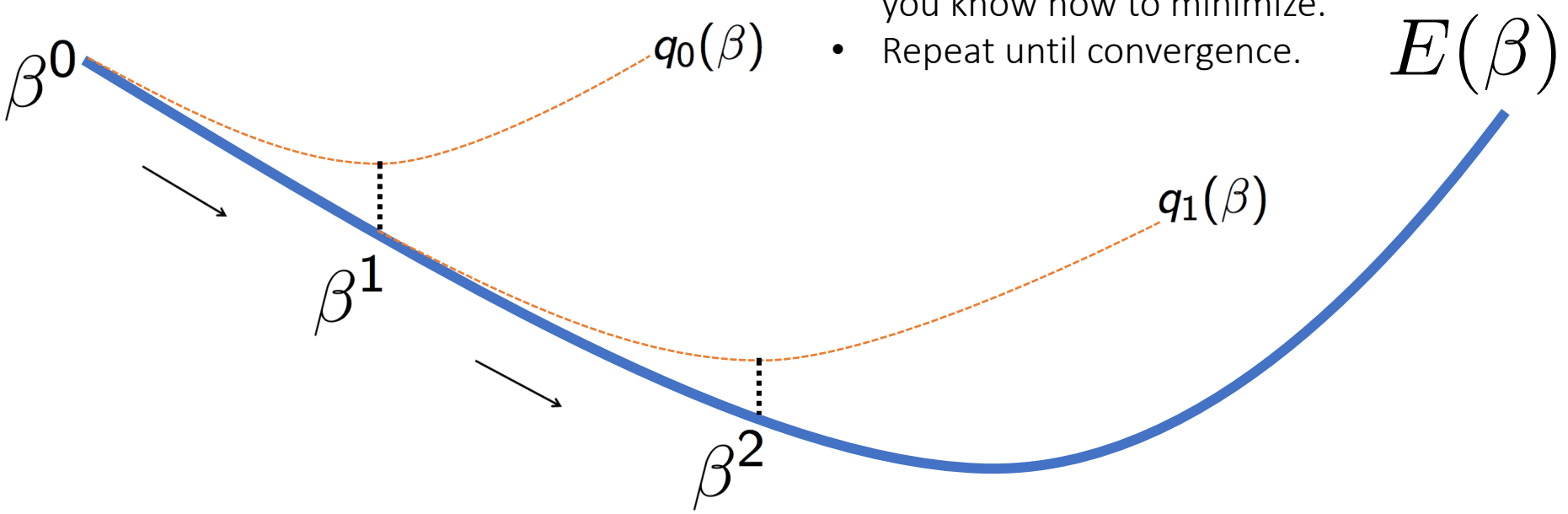
Loss minimization

- You start with an initial guess β^0 at t=0
- At each iteration t, you approximate your loss with a function $q_t(\beta)$ around β^t that you know how to minimize.



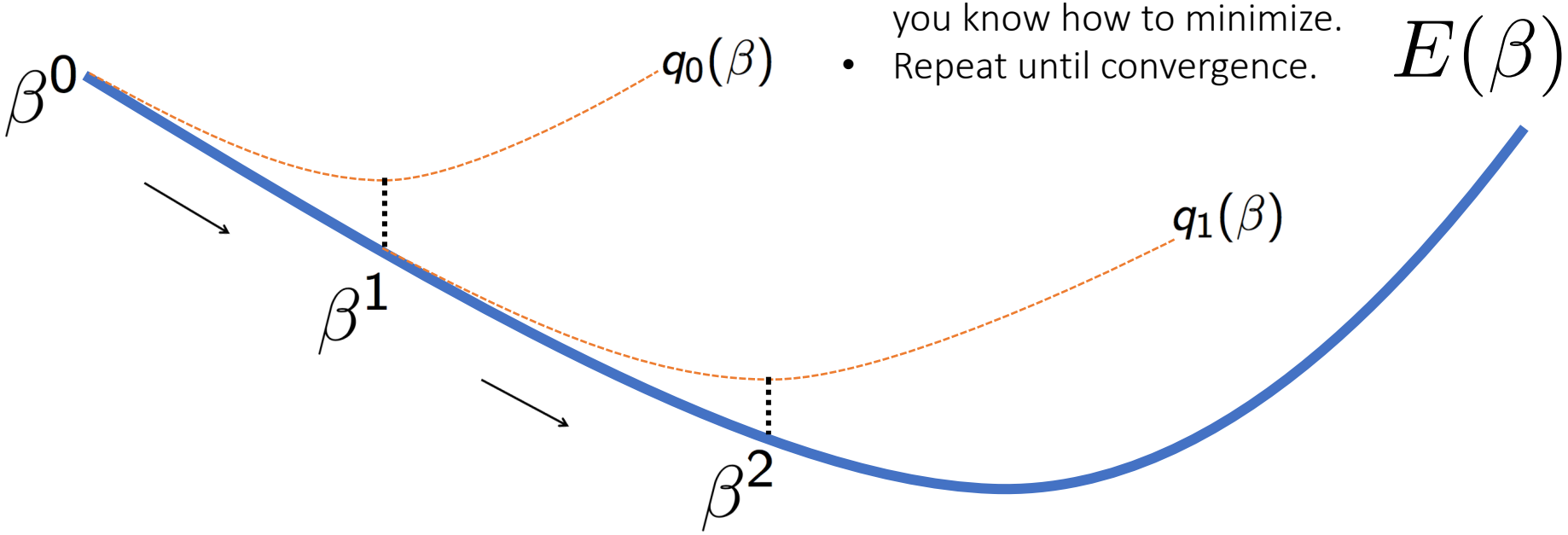
Loss minimization

- You start with an initial guess β^0 at t=0
- At each iteration t, you approximate your loss with a function $q_t(\beta)$ around β^t that you know how to minimize.
- Repeat until convergence.



Loss minimization

- You start with an initial guess β^0 at t=0
- At each iteration t, you approximate your loss with a function $q_t(\beta)$ around β^t that you know how to minimize.
- Repeat until convergence.



$q_t(\beta)$ is a Taylor series approximation of the error function. The update takes the form:

$$\beta^{t+1} = \text{argmin}_\beta q_t(\beta) = \beta^t - [\mathbf{H}^t]^{-1} \nabla E(\beta)$$

Summary: First and Second Order Methods

- $\mathbf{H}^t$ provides curvature information about the optimization landscape and determines the type of optimization method.
- $\mathbf{H}^t = \mathbf{I}$ reduces to gradient descent which is a first order method, i.e., only uses first order derivative information.
- $\mathbf{H}^t = \nabla^2 E(\beta^t)$ reduces to Newton's method which is a second order method, i.e., uses second order derivative information, e.g.

$$\beta^{t+1} = \beta^t - \nabla^2 E(\beta^t)^{-1} \nabla E(\beta^t)$$

  - These methods get faster **convergence rate**, since they use curvature information.
- Computing Hessian is numerically expensive. There are methods that approximate the Hessian (e.g. Quasi-Newton methods).

- The performance of an algorithm is determined by both its convergence rate and per-iteration cost.

Types of optimization

- Simple enough problems have a closed form solution:
  - $E(w) = w^2$
  - Linear regression
  - Still may need to rely on iterations. Why?

- If loss E(w) is convex, then we can use convex optimization techniques (most of machine learning uses these).

- If loss E(w) is non-convex we usually pretend it's convex and find a sub-optimal, but hopefully good enough solution (very active research area) (e.g., deep learning).

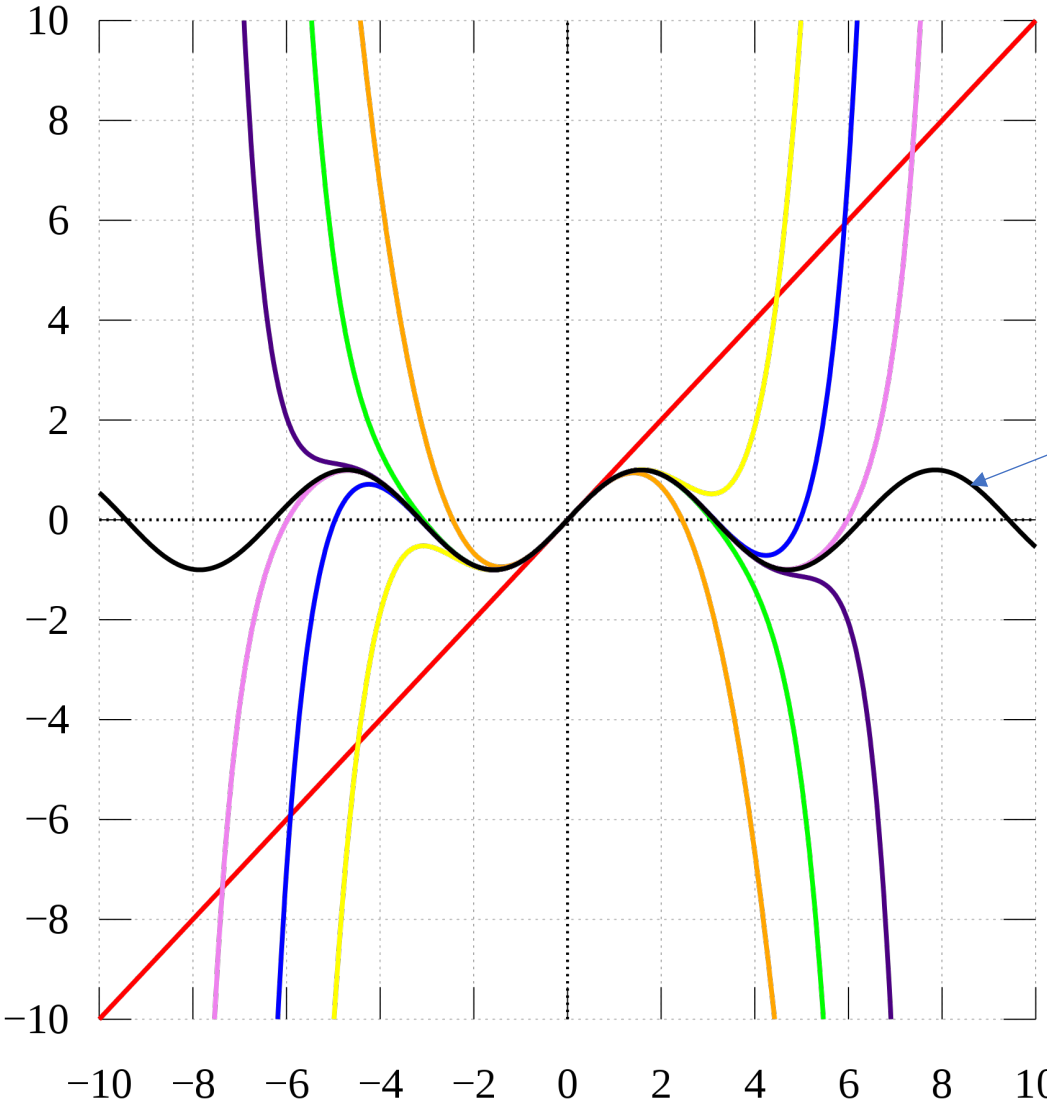
Taylor series

- A Taylor series is a polynomial series that converges to a function f.
- We say that the Taylor series expansion of f(x) around a point 'a', is:

$$f(a) + \frac{f'(a)}{1!}(x-a) + \frac{f''(a)}{2!}(x-a)^2 + \frac{f^{(3)}(a)}{3!}(x-a)^3 + \cdots.$$

- Truncating this series gives a polynomial approximation to a function.
- Approximation is always good around a.

Taylor series



Source:wikipedia

sin x

As the degree of the Taylor polynomial rises, it approaches the correct function.
This image shows sin *x* and its Taylor approximations, polynomials of degree 1, 3, 5, 7, 9, 11, 13.

Gradient descent derivation

- The first-order Taylor series expansion of a function E(w+d) around a point w is:

$$E(\mathbf{w}+\mathbf{d}) \approx E(\mathbf{w}) + \nabla E(\mathbf{w})^\top \mathbf{d}$$

- Suppose we are at **w** and we want to pick a direction **d** such that E(**w** + η**d**) is smaller than E(**w**) for a step size η.
- Using a linear approximation:

$$E(\mathbf{w}+\eta\mathbf{d}) \approx E(\mathbf{w}) + \eta\nabla E(\mathbf{w})^\top \mathbf{d}$$

- **d** should be in the negative direction of $\nabla E(\mathbf{w})$
- This approximation gets better as η gets smaller since as we zoom in on a differentiable function it will look more and more linear.

Gradient descent derivation

- We need to find a direction for d that minimizes

$$E(\mathbf{w}+\eta\mathbf{d}) \approx E(\mathbf{w}) + \eta\nabla E(\mathbf{w})^\top \mathbf{d}$$

- The best direction is $-\nabla E(\mathbf{w})$

This doesn't affect the problem, but it is common in practice to normalize with N

- For the least squares, $\text{minimize} \quad E(\mathbf{w}) = \frac{1}{2N}\sum_{n=1}^{N}(t_n - \mathbf{x}_n^T\mathbf{w})^2$
- we have

$$\nabla E(\mathbf{w}) = \frac{1}{N}\sum_{n=1}^{N}\mathbf{x}_n(\mathbf{x}_n^T\mathbf{w} - t_n)$$

- We choose an initial point $\mathbf{w}^0$, and do the following iterations

$$\mathbf{w}^{t+1} = \mathbf{w}^t - \eta\nabla E(\mathbf{w}^t)$$

How to choose the step size?

- Step size is referred to as learning rate in machine learning.
- It should be always in the interval (0,1).
- The sequence of step sizes is referred to as the learning rate schedule.
- One simple strategy: start with a big η and progressively make it smaller by e.g., halving it until the function decreases.
- There are more formal ways of choosing the step size. But in practice, they are not used for computational reasons.

When does the GD converged?

- When $\|\nabla E(\mathbf{w})\| = 0$

- This is never possible in practice. So we stop iterations if gradient is smaller than a threshold.

- If the function is convex then we have reached a global minimum.

- If the function is not convex, what did we obtain?

- Probably a local minimum or a saddle.

Newton's method

- To speed up convergence, we can use a more accurate approximation.
- Second order Taylor expansion:

$$E(\mathbf{w}+\mathbf{d}) \approx E(\mathbf{w}) + \nabla E(\mathbf{w})^\top \mathbf{d} + \frac{1}{2}\mathbf{d}^\top \nabla^2 E(\mathbf{w})\mathbf{d}$$

- *Hessian* matrix $\nabla^2 E(\mathbf{w})$ contains second derivatives.

$$\nabla^2 E(\mathbf{w})_{ij} = \frac{\partial^2 E(\mathbf{w})}{\partial w_i \partial w_j}$$

Newton's method

Again we are at **w** and we want to pick a direction **d** such that E(**w** + **d**) is smaller than E(**w**).

$$E(\mathbf{w}+\mathbf{d}) \approx E(\mathbf{w}) + \nabla E(\mathbf{w})^\top \mathbf{d} + \frac{1}{2}\mathbf{d}^\top \nabla^2 E(\mathbf{w})\mathbf{d}$$

Take derivatives with respect to d, and set it equal to 0.

$$\nabla E(\mathbf{w}) + \nabla^2 E(\mathbf{w})\mathbf{d} = 0$$

implies $$\mathbf{d} = -\nabla^2 E(\mathbf{w})^{-1} \nabla E(\mathbf{w})$$

The iterations look like

Can add a step size

$$\mathbf{w}^{t+1} = \mathbf{w}^t - \nabla^2 E(\mathbf{w}^t)^{-1} \nabla E(\mathbf{w}^t)$$

What is it doing?

- At each step, Newton's method approximates the function with a quadratic bowl, then goes to the minimum of this bowl.
- For convex functions, this is much faster than gradient descent.
- Con: computing Hessian requires too much computation time and storage (about ND^2 operations).
- Inverting the Hessian is also very expensive (about D^3 operations). This is problematic in high dimensions.
- Newton method gets fast convergence rate, but high per-iteration cost.
- Gradient descent gets slow convergence rate, but low per-iteration cost.

Newton's method

- For the least squares, $\text{minimize } E(\mathbf{w}) = \frac{1}{2}\sum_{n=1}^{N}(\mathbf{x}_n^T\mathbf{w} - t_n)^2$
- we have

$$\nabla E(\mathbf{w}) = \sum_{n=1}^{N}\mathbf{x}_n(\mathbf{x}_n^T\mathbf{w} - t_n) = \mathbf{X}^T\mathbf{X}\mathbf{w} - \mathbf{X}^T\mathbf{t} \qquad \nabla^2 E(\mathbf{w}) = \sum_{n=1}^{N}\mathbf{x}_n\mathbf{x}_n^T = \mathbf{X}^T\mathbf{X}$$

- We choose an initial point $\mathbf{w}^0$, and do the following iterations (step size is 1)

$$\mathbf{w}^{t+1} = \mathbf{w}^t - [\nabla^2 E(\mathbf{w}^t)]^{-1}\nabla E(\mathbf{w}^t)$$

$$\mathbf{w}^1 = \mathbf{w}^0 - [\mathbf{X}^T\mathbf{X}]^{-1}\left(\mathbf{X}^T\mathbf{X}\mathbf{w}^0 - \mathbf{X}^T\mathbf{t}\right)$$

Newton's method

- For the least squares, $\text{minimize } E(\mathbf{w}) = \frac{1}{2}\sum_{n=1}^{N}(\mathbf{x}_n^T\mathbf{w} - t_n)^2$
- we have

$$\nabla E(\mathbf{w}) = \sum_{n=1}^{N}\mathbf{x}_n(\mathbf{x}_n^T\mathbf{w} - t_n) = \mathbf{X}^T\mathbf{X}\mathbf{w} - \mathbf{X}^T\mathbf{t} \qquad \nabla^2 E(\mathbf{w}) = \sum_{n=1}^{N}\mathbf{x}_n\mathbf{x}_n^T = \mathbf{X}^T\mathbf{X}$$

- We choose an initial point $\mathbf{w}^0$, and do the following iterations (step size is 1)

$$\mathbf{w}^{t+1} = \mathbf{w}^t - [\nabla^2 E(\mathbf{w}^t)]^{-1}\nabla E(\mathbf{w}^t)$$

$$\mathbf{w}^1 = \mathbf{w}^0 - [\mathbf{X}^T\mathbf{X}]^{-1}\left(\mathbf{X}^T\mathbf{X}\mathbf{w}^0 - \mathbf{X}^T\mathbf{t}\right)$$

$$= [\mathbf{X}^T\mathbf{X}]^{-1}\mathbf{X}^T\mathbf{t}$$

Wow!

Stochastic Gradient Descent

- In most cases, the minimization is an average over data points:

$$\text{minimize} E(\mathbf{w}) = \frac{1}{N}\sum_{n=1}^{N} L(t_n, y(\mathbf{x}_n, \mathbf{w}))$$

Hard to compute when N is large

Recall that we can write the negative log-likelihood in the above form.

Gradient:

$$\nabla E(\mathbf{w}) = \frac{1}{N}\sum_{n=1}^{N} \nabla L(t_n, y(\mathbf{x}_n, \mathbf{w}))$$

At each iteration, sub-sample a small amount of data and use that to estimate the gradient.

$$\mathbf{w}^{t+1} = \mathbf{w}^{t} - \eta \frac{1}{|S|}\sum_{n\in S} \nabla L(t_n, y(\mathbf{x}_n, \mathbf{w}))$$

Here, |S| denotes the number of elements in the set S.
Standard SGD has |S|= 1, i.e., randomly samples an index
and takes a step based on that sample. |S|>1 is called mini-batch SGD.

Stochastic Gradient Descent

- Any iteration of a gradient descent (or quasi-Newton) method requires that we sum over the entire dataset to compute the gradient.
- SGD idea: at each iteration, sub-sample a small amount of data (even just 1 point can work) and use that to estimate the gradient.
- Each update is noisy, but very fast!
- This is the basis of optimizing ML algorithms with huge datasets (e.g., recent deep learning).
- Computing gradients using the full dataset is called batch learning, using subsets of data is called mini-batch learning.

Stochastic Gradient Descent

- For the least squares, $\text{minimize} E(\mathbf{w}) = \frac{1}{2N}\sum_{n=1}^{N}(t_n - \mathbf{x}_n^T\mathbf{w})^2$
- we have

$$\nabla E(\mathbf{w}) = \frac{1}{N}\sum_{n=1}^{N}\mathbf{x}_n(\mathbf{x}_n^T\mathbf{w} - t_n)$$

- We choose an initial point $\mathbf{w}^0$, and random subsample S_t from {1,2,…,N} and do the following iterations

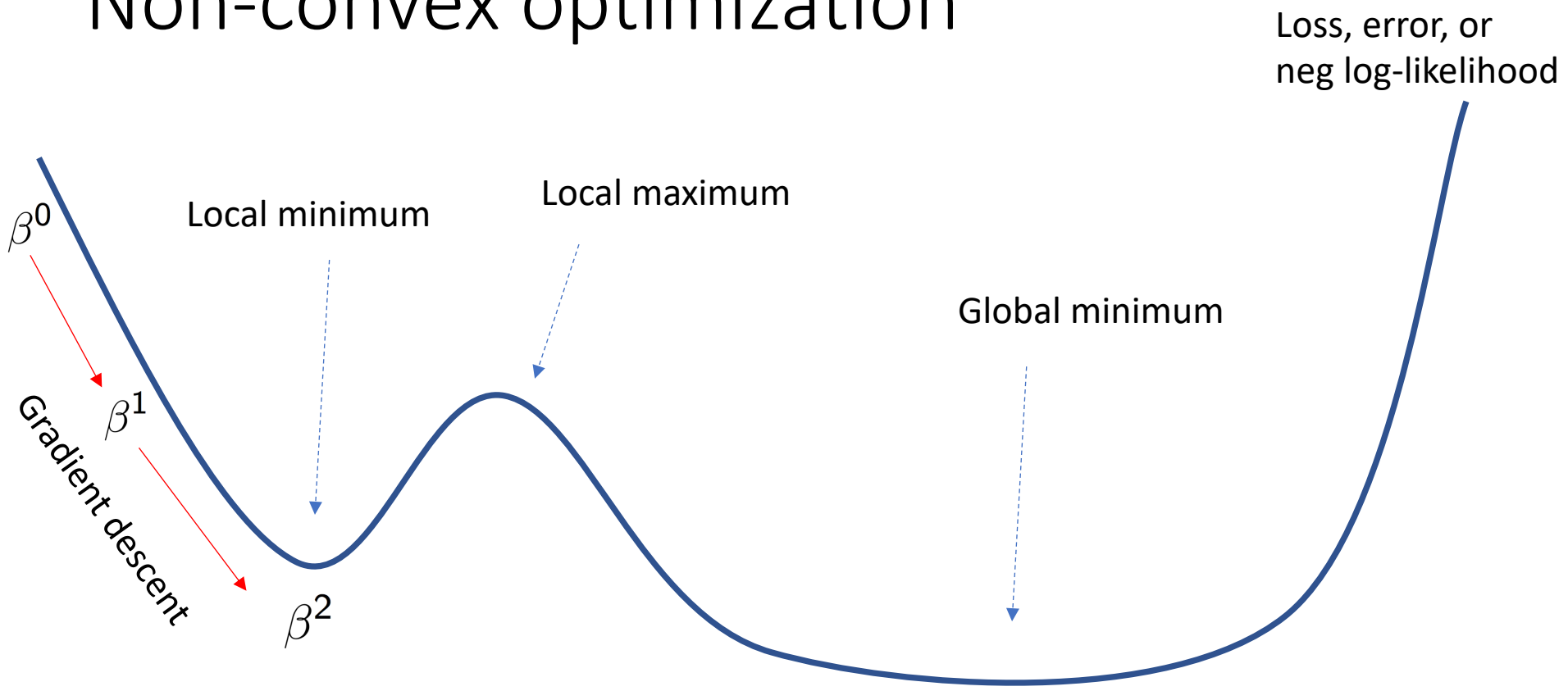
$$\mathbf{w}^{t+1} = \mathbf{w}^t - \frac{\eta_t}{|S_t|}\sum_{n\in S_t}\mathbf{x}_n(\mathbf{x}_n^T\mathbf{w}^t - t_n)$$

Here, both step size and the size of the subsample changes at every iteration.

Final words on SGD

- Tip 1: divide the log-likelihood estimate by the size of your mini-batches. This makes the step size invariant to mini-batch size.
- Tip 2: subsample without replacement so that you visit each point on each pass through the dataset (this is called an epoch).
- Tip 3: Initially, use a larger step size. Gradually reduce it near convergence.

Non-convex optimization

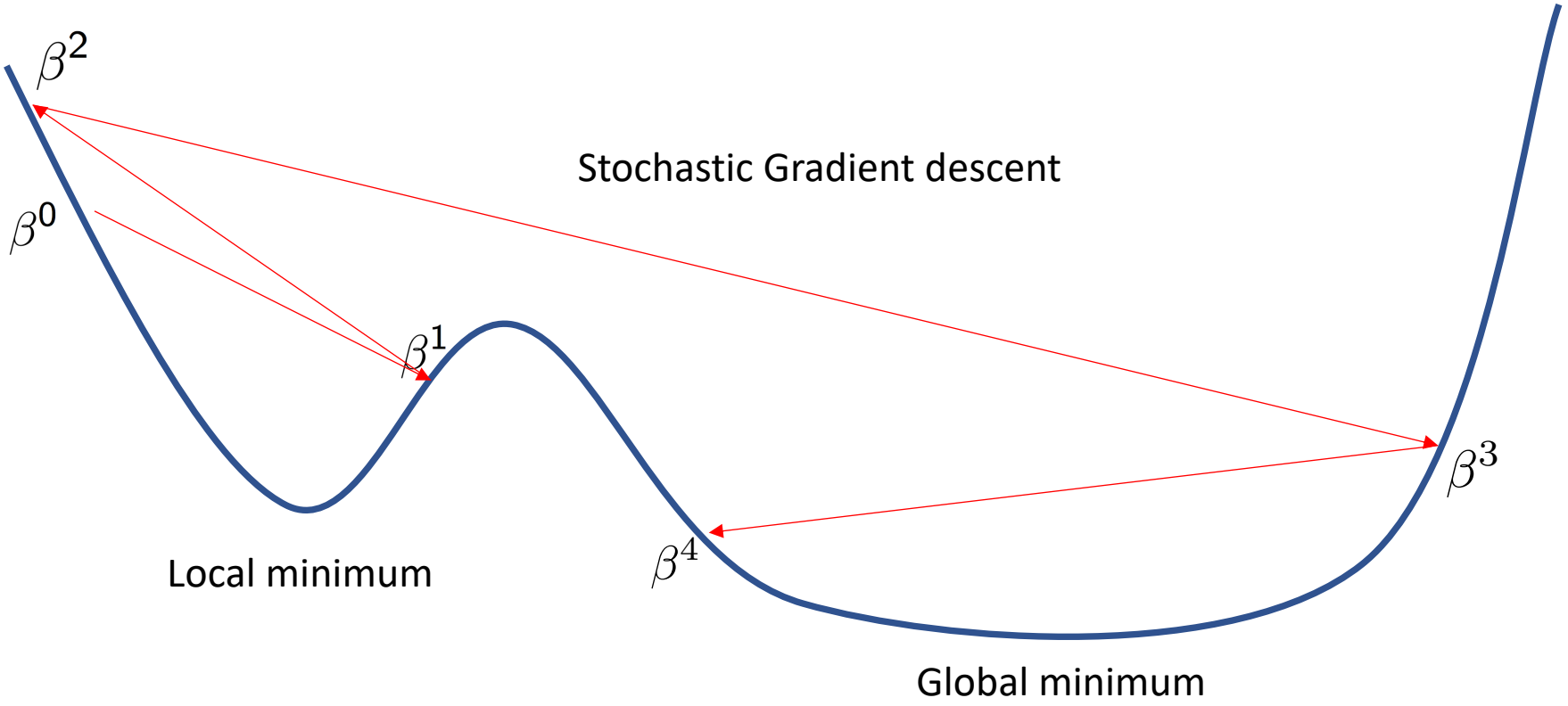


In machine learning, often times we need to rely on non-convex optimization.

Convergence guarantees are very limited, mostly based on heuristic.

In this example, gradient descent would converge to the closest local minimum.

Non-convex optimization



Stochastic methods have higher chance to escape “bad” minima, and converge to favorable regions.

Appendix: Matrix Derivatives

- For a function $f : \mathbb{R}^{m \times n} \to \mathbb{R}$

$$\frac{df(X)}{dX} = \nabla f(X) \in \mathbb{R}^{m \times n} \qquad \nabla f(X) = \begin{bmatrix} \frac{df(X)}{dX_{11}} & \frac{df(X)}{dX_{12}} & \cdots & \frac{df(X)}{dX_{1n}} \\ \vdots & \ddots & & \vdots \\ \frac{df(X)}{dX_{m1}} & \cdots & \cdots & \frac{df(X)}{dX_{mn}} \end{bmatrix}$$

- Ex: $X \in \mathbb{R}^{2 \times 2}$

$$f(X) = X_{11} + X_{12} + \tfrac{1}{2} X_{21}^2 + \tfrac{1}{3} X_{22}^3$$

$$\nabla f(X) = \begin{bmatrix} 1 & 1 \\ X_{21} & X_{22}^2 \end{bmatrix}$$

Appendix: Matrix derivatives

For matrices $A \in \mathbb{R}^{m \times n}, B \in \mathbb{R}^{n \times k}, C \in \mathbb{R}^{k \times l} \qquad X \in \mathbb{R}^{n \times n}$

Trace is given as $\mathrm{Tr}(X) = \sum_{i=1}^{n} X_{ii}$

$c \in \mathbb{R}$

and the following properties hold

$$\mathrm{Tr}(A) = \mathrm{Tr}(A^T) \qquad \mathrm{Tr}(cA) = c\mathrm{Tr}(A)$$

$$\mathrm{Tr}(A + B) = \mathrm{Tr}(A) + \mathrm{Tr}(B) \qquad \mathrm{Tr}(AB) = \mathrm{Tr}(BA)$$

$$\mathrm{Tr}(ABC) = \mathrm{Tr}(CAB) = \mathrm{Tr}(BCA) \qquad (AB)^T = B^T A^T$$

$$u \in \mathbb{R}^n \qquad \|u\|^2 = \mathrm{Tr}(uu^T)$$

Appendix: Matrix derivatives

$$\nabla_X \mathrm{Tr}(XA) = A^T$$

$$\nabla_{X^T} f(X) = (\nabla_X f(X))^T$$

$$\nabla_X \mathrm{Tr}(XBX^TC) = CXB + C^TXB^T$$

$$\nabla_X \det(X) = \det(X)(X^{-1})^T$$

$$E(\mathbf{w}) = \frac{1}{2}\|\mathbf{t} - \mathbf{X}\mathbf{w}\|^2 = \frac{1}{2}\sum_{i=1}^{n}(t_i - \mathbf{x_i}\mathbf{w})^2$$

$$= \frac{1}{2}\mathrm{Tr}((\mathbf{t} - \mathbf{X}\mathbf{w})(\mathbf{t} - \mathbf{X}\mathbf{w})^T)$$

$$= \frac{1}{2}\mathrm{Tr}(\mathbf{t}\mathbf{t}^T) - \mathrm{Tr}(\mathbf{X}\mathbf{w}\mathbf{t}^T) + \frac{1}{2}\mathrm{Tr}(\mathbf{X}\mathbf{w}\mathbf{w}^T\mathbf{X}^T)$$