COMP0171 Bayesian Deep Learning Lecture Notes

Melodie Li

UCL Department of Computer Science

Dec 2024

Contents

Chapter 1

Parameter Estimation and Inference

1.1 Maximum Likelihood Estimation (MLE)

Definition 1.1.1 (Likelihood Function)**.** The likelihood function $p(m|\mu, N)$ describes the probability of observing data m (e.g., number of heads) given parameter μ (e.g., probability of heads in a Bernoulli trial) and number of trials N.

Example 1.1.2. For $m = 4$ heads out of $N = 7$ flips:

$$p(m = 4|\mu, N = 7) = \binom{7}{4}\mu^4(1 - \mu)^3.$$

Theorem 1.1.3 (Maximum Likelihood Estimator)**.** *The Maximum Likelihood Estimate (MLE) is obtained by maximizing the likelihood function:*

$$\hat{\mu}_{ML} = \arg\max_{\mu} p(m|\mu, N).$$

For Bernoulli trials, this leads to:

$$\hat{\mu}_{ML} = \frac{m}{N},$$

where m is the number of successes and N is the total number of trials.

1.2 Maximum A Posteriori (MAP) Estimation

Definition 1.2.1 (MAP Estimator)**.** The MAP estimator combines the likelihood function with a prior distribution $p(\mu)$:

$$\hat{\mu}_{\text{MAP}} = \arg\max_{\mu} \left[\log p(m|\mu, N) + \log p(\mu)\right].$$

Remark. *The prior $p(\mu)$ acts as a regularizer, incorporating prior beliefs about μ and smoothing the estimate.*

Theorem 1.2.2 (Difference Between MLE and MAP)**.**

- ***MLE** considers only the likelihood $p(m|\mu, N)$, leading to estimates based solely on observed data.*
- ***MAP** balances the data (likelihood) with prior knowledge (prior distribution), which can prevent extreme estimates when data is scarce.*

1.3 Bayesian Estimation

Definition 1.3.1 (Posterior Distribution)**.** The posterior distribution updates our belief about the parameter θ after observing data D:

$$p(\theta|D, H) = \frac{p(D|\theta, H)p(\theta|H)}{p(D|H)},$$

where:

- $p(\theta|D, H)$: Posterior — updated belief after observing data.
- $p(D|\theta, H)$: Likelihood — probability of the data given parameters.
- $p(\theta|H)$: Prior — initial belief about parameters.
- $p(D|H)$: Evidence — normalization constant.

Definition 1.3.2 (Beta Distribution)**.** The Beta distribution serves as a conjugate prior for the Bernoulli process:

$$\text{Beta}(\mu|a, b) = \frac{\Gamma(a + b)}{\Gamma(a)\Gamma(b)}\mu^{a-1}(1 - \mu)^{b-1}.$$

Example 1.3.3 (Posterior Distribution for Coin Flips)**.** If we observe $m = 4$ successes out of $N = 7$, the posterior is:

$$p(\mu|m = 4, N = 7) \propto \mu^4(1 - \mu)^3.$$

Definition 1.3.4 (Bayesian Prediction)**.** To predict a new outcome x_{N+1} after observing data $D = \{x_1, \ldots, x_N\}$, we marginalize over all possible values of μ:

$$p(x_{N+1}|D) = \int p(x_{N+1}|\mu)p(\mu|D)d\mu.$$

Remark. *Unlike MLE and MAP, Bayesian predictions account for uncertainty in the parameter μ, leading to more robust predictions.*

Chapter 2

Markov Chain Monte Carlo

2.1 Monte Carlo Estimation

2.1.1 Basic Monte Carlo Identity

Definition 2.1.1 (Monte Carlo Sampling)**.** Monte Carlo methods are used to approximate expectations by averaging over random samples. For a function $f(x)$ under distribution $p(x)$, the expectation is:

$$\mathbb{E}[f] = \int p(x)f(x)dx.$$

If we can draw samples $x^{(i)}$ from $p(x)$, we can approximate this as:

$$\hat{f} = \frac{1}{N}\sum_{i=1}^{N} f(x^{(i)}), \quad x^{(i)} \sim p(x).$$

This is an unbiased estimator, meaning $\mathbb{E}[\hat{f}] = \mathbb{E}[f]$. The approximation becomes exact as $N \to \infty$.

Theorem 2.1.2 (Monte Carlo Unbiasedness)**.** *The Monte Carlo estimator $\hat{f}$ is unbiased:*

$$\mathbb{E}[\hat{f}] = \mathbb{E}\left[\frac{1}{N}\sum_{i=1}^{N} f(x^{(i)})\right] = \frac{1}{N}\sum_{i=1}^{N} \mathbb{E}_{p(x)}\left[f(x^{(i)})\right] = \frac{1}{N}\sum_{i=1}^{N} \mathbb{E}[f] = \mathbb{E}[f].$$

- *This expectation is taken over all random choices in the estimator.*
- *In general, expectation is a **linear operator**:*

$$\mathbb{E}_{p(x)}[ax + b] = a\mathbb{E}_{p(x)}[x] + b.$$

- *Each sample $x^{(i)}$ is assumed to be drawn independently from $p(x)$.*

Definition 2.1.3 (Monte Carlo Variance)**.** The variance of the Monte Carlo estimator $\hat{f}$ is:

$$\text{Var}[\hat{f}] = \text{Var}\left(\frac{1}{N}\sum_{i=1}^{N} f(x^{(i)})\right) = \frac{1}{N^2}\text{Var}\left(\sum_{i=1}^{N} f(x^{(i)})\right).$$

Since each $x^{(i)}$ is drawn independently from $p(x)$, this can be simplified to:

$$= \frac{1}{N^2} \cdot N\text{Var}[f(x)] = \frac{1}{N}\text{Var}[f(x)].$$

We can also express it as:

$$\text{Var}[\hat{f}] = \frac{1}{N}\mathbb{E}\left[(f - \mathbb{E}[f])^2\right].$$

Therefore, as $N \to \infty$, the variance approaches zero:

$$\lim_{N\to\infty} \text{Var}[\hat{f}] = 0.$$

2.1.2 Inverse CDF Sampling

Definition 2.1.4 (Inverse Transform Sampling)**.** A method to generate samples from a target distribution $p(x)$ by using a uniform random variable. For distributions with a tractable inverse CDF F_X^{-1}, the algorithm is:

1. Draw $u^{(i)} \sim \mathcal{U}([0,1])$.
2. Compute $x^{(i)} = F_X^{-1}(u^{(i)})$.

The resulting samples $x^{(i)}$ follow the target distribution $p(x)$.

Remark. *This method leverages the CDF $F_X(x)$, ensuring that $p(F_X^{-1}(u) \leq x) = p(u \leq F_X(x) = F_X(x)$, effectively matching the target distribution. i.e. the CDF of the transformed uniform random variable is exactly the CDF of the target distribution.*

2.1.3 Ancestral Sampling

Definition 2.1.5 (Ancestral Sampling)**.** Ancestral sampling allows us to build samples for complex joint distributions by sampling from simpler conditional distributions sequentially. This is particularly useful for generative models defined by sequences of conditional distributions.

Example 2.1.6. Consider a generative model for N coin flips:

$$x \sim \mathcal{U}([0,1]), \quad y_i|x \sim \text{Bernoulli}(x),\ i = 1,\ldots,N.$$

We can generate samples from the joint distribution $(x, y_1, \ldots, y_N)$ by drawing x from the uniform distribution and then generating each y_i conditionally on x.

2.1.4 Conditional Sampling: Guess and Check

Definition 2.1.7 (Conditional Sampling via Rejection)**.** To sample from a conditional distribution, a basic approach is to use rejection sampling:

1. Use ancestral sampling to draw samples (x, y) from the joint distribution $p(x, y)$.
2. To estimate $p(x|y = \mathbf{y})$, accept x as a sample if its corresponding $y = \mathbf{y}$.

Note: This method becomes inefficient as the dimensionality of y increases.

2.1.5 Importance Sampling

Definition 2.1.8 (Importance Sampling)**.** Importance sampling allows us to approximate expectations under a target distribution $\pi(x)$, which may be difficult to sample from, using samples from a different, related distribution $q(x)$, known as the proposal distribution. The goal is to estimate the expectation:

$$\mathbb{E}_{\pi(x)}[f(x)] = \int \pi(x) f(x) dx.$$

Since direct sampling from $\pi(x)$ might be impractical, we can instead rewrite this as:

$$\mathbb{E}_{\pi(x)}[f(x)] = \int q(x) \frac{\pi(x)}{q(x)} f(x) dx = \mathbb{E}_{q(x)}[W(x) f(x)],$$

where:

$$W(x) = \frac{\pi(x)}{q(x)},$$

are the importance weights.

Practical Algorithm

- Draw samples $x^{(i)} \sim q(x)$ for $i = 1, \ldots, N$.
- Compute the importance weights:

$$W(x^{(i)}) = \frac{\pi(x^{(i)})}{q(x^{(i)})}.$$

- Estimate the expectation:

$$\mathbb{E}_{\pi(x)}[f(x)] \approx \frac{1}{N} \sum_{i=1}^{N} W(x^{(i)}) f(x^{(i)}).$$

2.1.6 Self-Normalized Importance Sampling

Definition 2.1.9 (Self-Normalized Importance Sampling)**.** When exact weights $W(x)$ cannot be computed directly (but only evaluate up to a constant), we use unnormalized weights:

$$w(x) = \frac{p(x, y)}{q(x)}.$$

The posterior expectation can be approximated as:

$$\mathbb{E}_{p(x|y)}[f(x)] \approx \frac{\sum_{i=1}^{N} w(x^{(i)}) f(x^{(i)})}{\sum_{i=1}^{N} w(x^{(i)})},$$

where $x^{(i)}$ are samples from $q(x)$.

Remark. *Self-normalized importance sampling allows us to estimate the posterior without needing $p(y)$, making it useful for complex models. Weights W_i are defined as:*

$$W_i = \frac{w(x^{(i)})}{\sum_{j=1}^{N} w(x^{(j)})},$$

leading to:

$$\mathbb{E}_{p(x|y)}[f(x)] \approx \sum_{i=1}^{N} W_i f(x^{(i)}).$$

This has the added bonus of providing an estimator of the model evidence $p(y)$ automatically. (This is useful for comparing dierent models!).

2.1.7 Likelihood Weighting

Definition 2.1.10 (Likelihood Weighting)**.** Likelihood weighting is a method where we use the prior $p(x)$ as the proposal distribution $q(x)$ and compute weights based on the likelihood:

$$w(x) = p(y|x).$$

Instead of rejecting samples, the algorithm assigns ”soft” weights to each sample, and only sample the latent variables, effectively adjusting their contributions.

Remark. *Likelihood weighting avoids the inefficiency of hard rejection by assigning continuous weights, making it more suitable for scenarios where direct sampling from the posterior is impractical.*

2.2 MCMC Intuition

Challenges of Importance Sampling

- Importance sampling struggles as the dimension of the latent variables increases, unless the proposal distribution $q(x)$ is very close to the target distribution $p(x|y)$, which might be quite different from the prior.
- Common approaches like setting $q(x) = p(x)$ often lead to inefficient sampling.

Definition 2.2.1 (Markov Chain Monte Carlo)**.** MCMC methods are a class of algorithms that draw samples directly from a target distribution by constructing a Markov chain. These samples are generated through a biased random walk, allowing exploration of complex, high-dimensional distributions where traditional sampling methods (like importance sampling) may fail.

2.3 Markov Chains

2.3.1 Definition and Key Properties

Definition 2.3.1 (Markov Chain)**.** A Markov chain is a sequence of random variables $x_0, x_1, x_2, \ldots$ where the probability of transitioning to the next state depends only on the current state, not the entire history:

$$p(x_{n+1}|x_1, \ldots, x_n) = p(x_{n+1}|x_n).$$

This is known as the ”memoryless” property. If the transition probabilities are consistent across all steps, the chain is called **homogeneous**, with a transition probability denoted as:

$$T(x_{n+1}|x_n) = p(x_{n+1}|x_n).$$

- **Transition Probability**: The probability of moving from one state to another. For homogeneous chains, this probability remains unchanged across steps.
- **Stationary Distribution**: A distribution $\pi(x)$ is **invariant** or **stationary** if:

  $$\pi(x') = \sum_x T(x'|x)\pi(x).$$

  If the chain starts in this distribution, it will remain there indefinitely.
- **Detailed Balance Condition**: A sufficient condition for $\pi(x)$ to be stationary is if the chain satisfies:

  $$\pi(x)T(x'|x) = \pi(x')T(x|x').$$

  A chain satisfying this is known as **reversible**.

- **Ergodicity**: A Markov chain is **ergodic** if every state can be reached from any other (irreducible) and the chain is not trapped in cycles (aperiodic). This ensures convergence to a unique stationary distribution regardless of initial starting point x_0.

- **Convergence**: For ergodic chains, the distribution of x_n converges to $\pi(x)$ as $n \to \infty$, regardless of the initial state.

Remark. *Ergodicity is crucial for MCMC algorithms, as it guarantees that samples will eventually reflect the target distribution, even if the process starts from a different point.*

2.4 Constructing MCMC Algorithms

MCMC creates a Markov chain where the stationary distribution matches the target $p(x|y)$. The algorithm involves proposing a new state x' from a distribution $q(x'|x)$ and deciding to accept or reject based on an acceptance ratio A:

$$A(x \to x') = \min\left(1, \frac{p(y, x')q(x|x')}{p(y, x)q(x'|x)}\right).$$

This generates dependent samples. Also doesn't require the normalizing constant as cancelled out in the ratio!

2.4.1 Symmetric Proposals

Remark. *When the proposal distribution $q(x'|x)$ is symmetric (e.g., default choice Gaussian noise $\mathcal{N}(x, \sigma^2 I)$), the acceptance ratio simplifies to:*

$$A(x \to x') = \min\left(1, \frac{p(y, x')}{p(y, x)}\right).$$

Intuitively this looks like a noisy sort of hill climbing:

- sample a value $\mathbf{x}' \sim q(\mathbf{x}'|\mathbf{x})$

- if $p(\mathbf{y}, \mathbf{x}') > p(\mathbf{y}, \mathbf{x})$, then move to $\mathbf{x}'$ ($A = 1$).

- otherwise, maybe move to $\mathbf{x}'$

2.5 MCMC Diagnostics

2.5.1 Burn-in and Trace Plots

- **Trace Plots**: Useful diagnostic tools to observe how the Markov chain evolves over iterations and whether converges.
- **Burn-in**: Early samples might be biased by the initial values. It is common to discard a set number of initial samples until the chain reaches a stable state (equilibrium).

2.5.2 Pitfalls and Considerations

- **Proposal:** Choosing the right proposal distribution is crucial to ensure efficient sampling. Bad proposals may lead to too low/high acceptance rates (step size).
- **Convergence diagnostics** can be challenging; incorrect burn-in or slow exploration may lead to biased results.
- In large data settings, evaluating the acceptance ratio can be expensive.

Example 2.5.1 (Gaussian Approximation)**.** After sufficient sampling, a parametric approximation can be performed to estimate mean and variance:

$$\hat{\mu} = \frac{1}{S - s_0} \sum_{s=s_0}^{S} x^{(s)}, \quad \hat{\sigma}^2 = \frac{1}{S - s_0} \sum_{s=s_0}^{S} (x^{(s)} - \hat{\mu})^2,$$

where s_0 denotes the burn-in period.

Chapter 3

Graphical Models

3.1 Basic GM Identity

Definition 3.1.1 (Graphical Models)**.** Graphical models are a way to visually represent dependencies and conditional dependencies among random variables. They provide a compact representation of joint distributions by illustrating how variables interact through directed or undirected edges.

Theorem 3.1.2 (Factorizations and Dependency Structures)**.**

- *The joint distribution over variables $p(a, b, c)$ can be factorized using the product rule, e.g., $p(a, b, c) = p(a|b, c)p(b|c)p(c)$.*
- *Directed graphs (Bayesian networks) depict these factorized forms by connecting nodes with arrows based on conditional dependencies.*
- *The absence of links between nodes can indicate conditional independence.*

Definition 3.1.3 (Plate Notation)**.** Plate notation is a graphical representation used to simplify the depiction of models with repeated structures, such as multiple data points or repeated variables. It shows a single node with a surrounding box, indicating that multiple instances of the variable exist.

Example 3.1.4. For a Bayesian regression model:

$$p(t, w) = p(w) \prod_{n=1}^{N} p(t_n|w),$$

where w represents the weights, and t_n are data points. Plate notation can compactly represent this by enclosing t_n in a box labeled N.

Visual Example of Different Model Types

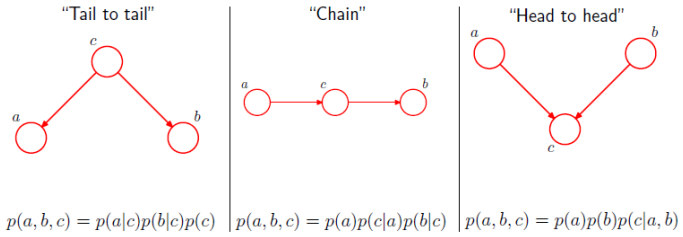


Figure 3.1: Structured Models, Conditional Independence.

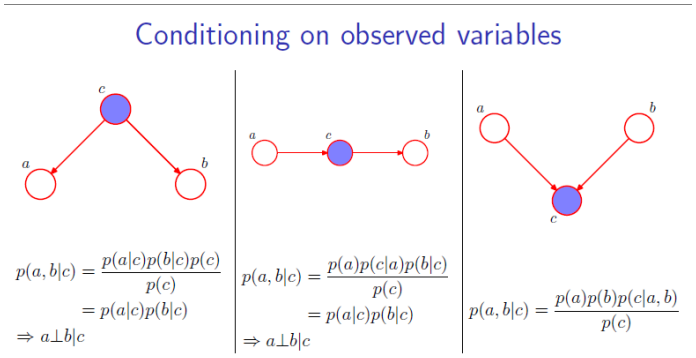


Figure 3.2: Conditioning on Observed Variables.

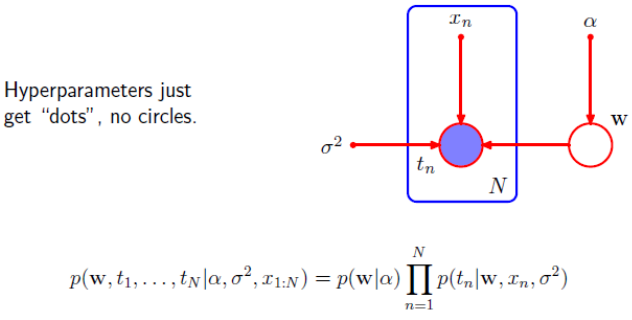


Figure 3.3: Including Deterministic Variables.

3.2 Bayesian Networks

3.2.1 Directed Graphs

Definition 3.2.1 (Bayesian Networks)**.** A Bayesian network is a directed acyclic graph (DAG) where each node represents a random variable, and edges denote conditional dependencies. Each node is conditionally independent of its non-descendants given its parents.

Example 3.2.2. Consider a network of three variables a, b, and c:

$$p(a, b, c) = p(a|c)p(b|c)p(c).$$

The graph will have directed arrows from c to both a and b, representing the dependencies.

3.3 Conditional Independence and d-Separation

Definition 3.3.1 (Conditional Independence)**.** Conditional independence, denoted $a \perp\!\!\!\perp b|c$, means that a and b are independent given c. Graphical models simplify detecting such independencies directly from the graph structure without requiring analytical calculations.

Theorem 3.3.2 (d-Separation)**.** *The d-separation criterion determines if a set of nodes is conditionally independent of another set, given a third set. It identifies blocked paths in the graph based on specific patterns, such as "tail-to-tail" or "head-to-head" connections.*

Chapter 4

Linear Models and Optimization

4.1 Linear Models with Fixed Features

Definition 4.1.1 (Linear Function)**.** A linear function satisfies two properties:

$$f(x+y) = f(x) + f(y), \quad f(cx) = cf(x).$$

The equation for a basic linear function can be represented as:

$$f(x) = mx + b,$$

and more generally as an affine function:

$$f(x) = Ax + b.$$

4.1.1 Linear Regression

Definition 4.1.2 (Linear Regression Model)**.** The goal is to learn a mapping from inputs x to outputs y using a linear model:

$$y_i = w^\top \phi(x_i),$$

where $\phi(x_i)$ represents feature map of the input, and w are the weights to be learned. Feature map for affine functions:

$$\phi(x_i) = \begin{bmatrix} 1 \\ x_i \end{bmatrix} \in \mathbb{R}^D$$

Note: For polynomial, adding more x_i to the feature map. Each of these entries forms a row in a "design matrix"

$$\Phi = \begin{bmatrix} \phi(x_1)^\top \\ \vdots \\ \phi(x_N)^\top \end{bmatrix} \in \mathbb{R}^{N \times D}.$$

Theorem 4.1.3 (Least Squares Objective)**.** *The objective of linear regression is to minimize the squared-error loss over the dataset* (x_i, y_i)*:*

$$L(w) = \sum_{i=1}^{N} (y_i - w^\top \phi(x_i))^2.$$

The optimal w *can be found analytically by minimizing this loss.*

4.1.2 Least Squares Solution

Theorem 4.1.4 (Closed-Form Solution)**.** *Let* Φ *be the design matrix, where each row is* $\phi(x_i)^\top$*. The loss function can be expressed as:*

$$L(w) = \|y - \Phi w\|_2^2.$$

The optimal weights w *can be computed as:*

$$w = (\Phi^\top \Phi)^{-1} \Phi^\top y.$$

4.1.3 Regularization

Definition 4.1.5 (Empirical Risk Minimization with Regularization)**.** Regularization modifies the objective to prevent overfitting (especially when D ¿N):

$$\min L(w) = \sum_{i=1}^{N} \ell(w^\top \phi(x_i), y_i) + \lambda r(w),$$

where l is some pre-instance loss function, $r(w)$ is a regularization term, and λ controls its strength and can be set using cross-validation.

Theorem 4.1.6 (L2 Regularization (Ridge Regression))**.** *For L2, we prevent large values of the weights by adding a quadratic penalty term:*

$$\min L(w) = \sum_{i=1}^{N} (y_i - w^\top \phi(x_i))^2 + \lambda \|w\|_2^2.$$

The closed-form solution can be expressed as:

$$w = (\Phi^\top \Phi + \lambda I)^{-1} \Phi^\top y.$$

Note: for redundant features (identical), both will have equal weights $w_1 = w_2$*.*

Remark.
- *Sum of squared weights*

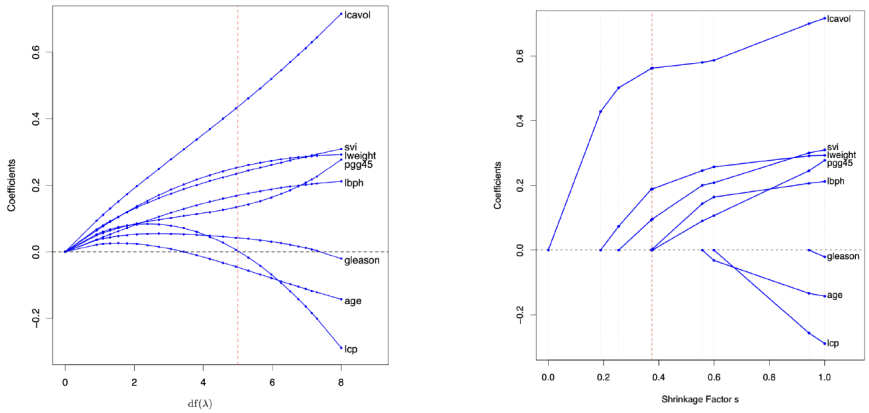
- *Heavily penalizes large w_d but only slightly penalizes small w_d*
- *Differentiable; closed-form solution*
- *Small weights still persist (don't reach exactly zero)*

Theorem 4.1.7 (L1 Regularization (Lasso))**.** *L1 regularization encourages sparsity by penalizing the sum of the absolute values of the weights:*

$$\min L(w) = \sum_{i=1}^{N} (y_i - w^\top \phi(x_i))^2 + \lambda \|w\|_1.$$

This can lead to some weights being exactly zero, effectively performing feature selection.

Remark.
- *Sum of the absolute values of the weights*
- *Non-differentiable at zero; need to use specialized optimization routines*
- *Only a subset of weights remain — redundant features will be "pruned" to zero*



Ridge (L_2) regularization LASSO (L_1) regularization

Figure 4.1: Comparing Solutions with Varying λ.

4.2 Probabilistic Treatment of Linear Regression

Definition 4.2.1 (Probabilistic Model)**.** A probabilistic interpretation assumes that the observed data y can be modeled as:

$$y_i = w^\top \phi(x_i) + \epsilon,$$

where ϵ is Gaussian noise with mean 0 and variance σ^2. The likelihood function can be written as:

$$p(y|x, w) = \mathcal{N}(y; w^\top \phi(x), \sigma^2).$$

Theorem 4.2.2 (MLE)**.** *The MLE approach estimates the weights w by maximizing the likelihood function. This is equivalent to minimizing the squared-error loss:*

$$\hat{w}_{MLE} = \arg\min_w \|y - \Phi w\|_2^2.$$

After simplification, the objective can be written as:

$$\hat{w}_{MLE} = \arg\max_w -\frac{1}{2\sigma^2} \sum_{i=1}^{N} (y_i - w^\top \phi(x_i))^2.$$

Theorem 4.2.3 (MAP)**.** *MAP estimation incorporates a Gaussian prior on w, typically expressed as:*

$$p(w) = \mathcal{N}(w; 0, \alpha^{-1} I),$$

leading to the posterior:

$$p(w|x, y) \propto p(y|x, w)p(w).$$

By maximizing this posterior, we obtain:

$$\hat{w}_{MAP} = \arg\max_w -\frac{1}{2\sigma^2} \sum_{i=1}^{N} (y_i - w^\top \phi(x_i))^2 - \frac{\alpha}{2} w^\top w.$$

Remark. *The MLE solution focuses solely on fitting the observed data, leading to the same objective as standard linear regression. The MAP solution, however, adds a regularization term $\frac{\alpha}{2} w^\top w$, effectively penalizing large weights. This corresponds to the L2 regularization term in ridge regression. Similarly, for L1 we can place a Laplace distribution prior on each w_d to be equivalent.*

Theorem 4.2.4 (Gaussian Prior and Likelihoods)**.** *Let $y_i \sim \mathcal{N}(\mathbf{w}^\top \phi(x_i), \sigma^2)$: that is, suppose labels y_i have a Gaussian likelihood with mean $\mathbf{w}^\top \phi(x_i)$ with some "error" that has (known) variance σ^2. Then*

$$\log p(y_i \mid x_i, \mathbf{w}) = -\frac{1}{2} \log 2\pi\sigma^2 \underbrace{-\frac{1}{2\sigma^2}(y - \mathbf{w}^\top \phi(x))^2}_{\text{const.}}$$

Now suppose we place a zero-mean Gaussian prior on $\mathbf{w}$, with an isotropic covariance $\Sigma = \alpha^{-1}\mathbf{I}$

$$\log p(\mathbf{w} \mid \alpha) = -\frac{D}{2} \log 2\pi + \frac{D}{2} \log \alpha \underbrace{-\frac{\alpha}{2} \mathbf{w}^\top \mathbf{w}}_{\text{const.}}$$

$$\hat{w}^{ML} = \arg\max_w -\frac{1}{2\sigma^2}\sum_{i=1}^{N}\Big(y_i - w^\top\phi(x_i)\Big)^2$$

$$\hat{w}^{MAP} = \arg\max_w -\frac{1}{2\sigma^2}\sum_{i=1}^{N}\Big(y_i - w^\top\phi(x_i)\Big)^2 - \frac{\alpha}{2}w^\top w$$

Flipping back to the previous slide, you can see that the MAP objective corresponds to the regularized empirical risk objective, with an L_2 regularizer, where $\lambda \equiv \alpha\sigma^2$.

Theorem 4.2.5. *There's a similar equivalency for L_1 regularization, by placing a Laplace distribution prior on each w_d:*

$$p(w_d \mid \mu, b) = \frac{1}{2b}\exp\left(-\frac{|w_d - \mu|}{b}\right)$$

This kinda looks like a normal distribution, but has heavier tails.

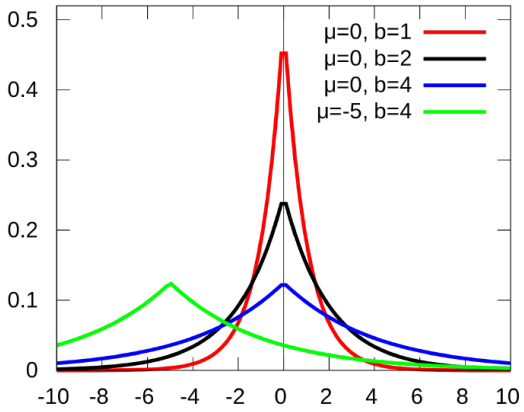


Figure 4.2: Laplace Distribution.

4.3 Bayesian Linear Model

4.3.1 Bayesian Linear Regression

Definition 4.3.1 (Bayesian Linear Regression)**.** As it turns out, Bayesian linear regression is one of the only models where the posterior has a closed form. If we define the model as (Gaussian case):

$$\mathbf{w} \sim \mathcal{N}(0, \alpha^{-1}\mathbf{I}_d) \quad \text{and} \quad y_i \sim \mathcal{N}(\mathbf{w}^\top\phi(x_i), \beta^{-1}),$$

then the posterior is also a normal distribution:

$$p(\mathbf{w} \mid \mathbf{x}_{1:N}, y_{1:N}) = \mathcal{N}(\mathbf{w} \mid \mathbf{m}, \mathbf{S}),$$

where

$$\mathbf{m} = \beta(\alpha\mathbf{I} + \beta\Phi^\top\Phi)^{-1}\Phi^\top\mathbf{y}, \quad \mathbf{S} = (\alpha\mathbf{I} + \beta\Phi^\top\Phi)^{-1}.$$

4.3.2 Bayesian Linear Classifiers

BLC Overview

Bayesian linear classifiers focus on modeling binary classification problems using a Bayesian approach. Given data $\mathcal{D} = \{x_i, y_i\}, i = 1, \ldots, N$, and a fixed feature map ϕ such that $\phi(x_i) \in \mathbb{R}^D$, we have:

$$\mathbf{w} \sim \mathcal{N}(\mathbf{m}_0, \mathbf{S}_0), \quad \mathbf{w} \in \mathbb{R}^D$$

$$y_i \mid x_i \sim \text{Bernoulli}(\sigma(\mathbf{w}^\top\phi(x_i))), \quad i = 1, \ldots, N$$

where

$$\sigma(z) = \frac{1}{1 + \exp(-z)}.$$

Typically, we assume a prior $\mathbf{m}_0 = \mathbf{0}$ and $\mathbf{S}_0 = \alpha^{-1}\mathbf{I}$.

BLC Goals

Our primary goals are:

1. **Estimate the Posterior:**

$$p(\mathbf{w} \mid \mathcal{D}) = \frac{p(\mathbf{w}) \prod_{i=1}^N p(y_i \mid \mathbf{w}, x_i)}{p(y_{1:N} \mid x_{1:N})}.$$

2. **Make Predictions for New Data:**

$$p(y' \mid x', \mathcal{D}) = \int p(y' \mid \mathbf{w}, x') p(\mathbf{w} \mid \mathcal{D})\, d\mathbf{w}.$$

MAP Estimation for Bayesian Linear Classification

Maximum A Posteriori (MAP) estimation seeks to find the most probable $\mathbf{w}$ given the data. The log posterior is given by:

$$\mathcal{L}(\mathbf{w}) = \log p(\mathbf{w}) + \sum_{i=1}^N \log p(y_i \mid \mathbf{w}, x_i)$$

$$\equiv \log p(\mathbf{w} \mid \mathcal{D}) + \text{const.}$$

The MAP estimate is:

$$\hat{\mathbf{w}}^{MAP} = \arg\max_{\mathbf{w}} \mathcal{L}(\mathbf{w}),$$

which can be optimized using techniques like gradient descent.

Warning: Minibatches and Priors

When using minibatches, it's crucial to properly scale the prior. Suppose we evaluate the objective function on a minibatch of size $M \ll N$. The likelihood is rescaled as:

$$\sum_{i=1}^{N} \log p(y_i \mid \mathbf{w}, x_i) \approx \frac{N}{M} \sum_{j=1}^{M} \log p(y_j \mid \mathbf{w}, x_j).$$

The minibatch estimator for the loss becomes:

$$\hat{\mathcal{L}}_M(\mathbf{w}) = \log p(\mathbf{w}) + \frac{N}{M} \sum_{j=1}^{M} \log p(y_j \mid \mathbf{w}, x_j).$$

In practice, we often optimize $\frac{1}{N}\hat{\mathcal{L}}_M(\mathbf{w})$ instead.

What to Do for Non-Gaussian Posteriors?

When the posterior $p(\mathbf{w} \mid \mathcal{D})$ is non-Gaussian, there are two common approaches:

- **Finite Sample Approximation:**
  - Use a collection of candidate weights $\mathbf{w}^{(1)}, \ldots, \mathbf{w}^{(L)}$.
  - Example: Markov chain Monte Carlo (MCMC).
- **Parametric Approximation:**
  - Approximate the posterior with a parametric distribution $q(\mathbf{w})$, e.g., a Gaussian distribution.
  - Examples: Variational Bayes, Laplace approximations.

While the Gaussian assumption may not hold, we can aim to find a "best choice" $q(\mathbf{w})$ that closely approximates $p(\mathbf{w} \mid \mathcal{D})$.

4.3.3 Laplace Approximation

Definition 4.3.2 (Laplace Approximation Justification)**.** The Laplace approximation is a method for approximating a non-Gaussian posterior distribution $p(w|D)$ with a Gaussian distribution. It involves expanding the log-posterior around its mode w^* using a Taylor series.

In general: suppose we have found the value $\mathbf{w}^*$ that maximizes $\log p(\mathbf{w} \mid \mathcal{D})$.

- Take a Taylor expansion of $\log p(\mathbf{w} \mid \mathcal{D})$ in the vicinity of $\mathbf{w}^*$. For small δ,

$$\log p(\mathbf{w}^* + \delta \mid \mathcal{D}) \approx \log p(\mathbf{w}^* \mid \mathcal{D}) + \delta^\top \mathbf{g} + \frac{1}{2}\delta^\top \mathbf{H}\delta,$$

  where the gradient and Hessian are, respectively,

$$\mathbf{g} = \nabla \log p(\mathbf{w} \mid \mathcal{D})\big|_{\mathbf{w}=\mathbf{w}^*}, \quad \mathbf{H} = \nabla\nabla^\top \log p(\mathbf{w} \mid \mathcal{D})\big|_{\mathbf{w}=\mathbf{w}^*}.$$

- Since $\mathbf{w}^*$ is a mode of $\log p(\mathbf{w} \mid \mathcal{D})$, $\mathbf{g} = 0$. So taking $\mathbf{w} = \mathbf{w}^* + \delta$, we have:

$$\log p(\mathbf{w} \mid \mathcal{D}) \approx \log p(\mathbf{w}^* \mid \mathcal{D}) + \frac{1}{2}(\mathbf{w} - \mathbf{w}^*)^\top \mathbf{H}(\mathbf{w} - \mathbf{w}^*).$$

- The Taylor expansion around w^* gives:

$$\log p(w|D) \approx \log p(w^*|D) - \frac{1}{2}(w - w^*)^\top H(w - w^*),$$

  where H is the Hessian matrix of the log-posterior evaluated at w^*. The first term can be considered as constant w.r.t. w. The resulting Gaussian approximation suggests:

$$p(w|D) \approx \mathcal{N}(w^*, H^{-1}),$$

  where H^{-1} serves as the covariance.

Remark. *The Laplace approximation works well if the posterior is approximately Gaussian around its mode. However, computing the Hessian H can be challenging in high-dimensional settings. Despite its limitations, it can be used effectively for tasks like model selection and online learning.*

Theorem 4.3.3 (Hessian Computation for Bayesian Logistic Regression)**.** *To apply this to logistic regression, first write the negative log target density,*

$$-\log p(\mathbf{w} \mid \mathcal{D}) = -\log p(\mathbf{w}) - \sum_{i=1}^{N} \log p(y_i \mid \mathbf{w}, \mathbf{x}_i) + \textit{const}$$

$$= \frac{1}{2}(\mathbf{w} - \mathbf{m}_0)^\top \mathbf{S}_0^{-1}(\mathbf{w} - \mathbf{m}_0) - \sum_{i=1}^{N} \left(y_i \log \hat{y}_i + (1 - y_i)\log(1 - \hat{y}_i)\right) + \textit{const}$$

where $\hat{y}_i = \sigma(\mathbf{w}^\top \phi(\mathbf{x}_i))$*. The covariance* $\mathbf{S}_N$ *is then its inverse Hessian:*

$$\mathbf{S}_N^{-1} = -\nabla\nabla \log p(\mathbf{w} \mid \mathcal{D}) = \mathbf{S}_0^{-1} + \sum_{i=1}^{N} \hat{y}_i(1-\hat{y}_i)\phi(\mathbf{x}_i)\phi(\mathbf{x}_i)^\top.$$

Theorem 4.3.4 (Prediction Approximation using Laplace)**.** *Given a posterior approximation* $p(\mathbf{w} \mid \mathcal{D}) \approx q(\mathbf{w}) \equiv \mathcal{N}(\mathbf{w} \mid \hat{\mathbf{w}}_{MAP}, \mathbf{S}_N)$*, we aim to make predictions for a new point* $\mathbf{x}$*. The predictive probability*

$$p(y = 1 \mid \mathbf{x}, \mathcal{D}) \approx \int \sigma(\mathbf{w}^\top \phi(\mathbf{x}))q(\mathbf{w})\, d\mathbf{w}$$

cannot be computed exactly.

Monte Carlo Approach: *Sample* $\mathbf{w}^{(1)}, \ldots, \mathbf{w}^{(L)} \sim q(\mathbf{w})$*, and estimate:*

$$\mathbb{E}_{q(\mathbf{w})}\left[\sigma(\mathbf{w}^\top \phi(\mathbf{x}))\right] \approx \frac{1}{L}\sum_{\ell=1}^{L} \sigma\left(\mathbf{w}^{(\ell)\top}\phi(\mathbf{x})\right).$$

Dimensionality Reduction for Integration: *Alternatively, the* D*-dimensional integral can be reduced to a one-dimensional integral. Define:*

$$\mu_a = \hat{\mathbf{w}}_{MAP}^\top \phi(\mathbf{x}), \quad \sigma_a^2 = \phi(\mathbf{x})^\top \mathbf{S}_N \phi(\mathbf{x}),$$

where $a \sim \mathcal{N}(a \mid \mu_a, \sigma_a^2)$*.*

The expected value becomes:

$$\mathbb{E}_{q(\mathbf{w})}\left[\sigma(\mathbf{w}^\top \phi(\mathbf{x}))\right] = \mathbb{E}_{p(a)}[\sigma(a)] = \int \sigma(a)\mathcal{N}(a \mid \mu_a, \sigma_a^2)\, da.$$

This integral can be numerically approximated, or closely approximated as:

$$p(y = 1 \mid \mathbf{x}, \mathcal{D}) \approx \sigma\left(\frac{\mu_a}{\sqrt{1 + \pi\sigma_a^2/8}}\right).$$

Thoughts on Laplace Approximations

Laplace approximations provide a simple yet limited method for approximating posterior distributions. Key considerations include:

- **Advantages:**
  - A significant improvement over doing nothing, despite being Gaussian-based and potentially mismatched to the true posterior shape.

  - Useful for model selection and online learning applications.

- **Limitations:**
  - The curvature at the mode may not reflect global properties of the distribution.
  - Computing Hessian matrices becomes challenging for high-dimensional problems.

- **Extensions:**
  - Can be applied to multi-class problems by adjusting the Hessian matrix.
  - Monte Carlo integration remains necessary for predictions, as no direct alternatives exist.

4.3.4 Variational Bayes

Definition 4.3.5 (Variational Inference)**.** The Laplace approximation isn't the only way to find a Gaussian posterior approximation. Another approach is **variational inference**, in which we directly search for a Gaussian:

$$q_\lambda(\mathbf{w}) = \mathcal{N}(\mathbf{w} \mid \boldsymbol{\mu}, \boldsymbol{\Sigma}) \approx p(\mathbf{w} \mid \mathcal{D}).$$

The most popular way to do this (for deep networks) is to minimize the **Kullback-Leibler (KL) divergence** with respect to $\lambda = \{\boldsymbol{\mu}, \boldsymbol{\Sigma}\}$,

$$D_{KL}(q_\lambda(\mathbf{w}) \| p(\mathbf{w} \mid \mathcal{D})) = \int q_\lambda(\mathbf{w}) \log \frac{q_\lambda(\mathbf{w})}{p(\mathbf{w} \mid \mathcal{D})} \, d\mathbf{w}.$$

The **KL divergence** isn't a "distance", but it measures dissimilarity between distributions:

$$D_{KL} \geq 0, \quad \text{and } D_{KL}(q \| p) = 0 \text{ only if } q = p \text{ almost everywhere.}$$

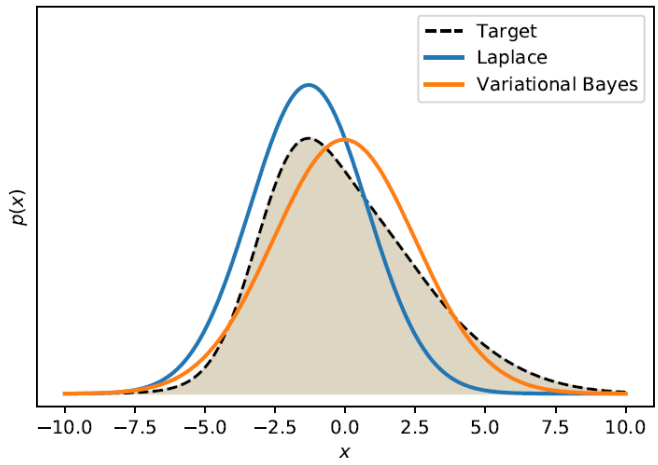


Figure 4.3: Variational Bayes Visual Illustration.

Theorem 4.3.6 (KL Divergence Behavior)**.** *The Kullback-Leibler (KL) divergence $D_{KL}(q_\lambda(\mathbf{z})\|p(\mathbf{z} \mid \mathcal{D}))$ is inherently asymmetric. When we maximize the Evidence Lower Bound (ELBO), we effectively minimize:*

$$D_{KL}(q_\lambda(\mathbf{z})\|p(\mathbf{z} \mid \mathcal{D})) = \mathbb{E}_{q_\lambda(\mathbf{z})}\left[\log \frac{q_\lambda(\mathbf{z})}{p(\mathbf{z} \mid \mathcal{D})}\right].$$

This form of the KL divergence exhibits mode-seeking behavior. It tends to concentrate probability mass near high-density regions of $p(\mathbf{z} \mid \mathcal{D})$ while avoiding regions where the true posterior is small, often resulting in an underestimation of the marginal variance.

In contrast, the reverse KL divergence,

$$D_{KL}(p(\mathbf{z} \mid \mathcal{D})\|q_\lambda(\mathbf{z})) = \mathbb{E}_{p(\mathbf{z}|\mathcal{D})}\left[\log \frac{p(\mathbf{z} \mid \mathcal{D})}{q_\lambda(\mathbf{z})}\right],$$

has a mass-covering or mean-seeking behavior. This version spreads probability mass to cover all modes of the target distribution and may lead to an overestimation of the variance to ensure that all significant regions of $p(\mathbf{z} \mid \mathcal{D})$ are covered.

Remark. *The practical choice of which KL divergence to minimize has important implications. Variational Bayes (VB) typically minimizes $D_{KL}(q_\lambda\|p)$, leading to approximations that can systematically underestimate uncertainty. In cases where the true posterior is multimodal, minimizing $D_{KL}(p\|q_\lambda)$ would be preferable to better represent multiple modes. However, computing expectations under $p(\mathbf{z} \mid \mathcal{D})$ can be challenging, which is why the mode-seeking form is often used.*

Theorem 4.3.7 (Evidence Lower Bound (ELBO))**.** *We can re-arrange the KL divergence so the integrand only contains quantities we can evaluate pointwise:*

$$\begin{aligned}
D_{KL}(q_\lambda(\mathbf{w})\|p(\mathbf{w} \mid \mathcal{D})) &= \mathbb{E}_{q_\lambda(\mathbf{w})}\left[\log \frac{q_\lambda(\mathbf{w})}{p(\mathbf{w} \mid \mathcal{D})}\right] \\
&= \mathbb{E}_{q_\lambda(\mathbf{w})}\left[\log \frac{q_\lambda(\mathbf{w})p(\mathcal{D})}{p(\mathbf{w}, \mathcal{D})}\right] \\
&= \mathbb{E}_{q_\lambda(\mathbf{w})}[\log p(\mathcal{D})] + \mathbb{E}_{q_\lambda(\mathbf{w})}\left[\log \frac{q_\lambda(\mathbf{w})}{p(\mathbf{w}, \mathcal{D})}\right] \\
&= \log p(\mathcal{D}) - \mathbb{E}_{q_\lambda(\mathbf{w})}\left[\log \frac{p(\mathbf{w}, \mathcal{D})}{q_\lambda(\mathbf{w})}\right],
\end{aligned}$$

where

$$\textit{ELBO} = \mathbb{E}_{q_\lambda(\mathbf{w})}\left[\log \frac{p(\mathbf{w}, \mathcal{D})}{q_\lambda(\mathbf{w})}\right].$$

Re-arranging the result on the last slide:

$$\mathbb{E}_{q_\lambda(\mathbf{w})}\left[\log \frac{p(\mathbf{w}, \mathcal{D})}{q_\lambda(\mathbf{w})}\right] = \log p(\mathcal{D}) - D_{KL}(q_\lambda(\mathbf{w})\|p(\mathbf{w} \mid \mathcal{D})),$$

where

$$\textit{ELBO}, \mathcal{L}(\lambda, \mathcal{D}) = \mathbb{E}_{q_\lambda(\mathbf{w})}\left[\log \frac{p(\mathbf{w}, \mathcal{D})}{q_\lambda(\mathbf{w})}\right],$$

$$\textit{const.} = \log p(\mathcal{D}),$$

and

$$D_{KL}(q_\lambda(\mathbf{w}) \| p(\mathbf{w} \mid \mathcal{D})) \geq 0.$$

Thus,

$$\mathcal{L}(\lambda, \mathcal{D}) \leq \log p(\mathcal{D}).$$

*This is why it is called the **ELBO**.*

Theorem 4.3.8 (Score Function Estimator)**.** *Interestingly, it's possible to construct a "score function" estimator in nearly any setting. To do this, we will need the following two useful identities.*

From the chain rule for logarithms:

$$\nabla_\lambda q_\lambda(\mathbf{w}) = q_\lambda(\mathbf{w}) \nabla_\lambda \log q_\lambda(\mathbf{w}).$$

Then, we have

$$\mathbb{E}_{q_\lambda(\mathbf{w})}[\nabla_\lambda \log q_\lambda(\mathbf{w})] = 0.$$

$$\mathbb{E}_{q_\lambda(\mathbf{w})}[\nabla_\lambda \log q_\lambda(\mathbf{w})] = \int q_\lambda(\mathbf{w}) \nabla_\lambda \log q_\lambda(\mathbf{w}) \, d\mathbf{w}$$

$$= \int \nabla_\lambda q_\lambda(\mathbf{w}) \, d\mathbf{w}$$

$$= \nabla_\lambda \int q_\lambda(\mathbf{w}) \, d\mathbf{w} = \nabla_\lambda 1 = 0.$$

$$\nabla_\lambda \mathbb{E}_{q_\lambda(\mathbf{w})}[f(\mathbf{w}, \lambda)] = \int \nabla_\lambda q_\lambda(\mathbf{w}) f(\mathbf{w}, \lambda) \, d\mathbf{w}$$

$$= \int f(\mathbf{w}, \lambda) \nabla_\lambda q_\lambda(\mathbf{w}) + q_\lambda(\mathbf{w}) \nabla_\lambda f(\mathbf{w}, \lambda) \, d\mathbf{w}$$

$$= \int q_\lambda(\mathbf{w}) (f(\mathbf{w}, \lambda) \nabla_\lambda \log q_\lambda(\mathbf{w}) + \nabla_\lambda f(\mathbf{w}, \lambda)) \, d\mathbf{w}$$

$$= \mathbb{E}_{q_\lambda(\mathbf{w})}[f(\mathbf{w}, \lambda) \nabla_\lambda \log q_\lambda(\mathbf{w})] + \mathbb{E}_{q_\lambda(\mathbf{w})}[\nabla_\lambda f(\mathbf{w}, \lambda)].$$

With $f(\mathbf{w}, \lambda) = \log p(\mathbf{w}, \mathcal{D}) - \log q_\lambda(\mathbf{w})$, then the second term:

$$\mathbb{E}_{q_\lambda(\mathbf{w})}\left[\nabla_\lambda (\log p(\mathbf{w}, \mathcal{D}) - \log q_\lambda(\mathbf{w}))\right] = -\mathbb{E}_{q_\lambda(\mathbf{w})}[\nabla_\lambda \log q_\lambda(\mathbf{w})] = 0.$$

This leaves us with

$$\nabla_\lambda \mathbb{E}_{q_\lambda(\mathbf{w})}[f(\mathbf{w},\lambda)] = \mathbb{E}_{q_\lambda(\mathbf{w})}\left[\log \frac{p(\mathbf{w},\mathcal{D})}{q_\lambda(\mathbf{w})} \nabla_\lambda \log q_\lambda(\mathbf{w})\right].$$

Since this is an expectation under $q_\lambda(\mathbf{w})$, we can Monte Carlo evaluate it with a finite set of samples.

Unfortunately, this typically leads to very high-variance estimates of the gradient. Intuitively, this estimator involves:

1. *Drawing samples from the approximation $q_\lambda(\mathbf{w})$.*
2. *Computing the gradients $\nabla_\lambda \log q_\lambda(\mathbf{w})$.*
3. *Estimating the expectation by "weighting" those according to $\log \frac{p(\mathbf{w},\mathcal{D})}{q_\lambda(\mathbf{w})}$.*

Theorem 4.3.9 (Re-parameterization Trick)**.** *Let $p(\epsilon) = \mathcal{N}(\epsilon \mid 0, \mathbf{I})$, and define $\mathbf{w} = r(\lambda,\epsilon) \equiv r(\boldsymbol{\mu},\boldsymbol{\Sigma},\epsilon) = \boldsymbol{\mu} + \boldsymbol{\Sigma}^{1/2}\epsilon$. Then, the gradient of the expected log-posterior with respect to λ can be written as:*

$$\nabla_\lambda \mathbb{E}_{q_\lambda(\mathbf{w})}\left[\log \frac{p(\mathbf{w},\mathcal{D})}{q_\lambda(\mathbf{w})}\right] = \nabla_\lambda \mathbb{E}_{p(\epsilon)}\left[\log \frac{p(r(\lambda,\epsilon),\mathcal{D})}{q_\lambda(r(\lambda,\epsilon))}\right].$$

By interchanging the expectation and the gradient, this becomes:

$$\mathbb{E}_{p(\epsilon)}\left[\nabla_\lambda \log \frac{p(r(\lambda,\epsilon),\mathcal{D})}{q_\lambda(r(\lambda,\epsilon))}\right].$$

This allows gradients to be computed efficiently using autodiff techniques.

The re-parameterization gradient can be computed as:

$$\nabla_\lambda \mathbb{E}_{q_\lambda(\mathbf{w})}\left[\log \frac{p(\mathbf{w},\mathcal{D})}{q_\lambda(\mathbf{w})}\right] = \mathbb{E}_{p(\epsilon)}\left[\nabla_\lambda \log \frac{p(r(\lambda,\epsilon),\mathcal{D})}{q_\lambda(r(\lambda,\epsilon))}\right].$$

In practice:

- *Draw a single sample $\epsilon \sim p(\epsilon)$, and use it to compute $\boldsymbol{\mu}$ and $\boldsymbol{\Sigma}$.*
- *The computation of $\boldsymbol{\mu}$ is exactly the same as if it were a fixed parameter $\mathbf{w}$ we were optimizing.*
- *Using a diagonal approximation $\boldsymbol{\Sigma} = diag(\sigma)$, we can estimate a per-weight posterior standard deviation with twice the number of parameters relative to MAP optimization.*

Theorem 4.3.10 (Mini-Batch Gradient Estimation)**.** *For models where $\log p(\mathbf{w},\mathcal{D}) = \sum_{i=1}^N \log p(y_i \mid x_i,\mathbf{w}) + \log p(\mathbf{w})$, we can use mini-batch gradient estimators:*

$$\mathbb{E}_{q_\lambda(\mathbf{w})}\left[\log \frac{p(\mathbf{w},\mathcal{D})}{q_\lambda(\mathbf{w})}\right] = \mathbb{E}_{q_\lambda(\mathbf{w})}\left[\sum_{i=1}^N \log p(y_i \mid x_i,\mathbf{w})\right] + \mathbb{E}_{q_\lambda(\mathbf{w})}\left[\log \frac{p(\mathbf{w})}{q_\lambda(\mathbf{w})}\right].$$

$$= \sum_{i=1}^{N} \mathbb{E}_{q_\lambda(\mathbf{w})}[\log p(y_i \mid x_i, \mathbf{w})] - D_{KL}(q_\lambda(\mathbf{w}) \| p(\mathbf{w})).$$

Using a mini-batch of size $M \ll N$, this becomes:

$$\mathbb{E}_{q_\lambda(\mathbf{w})}\left[\log \frac{p(\mathbf{w}, \mathcal{D})}{q_\lambda(\mathbf{w})}\right] \approx \frac{N}{M} \sum_{j=1}^{M} \mathbb{E}_{q_\lambda(\mathbf{w})}[\log p(y_j \mid x_j, \mathbf{w})] - D_{KL}(q_\lambda(\mathbf{w}) \| p(\mathbf{w})).$$

Remark. *Variational Bayes provides more flexibility than the Laplace approximation, as it can accommodate non-Gaussian approximations and scale to larger models.*

Definition 4.3.11 (MLE with Latent Variables using ELBO)**.** For models with latent variables z, the ELBO can be used as a surrogate for maximum likelihood estimation when the exact posterior $p(z|D)$ is intractable. The objective can be defined as:

$$\text{ELBO} = \mathbb{E}_{q_\lambda(z)}\left[\log p(z, D) - \log q_\lambda(z)\right],$$

which provides a lower bound on the log marginal likelihood $\log p(D)$.

Remark. *By simultaneously optimizing the ELBO over λ (variational parameters) and θ (model parameters), we can find approximate posteriors and maximum likelihood estimates, linking variational inference with model learning.*

Example 4.3.12 (Handling Discrete Latent Variables)**.** When latent variables z are discrete, the reparameterization trick cannot be applied directly. Instead, approaches such as score-function estimators or explicit marginalization are used. For example, if z follows a Bernoulli distribution, we can marginalize out z to compute expectations over discrete outcomes:

$$\mathbb{E}[f(z)] = \sum_{z \in \{0,1\}} p(z|D) f(z).$$

Variational Inference in Bayesian Logistic Regression

To apply variational inference to Bayesian logistic regression, we drop in our likelihood, prior, and approximate posterior:

$$\mathcal{L}(\lambda, \mathcal{D}) \approx \mathbb{E}_{p(\epsilon)}\left[\frac{N}{M} \sum_{j=1}^{M} \log \text{Bernoulli}(y_j \mid x_j, r(\lambda, \epsilon)) + \log \mathcal{N}(r(\lambda, \epsilon) \mid \mathbf{m}_0, \mathbf{S}_0) - \log \mathcal{N}(r(\lambda, \epsilon) \mid \boldsymbol{\mu}, \boldsymbol{\Sigma})\right].$$

Here, ϵ is sampled, and the ELBO is estimated using autodiff, followed by gradient descent updates.

To fit an approximate posterior $q(\mathbf{w} \mid \boldsymbol{\mu}, \boldsymbol{\Sigma})$, we must handle constraints:

- While $\boldsymbol{\mu}$ can take any value, $\boldsymbol{\Sigma}$ must be symmetric and positive definite to be a valid covariance matrix.
- The challenge lies in ensuring that gradient updates do not violate this constraint.

Transforming Parameters

To address constraints, current best practices involve transforming parameters to an unconstrained space:

- For $\mathbf{\Sigma} = \text{diag}(\sigma^2)$, optimize with respect to an "untransformed" variable z_d, such that $\sigma_d^2 = T(z_d)$, where $T(z)$ could be:

$$T(z) = \exp(z) \quad \text{or} \quad T(z) = \log(1 + \exp(z)).$$

- For arbitrary $\mathbf{\Sigma}$, parameterize as:

$$\mathbf{\Sigma} = \mathbf{A}\mathbf{A}^\top + \text{diag}(\sigma^2),$$

  where $\mathbf{A} \in \mathbb{R}^{D \times D}$ is an unconstrained matrix, and $\sigma \geq 0$ is enforced as in the diagonal case.

- "Low-rank plus diagonal" approximate posteriors are also common, using a rectangular matrix $\mathbf{A}$ to allow non-diagonal covariance matrices with far fewer than $\mathcal{O}(D^2)$ parameters.

4.3.5 Key Highlights of Laplace and Variational Bayes

- The KL divergence $D_{KL}(q_\lambda(\mathbf{w}) \| p(\mathbf{w}))$ has a closed-form solution if both distributions are Gaussians, enabling analytical gradient computation.
- Monte Carlo estimation is only required for gradients of $\mathbb{E}_{p(\epsilon)}[\log p(y_j \mid x_j, r(\lambda, \epsilon))]$.
- The variational Bayes approach presented here differs from traditional textbook approaches that derive closed-form coordinate ascent updates for conjugate exponential family models.
- By assuming model differentiability with respect to latent variables, this approach enables finding Gaussian posteriors for any differentiable model.
- Alternative non-Gaussian distributions for $q_\lambda(\mathbf{w})$ can also be explored.

4.4 Linear Classification

Definition 4.4.1 (Linear Classifier)**.** A linear classifier is a model that predicts the class label $y \in \{0, 1\}$ for an input x by using a linear function of the features:

$$y = \begin{cases} 1, & \text{if } w^\top x + b > 0, \\ 0, & \text{otherwise.} \end{cases}$$

Here, w represents the weights, and b is the bias term.

Definition 4.4.2 (Decision Boundary)**.** The decision boundary for a linear classifier is the hyperplane defined by:

$$w^\top x + b = 0.$$

This hyperplane separates the feature space into regions that correspond to different predicted class labels.

Definition 4.4.3 (Loss Functions for Linear Classification)**.**

- **Hinge Loss**: Commonly used for support vector machines (SVMs):

$$L(y, \hat{y}) = \max(0, 1 - y(w^\top x + b)).$$

- **Logistic Loss**: Used for logistic regression, also known as the cross-entropy loss:

$$L(y, \hat{y}) = -y \log(\sigma(w^\top x + b)) - (1 - y) \log(1 - \sigma(w^\top x + b)),$$

  where $\sigma(z) = \frac{1}{1+e^{-z}}$ is the sigmoid function.

Theorem 4.4.4 (Logistic Regression as a Probabilistic Model)**.** *Logistic regression can be interpreted as a probabilistic model, where the probability of $y = 1$ given x is modeled using the sigmoid function:*

$$\sigma(z) = \frac{1}{1 + e^{-z}},$$

which "squashes" a real value z into the interval $(0, 1)$. In the context of logistic regression, this is applied as:

$$p(y = 1|x) = \sigma(w^\top x + b) = \frac{1}{1 + e^{-(w^\top x + b)}}.$$

The sigmoid function ensures that the output is interpretable as a probability. The model is trained by maximizing the likelihood of the observed data, which is equivalent to minimizing the logistic loss.

Definition 4.4.5 (Regularized Linear Classifier)**.** To avoid overfitting, regularization terms can be added to the loss function. The objective then becomes:

$$\min L(y, \hat{y}) + \lambda r(w),$$

where $r(w)$ is a regularization function, such as $\|w\|_2^2$ for L2 regularization (ridge) or $\|w\|_1$ for L1 regularization (lasso).

Remark. *Regularization encourages simpler models by penalizing large weights, which helps prevent overfitting. L1 regularization can also induce sparsity, effectively performing feature selection.*

Definition 4.4.6 (Multi-class Classification)**.** For multi-class classification, assume we have K classes.

The probabilistic approach is to choose a likelihood that defines a categorical distribution over these K classes. This is typically done with the **softmax function**.

- We need a one-hot encoding of y_i, i.e., $\mathbf{y}_i$ is a vector of length K with a single non-zero entry, with $y_{i,k} = 1$ for the observed class k.
- We need a matrix $\mathbf{W} \in \mathbb{R}^{K \times D}$, in order to output all K values of $\mathbf{y}_i$.

Logits: Define logits, unnormalized log probabilities, as a vector $\mathbf{z}_i \in \mathbb{R}^K$, with

$$\mathbf{z}_i = \mathbf{W}^\top \phi(\mathbf{x}_i).$$

The softmax function computes a probability vector $\hat{\mathbf{y}}_i$ over the different classes:

$$\hat{y}_{i,k} = \frac{\exp(z_{i,k})}{\sum_{j=1}^{K} \exp(z_{i,j})}.$$

The log-likelihood of a discrete probability distribution yields the loss:

$$\ell(\hat{\mathbf{y}}_i, \mathbf{y}_i) = -\sum_{j=1}^{K} y_{i,j} \log \hat{y}_{i,j}.$$

4.5 Performance and Feature Selection

Definition 4.5.1 (Feature Engineering)**.** The performance of linear models depends significantly on the choice of input features $\phi(x)$. Transforming inputs into higher-dimensional feature spaces can enable better model fitting.

Theorem 4.5.2 (Random Features for Complex Data)**.** *Random cosine features, defined as:*

$$\phi(x) = \cos(Ax + b), \quad A \sim \mathcal{N}(0, 1),\ b \sim \mathcal{U}([0, 2\pi]),$$

allow linear models to approximate more complex patterns by expanding the feature space.

Example 4.5.3 (Feature Transformations in Classification)**.** Consider a two-class classification task. Transforming input data into a new feature space can improve separability. For instance, the following features:

$$\phi_1(x) = \exp\left(-\frac{(x+1)^2}{2}\right), \quad \phi_2(x) = \exp\left(-\frac{x^2}{2}\right),$$

project the data into a transformed space where linear classification becomes feasible.

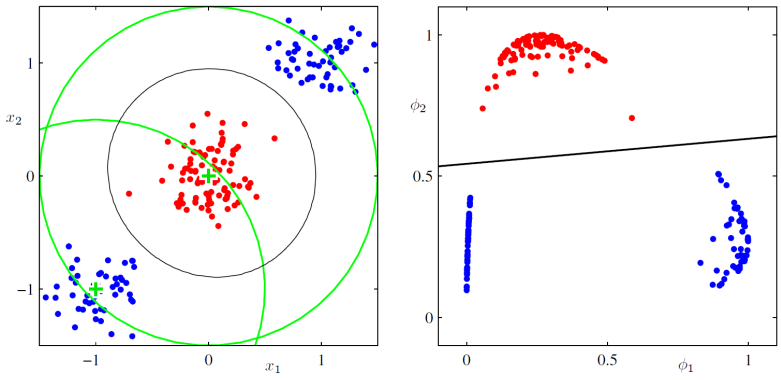


Figure 4.4: Two-class Classification.

4.5.1 Performance Depends on the Features

How well a linear learner works depends on the input features ϕ:

- If you have a lot of inputs x, and they are fairly "useful," you'll likely do well.
- If your inputs are good but your target variable isn't linear in x, define appropriate transforms $\phi(x)$.

What if you have no idea how to construct $\phi(x)$, and x on its own isn't good enough? In this case:

- **Kernel Methods:** These methods create large (potentially infinite) numbers of implicit features.
- **Deep Learning:** Deep models learn appro

Chapter 5

Optimization

5.1 Taylor Expansion and Function Approximation

Definition 5.1.1 (First and Second Order Approximations)**.** Optimization methods rely on approximating functions. Using a Taylor expansion:

- **First-order approximation**:

$$f(\theta + \delta) \approx f(\theta) + \delta^\top \nabla f(\theta),$$

  where $\nabla f(\theta)$ is the gradient.

- **Second-order approximation**:

$$f(\theta + \delta) \approx f(\theta) + \delta^\top \nabla f(\theta) + \frac{1}{2}\delta^\top H(\theta)\delta,$$

  where $H(\theta)$ is the Hessian matrix for second derivative.

5.2 Gradient Descent

Definition 5.2.1 (Gradient Descent)**.** Gradient descent is an iterative first-order optimization method. As part of an iterative optimization algorithm, for $\boldsymbol{\theta}_{t+1} \approx \boldsymbol{\theta}_t$, the objective function $f(\boldsymbol{\theta})$ can be approximated as:

$$f(\boldsymbol{\theta}_{t+1}) \approx f(\boldsymbol{\theta}_t) + (\boldsymbol{\theta}_{t+1} - \boldsymbol{\theta}_t)^\top \nabla f(\boldsymbol{\theta}_t),$$

where $\mathbf{s} = \boldsymbol{\theta}_{t+1} - \boldsymbol{\theta}_t$ represents the step direction.

The simplest steepest descent approach chooses a step direction $\mathbf{s}$ (or equivalently, the next location $\boldsymbol{\theta}_{t+1}$) as:

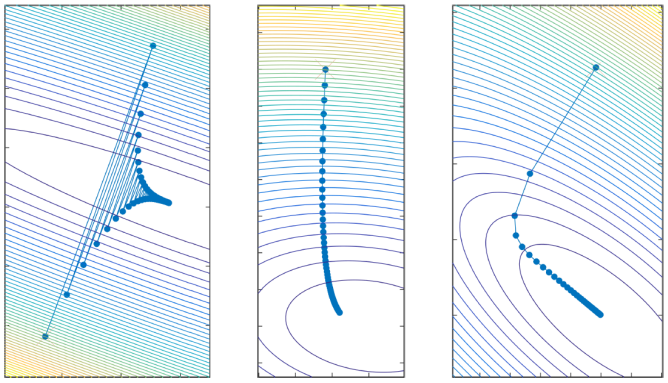
$$\mathbf{s} = \boldsymbol{\theta}_{t+1} - \boldsymbol{\theta}_t = -\epsilon \nabla f(\boldsymbol{\theta}_t),$$

for a small learning rate $\epsilon > 0$.

Assuming this approximation holds, f decreases at each iteration because:

$$f(\boldsymbol{\theta}_{t+1}) \approx f(\boldsymbol{\theta}_t) - \epsilon \nabla f(\boldsymbol{\theta}_t)^\top \nabla f(\boldsymbol{\theta}_t) \leq f(\boldsymbol{\theta}_t),$$

since $\nabla f(\boldsymbol{\theta}_t)^\top \nabla f(\boldsymbol{\theta}_t) \geq 0$.



(a) Learning rate too large

(b) Learning rate too small

(c) Learning rate OK

Figure 5.1: Learning Rates for Gradient Descent.

Remark. *Choosing an appropriate learning rate ϵ is crucial. Too large a rate may cause divergence, while too small a rate slows convergence. This is usually determined by experimenting with different values or learning schedules.*

5.2.1 Stochastic Gradient Descent (SGD)

Definition 5.2.2 (Stochastic Gradient Descent)**.** Instead of computing gradients over the entire dataset, SGD approximates the gradient using a single data point or a mini-batch:

$$\theta_{t+1} = \theta_t - \epsilon \nabla L_i(\theta),$$

where $L_i(\theta)$ is the loss for the i-th data point. This makes the optimization faster, especially for large datasets.

Definition 5.2.3 (Mini-batch Stochastic Gradient Descent)**. Problem:** Gradient estimates from one datapoint are noisy.

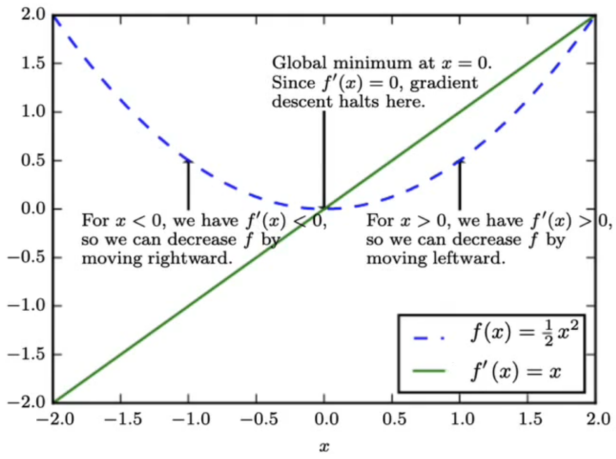


Figure 5.2: Gradient Descent Example.

A common solution is to use mini-batches of training examples and compute the gradient estimate based on those. For $j = 1, \ldots, M$ subsampled from $i = 1, \ldots, N$,

$$\nabla_\theta L(\theta) = \frac{1}{N} \sum_{i=1}^{N} \nabla_\theta L_i(\theta) = \frac{1}{M} \sum_{j=1}^{M} \nabla_\theta L_j(\theta).$$

Trade-offs for different batch sizes:

- Large batch size: less noisy estimates, convergence might be faster (in number of iterations).
- Smaller batch size: faster parameter updates.
- GPUs usually allow for larger batch sizes.

Stochastic gradient descent introduces noise into the training process. Each update does not necessarily point toward the direction of true gradient. It may even increase the loss! Reducing learning rate reduces fluctuations (but may slow down convergence): a trade-off between learning rate and batch size.

5.3 Momentum and Adaptive Learning Rates

Definition 5.3.1 (Momentum)**.** Momentum is a technique used in optimization to limit "zig-zag" behavior by incorporating a moving average of previous updates into the step direction.

Moving Average: For a set of numbers $x_1, \ldots, x_t$, the moving average a_t is given by:

$$a_t = \frac{1}{t}\sum_{\tau=1}^{t} x_\tau = \frac{1}{t}(x_t + (t-1)a_{t-1}) = \epsilon_t x_t + \mu_t a_{t-1},$$

where $0 \leq \mu_t \leq 1$, ϵ_t is suitably chosen, and μ_t controls the contribution of recent values.

Update Rule: In momentum, this concept is applied to optimization updates as:

$$\widetilde{g}_{t+1} = \mu\widetilde{g}_t - \epsilon\nabla f(\boldsymbol{\theta}_t),$$

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \widetilde{g}_{t+1}.$$

Instead of the simple update rule $-\epsilon\nabla f(\boldsymbol{\theta})$, the moving average $\widetilde{g}_{t+1}$ defines the step direction.

Key Features of Momentum:

- **Speed and Stability:** Momentum accelerates convergence for smooth objectives by reducing oscillations and moving in the average direction.
- **Noise Reduction:** By averaging gradients, momentum reduces the impact of noisy updates, resulting in smoother optimization paths.
- **Avoiding Saddle Points:** It helps escape saddle points (where gradients are zero but not minima) by maintaining updates through flat regions.

Definition 5.3.2 (Adaptive Learning Rates: Adagrad, RMSprop, and Adam)**.** These methods adjust the learning rate based on past gradients:

- **Adagrad**: Useful for objective functions sensitive to small changes in some directions but less sensitive in others. Adagrad sets a different learning rate for each parameter:

$$g_{t,i} = (\nabla L(\theta^t))_i, \quad G_{t,ii} = \sum_{\tau=1}^{t} g_{\tau,i}^2,$$

$$\theta_i^{t+1} = \theta_i^t - \frac{\eta}{\sqrt{G_{t,ii} + \epsilon}} g_{t,i}.$$

  Good: Per-parameter learning rates scale inversely with the squared magnitude of gradients, often using $\eta = 1.0$ or $\eta = 0.1$ as default values. However, it may slow down excessively due to accumulated gradients.

  Not Good: The accumulated gradients eventually make learning very slow and learning rate never recovers.

- **RMSprop**:RMSprop fixes the problem with Adagrad vanishing learning rates by using moving average of past squared gradients instead (like momentum is doing for the gradients themselves):

$$s_{t,i} = \gamma s_{t-1,i} + (1-\gamma) g_{t,i}^2$$

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{s_{t,i}+\epsilon}} g_{t,i}$$

  The algorithm was first presented in Geoff Hinton Coursera course (Lecture 6e).

  (Note: You also might see this quoted as using a fixed "rolling window" average of the past few gradient updates, rather than a moving average.)

- **Adam**: Adam is a very popular optimizer, specifically for deep learning models. Essentially, it combines RMSprop with momentum:

$$m_{t,i} = \beta m_{t-1,i} + (1-\beta) g_{t,i}$$

$$\hat{m}_{t,i} = \frac{m_{t,i}}{1-\beta^t}$$

$$s_{t,i} = \gamma s_{t-1,i} + (1-\gamma) g_{t,i}^2$$

$$\hat{s}_{t,i} = \frac{s_{t,i}}{1-\gamma^t}$$

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{\hat{s}_{t,i}}+\epsilon} \hat{m}_{t,i}$$

  The $\hat{m}$ and $\hat{s}$ are "bias-corrected", accounting for the fact that we typically initialize with $m_0 = s_0 = 0$, which is intended to help in the early iterations.

  All of these algorithms are implemented in common ML and DL libraries.

5.4 Second-Order Methods

Definition 5.4.1 (Newton's Method)**.** Newton's method uses a second-order approximation to find the minimum:

$$\theta_{t+1} = \theta_t - \epsilon \mathbf{H}^{-1} \nabla f(\theta_t),$$

where $\mathbf{H}$ is the Hessian matrix.

For **Newton's method**, we consider

$$f(\theta + s) \approx f(\theta) + s^\top \nabla f(\theta) + \frac{1}{2} s^\top H(\theta) s$$

and find the step direction by minimizing it with respect to s. Differentiating the right-hand side yields

$$\nabla_s f(\theta + s) \approx \nabla f(\theta) + H(\theta)s;$$

setting equal to zero and solving for s, we find the minimum to be at

$$s = -H(\theta)^{-1}\nabla f(\theta).$$

Often, a learning rate $\epsilon \in (0, 1)$ is included to dampen the update,

$$\theta_{t+1} = \theta_t - \epsilon H^{-1}\nabla f(\theta_t).$$

When we ran gradient descent, we made a linear approximation to $f(\theta)$ and then stepped a short distance ϵ.

Here, we instead make a quadratic approximation to $f(\theta)$, and move *directly to its minimum*. The rough idea (Newton's method):

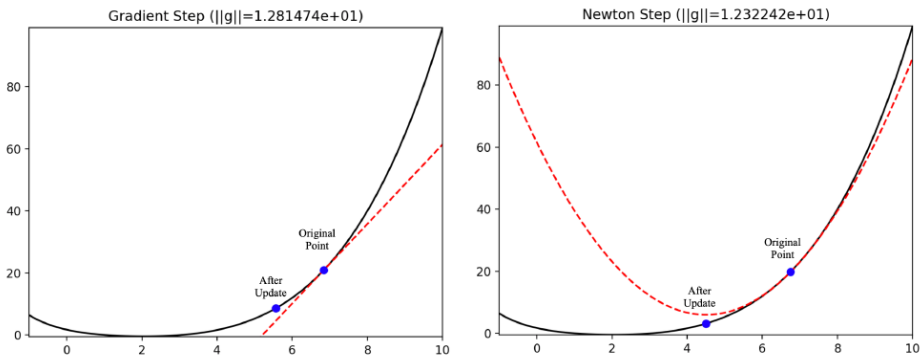


Figure 5.3: Newton's Method Visual.

This method can converge rapidly but has some practical challenges:

- Storing and solving the linear system involving the Hessian $\mathbf{H}^{-1}\nabla f$ is computationally expensive. One way to reduce the cost is to use a diagonal approximation to the Hessian.
- Newton's method does not always guarantee a downhill step, especially if the step size ϵ is too large or if $\mathbf{H}$ is not positive definite.
- If $\mathbf{H}$ is positive definite, a line search can be employed to ensure the step direction leads to a non-trivial descent.
- In general, Newton's method is not well-suited for stochastic objectives due to the variability in gradient and Hessian estimates.

Definition 5.4.2 (Quasi-Newton Methods)**.** Quasi-Newton methods are designed to avoid the full computation of the Hessian matrix, using an approximation instead. The goal is to find a matrix $\mathbf{B}$ that satisfies:

$$\nabla f(\theta_{t+1}) = \nabla f(\theta_t) + \mathbf{B}(\theta_{t+1} - \theta_t),$$

which corresponds to a Taylor expansion of the gradient. This leads to the optimization step:

$$\theta_{t+1} = \theta_t - \epsilon \mathbf{B}_t^{-1} \nabla f(\theta_t).$$

- In the Broyden-Fletcher-Goldfarb-Shannon (BFGS) method, an approximate inverse Hessian $\mathbf{B}_t^{-1}$ is constructed iteratively.
- Limited-memory BFGS (L-BFGS) is a practical variant that reduces memory requirements, making it suitable for high-dimensional problems.
- The update steps in these methods share similarities with adaptive gradient methods like Adagrad and RMSprop, as they incorporate past gradient information.

Definition 5.4.3 (Outer Product Approximation)**.** Suppose we are optimizing a function that decomposes into a sum of squared errors, e.g.

$$L(\theta) = \frac{\lambda}{2} \sum_{n=1}^{N} \big(y_n - f(\theta, x_n)\big)^2.$$

The Hessian of this overall loss function has entries

$$\frac{\partial L}{\partial \theta_i \partial \theta_j} = \lambda \sum_{n=1}^{N} \Big((f(\theta, x_n) - y_n) \frac{\partial f_n}{\partial \theta_i} \frac{\partial f_n}{\partial \theta_j} + \frac{\partial f_n}{\partial \theta_i} \frac{\partial^2 f_n}{\partial \theta_i \partial \theta_j} \Big).$$

Near the minimum, the first term will be small, as $f(\theta, x_n) \approx y_n$. This suggests that we can construct a Hessian approximation by using the outer product of the gradients,

$$B = \lambda \sum_{i=1}^{N} \nabla f_n(\theta) \nabla f_n(\theta)^\top.$$

Remark (Summary)**.** *[leftmargin=*]*

- *Gradient descent: general-purpose optimization of real-valued scalar functions.*
- *Stochastic / mini-batch methods can speed up runtime by fast parallel estimation of an approximate step direction.*
- *Momentum and adaptive learning rates can speed up rate of optimization (and will be essential for training deep networks with millions of parameters!).*

- *Second-order methods are memory-intensive and not always "worth it" on stochastic objectives, but can converge in far fewer iterations. There's some current work on approximating Hessians for deep networks that we may discuss later.*

5.5 Practical Considerations

- **Learning Rate Tuning**: Experiment with different rates or schedules to improve convergence.
- **Batch Size Trade-offs**: Larger batches provide smoother gradients, while smaller ones offer quicker updates.
- **Handling Non-Convexity**: Non-convex loss functions may lead to local minima; adaptive methods can help escape such traps.

Example 5.5.1 (Mini-Batch Gradient Descent)**.** Using mini-batches, we approximate gradients:

$$\nabla L(\theta) \approx \frac{1}{M} \sum_{j=1}^{M} \nabla L_j(\theta),$$

where M is the batch size. This approach balances gradient smoothness and update speed.

5.6 Automatic Differentiation

5.6.1 Modes of Auto-differentiation

Autodiff takes a function $f(\theta)$ and returns an exact value of the gradient g

- **Forward Mode**: Efficient when the number of inputs is less than the number of outputs.
- **Reverse Mode**: Efficient when there are many inputs but few outputs, such as in deep learning where it enables backpropagation.

Forward Mode

Definition 5.6.1 (Forward Mode AD)**. Forward mode** autodiff calculates a derivative along a single direction (e.g. a single partial derivative or directional derivative).

- It's low memory reqruiments and easy to implement. It also computes the full gradient, costing $O(Df)$.

- It introduces an idempotent infinitesimal variable ϵ, such that $\epsilon^2 = 0$. By defining a function $\texttt{DualPart}(\cdot)$, which extracts the "dual" component (i.e., the coefficient of ϵ), we can calculate derivatives directly.
- Forward computation produces a scalar output eventually as the root node of the computation graph.

Remark (Primitive Operations and Overloading)**.** *Forward mode is ingenious in its simplicity. It uses* ***dual arithmetic****, which resembles complex numbers.*

- *Define an idempotent infinitesimal variable ϵ, such that $\epsilon^2 = 0$.*
- *Define a function DualPart$(\cdot)$, which returns the "dual" component (i.e., the coefficient of ϵ).*
- *For any function $f(x)$, we have $f'(x) =$ DualPart$(f(x))$.*

This requires some primitive operations to be overloaded in order to compute derivatives:

$$(v_1 + \epsilon) + v_2 = v_1 + v_2 + \epsilon$$

$$(v_1 + \epsilon)v_2 = v_1 v_2 + v_2\epsilon$$

$$\sin(v_1 + \epsilon) = \sin(v_1) + \cos(v_1)\epsilon$$

Example 5.6.2 (Single Variable)**.** Consider $f(x) = x^2$. Using forward mode:

$$f(x + \epsilon) = (x + \epsilon)^2 = x^2 + 2x\epsilon,$$

where the derivative $f'(x) = 2x$ is obtained as the coefficient of ϵ. We can express this as:

$$f'(x) = \texttt{DualPart}(f(x + \epsilon)) = \texttt{DualPart}(x^2 + 2x\epsilon) = 2x.$$

Example 5.6.3 (Partial Derivatives for Multivariable Functions)**.** For a function $f(v_1, v_2) = v_1 v_2 - \sin(v_2)$, the two partial derivatives are computed separately:

$$f(v_1 + \epsilon, v_2) = v_1 v_2 - \sin(v_2) + v_2\epsilon \quad \Rightarrow \quad \frac{\partial f}{\partial v_1} = v_2.$$

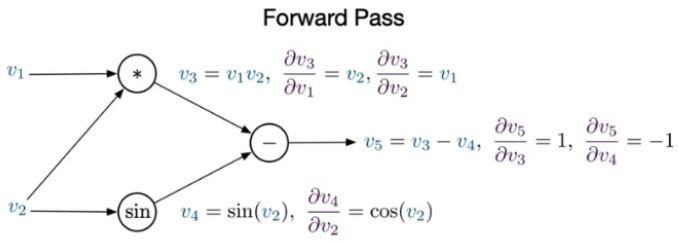
Similarly:

$$f(v_1, v_2 + \epsilon) = v_1(v_2 + \epsilon) - \sin(v_2 + \epsilon) = v_1 v_2 - \sin(v_2) + (v_1 - \cos(v_2))\epsilon,$$

resulting in:

$$\frac{\partial f}{\partial v_2} = v_1 - \cos(v_2).$$

These examples illustrate how forward mode efficiently computes derivatives by following the chain rule through overloaded operations.



$$
\begin{array}{lll}
v_1 \quad v_2 \quad v_3 = v_1 v_2 & v_4 = \sin(v_2) & v_5 = v_3 - v_4 \\
\dfrac{\partial v_3}{\partial v_2} = v_1 & \dfrac{\partial v_4}{\partial v_2} = \cos(v_2) & \dfrac{\partial v_5}{\partial v_2} = \dfrac{\partial v_5}{\partial v_3}\dfrac{\partial v_3}{\partial v_2} + \dfrac{\partial v_5}{\partial v_4}\dfrac{\partial v_4}{\partial v_2} \\
 & & = (1)(v_1) + (-1)(\cos(v_2))
\end{array}
$$

Figure 5.4: Forward Computation Graph Illustration of the Multivariate Example.

Reverse Mode

Definition 5.6.4 (Reverse Mode AD)**.** Reverse mode involves computing a full gradient by first running a forward pass to store intermediate values, followed by a backward pass to propagate derivatives.

- Requires keeping around pointers from every computed value to its parents.
- Memory intensive (can't free any intermediate computed values. . .), but runtime is $O(f)$, the same as the original function.
- The **backprop** algorithm is a special case of reverse-mode autodiff.

Example 5.6.5 (Multi-variable Functions(same as forward))**.** The calculation of partial derivatives in the reverse pass follows the chain rule. Each derivative is computed by propagating gradients backward through the computational graph. The steps are as follows:

$$\frac{\partial f}{\partial v_5} = 1$$

$$\frac{\partial f}{\partial v_4} = \frac{\partial f}{\partial v_5} \cdot (-1) = (1)(-1)$$

$$\frac{\partial f}{\partial v_3} = \frac{\partial f}{\partial v_5} \cdot (1) = (1)(1)$$

$$\frac{\partial f}{\partial v_2} = \frac{\partial f}{\partial v_3} \cdot v_1 + \frac{\partial f}{\partial v_4} \cdot \cos(v_2)$$

$$= (1)(v_1) + (-1)(\cos(v_2))$$

$$\frac{\partial f}{\partial v_1} = \frac{\partial f}{\partial v_3} \cdot v_2 = (1)(v_2)$$

Reverse Pass Overview: 1. **Initialization:** Start by setting $\frac{\partial f}{\partial v_5} = 1$, as v_5 represents the output

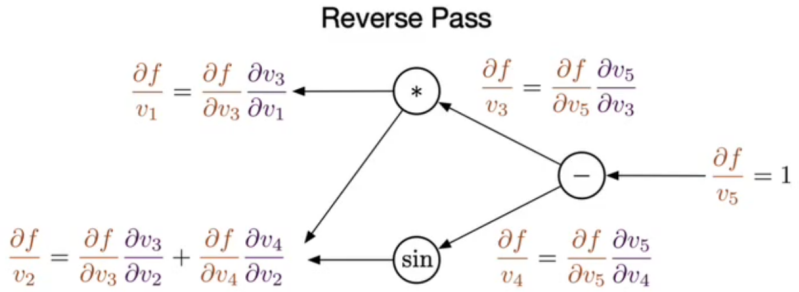


Figure 5.5: Reverse Computation Graph.

of the function f.

2. **Backward Propagation:** - Propagate gradients through addition and subtraction operations:

$$\frac{\partial f}{\partial v_4} = \frac{\partial f}{\partial v_5} \cdot (-1)$$

$$\frac{\partial f}{\partial v_3} = \frac{\partial f}{\partial v_5} \cdot (1)$$

- Propagate through the sine function:

$$\frac{\partial f}{\partial v_2} = \frac{\partial f}{\partial v_3} \cdot v_1 + \frac{\partial f}{\partial v_4} \cdot \cos(v_2)$$

- Propagate through multiplication:

$$\frac{\partial f}{\partial v_1} = \frac{\partial f}{\partial v_3} \cdot v_2$$

Summary: The reverse pass computes derivatives by systematically applying the chain rule at each node of the computational graph. This approach efficiently calculates gradients for all variables in the graph by propagating information from the output back to the inputs.

Reverse Mode Overview

Reverse mode differentiation is an efficient technique to compute gradients for functions with a scalar output. The process can be summarized as follows:

1. **Constructing the Computation Graph:**
   - During the forward computation, construct a "computation graph" where each node:
     - Stores a pointer to its parent nodes.
     - Computes the partial derivatives with respect to each input.
2. **Producing the Scalar Output:**
   - The forward computation produces a scalar output, which acts as the "root" node of the computation graph.
3. **Backward Computation:**
   - Run a backward computation, rolling backward through the computation graph in reverse order:
     - At each intermediate node, accumulate the "incoming" partial derivatives from its parents.
     - At each leaf node, report the entry of the gradient.

Preventing Duplicate Computation

Reverse mode differentiation prevents duplicate computation by reusing shared branches of the computation graph. For example, consider the derivative of f_3 with respect to x_1:

$$\frac{df_3}{dx_1} = \frac{\partial f_3}{\partial f_2} \cdot \frac{\partial f_2}{\partial f_1} \cdot \frac{\partial f_1}{\partial x_1}.$$

Similarly, the derivative with respect to x_2 is:

$$\frac{df_3}{dx_2} = \frac{\partial f_3}{\partial f_2} \cdot \frac{\partial f_2}{\partial f_1} \cdot \frac{\partial f_1}{\partial x_2}.$$

In both cases, the term $\frac{\partial f_3}{\partial f_2}$ and the computation of $\frac{\partial f_2}{\partial f_1}$ are shared across branches. This reuse avoids redundant computations and makes reverse mode differentiation highly efficient for functions with a scalar output.

Chapter 6

Model Comparison

6.1 Marginal Likelihood (Model Evidence)

Definition 6.1.1 (Model Evidence)**.** The model evidence, also known as the marginal likelihood, is defined as:

$$p(D|M) = \int p(D|\theta, M)p(\theta|M)d\theta,$$

where $p(D|\theta, M)$ is the likelihood of the data given parameters θ, and $p(\theta|M)$ is the prior distribution over parameters under model M. It represents the probability of the data D with the parameters marginalized out, effectively acting as a normalizing constant for the posterior. We see that as model complexity increases ((more possible "explanations" for the data, it reduces training errors but also introduce variance.

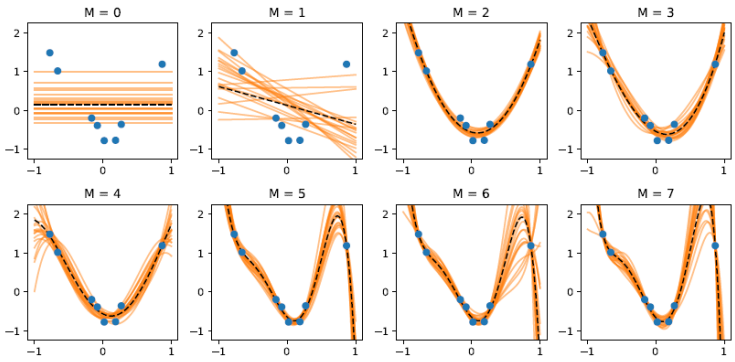


Figure 6.1: Bayesian Estimation with different Model Complexity.

6.2 Bayesian Occam's Razor

Definition 6.2.1 (Occam Factor)**.** The Occam factor penalizes model complexity, reflecting the preference for simpler models when two models fit the data equally well. It arises naturally from the Laplace approximation:

$$\log p(D) \approx \log p(D|\theta^*) + \log p(\theta^*) + \frac{D}{2}\log(2\pi) - \frac{1}{2}\log|\Lambda|,$$

where the term $-\frac{1}{2}\log|\Lambda|$ acts as a complexity penalty.

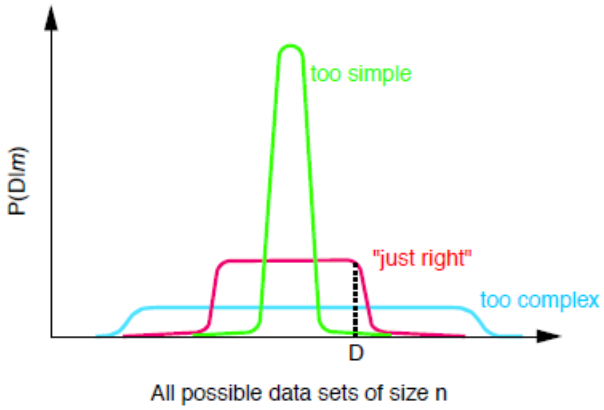


Figure 6.2: Bayesian Occam's Razor.

Remark. *The Occam factor penalizes models that have a narrow posterior around the MAP estimate, as such models are more "peaky" and thus less likely to generalize. It encourages broad, smooth posterior distributions, thus incorporating a regularization effect.*

Theorem 6.2.2 (Bayesian Model Comparison as Inference)**.** *To compare two models M_0 and M_1, we use the ratio of their marginal likelihoods (also called the Bayes factor):*

$$\frac{p(M_0|D)}{p(M_1|D)} = \frac{p(D|M_0)p(M_0)}{p(D|M_1)p(M_1)},$$

where $p(M_0|D)$ and $p(M_1|D)$ are the posterior probabilities of the models, and $p(M_0)$, $p(M_1)$ are the prior probabilities. This ratio evaluates which model is more probable given the observed data. Higher marginal likelihoods give better model.

Theorem 6.2.3 (Estimating Marginal Likelihood via Laplace Approximation)**.** *The **Laplace approximation** approximates posterior distributions with Gaussians.*

To estimate the normalizing constant instead of the posterior, start by taking the same Taylor approximation at the mode $\theta^* \in \mathbb{R}^D$,

$$\log p(D,\theta) \approx \log p(D,\theta^*) + \frac{1}{2}(\theta-\theta^*)^\top H(\theta-\theta^*),$$

$$p(D,\theta) \approx p(D,\theta^*)\exp\left\{-\frac{1}{2}(\theta-\theta^*)^\top \Lambda(\theta-\theta^*)\right\},$$

where $\Lambda = -H = -\nabla\nabla \log p(D,\theta) = -\nabla\nabla \log p(\theta|D)$.

This suggests we can (approximately) normalize the distribution by normalizing the approximation, as

$$p(D) = \int p(D,\theta)d\theta.$$

First, note that for $\theta \in \mathbb{R}^D$

$$\int \exp\left\{-\frac{1}{2}(\theta-\theta^*)^\top \Lambda(\theta-\theta^*)\right\} d\theta = \frac{(2\pi)^{D/2}}{|\Lambda|^{1/2}}.$$

(This is exactly the normalization constant of a multivariate Gaussian.)

$$\begin{aligned}\int p(D,\theta)d\theta &\approx \int p(D,\theta^*)p(\theta^*)\exp\left\{-\frac{1}{2}(\theta-\theta^*)^\top \Lambda(\theta-\theta^*)\right\} d\theta \\ &= p(D,\theta^*)p(\theta^*)\int \exp\left\{-\frac{1}{2}(\theta-\theta^*)^\top \Lambda(\theta-\theta^*)\right\} d\theta \\ &= p(D,\theta^*)p(\theta^*)\frac{(2\pi)^{D/2}}{|\Lambda|^{1/2}}.\end{aligned}$$

Taking logarithms of both sides, we have the approximation

$$\log p(D) \approx \log p(D|\theta^*) + \log p(\theta^*) + \frac{D}{2}\log(2\pi) - \frac{1}{2}\log|\Lambda| \quad \underbrace{\text{“Occam factor”}}.$$

*The **Occam factor** here is a general means for penalizing model complexity. Of particular note:*

[leftmargin=]Partially, this depends on the prior. But it is not just the prior! It also depends on $\Lambda = -\nabla\nabla \log p(\theta|D)|_{\theta=\theta^*}$, the Hessian at the MAP estimate θ^*.*

Intuitively, this will prefer “broader” rather than “peaky” posteriors.

Remark. *The Laplace approximation favors models that have higher $p(D|\theta^*, M)$ and broader, rather than peaky, posterior distributions, implicitly penalizing overly complex models.*

6.3 Practical Considerations for Approximating Model Evidence

- **Laplace Approximation**: Effective for models where the posterior is well-approximated by a Gaussian, but can struggle with multimodal or skewed distributions.
- **ELBO as Proxy**: Variational methods, using the Evidence Lower Bound (ELBO), can sometimes serve as a proxy for estimating model evidence. The tightness of the bound determines the accuracy.
- **Importance Sampling**: An alternative method to approximate the marginal likelihood, particularly useful when exact integration is not feasible.

Chapter 7

Deep Learning Overview

7.1 Overview of Learning Paradigms

Theorem 7.1.1. *Traditional machine learning involves estimating parameters w in a fixed feature space $\Phi(x)$. This process, often called **shallow learning**, follows the formulation:*

$$f(x) = w^\top \Phi(x),$$

where:

- Φ*: Predefined feature map.*
- w*: Model parameters learned during training.*

Deep learning differs by jointly learning the feature representation Φ along with the model parameters w. This is formulated as:

$$f(x) = w^\top \Phi_\theta(x),$$

where θ includes all parameters of the model (e.g., weights and biases in a neural network).

7.2 Basic Components of Deep Learning Models

7.2.1 Hidden Layers and Nonlinearities

Definition 7.2.1 (The simplest "deep" model)**.** The simplest deep learning model is a single-layer perceptron:

$$f(x) = w^\top g(Ax + b),$$

where:

- $g(\cdot)$: Nonlinear activation function.
- A: Weights of the hidden layer.
- b: Biases of the hidden layer.
- $h = g(Ax + b)$: Hidden layer representation. With $\mathbf{x} \in \mathbb{R}^D$, the hidden layer can be any $\mathbf{h} \in \mathbb{R}^H$, even with $H \gg D$.

  Generically, we will refer to networks as $f_\theta(\mathbf{x})$, where θ represents all the parameters in the model. In this case, $\theta = \{\mathbf{A}, \mathbf{b}, \mathbf{w}\}$.

Remark. *Common activation functions $g(\cdot)$ include:*

- ***ReLU (Rectified Linear Unit):*** $g(z) = \max(z, 0)$.
- ***Softplus:*** $g(z) = \log(1 + \exp(z))$.
- ***Sigmoid:*** $g(z) = \frac{1}{1+\exp(-z)}$.
- ***Tanh:*** $g(z) = \frac{\exp(z)-\exp(-z)}{\exp(z)+\exp(-z)}$.

7.2.2 Multi-Layer Perceptron (MLP)

Definition 7.2.2. A multi-layer perceptron (MLP), or feed-forward network, is a neural network that stacks multiple layers of hidden units. It is defined as:

$$h_1 = g(A_1 x + b_1), \quad h_2 = g(A_2 h_1 + b_2), \quad \ldots, \quad f(x) = w^\top h_L,$$

where:

- L is the number of layers.
- $g(\cdot)$ is a non-linear activation function (e.g., ReLU, sigmoid, or tanh).
- $\mathbf{A}_i$ and $\mathbf{b}_i$ are the weight matrix and bias vector for the i-th layer, respectively.
- h_i represents the hidden layer activations at layer i.

Theorem 7.2.3 (MLP: Hugely Popular framework due to Flexibility)**.** *MLPs are trained using the backpropagation algorithm combined with stochastic gradient descent (SGD). The training process involves:*

- ***Automatic Differentiation (Autodiff):*** *A computational tool that constructs a computational graph of $f_\theta(x)$ and computes gradients $\nabla_\theta f_\theta(x)$ with runtime complexity comparable to evaluating $f_\theta(x)$ itself.*

- ***Stochastic Gradient Descent (SGD):*** *Optimizes the parameters $\theta = \{\mathbf{A}, \mathbf{b}, \mathbf{w}\}$ by iteratively minimizing a loss function.*
- *Classic models include convolutions (for images), recurrent temporal models, etc.; new architectures developed constantly*

Deep learning libraries such as PyTorch and TensorFlow make it straightforward to construct and differentiate computation graphs, enabling efficient implementation of backpropagation and gradient descent.

Remark. *The flexibility of MLPs stems from their ability to define both the feature transformation $\Phi(x)$ and the overall model $f_\theta(x)$ as arbitrary code, with parameters to be optimized. This makes MLPs adaptable to:*

- ***Classic Architectures:*** *Extensions include convolutional models for image data and recurrent models for temporal sequences.*
- ***Evolving Architectures:*** *New designs are continuously developed to address diverse tasks and data modalities.*

Moreover, MLPs are capable of approximating complex, non-linear functions when equipped with sufficient depth and appropriate activation functions.

7.3 Theoretical Foundations

7.3.1 Universal Approximation Theorems

Theorem 7.3.1. *A single-layer neural network with a sufficient number of hidden units can approximate any Borel-measurable function to arbitrary precision. However:*

- *The required number of hidden units may be exponentially large.*
- *There are no guarantees for any particular optimization procedures to converge to the optimal network.*

7.3.2 Depth vs. Width Trade-offs

Remark. *For a fixed number of parameters, should you have more "narrow" layers, or fewer "wide" layers?*
Generally the consensus is that ***it's useful for networks to be deeper****:*

- *Often much fewer-parameter "deep" networks are needed, relative to a wide single-hidden-layer network.*

- *Composing "simple" features into more complex features makes sense conceptually.*
- *Empirically, it seems like deeper networks generalize better to unseen data.*
- *They also tend to be easier to optimize by gradient descent.*

7.3.3 Activation Function

1. **Sigmoid Function:**
$$g(z) = \frac{1}{1 + e^{-z}}$$
   - Outputs values in the range $(0, 1)$.
   - Useful for binary classification problems, as it maps inputs to probabilities.
   - **Limitations:**
     - Gradients can vanish for large positive or negative inputs, leading to slow learning.
     - Activations saturate at extreme values, causing poor gradient flow during backpropagation.
2. **Tanh (Hyperbolic Tangent):**
$$g(z) = \frac{\exp(z) - \exp(-z)}{\exp(z) + \exp(-z)}$$
   - Outputs values in the range $(-1, 1)$, making it zero-centered.
   - Preferred over the sigmoid function in deep networks due to improved gradient flow near zero.
   - **Limitations:**
     - Similar to sigmoid, suffers from vanishing gradients for large input magnitudes.
3. **ReLU (Rectified Linear Unit):**
$$g(z) = \max(0, z)$$
   - Outputs values in the range $[0, \infty)$, introducing sparsity in activations.
   - Computationally efficient and commonly used in modern neural networks.
   - **Advantages:**
     - Avoids vanishing gradients by having a linear derivative for positive inputs.
   - **Limitations:**
     - Gradients are zero for negative inputs, leading to "dead neurons" if many neurons become inactive.

4. **Softplus:**

$$g(z) = \log(1 + e^z)$$

- A smooth approximation of the ReLU function.
- Outputs values in the range $(0, \infty)$.
- **Advantages:**
  - Avoids the problem of "dead neurons" since it is always differentiable and non-zero.
- **Limitations:**
  - Computationally more expensive than ReLU.

Remark. *The choice of activation function depends on the task and network architecture. While ReLU and its variants dominate modern applications due to their computational efficiency and gradient flow properties, functions like sigmoid and tanh still find use in specialized tasks, such as binary classification or autoencoders.*

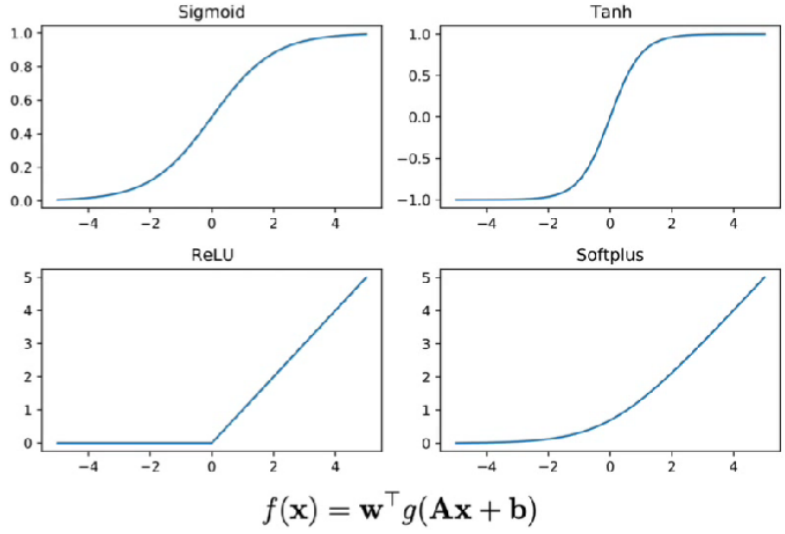


Figure 7.1: Activation Functions.

Remark. ***What about a linear activation function?*** *If we choose $g(z) = z$, the implications are as follows:*

- ***Bad news:*** *The model reduces to a linear transformation, regardless of the number of layers:*

$$f(x) = w^\top h_1 = w^\top (A_1 h_2 + b_1) = w^\top A_1 h_2 + w^\top b_1,$$

where $w^\top A_1$ and $w^\top b_1$ can be treated as new weight and bias terms, w' and b', resulting in:

$$f(x) = w'^\top h_2 + b'.$$

This linearity removes the network's ability to model complex, non-linear patterns.

- ***Possibly interesting:*** *When dealing with a function mapping $\mathbb{R}^{D_1} \to \mathbb{R}^{D_2}$ with D_1, D_2 being large, a linear activation might still be useful in specific contexts:*

$$f(x) = W^\top h = W^\top (Ax + b) = W^\top Ax + W^\top b,$$

where:

  - *$W \in \mathbb{R}^{H \times D_2}$, $A \in \mathbb{R}^{H \times D_1}$.*
  - *The projection matrix $W^\top A \in \mathbb{R}^{D_2 \times D_1}$.*

If the hidden dimension H is small, this approach introduces fewer parameters ($H(D_1 + D_2)$) compared to directly learning a large projection matrix ($D_1 D_2$).

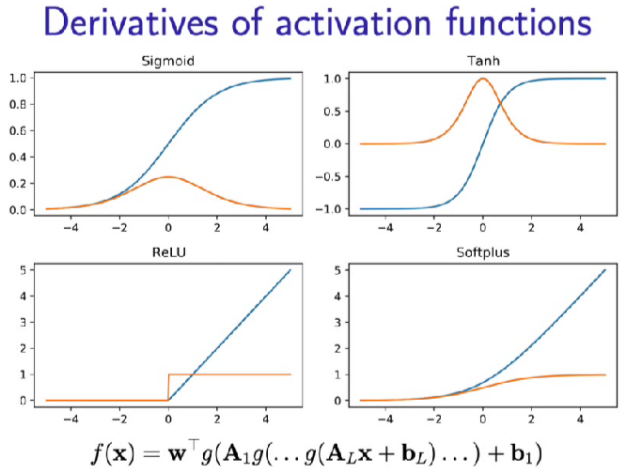


Figure 7.2: Derivatives of Activation Functions.

7.3.4 Loss Functions for Deep Learning

Remark. *Loss functions are a critical component of deep learning models, as they quantify the difference between predicted outputs $\hat{y}$ and ground truth y. Most standard loss functions have a* ***probabilistic interpretation****, linking them to specific likelihood functions under certain assumptions. These include:*

Common Loss Functions and Their Probabilistic Interpretations

1. **Squared Error (Mean Squared Error - MSE):**

   - Probabilistic interpretation: Squared error corresponds to the negative log-likelihood of a Gaussian distribution with a fixed variance:

   $$-\log \mathcal{N}(y \mid \hat{y}, \sigma^2) = \frac{1}{2}\log(2\pi\sigma^2) + \frac{1}{2\sigma^2}(y - \hat{y})^2.$$

   Ignoring constants, this reduces to:

   $$\alpha(y - \hat{y})^2 + \text{const},$$

   where α is a constant dependent on σ^2.

   - **Usage:** Squared error is commonly used for regression tasks where the output is continuous.

2. **Binary Cross Entropy (Log-Loss):**

   - Probabilistic interpretation: Cross-entropy loss corresponds to the negative log-likelihood of a Bernoulli distribution:

   $$-\log \text{Bernoulli}(y \mid \hat{y}) = y \log \hat{y} + (1 - y)\log(1 - \hat{y}).$$

   - **Usage:** Binary cross-entropy is used in binary classification tasks, where the output is a probability in the range $[0, 1]$.

3. **Absolute Error (Mean Absolute Error - MAE):**

   - Probabilistic interpretation: Absolute error corresponds to the negative log-likelihood of a Laplace distribution:

   $$-\log \text{Laplace}(y \mid \hat{y}) = \frac{|y - \hat{y}|}{b} + \text{const},$$

   where b is the scale parameter.

   - **Usage:** Absolute error is robust to outliers and is used in regression tasks where the data distribution has heavy tails.

Key Observations

Remark. *The choice of loss function depends on the nature of the task and the data:*

- ***Gaussian Likelihood (Squared Error):*** *Assumes normally distributed noise and penalizes large deviations quadratically.*
- ***Bernoulli Likelihood (Binary Cross-Entropy):*** *Assumes binary outcomes and penalizes incorrect predictions logarithmically.*
- ***Laplace Likelihood (Absolute Error):*** *Assumes noise with heavier tails, making it more robust to outliers.*

These probabilistic interpretations help connect machine learning to statistical modeling, guiding principled choices of loss functions for specific problems.

Practical Considerations

Remark. *In practice, the loss function not only guides optimization but also affects the convergence behavior and generalization of the model:*

- *Squared error tends to amplify the impact of outliers due to its quadratic penalty.*
- *Cross-entropy is preferred for classification tasks as it focuses on the confidence of the predicted probabilities.*
- *Absolute error is robust to noisy data and outliers but can lead to slower convergence due to its non-differentiability at $y = \hat{y}$.*

7.4 Typical Challenges in Deep Learning

7.4.1 Vanishing and Exploding Gradients

Remark. *Gradients can diminish or grow exponentially with depth, causing:*

- *Slow convergence or failure in training deep networks.*
- *Mitigation through ReLU activation, which avoids saturation.*

7.4.2 Model Complexity and Regularization

Definition 7.4.1. Regularization and model complexity are two interconnected aspects of deep learning. Controlling the complexity of a model helps to balance its capacity to fit training data and its ability to generalize to unseen data. Regularization techniques aim to prevent overfitting by introducing constraints or modifications to the training process:

- **L2 Regularization:** Adds a penalty proportional to the squared magnitude of the model parameters to the loss function:
$$\mathcal{L}_{\text{reg}} = \mathcal{L} + \lambda \|\theta\|_2^2,$$
where λ controls the strength of the penalty.
- **Dropout:** During training, randomly disables a fraction of neurons in each layer, forcing the network to develop redundant representations that are robust to missing activations.
- **Early Stopping:** Monitors validation error during training and halts optimization when the error stops decreasing, preventing the model from overfitting to the training data.

Remark. ***Controlling Model Complexity:*** *The complexity of a neural network is influenced by the number of hidden units and layers. For example:*

- *With a small number of hidden units (e.g., $M = 1$), the model underfits, failing to capture the underlying patterns in the data.*
- *Increasing the number of hidden units (e.g., $M = 3$ or $M = 10$) allows the model to learn more complex functions but risks overfitting if the capacity becomes too large.*

This trade-off highlights the importance of carefully selecting model size to match the complexity of the task.

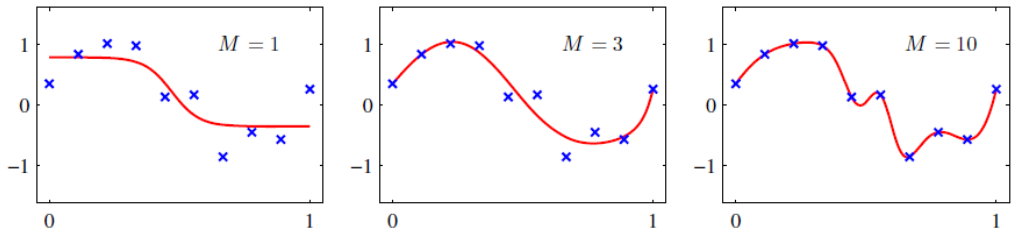


Figure 7.3: Controlling Model Complexity.

Remark. ***Balancing Complexity and Regularization:***

- *A too-simple model (e.g., with very few hidden units or regularized too heavily) will underfit, failing to capture the nuances of the data.*
- *A too-complex model (e.g., with many hidden units and no regularization) will overfit, memorizing the training data rather than generalizing to new inputs.*

Techniques like adjusting the number of hidden units and applying regularization allow for finding the optimal balance.

7.5 Bayesian Deep Learning

7.5.1 Bayesian Approaches

Definition 7.5.1. Bayesian deep learning estimates distributions over weights and activations by modeling uncertainty in both predictions and parameters. The predictive distribution is given by:

$$p(y \mid \mathbf{x}) = \int p(y \mid \theta) p(\theta \mid \mathcal{D}) d\theta,$$

where:

- θ represents the model parameters (e.g., weights and biases).
- $\mathcal{D}$ represents the training dataset.
- $p(\theta \mid \mathcal{D})$ is the posterior distribution over parameters.

Remark. ***Bayesian and Non-Bayesian Approaches:***

$$p(y|\mathbf{x}) = p(y|\mathbf{w}^\top \boldsymbol{\phi}(\mathbf{x}))$$

- ***Not deep, not Bayesian:*** *Fix ϕ, the feature mapping, and find a point estimate $\hat{\mathbf{w}}$ of the weights.*
- ***Not deep, Bayesian:*** *Fix ϕ, estimate the posterior distribution $p(\mathbf{w} \mid \mathcal{D})$ over the weights.*

$$p(y|\mathbf{x}) = p(y|\mathbf{w}^\top g(\mathbf{A}_1 g(\ldots \mathbf{x}) + \mathbf{b}_1))$$

- ***Deep, but not Bayesian:*** *Allow the feature mapping to depend on learned parameters, $\phi_\ell(\mathbf{h}) = g(\mathbf{A}_\ell \mathbf{h} + \mathbf{b}_\ell)$, and find point estimates for all parameters.*
- ***Deep and Bayesian:*** *Estimate the posterior distribution over all parameters, $p(\{\mathbf{A}_\ell, \mathbf{b}_\ell\}_{\ell=1}^L \mid \mathcal{D})$, and use it to compute the predictive distribution.*
- ***Deep and Sorta Bayesian (Bayesian Last Layer):***

  - *Find point estimates for the hidden layers,* $\{\mathbf{A}_\ell, \mathbf{b}_\ell\}_{\ell=1}^{L-1}$.
  - *Estimate a posterior distribution for the weights of the last layer,* $p(\mathbf{w} \mid \mathcal{D})$.

- ***Deep and Sorta Bayesian (Deep Ensembles):*** *Train* S *different models with random initializations to obtain* S *sets of point estimates,* $\{\hat{\mathbf{w}}^s, \{\hat{\mathbf{A}}_\ell^s, \hat{\mathbf{b}}_\ell^s\}_{\ell=1}^L\}_{s=1}^S$*, and use these as samples from an implicit posterior.*

7.5.2 Bayesian Estimation and Learning in Deep Models

From Linear to Nonlinear Classification with Bayesian Methods

- In linear classification, Bayesian methods estimate the posterior distribution over weights $p(\mathbf{w} \mid \mathcal{D})$, where $\mathcal{D}$ is the training dataset:
  - The maximum a posteriori (MAP) estimate of weights, $\mathbf{w}^\star$, defines the best-fit decision boundary:
$$\mathbf{w}^\star = \arg\max_{\mathbf{w}} p(\mathbf{w} \mid \mathcal{D}).$$
  - Samples $\mathbf{w}^{(k)} \sim p(\mathbf{w} \mid \mathcal{D})$ from the posterior can be used to visualize uncertainty, as shown by multiple plausible decision boundaries.
- For problems where data is not linearly separable, feature transformations $\phi(x)$ can project the data into a higher-dimensional space where linear separation becomes possible. For example:
$$\phi(x) = [x, (x - 5.5)^2 - 10]^\top.$$
- This highlights the importance of designing feature maps or learning them using deeper networks.

Deep Bayesian Networks: From Random to Learned Representations

- **Random One-Layer Networks:**
  - A random one-layer network generates features $\mathbf{h}$ using fixed random parameters:
$$\mathbf{h} = g(\mathbf{A}x + \mathbf{b}),$$
    where $\mathbf{A}$ and $\mathbf{b}$ are sampled from $\mathcal{N}(0, 1)$, and $g(z) = \log(1 + \exp(z))$ is the softplus activation function.
  - This approach defines a 2D feature space, enabling nonlinear classification without learning the parameters.

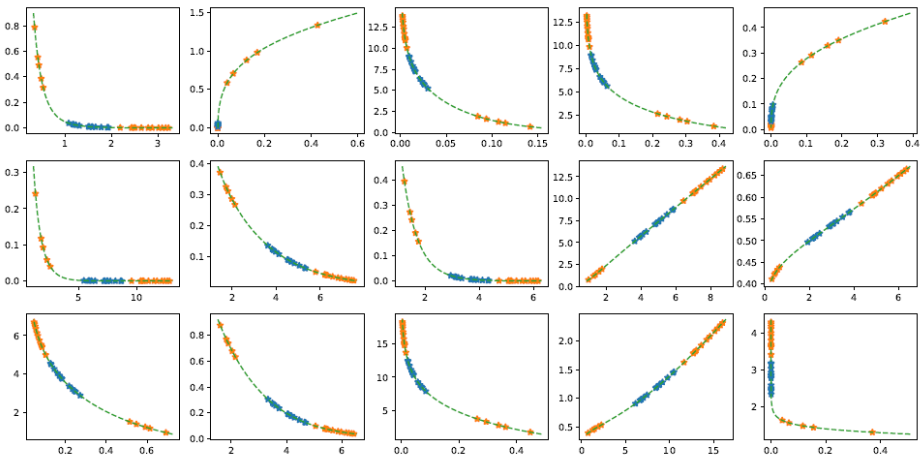


Figure 7.4: Random one-layer networks.

- **Random two-layer networks:**
  - Increasing depth allows the network to represent more complex functions. For example, in a two-layer network:

$$p(y = 1 \mid x) = \sigma(\mathbf{w}^\top \mathbf{h}_1), \quad \mathbf{h}_1 = g(\mathbf{A}_1 \mathbf{h}_2 + \mathbf{b}_1), \quad \mathbf{h}_2 = g(\mathbf{A}_2 x + \mathbf{b}_2),$$

    where the parameters are still sampled randomly.

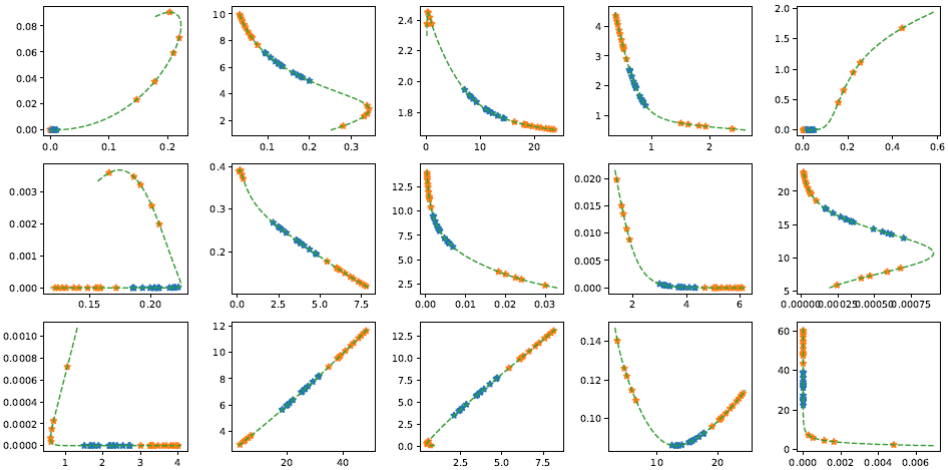


Figure 7.5: Random two-layer networks.

- **Learning a two-layer network:**
  - Bayesian learning involves inferring the posterior distribution over parameters:

$$p(y = 1 \mid x) = \sigma(\mathbf{w}^\top g(\mathbf{A}_1 g(\mathbf{A}_2 x + \mathbf{b}_2) + \mathbf{b}_1)).$$

  - Maximum likelihood or MAP estimation learns a usable representation space, improving classification accuracy.

Uncertainty Quantification in Bayesian Models

- Bayesian inference can capture uncertainty in both model parameters and features:
  - Running MCMC over all parameters provides posterior samples, visualizing the variability in decision boundaries or feature representations.
  - This uncertainty modeling is particularly useful in applications requiring reliable confidence estimates, such as medical diagnosis or safety-critical systems.

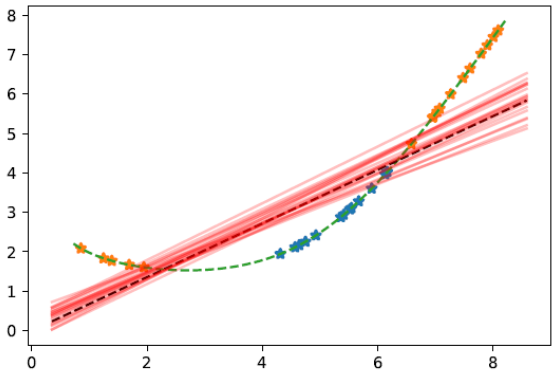


Figure 7.6: Uncertainty Quantification in the Features.

- The variability in feature spaces and decision boundaries emphasizes the interpretability of Bayesian deep learning, offering insights beyond point estimates.

7.5.3 A Multi-Modal Posterior Distribution

Remark. *Retraining a Bayesian model from scratch often reveals interesting characteristics about its posterior distribution:*

- *Ideally, we might expect no significant changes when retraining under the same data and conditions.*

- *Observations, however, reveal that:*

  - *The model seems to fit well, producing consistent performance metrics.*
  - *Posterior samples yield a variety of decision boundaries and representations, demonstrating the model's uncertainty and the multi-modal nature of the posterior.*

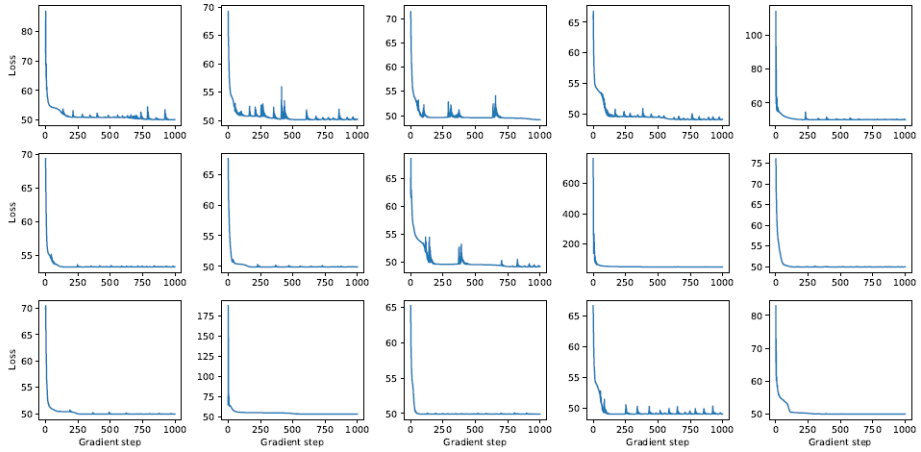


Figure 7.7: Loss trajectories from 15 optimization runs, showcasing variability in the optimization process.

Learning Different Embeddings

Remark. *When the same model is trained multiple times with different initializations, it can produce diverse embeddings:*

- *Across 15 runs, each trained model achieves high accuracy (often 100*

- *This highlights the sensitivity of deep models to initialization and the multi-modal nature of their solution space.*

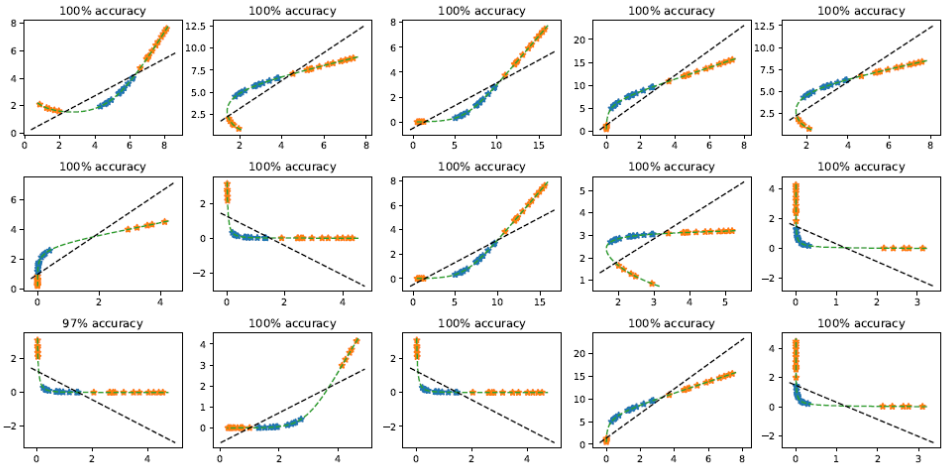


Figure 7.8: Decision boundaries and embeddings learned over 15 runs. Despite achieving similar accuracies, the learned representations vary significantly.

Comparison with MCMC Samples

Remark. *Bayesian inference with Markov Chain Monte Carlo (MCMC) provides a principled way to sample from the posterior distribution:*

- *By sampling from the posterior, we observe the variability among decision boundaries.*

- *This variability aligns with the diversity observed in embeddings learned through optimization-based methods.*

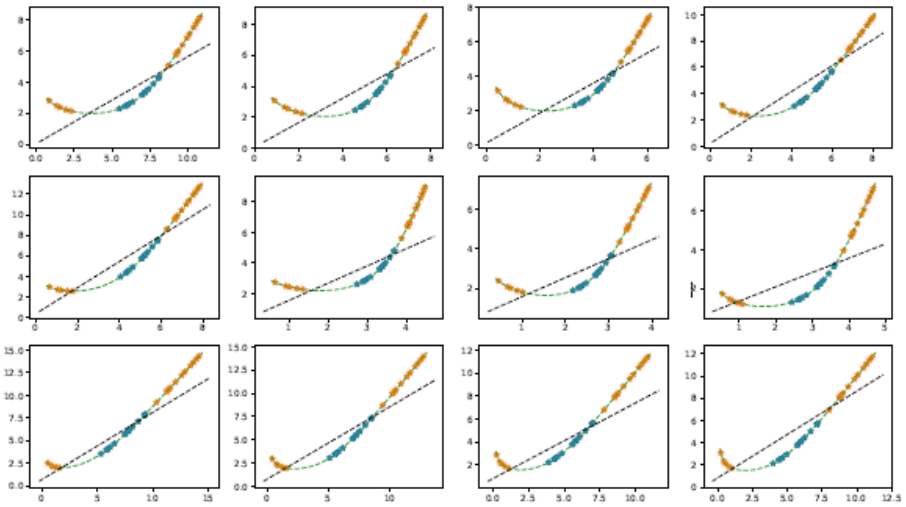


Figure 7.9: Variability in decision boundaries obtained from MCMC sampling, illustrating uncertainty in the posterior distribution.

Local and Global Uncertainty in Bayesian Deep Learning

Remark. *Bayesian deep learning allows us to capture different types of uncertainty in models, with particular focus on local and global variations:*

- *Many Bayesian inference algorithms struggle with disconnected modes in the posterior distribution, making it challenging to explore the entire solution space.*
- *Popular algorithms include:*
  - *Laplace approximation.*
  - *Variational Bayes.*
  - *Markov Chain Monte Carlo (MCMC).*
- *Ensemble methods can effectively capture "global" variation in uncertainty by combining predictions from multiple independently trained models.*

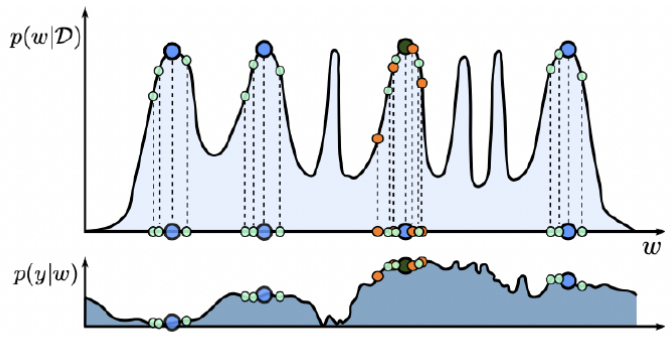


Figure 7.10: Illustration of local and global uncertainty in Bayesian models. Ensembles capture global variation, while certain algorithms struggle with disconnected modes.

Comparing Predictive Distributions

Remark. *Predictive distributions $p(y \mid x, \theta)$ allow us to evaluate the variability in predictions across different model configurations:*

- *Variability in predictive distributions can result from differences in optimized parameters θ.*
- *Bayesian methods, such as MCMC, provide principled ways to explore these distributions.*

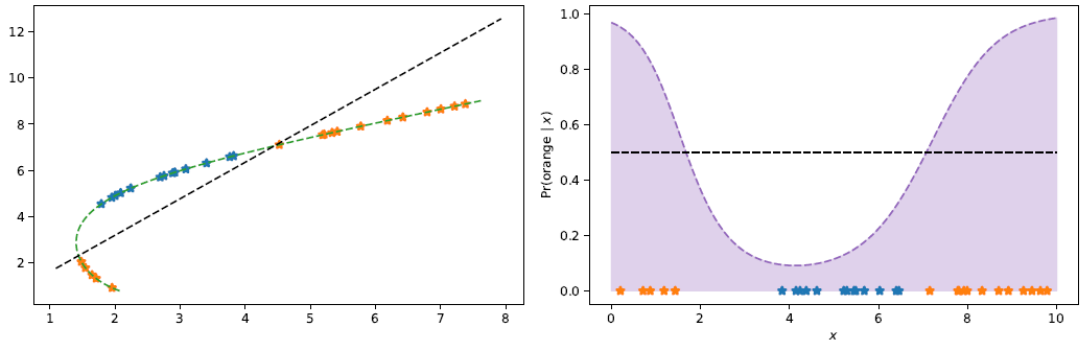


Figure 7.11: Predictive distributions across different optimized parameters, showing variability in predictions and confidence intervals.

MCMC Sampling for the Final Layer

Remark. *Sampling from the posterior over weights* $\mathbf{w}$ *in the final layer provides an effective way to capture uncertainty in predictions:*

- *MCMC sampling focuses on the variability in the weights* $\mathbf{w}$ *while keeping other parameters fixed.*
- *This approach can highlight how uncertainty in the final layer propagates to predictions.*

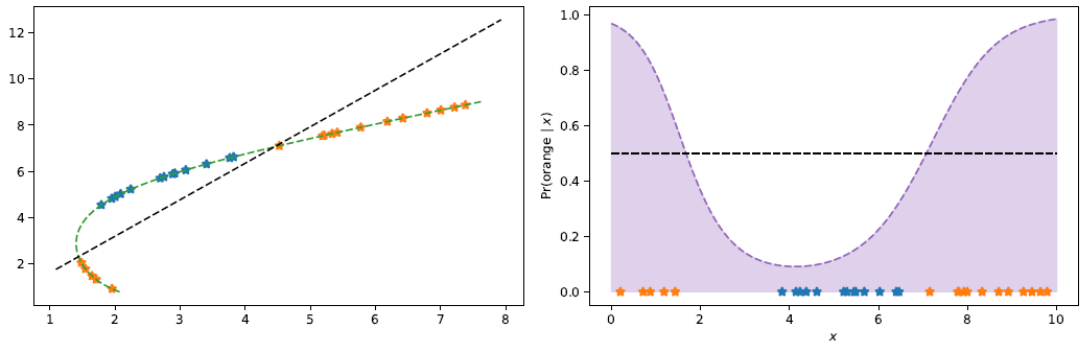


Figure 7.12: Predictive distributions obtained from MCMC sampling over the final layer weights, illustrating uncertainty in predictions.

Summary of Bayesian Deep Learning Concepts

Remark. *Key takeaways from Bayesian deep learning include:*

- ***Representation Learning:*** *Deep learning learns feature representations, while Bayesian deep learning extends this to learning distributions over representations.*
- ***Inference Challenges:*** *Handling large datasets and non-convex posteriors remains challenging for Bayesian inference algorithms.*
- ***Uncertainty Matters:*** *It is non-obvious which type of posterior uncertainty is most relevant:*
  - *Should we focus on local or global uncertainty?*
  - *Should we model uncertainty over the entire representation or just the predictor?*

7.6 Convolutional Neural Network

7.6.1 Key Principles of Model Architecture

- **Guided by Data Structure:** Deep learning architectures should align with the structure of the data, leveraging inherent patterns or invariances.
- **Convolutions:** A foundational method for encoding invariances in image data, enabling parameter efficiency and local feature extraction.

7.6.2 Desired Model Invariances

Remark (Label-Invariant Transformations)**.** *To maintain classification consistency (e.g., detecting a bird regardless of location), we could do the following operations to the training set that won't affect the class label:*

- *Horizontal flipping.*
- *Small translations, rotations, or scaling.*
- *Small scaling up or down*
- *Color adjustments (e.g., grayscale conversion).*

Definition 7.6.1 (Data Augmentation)**.** A technique to enhance model robustness by applying label-preserving transformations to training data, thereby creating a richer dataset. It takes the original training set and constructs a new, larger training set that applies label-preserving transformations. Transformations can also be applied dynamically during mini-batch processing. However, this gives you extra things for the network to learn! So we ideally want to build invariances directly into the network architecture.

7.6.3 Convolutional Layers as Invariant Features

Definition 7.6.2 (Convolution)**.** Convolution is an operation that combines a signal $f(t)$ and a weighting function (or filter) $g(t)$ to produce a weighted sum of overlapping elements. For continuous functions, it is defined as:

$$(f * g)(t) = \int f(a)g(t-a)\,da.$$

This operation is commutative, meaning:

$$f * g = g * f.$$

Remark (Intuition for Convolution)**.** *Convolution involves sliding the filter $g(t)$ over the input signal $f(t)$ and computing a weighted overlap. At each position t, the values of $f(t)$ and $g(t)$ are multiplied and summed, creating a new signal that reflects the alignment of features in $f(t)$ with the filter $g(t)$.*

Definition 7.6.3 (Discrete Convolution)**.** For a discrete input $x \in \mathbb{R}^D$ and a filter $w \in \mathbb{R}^M$ ($M < D$), a one-dimensional discrete convolution is defined as:

$$(x * w)_d = \sum_m x_m w_{d-m}.$$

Padding: Inputs may be padded with zeros or boundary values to preserve dimensions. For example:

- **Valid Convolution:** Only regions where the filter fully overlaps the input are computed.
- **Zero Padding:** The input is extended by zeros at the boundaries, e.g., $[0, 0, x_1, x_2, \ldots, x_D, 0, 0]$.

Example 7.6.4 (Discrete Convolution As Matrix Multiplication)**.** The convolution can be represented as a specialized linear operation. For example, with $x = [x_1, x_2, x_3, x_4, x_5]^T$ and $w = [w_1, w_2, w_3]$, the convolution can be expressed as:

$$x * w = \begin{bmatrix} w_3 & w_2 & w_1 & 0 & 0 \\ 0 & w_3 & w_2 & w_1 & 0 \\ 0 & 0 & w_3 & w_2 & w_1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{bmatrix}.$$

This example computes only the *valid* section of the convolution.

Remark (Handling Edges)**.** *Alternative approaches include padding the input with zeros, boundary values, or reflections, allowing computations over the entire input range. This is common in image processing where preserving spatial dimensions is important.*

Remark (Convolution on 2D Images)**.** *For 2D inputs, such as an image $X \in \mathbb{R}^{H \times W}$ and a filter $W \in \mathbb{R}^{M \times N}$, the convolution is defined as:*

$$(X * W)_{i,j} = \sum_m \sum_n x_{m,n} w_{i-m,j-n}.$$

While this can theoretically be represented as a doubly block circulant matrix, such constructions are avoided in practice due to inefficiency.

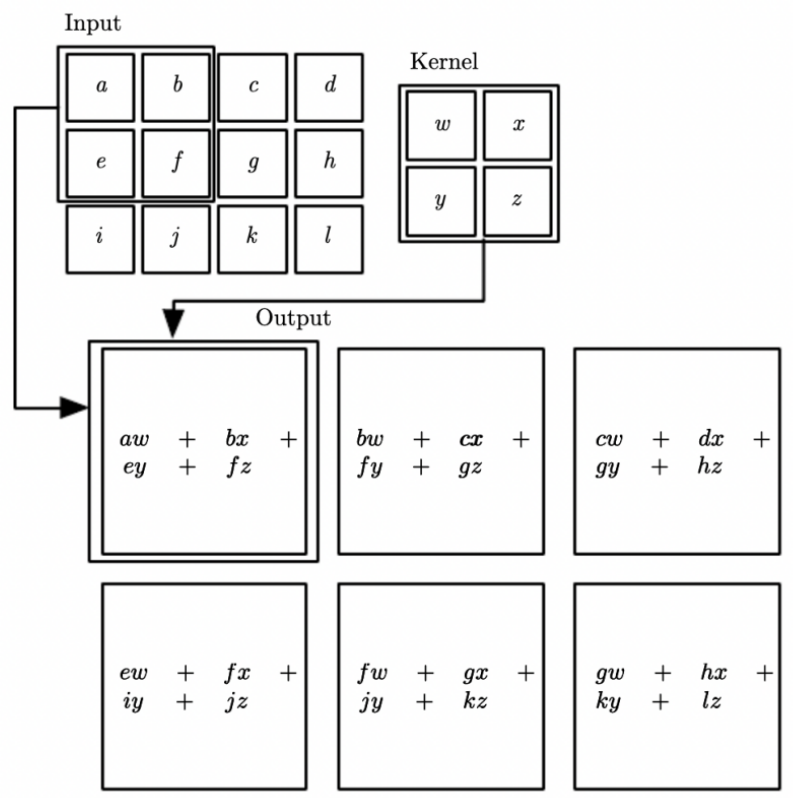


Figure 7.13: 2D Convolution Schematic.

Example 7.6.5 (Edge Detection via 2D Convolution)**.** A simple edge detection filter $W = \begin{bmatrix} 1 & -1 \end{bmatrix}$ applied to an image X highlights changes in intensity across adjacent pixels. Convolutions like this are foundational in classical computer vision tasks.

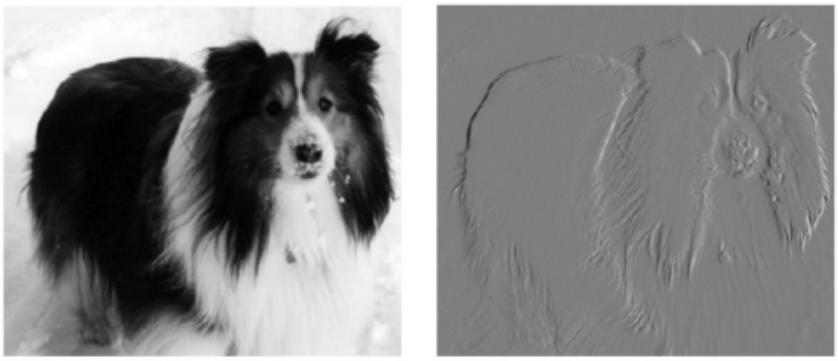


Figure 7.14: 2D Convolution example in edge detections.

Remark (Efficient Implementations)**.** *Modern deep learning frameworks optimize convolutions without explicitly instantiating transformation matrices. Techniques such as strided convolutions and padding further enhance computational efficiency.*

7.6.4 Building Blocks of Convolutional Networks

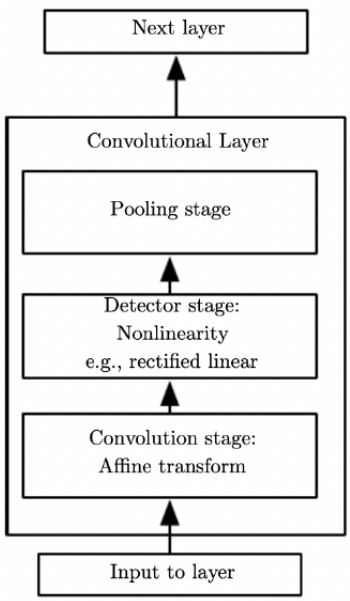


Figure 7.15: Convolutional Block Structure.

Definition 7.6.6 (Convolutional Block)**.** A modular unit in convolutional networks, consisting of:

- Convolutional layers.
- Non-linear activation functions.
- Optional pooling layers (e.g., max pooling or average pooling) to summarize spatial regions.

  In deep learning models, we typically de

  ne a series of filters, interspersed with nonlinearities, and sometimes also with pooling layers, replace a spatial region of the output with a summary statistic. Most common is "max pooling", where we take the maximum. "Average pooling" is also common. We often refer to this entire structure as a "convolutional block".

Definition 7.6.7 (Strided Convolutions)**.** A **strided convolution** is a convolutional operation where the filter (or kernel) is applied to the input with a step size greater than one, known as the *stride*.

Stride (s): The stride determines the number of steps the filter moves across the input. A stride of $s = 1$ moves the filter one step at a time, while $s > 1$ skips steps, reducing the spatial dimensions of the output.

Mathematical Representation: Given an input matrix $\mathbf{X}$ of size $H \times W$ and a filter $\mathbf{K}$ of size $k \times k$, applied with stride s, the output dimensions are:

$$H_{\text{out}} = \left\lfloor \frac{H-k}{s} + 1 \right\rfloor, \quad W_{\text{out}} = \left\lfloor \frac{W-k}{s} + 1 \right\rfloor.$$

The output values are computed as:

$$Y[i,j] = \sum_{m=0}^{k-1} \sum_{n=0}^{k-1} \mathbf{K}[m,n] \cdot \mathbf{X}[i \cdot s + m, j \cdot s + n].$$

Key Note: Strided convolutions act as an alternative to pooling, reducing the size of the output while preserving critical spatial information.

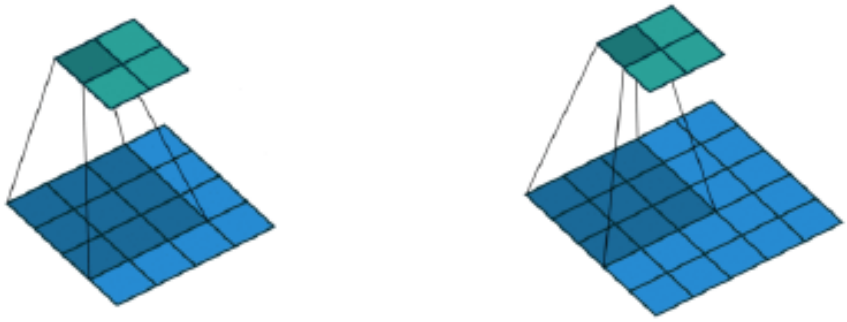


Figure 7.16: Strided Convolutions.

Definition 7.6.8 (Batch Normalization (BatchNorm))**.** A layer that standardizes input features to have unit mean and variance within a mini-batch:

$$\mu_d = \frac{1}{M}\sum_{i=1}^{M} x_d, \quad \sigma_d^2 = \frac{1}{M}\sum_{i=1}^{M}(x_d - \mu_d)^2,$$

$$\hat{x}_d = \frac{x_d - \mu_d}{\sqrt{\sigma_d^2 + \epsilon}}, \quad y_d = \gamma_d \hat{x}_d + \beta_d,$$

where γ_d and β_d are trainable parameters, and $\epsilon > 0$ ensures numerical stability.

Remark (Advantages of BatchNorm)**.** *Empirically, BatchNorm improves optimization and generalization in deep networks, though its theoretical benefits remain partially understood.*

7.6.5 Advanced Architectures

Definition 7.6.9 (VGGNet (Visual Geometry Group Network))**.** VGGNet is a deep convolutional neural network designed for large-scale image recognition tasks. Mathematically, VGGNet can be expressed as a sequential composition of convolutional layers, where each layer applies a convolution operation $F(W, X)$, followed by a non-linearity such as ReLU, to produce an output:

$$h^{(l)} = \text{ReLU}(W^{(l)} * h^{(l-1)} + b^{(l)}),$$

where $W^{(l)}$ are the weights, $b^{(l)}$ are the biases, $*$ denotes the convolution operation, and $h^{(l)}$ is the output at layer l. The architecture is characterized by:

- Small 3×3 convolutional filters, which capture local spatial patterns while reducing the number of parameters compared to larger filters.

- A depth of 16 to 19 layers, with blocks of convolutional layers followed by max-pooling layers for down-sampling.
- Fully connected layers represented as:

$$h_{\text{fc}} = \sigma(W_{\text{fc}} h^{(L)} + b_{\text{fc}})$$

where $h^{(L)}$ is the flattened feature map from the last convolutional layer, σ is the softmax function, and h_{fc} gives the class probabilities.

While VGGNet achieves high accuracy, it suffers from high computational cost and vanishing gradients in deeper layers.

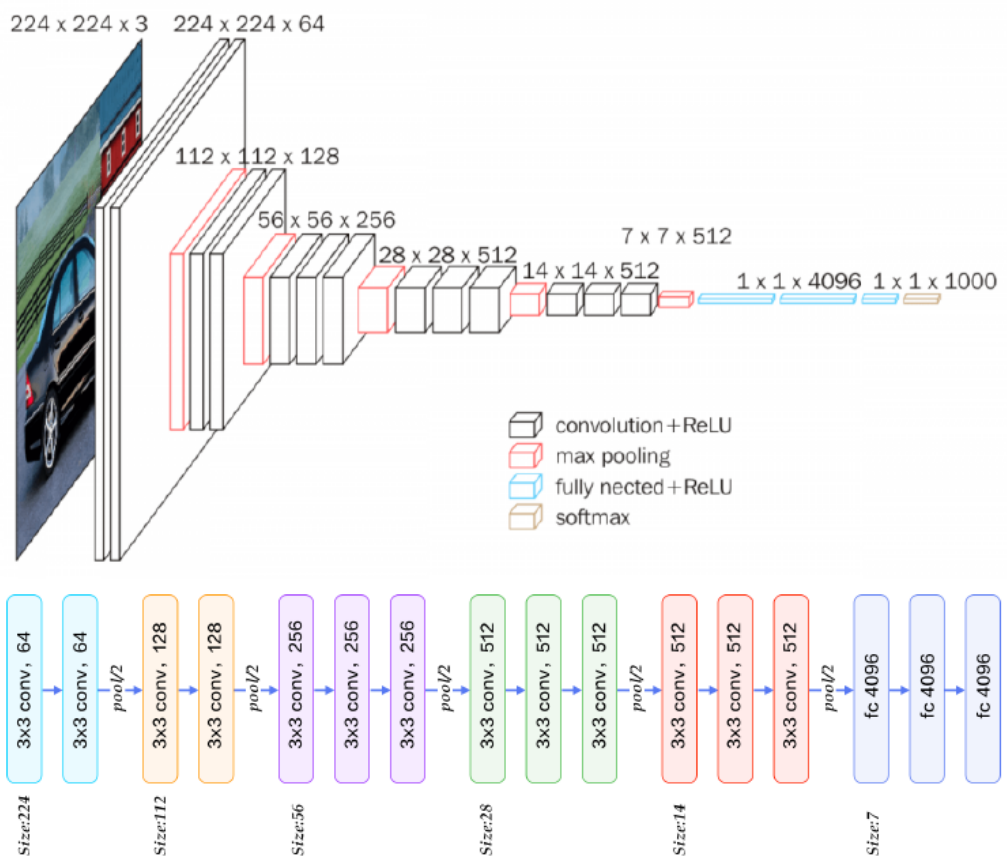


Figure 7.17: Visual of VGGNet (Simonyan and Zisserman, 2014).

Definition 7.6.10 (Connection Between VGGNet and ResNet)**.** ResNet (Residual Network) builds on VGGNet by introducing *residual connections*, which address the vanishing gradient problem. In ResNet, instead of learning a direct mapping $h^{(l)} = F(h^{(l-1)})$, the network learns a residual mapping:

$$h^{(l)} = F(h^{(l-1)}) + h^{(l-1)},$$

where h is the hidden layer and $F(h^{(l-1)})$ represents the transformation performed by the layer (e.g., convolution and ReLU). The addition of the identity shortcut $h^{(l-1)}$ simplifies the optimization problem by ensuring gradients can flow directly through the network during backpropagation.

While VGGNet stacks layers sequentially, ResNet modifies the composition rule to facilitate the training of much deeper networks, reaching hundreds or thousands of layers. This is mathematically equivalent to solving an easier optimization problem where the network learns a residual function $F(x) = h(x) - x$ instead of directly learning $h(x)$.

ResNets achieve superior depth (e.g., 151 layers) with fewer parameters compared to architectures like VGGNet (19 layers).

Definition 7.6.11 (Residual Networks (ResNets))**.** ResNets introduce skip connections:

$$h_1 = \text{NN}(h_2) + Ah_2,$$

where A may be the identity matrix. This design facilitates gradient flow, simplifies learning, and enhances performance in very deep networks.

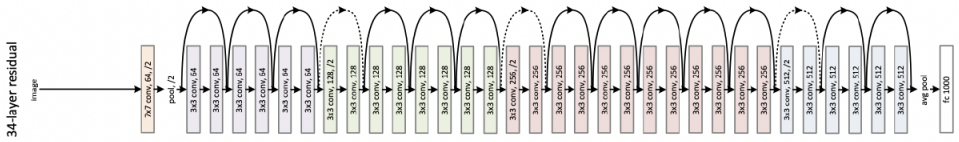
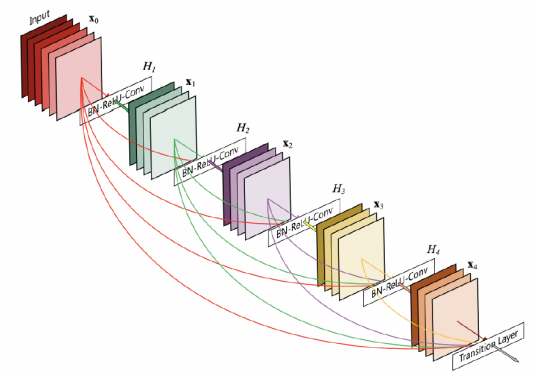


Figure 7.18: Visual of ResNet (He et al., 2015).



Densely-connected convolutional networks ("DenseNet")

Figure 7.19:

Definition 7.6.12 (Dense Connections (DenseNet))**.** DenseNet extends ResNet by densely connecting each layer to every subsequent layer, encouraging feature reuse and efficiency.

7.6.6 Applications in Vision Tasks

- **Image Segmentation:** Architectures like U-Net are tailored for pixel-level classification in tasks like medical imaging.

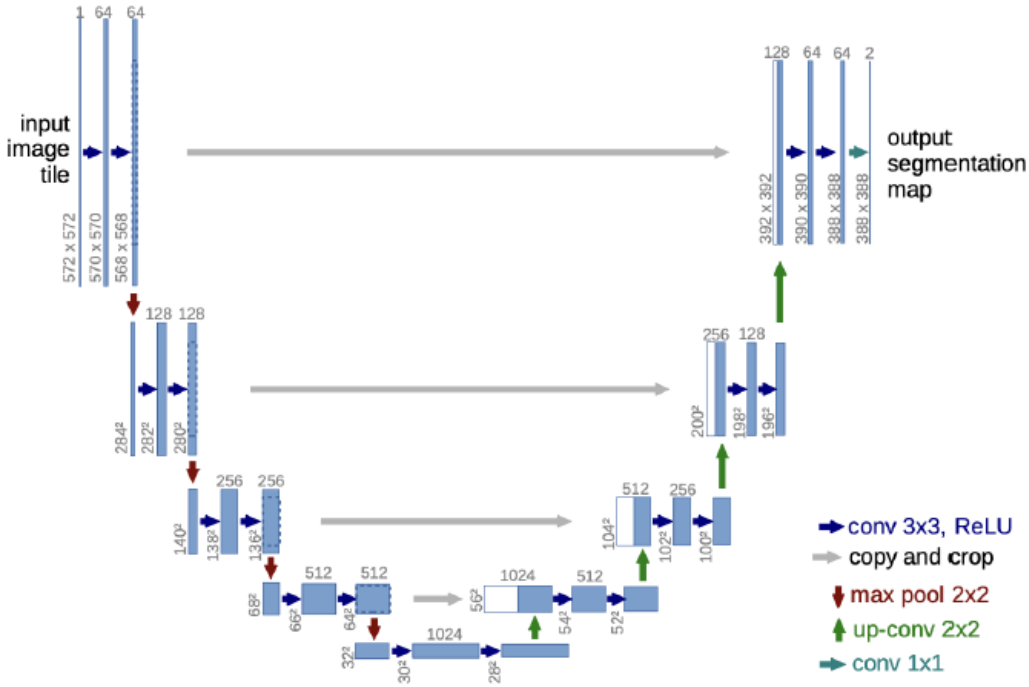


Figure 7.20: U-Net architecture

- **Object Detection:** Layers progressively detect edges, contours, and higher-level object components.

7.6.7 Key Takeaway of Convolutional Networks

- Convolutions capture spatial invariances and dependencies efficiently.

- Modular components (e.g., convolutional blocks, BatchNorm) enhance flexibility and performance.

- Advances like skip connections (ResNet) and dense connections (DenseNet) address challenges in optimization and depth.

7.7 Uncertainty Quantification

7.7.1 Predictive Uncertainty in Deep Learning

Definition 7.7.1 (Probabilistic DL)**.** In deep learning, we aim to model predictive uncertainty by defining:

- A network architecture $f_\theta(x_i)$.
- A likelihood function $p(y_i|x_i,\theta)$ (e.g., $p(y|x,\theta) = \text{Categorical}(y|f_\theta(x))$).

The parameters θ are learned by maximizing the likelihood:

$$\theta^* = \arg\max_\theta p(D|\theta),$$

using stochastic gradient descent, where $D = \{(x_i, y_i)\}_{i=1}^N$ is the training dataset.

Remark (Probabilities in Classification)**.** *For a 100-class problem, the model outputs $f_\theta(x) = \hat{y} \in [0,1]^{100}$ with $\sum_{k=1}^{100} \hat{y}_k = 1$. Here, $\hat{y}_k = \Pr(y = k)$ represents the probability of assigning label k to input x.*

7.7.2 Uncertainty Calibration

Definition 7.7.2 (Calibration of a Classifier)**.** A classifier is well-calibrated if:

$$\Pr(\hat{y} = y \,|\, \hat{p} = p) = p, \quad \forall p \in [0,1],$$

where $\hat{p}$ is the predicted probability of particular output and $\hat{y}$ is the single class with highest probability. This probability means: for any choice of p, the probability of $\hat{y}$ being the true label given our estimate is p, should be p.

Remark. *For example, consider all test points i for which the classifier had $\hat{p}_i \in [0.3, 0.4]$. We would expect between 30% and 40% of those points to have been classified correctly.*

Theorem 7.7.3 (Estimating Calibration Error)**.** *To quantify calibration, partition $[0,1]$ into M consecutive intervals $I_m = \left(\frac{m-1}{M}, \frac{m}{M}\right]$. For each bin B_m:*

$$\textit{Accuracy: acc}(B_m) = \frac{1}{|B_m|} \sum_{i \in B_m} \mathbf{1}[\hat{y}_i = y_i], \quad \textit{Confidence: conf}(B_m) = \frac{1}{|B_m|} \sum_{i \in B_m} \hat{p}_i.$$

A well calibrated model would have $acc(B_m) = conf(B_m)$ for each m.

Since some bins may have more examples than other, a useful statistic is the Expected Calibration Error (ECE):

$$\textit{ECE} = \sum_{m=1}^{M} \frac{|B_m|}{N} \left|\textit{acc}(B_m) - \textit{conf}(B_m)\right|.$$

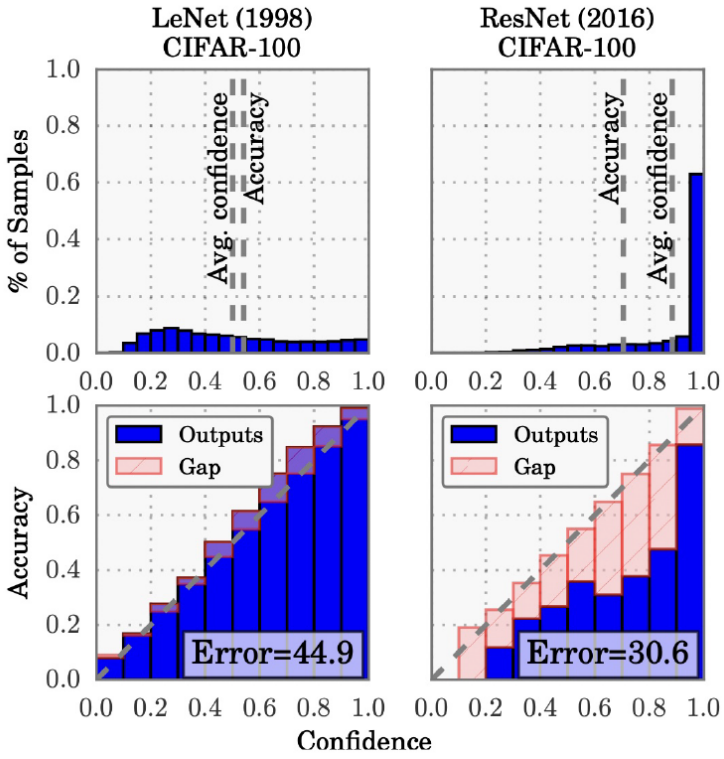


Figure 7.21: Deep network uncertainty calibration.

Remark (Calibration Challenges in Modern Deep Networks)**.** *Modern deep networks, such as ResNet (110-layer) and LeNet (5-layer CNNs), often suffer from poor calibration:*

- *While deeper networks like ResNet achieve higher accuracy, they tend to be more overconfident compared to shallower networks like LeNet.*
- *This observation emphasizes the need for post-hoc calibration methods to correct overconfidence.*

Definition 7.7.4 (Temperature Scaling)**.** Temperature scaling is a simple post-hoc method for improving calibration. Given the network logits z, the scaled probabilities are computed as:

$$\text{softmax}(\tau z),$$

where $\tau > 0$ is a scalar temperature parameter.

- τ is tuned on a validation set to minimize calibration error.
- This method adjusts the predicted probabilities and uncertainties without altering the predicted class labels.

- Works in regression problems too (taking continuous likelihood and check the predictions within different standard deviations).

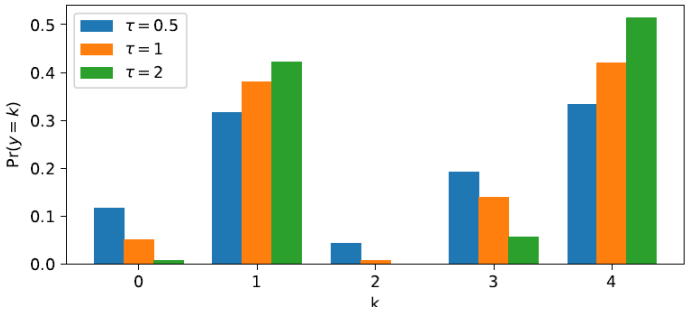


Figure 7.22: Temperature Scaling for Network Outputs.

Example 7.7.5 (Adversarial Perturbations and Calibration)**.** Consider a sentiment analysis task:

- Original input: ”This movie is terrible.” Prediction: **negative sentiment**.
- Modified input: ”This movie is not terrible.” Prediction: **positive sentiment**.

A single, minor perturbation in the input can cause the model to change its prediction drastically, indicating that the model's confidence in its predictions may not always reflect true uncertainty.

There is really but one thing to say about **this** sorry movie It should never have been made The first one one of my favourites An American Werewolf in London is a great movie with a good plot good actors and good FX But this one It stinks to heaven with a cry of helplessness	There is really but one thing to say about **that** sorry movie It should never have been made The first one one of my favourites An American Werewolf in London is a great movie with a good plot good actors and good FX But this one It stinks to heaven with a cry of helplessness

Figure 7.23: Example of the Adversarial Perturbations in Sentiment Analysis.

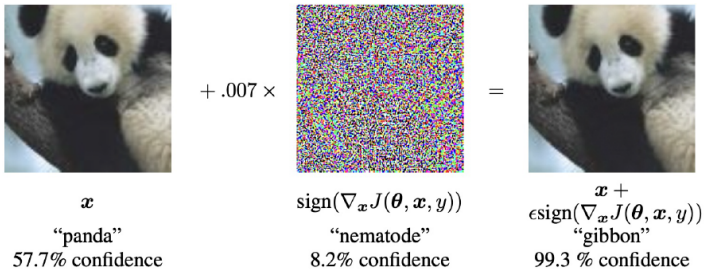


Figure 7.24: Example of the Adversarial Perturbations in Pandas Image

7.7.3 Uncertainty in Bayesian Deep Learning

Definition 7.7.6 (Bayesian Inference)**.** Now define a prior $p(\theta)$. Optimizing parameters θ would find:

$$\theta^* = \arg\min_{\theta} \sum_{i=1}^{N} \log p(y_i|x_i, \theta) + \log p(\theta),$$

and make predictions using the distribution $p(y|x, \theta^*)$.

If we use a Bayesian approach and *marginalize* over the parameters θ, to average across uncertainty in the parameters, we will make predictions with:

$$p(y|x, D) = \int p(y|x, \theta)p(\theta|D)d\theta,$$

effectively decomposing our predictive distribution uncertainty into uncertainty over θ and uncertainty over $y|\theta$.

Definition 7.7.7 (Aleatoric and Epistemic Uncertainty)**.** Predictive uncertainty can be decomposed as:

$$p(y|x, D) = \int p(y|x, \theta)p(\theta|D)\, d\theta.$$

- **Aleatoric Uncertainty:** $p(y|x, \theta)$, captures irreducible uncertainty in the measurement process. Can improve it by adding more features.
- **Epistemic Uncertainty:** $p(\theta|D)$, model uncertainty due to limited data or knowledge. It represents our uncertainty over multiple different hypothetical values of θ. More sufficient data means less epistemic uncertainty.
- When having a new input, if high Epistemic uncertainty, means this input is very different from my training input. If high Aleatoric uncertainty is high, means it's very close to the training dataset.

Accurate uncertainty estimate is essential for many downstream tasks such as out-of-distribution detection and active learning.

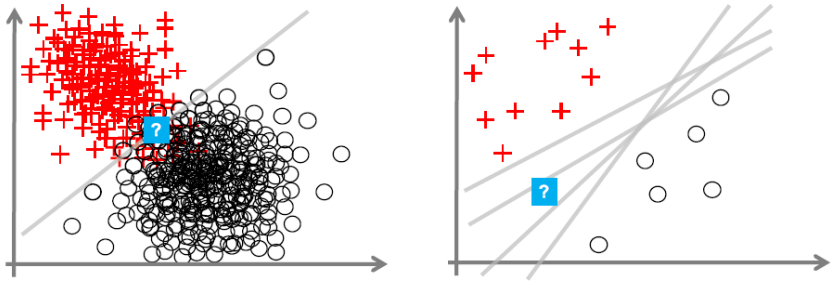


Figure 7.25: Epistemic vs Aleatoric Uncertainty. (Example where more data will improve the Epistemic Uncertainty)

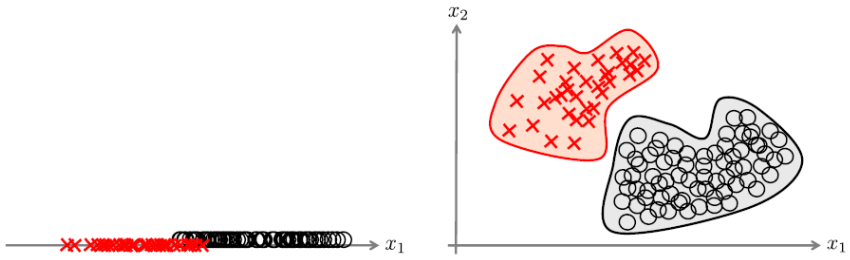


Figure 7.26: Epistemic vs Aleatoric Uncertainty. (Example where more features will improve the Aleatoric Uncertainty)

Example 7.7.8 (Hypothetical Scenarios)**.** In a binary classification problem with test point x, large uncertainty may arise in two ways:

1. (aleatoric): $\Pr(y = 1|x, \theta) \approx 0.5$, indicating irreducible label ambiguity. Here the posterior over θ is concentrated around a particular value θ^*.

2. (epistemic): The posterior $p(\theta|D)$ supports multiple models θ_1, θ_2, with $\Pr(y = 1|x, \theta_1) \approx 1$ and $\Pr(y = 1|x, \theta_2) \approx 0$.

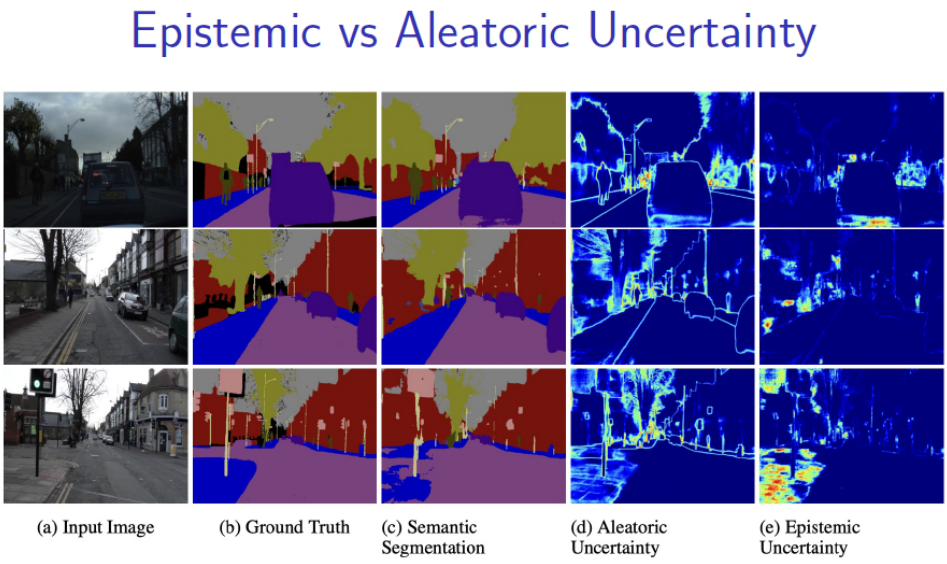


Figure 7.27: Epistemic vs Aleatoric Uncertainty in a Car Detection Example.

Theorem 7.7.9 (Computing Epistemic and Aleatoric Uncertainty)**.** *The predictive variance in a regression setting can be decomposed as:*

$$Var(y) = Var[\mathbb{E}[y|z]] + \mathbb{E}[Var[y|z]],$$

where $z = f_\theta(x)$ represents the network output, providing parameters $\{\mu, \sigma\}$ for the likelihood $\mathcal{N}(y|\mu, \sigma^2 I)$. Specifically:

- $Var[\mathbb{E}[y|z]] = Var[\mu]$*: Represents epistemic uncertainty, which arises from model uncertainty and depends on $p(\theta|D)$.*
- $\mathbb{E}[Var[y|z]] = \mathbb{E}[p(\theta|D)[\sigma^2 I]]$*: Represents aleatoric uncertainty, which captures irreducible noise inherent to the data.*

Law of Total Variance: *The first line is a general result that always holds, while the second line is model-specific and often estimated using Monte Carlo methods.*

7.7.4 Summary

- Predictive uncertainty is crucial for safe real-world deployment of machine learning models.
- Uncertainty can be decomposed into aleatoric (data-dependent) and epistemic (model-dependent) components.

- Calibration techniques like temperature scaling can address overconfidence in deep networks.
- Bayesian deep learning provides a principled framework for quantifying uncertainty but is computationally intensive.

Chapter 8

Deep Learning Inference

8.1 Posterior Approximation Techniques

Definition 8.1.1 (Posterior Approximation)**.** Approximating the posterior distribution $p(\theta|D)$ is essential in Bayesian deep learning. Two primary approaches include:

- **Parametric Approximation:** Assume a parametric form for the posterior, e.g. Gaussian distribution $q(\theta)$ over weights. Variational Bayes, Laplace approximations.
- **Finite Sample Approximation:** Represent the posterior using a collection of candidate weights: $\{\theta_1, \ldots, \theta_K\}$.Markov Chain Monte Carlo (MCMC), Stein Variational Gradient Descent, model ensembles.

8.2 Laplace Approximation in DL Inference

8.2.1 Definition: Laplace Approximation

The Laplace approximation provides a Gaussian approximation to the posterior distribution $p(\theta|\mathcal{D})$, centered at its mode θ^*(local approx, unimodal). Using a second-order Taylor expansion around θ^*, the log-posterior can be approximated as:

$$\log p(\theta|D) \approx \log p(\theta^*|D) + \underbrace{\text{const}}_{\text{constant}} + \frac{1}{2}(\theta - \theta^*)^\top H(\theta - \theta^*),$$

where:

- θ^*: The mode (maximum a posteriori estimate) of the posterior distribution.
- H: The Hessian matrix, $H = \nabla^2 \log p(\theta|D)$, representing the second derivative of the log-posterior.

- D: The observed data.

This suggests a Gaussian approximation with

$$\log p(\theta|D) \approx \log \mathcal{N}(\theta|\theta^*, \Lambda^{-1}) = -\frac{1}{2}(\theta - \theta^*)^\top \Lambda(\theta - \theta^*) + \text{const},$$

where:

- $\Lambda = -H = -\nabla^2 \log p(\theta|D)$: The precision matrix (inverse of the covariance matrix), equal to $-H$.
- $\mathcal{N}(\theta|\theta^*, \Lambda^{-1})$: A multivariate Gaussian distribution with mean θ^* and covariance Λ^{-1}.

Two apparent problems:

- **Neural network posteriors are not even remotely Gaussian.**
- **The Hessian for large networks will be impossible to compute or store.**

8.2.2 Model Setup for Regression Problems

For regression problems in deep learning, the model can be specified as:

$$p(y_i|x_i, \theta, \beta) = \mathcal{N}(y_i|f_\theta(x_i), \beta^{-1}), \quad p(\theta|\alpha) = \mathcal{N}(\theta|0, \alpha^{-1}I),$$

where:

- $f_\theta(x)$ is the neural network model,
- $\alpha > 0$ and $\beta > 0$ are precisions (inverse variances).

The posterior distribution can then be expressed as:

$$p(\theta, \alpha, \beta|\mathcal{D}) \propto p(\theta|\alpha) \prod_{i=1}^{N} p(y_i|x_i, \theta, \beta).$$

8.2.3 Step 1: Finding the Posterior Mode

The first step in applying the Laplace approximation is to find the posterior mode θ^*. This is achieved by maximizing the log-posterior:

$$\log p(\theta|\mathcal{D}) = \log p(\theta|\alpha) + \sum_{i=1}^{N} \log p(y_i|x_i, \theta, \beta),$$

which simplifies to:

$$\log p(\theta|\mathcal{D}) = -\frac{\alpha}{2}\theta^\top\theta - \frac{\beta}{2}\sum_{i=1}^{N}(f_\theta(x_i) - y_i)^2 + \text{const.}$$

The mode θ^* is found by solving the Maximum A Posteriori (MAP) estimate.

8.2.4 Step 2: Computing the Hessian

The precision matrix for the Gaussian approximation is defined as:

$$\Lambda = -\nabla^2 \log p(\theta|\mathcal{D}) = \alpha I + \beta H,$$

where H is the Hessian of the sum of squared errors:

$$H = \nabla\nabla^\top \sum_{i=1}^{N}(f_\theta(x_i) - y_i)^2.$$

Expanding H, we obtain:

$$H = \sum_{i=1}^{N} \nabla f_\theta(x_i)\nabla f_\theta(x_i)^\top + \sum_{i=1}^{N}(f_\theta(x_i) - y_i)\nabla\nabla^\top f_\theta(x_i).$$

If computing the full Hessian is infeasible, a common approximation is:

$$H \approx \sum_{i=1}^{N} \nabla f_\theta(x_i)\nabla f_\theta(x_i)^\top.$$

The intuition here is that the residual $f_\theta(x_i) - y_i$ in the second term should be small at the mode θ^* and hopefully in expectation zero.

8.2.5 Step 3: Making Predictions by Linearization

Once the precision matrix Λ is computed, we approximate the posterior distribution as:

$$p(\theta|\mathcal{D}) \approx q(\theta|\mathcal{D}) \equiv \mathcal{N}(\theta^*, \Lambda^{-1}).$$

For predictions, the posterior predictive distribution is:

$$p(y|x, \mathcal{D}) = \int p(y|x, \theta)q(\theta|\mathcal{D})d\theta.$$

Using a first-order Taylor expansion of $f_\theta(x)$, we approximate:

$$f_\theta(x) \approx f_{\theta^*}(x) + g(x)^\top(\theta - \theta^*),$$

where $g(x) = \nabla f_\theta(x)\big|_{\theta=\theta^*}$. Substituting this back, the predictive distribution becomes:

$$p(y|x, \mathcal{D}) \approx \mathcal{N}(y|f_{\theta^*}(x) + g(x)^\top(\theta - \theta^*), \beta^{-1}).$$

which is our linearized approximate posterior.

8.2.6 Classification with Laplace Approximation

For classification problems, the posterior predictive distribution is given by:

$$p(y = 1|x, \mathcal{D}) \approx \int \sigma\big(f_\theta(x) + g(x)^\top(\theta - \theta^*)\big)\mathcal{N}(\theta^*, \Lambda^{-1})d\theta,$$

where $\sigma(\cdot)$ is the sigmoid function, $f_\theta(x)$ is the neural network output, and $g(x) = \nabla f_\theta(x)\big|_{\theta=\theta^*}$.

Remark (Hessian Approximation). • *We can use a similar outer-product approximation of the Hessian; the only difference is that the likelihood changes*

- *The details for a logistic output are in Bishop PRML (and look a lot like the Hessian in your coursework), but it's not necessary to know this. You can actually just use autodiff on the likelihood function, for any likelihood.*
- *For predictions, we can combine the "linearize the predictor" trick from this lecture with the "avoid high-dimensional Monte Carlo sampling" trick from the linear model example*

8.2.7 Locally-Linear Classification

Combining "locally-linear" and "low-dimensional integration" tricks, the posterior predictive distribution can be further approximated as:

$$\begin{aligned} p(y = 1|x, \mathcal{D}) &\approx \int \sigma\big(f_\theta(x)\big)\mathcal{N}(\theta^*, \Lambda^{-1})d\theta, \\ &\approx \int \sigma\big(f_\theta(x) + g(x)^\top(\theta - \theta^*)\big)\mathcal{N}(\theta^*, \Lambda^{-1})d\theta, \\ &= \int \sigma\big(a\big)\mathcal{N}(a|f_{\theta^*}(x), g(x)^\top\Lambda^{-1}g(x))da. \end{aligned}$$

The key here is to introduce the a. This integral can be evaluated using Monte Carlo sampling or numerical approximations.

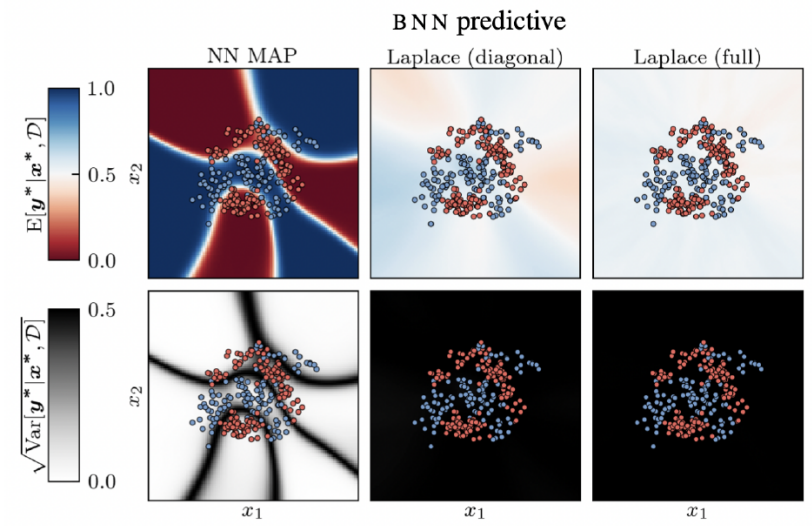


Figure 8.1: Use locally-linear classifier to make predictions - Underfitting Issue

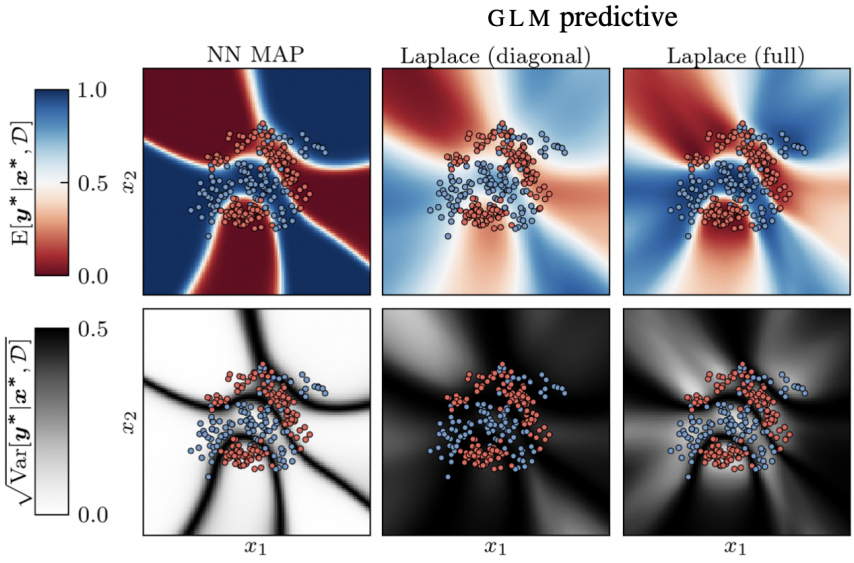


Figure 8.2: Use locally-linear classifier to make predictions - Underfitting Issue

Remark (Linearization for Predictions)**.** *Linearization of the model at the mode θ^* is often necessary to avoid severe underfitting in predictions. The first-order Taylor expansion of the network output*

enables tractable computation of the posterior predictive distribution.

8.2.8 Summary of Laplace Approximations

- The Laplace approximation provides a Gaussian approximation to the posterior and posterior predictive distributions.
- Computing the Hessian is computationally expensive, but several approximations are available:
  - Compute the Hessian only for certain parameters (e.g., last layer weights or subset of weights).
  - Use block-diagonal approximations (e.g., Kronecker factorizations).
- **Good:** The approximation provides a tractable marginal likelihood, enabling optimization of hyperparameters (e.g., α, β) and model comparison.
- **Bad:** It only captures uncertainty around a single mode of the posterior.

8.3 Markov Chain Monte Carlo in DL Inference

8.3.1 MCMC Recap

- **Target Distribution:** Assume we have a target density $\pi(\theta) \propto p(\theta, \mathcal{D})$ that is hard to sample from directly.
- **Transition Operator:** Starting with an initial state $\theta^{(0)}$, we define a Markov transition operator $A(\theta'|\theta^{(t)})$ to generate a sequence of correlated samples $\{\theta^{(t)}\}$, such that $\theta^{(t)} \sim \pi(\theta)$ after sufficient steps.

The Metropolis-Hastings (MH) algorithm is a widely used MCMC method:

1. Given the current state $\theta^{(t)}$, propose a new state θ' using a proposal distribution $q(\theta'|\theta^{(t)})$.
2. Compute the acceptance ratio:

$$\alpha(\theta', \theta^{(t)}) = \min\left(1, \frac{\pi(\theta')q(\theta^{(t)}|\theta')}{\pi(\theta^{(t)})q(\theta'|\theta^{(t)})}\right).$$

   For symmetric proposal such as Gaussian centered at the current location, the acceptance ratio only depends on π.
3. With probability α, accept the proposal ($\theta^{(t+1)} = \theta'$); otherwise, reject ($\theta^{(t+1)} = \theta^{(t)}$).

8.3.2 Langevin Dynamics: Noisy Gradient Descent

Given MCMC's challenge of proposal distribution for high-dimensional θ, we introduce Langevin Monte Carlo.

Definition 8.3.1 (Langevin Dynamics)**.** Langevin Dynamics combines gradient descent with stochastic noise, enabling exploration of parameter space rather than converging to a single estimate.

Maximizing the Log Joint Probability: To find a single estimate of the weights, we maximize the log joint probability:

$$\log p(\theta, D) = \sum_{i=1}^{N} \log p(y_i|x_i, \theta) + \log p(\theta).$$

This is achieved through gradient descent, where:

$$g_t = -\nabla_\theta \log p(\theta^{(t)}, D),$$

and the update rule is:

$$\theta^{(t+1)} = \theta^{(t)} - \eta g_t,$$

with a small learning rate $\eta > 0$.

Definition 8.3.2 (Langevin Monte Carlo (LMC))**.** Langevin Monte Carlo interprets Langevin dynamics as a Metropolis-Hastings proposal:

$$q(\theta'|\theta) = \mathcal{N}(\theta'|\theta - \eta g, \epsilon^2 I).$$

The acceptance probability for θ' is:

$$\alpha(\theta', \theta) = \min\left(1, \frac{\pi(\theta')q(\theta|\theta')}{\pi(\theta)q(\theta'|\theta)}\right).$$

Relation to HMC
Langevin dynamics can be extended to Hamiltonian Monte Carlo (HMC) by:

- Introducing momentum variables to explore posterior landscapes more effectively.
- Performing multiple gradient updates (leapfrog steps) before the accept-reject step.

This allows HMC to achieve superior performance in navigating complex, high-dimensional distributions.

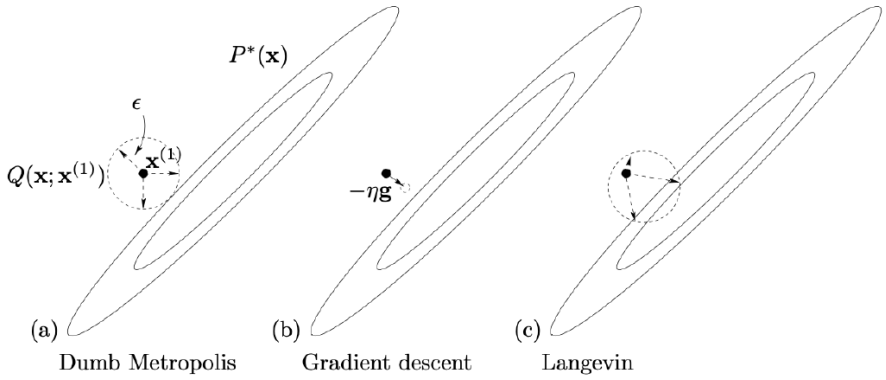


Figure 8.3: Langevin Dynamics Example.

Example and Intuition: In high-dimensional spaces, including gradient information in the proposal distribution improves sampling efficiency compared to simple Gaussian proposals:

- **Dumb Metropolis:** Proposals are randomly generated without gradient information, leading to inefficient exploration.
- **Gradient Descent:** Exploits gradient information for deterministic optimization but lacks stochasticity for exploration.
- **Langevin Dynamics:** Combines gradient information with stochastic noise, allowing efficient exploration of the posterior.

Hamiltonian Monte Carlo extends this further by taking multiple gradient updates before the accept-reject step, improving efficiency.

Remark (Interpretation of Langevin Dynamics)**.** *Langevin dynamics can be interpreted as:*

- ***Noisy Gradient Descent:*** *Gradient information drives the updates toward regions of high posterior probability, while the noise ensures stochastic exploration.*
- ***Physical Analogy:*** *Viewed as a special case of Hamiltonian Monte Carlo (HMC), where random noise introduces momentum-like effects.*

Remark (Efficiency in High Dimensions)**.** *Langevin dynamics is particularly effective for high-dimensional distributions due to:*

- *Exploiting gradient information to align proposals with posterior contours.*
- *Reducing rejection rates by generating more informed proposals compared to standard random-walk methods.*

8.3.3 Stochastic Gradient Langevin Dynamics (SGLD)

Definition 8.3.3 (Stochastic Gradient Langevin Dynamics)**.** Stochastic Gradient Langevin Dynamics (SGLD) modifies Langevin dynamics to handle large datasets by using **mini-batches** for gradient estimation. Instead of computing the full log-posterior, it approximates it using a subset of the data:

$$\log \pi_M(\theta) \approx \frac{N}{M} \sum_{i=1}^{M} \log p(y_i|x_i, \theta) + \log p(\theta),$$

where:

- M is the mini-batch size,
- N is the total dataset size,
- $\pi_M(\theta)$ is the mini-batch approximation of the posterior $\pi(\theta)$.

In optimization, the parameters are updated via stochastic gradient steps:

$$\theta^{(t+1)} = \theta^{(t)} + \epsilon_t \hat{g}_t,$$

where $\hat{g}_t = \nabla_\theta \log \pi_M(\theta)$ is the stochastic gradient.

When used as an MCMC proposal, the parameters are sampled from the proposal distribution:

$$q(\theta^{(t+1)}|\theta^{(t)}) = \mathcal{N}\big(\theta^{(t+1)}|\theta^{(t)} + \eta_t \hat{g}_t, \epsilon_t^2 I\big),$$

where:

- $\eta_t = \frac{\epsilon_t^2}{2}$ controls the magnitude of the gradient step,
- $\epsilon_t^2 I$ adds isotropic Gaussian noise to ensure stochasticity.

This framework allows sampling from the posterior without requiring access to the entire dataset at each iteration, making it computationally efficient.

Remark (Skipping the Acceptance Step)**.** *Unlike standard Langevin dynamics, SGLD often skips the Metropolis-Hastings acceptance step to avoid evaluating the entire dataset (i.e. always accept). This is justified by:*

- *Small step sizes ϵ_t ensuring that proposals approximate the target distribution well.*
- *A reduction in computational cost, which is critical for large datasets.*

Theorem 8.3.4 (Algorithm for SGLD)**.** *The SGLD algorithm proceeds as follows:*

1. ***Sample Noise:*** *Draw $\xi_t \sim \mathcal{N}(0, I)$.*

2. ***Compute Update:*** *Update parameters using:*

$$\theta^{(t+1)} = \theta^{(t)} + \frac{\epsilon_t^2}{2}\nabla_\theta \log \pi_M(\theta) + \epsilon_t \xi_t.$$

3. *Optionally, reduce the sequence of step sizes (learning rates) ϵ_t over time for guaranteed convergence.*

For SGLD to converge to the true posterior, the step sizes ϵ_t must decrease over time according to:

$$\sum_{t=1}^{\infty} \epsilon_t^2 < \infty \quad \textit{and} \quad \sum_{t=1}^{\infty} \epsilon_t = \infty.$$

However, this is rarely implemented in practice due to computational constraints.

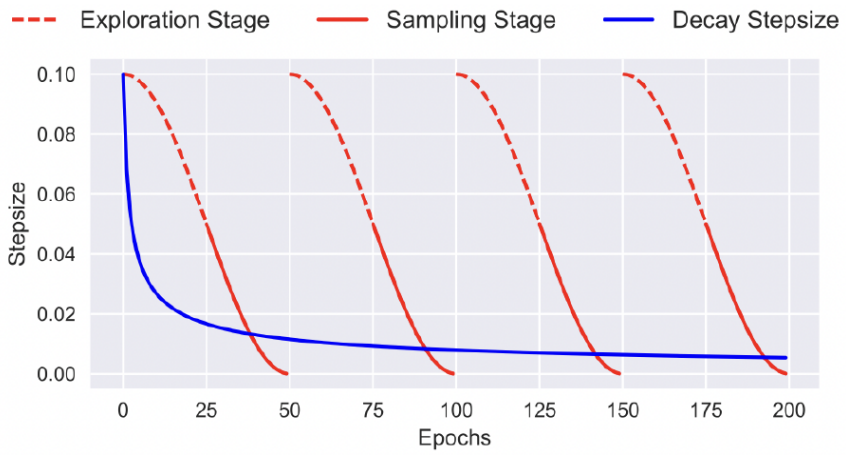


Figure 8.4: Cyclic Learning Rates in SGLD.

Theorem 8.3.5 (Mixing Across Modes in MCMC)**.** *SGLD sometimes struggles with multi modal posteriors for multiple models because MCMC won't move between disconnected regions. A solution is to use cyclic learning rates, as illustrated:*

- ***SGLD Without Cyclic Rates:*** *Tends to stay stuck in a single mode.*
- ***Cyclic SGLD (cSGLD):*** *Encourages exploration of multiple modes by periodically increasing and decreasing the step size (red curve).*

When using Cyclic SGLD, we need to be careful that the cyclic stage is long enough to find the real samples.

(a) Target (b) SGLD (c) cSGLD

Figure 8.5: Mixing across modes is easier with a cyclic learning rate schedule.

Practical Improvements on SGLD

- **Preconditioning Matrix:** Using a preconditioning matrix can adaptively scale the gradient, improving convergence.
- **Cyclic Learning Rates:** Mixing across posterior modes can be enhanced by periodically varying ϵ_t in a cyclic schedule.
- **SWAG (Stochastic Weight Averaging in Gaussian Space):** Techniques like SWAG approximate posterior uncertainty by treating SGD as a sampler.
- Stochastic Hamiltonian Monte Carlo is a related method but can be sensitive to hyperparameter tuning.

8.4 Variational Inference in DL Inference

8.4.1 Variational Inference

Definition 8.4.1 (Variational Inference and ELBO)**.** Variational inference approximates the posterior $p(\theta|D)$ by directly optimizing a parametric distribution $q_\lambda(\theta)$. This is achieved by minimizing the KL divergence:

$$D_{KL}(q_\lambda(\theta)\|p(\theta|D)) = \int q_\lambda(\theta) \log \frac{q_\lambda(\theta)}{p(\theta|D)} d\theta.$$

Since the true posterior $p(\theta|D)$ is intractable, we maximize the Evidence Lower Bound (ELBO), which is equivalent to minimizing the KL divergence:

$$\text{ELBO}(\lambda, D) = \mathbb{E}_{q_\lambda(\theta)} \left[\log p(\theta, D) - \log q_\lambda(\theta)\right].$$

8.4.2 Evidence Lower Bound (ELBO)

Theorem 8.4.2 (Relationship Between ELBO and Marginal Likelihood)**.** *The ELBO is a lower bound on the log-marginal likelihood* $\log p(D)$*:*

$$\textit{ELBO}(\lambda, D) = \log p(D) - D_{KL}(q_\lambda(\theta) \| p(\theta|D)),$$

where the KL divergence $D_{KL} \geq 0$*. Therefore, maximizing the ELBO minimizes the KL divergence and improves the approximation* $q_\lambda(\theta) \approx p(\theta|D)$*.*

Remark (Gradient Ascent Maximization of ELBO)**.** *Reparameterized gradient estimators enable efficient computation of gradients with respect to variational parameters* $\lambda = \{\mu, \Sigma\}$*. Using reparameterized gradient estimators, we can compute:*

$$\nabla_\lambda \mathbb{E}_{q_\lambda(\theta)} \left[\log \frac{p(\theta, D)}{q_\lambda(\theta)} \right] = \mathbb{E}_{p(\epsilon)} \left[\nabla_\lambda \log \frac{p(r(\lambda, \epsilon), D)}{q_\lambda(r(\lambda, \epsilon))} \right].$$

Key Notes:

- *In practice, draw a single sample* $\epsilon \sim p(\epsilon)$*, and use it to compute* μ *and* σ*.*
- *At a cost of* $\times 2$ *parameters, we also estimate a per-weight standard deviation (note: this is* ***bad*** *if we use diagonal only).*
- *Blundell et al. (2015), Weight uncertainty in neural networks, writes out derivatives with respect to* μ, σ *explicitly. But with modern tools, we can "solve" the VB inference problem using reparameterized sampling with autodiff.*

8.4.3 Using Mini-Batches for ELBO Optimization

Definition 8.4.3 (Mini-Batch Estimation of ELBO)**.** For models where the likelihood factorizes over data points, the ELBO can be approximated using mini-batches:

$$\text{ELBO}(\lambda, D) \approx \frac{N}{M} \sum_{j=1}^{M} \mathbb{E}_{q_\lambda(\theta)} \left[\log p(y_j | x_j, \theta) \right] - D_{KL}(q_\lambda(\theta) \| p(\theta)),$$

where M is the mini-batch size. Careful to include the N in the ratio.

Remark (Reducing Monte Carlo Integration)**.** *If the priors and posteriors on the weights are Gaussian, Monte Carlo integration for linear layers can be made more efficient using the* ***local reparameterization trick*** *(Kingma et al., 2015):*

- *Suppose the entries of* $A \in \mathbb{R}^{H \times D}$ *and* $b \in \mathbb{R}^H$ *are Gaussian distributed.*

- *For a given input x, the output $h = Ax + b \in \mathbb{R}^H$ is also Gaussian distributed, but with a much lower dimensionality.*
- *Sampling from h directly only requires drawing H random values, as opposed to $(H + 1)D$, which reduces variance in the estimated loss and gradients.*

8.4.4 Practical Implementation of Variational Inference

- **Reparameterization Trick:** Variational distributions like $q_\lambda(\theta) = \mathcal{N}(\mu, \Sigma)$ can be reparameterized as $\theta = \mu + \Sigma\epsilon$, where $\epsilon \sim \mathcal{N}(0, I)$. This simplifies gradient estimation.
- **Per-Weight Variances:** At a cost of $2\times$ the parameters, one can estimate a diagonal covariance matrix Σ, although this limits expressiveness.
- **Example:** Blundell et al. (2015) proposed using this approach for neural networks, showing how derivatives with respect to μ and Σ can be computed using automatic differentiation.
- **Challenges**: Limited by the choice of variational family (e.g., mean-field or diagonal Gaussian assumptions). Optimization is sensitive to initialization (by first finding a MAP estimate for μ) and hyperparameter choices.

8.5 Ensembles and other Approaches for Uncertainty Estimate

When seeking a quick estimate of uncertainty from a deep learning model, practitioners often use alternative methods that do not involve complex Bayesian approaches. Here are three common strategies:

- **Ensemble Models:** Train multiple models independently with different initializations and average their predictions. This approach has been studied in works such as Lakshminarayanan et al. (2017).
- **Monte Carlo Dropout:** Introduced by Gal and Ghahramani (2016), this method uses dropout at both training and test time, interpreting the stochastic behavior as a Bayesian approximation.
- **Deterministic Networks with Bayesian Layers:** Combine a deterministic network with a Bayesian linear model as the final layer. This can provide uncertainty estimates:
  - For regression tasks, this method can be exact.
  - For classification tasks, methods such as Variational Bayes (VB) or Laplace approximations can be applied.

Ideally, the Bayesian layer is trained jointly with the rest of the network, rather than pre-training the network and freezing its features.

In addition, uncertainty estimates can also be derived from unconventional sources, such as using data augmentation techniques at test time.

Summary of DL Inference Methods

- Local Gaussian approximations can be useful for deep learning models, despite multimodality in the posterior
- Stochastic gradient MCMC methods are more able to mix across modes and can scale, but they are sensitive to parameter tuning issues
- Challenges for Laplace approximations are largely computational
- Challenges for VB lie in optimization, as well as opening up conceptual questions (what sort of posterior approximation to use, etc).

Chapter 9

Deep Generative Models

9.1 Variational Autoencoders (VAEs)

9.1.1 Generative Models: Overview

Definition 9.1.1 (Generative vs Discriminative Models)**.**

- **Discriminative models:** Learn $p_\theta(y|x)$, focusing on the conditional distribution of labels y given data x. Typically do this by estimating a parameter θ to minimize the prediction error on training set.
- **Generative models:** Learn an approximation to $p_\theta(x)$ or $p_\theta(x, y)$, aiming to model the data distribution itself.

9.1.2 Linear Latent Variable Models and PCA

A Tale of Two Binary Linear Classifiers

- **Logistic Regression:** Learns the conditional probability $p(y|x)$.
- **Gaussian Naïve Bayes:** Models the joint distribution $p(y, x)$ by learning $p(y)$, $p(x|y = 1)$, and $p(x|y = 0)$.
- **Observation:** The choice of model depends on the data regime:
  - For **small data**, Gaussian Naïve Bayes can outperform logistic regression by leveraging prior assumptions.
  - For **large data**, logistic regression benefits from its ability to directly optimize $p(y|x)$ without additional assumptions.

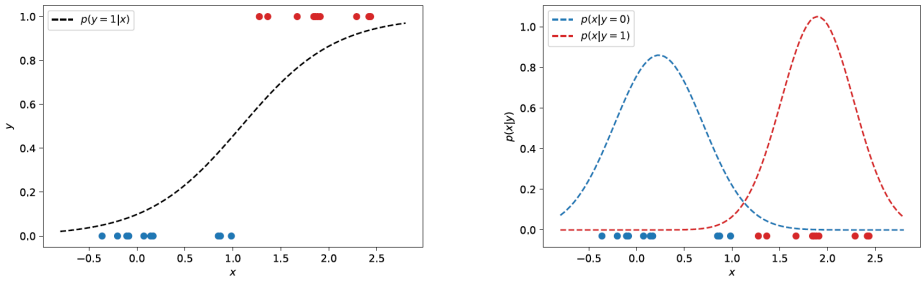


Figure 9.1: Two Binary Linear Classifiers.

Learning Linear Latent Variable Models

Definition 9.1.2 (Linear Latent Variable Models)**.** Linear latent variable models aim to find a low-dimensional representation of the data by factorizing a design matrix $X \in \mathbb{R}^{N \times D}$ into:

$$X \approx ZW^\top,$$

where:

- $Z \in \mathbb{R}^{N \times K}$ is a matrix of latent variables.
- $W \in \mathbb{R}^{D \times K}$ is a matrix of weights.

Examples include:

- Principal Component Analysis (PCA).
- Independent Component Analysis (ICA).
- Factor Analysis.
- Non-Negative Matrix Factorization (NMF).

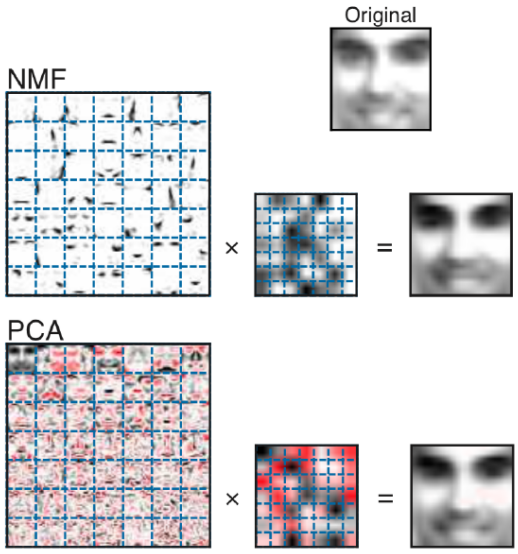


Figure 9.2: Linear Latent Variable Models.

PCA as Reconstruction Error Minimization

Theorem 9.1.3 (PCA Objective)**.** *Suppose design matrix $X \in \mathbb{R}^{N\times D}$ is a centered data matrix ($\frac{1}{N}\sum_{i=1}^{N} x_i = 0$). PCA approximates X as:*

$$\widehat{X} = ZW^\top, \quad Z \in \mathbb{R}^{N\times K},\ W \in \mathbb{R}^{D\times K},\ K \ll D.$$

where K is the latent variable. PCA solves the least squares problem:

$$\arg\min_{W,Z} \|X - ZW^\top\|_F^2, \quad \textit{s.t. } W^\top W = I.$$

For each $x_i \in \mathbb{R}^D$, the low-dimensional representation is $z_i = W^\top x_i$, and the reconstruction is $x_i \approx Wz_i$.

Finding a PCA Solution

1. **Alternate Minimization:** We can find the solution to this minimization problem by **alternate minimization** of:

$$\sum_{i=1}^{N} \|\mathbf{x}_i - \mathbf{W}\mathbf{z}_i\|_2^2 :$$

(a) **Step 1: Fix W and solve for $\mathbf{z}_i$:**

$$\frac{\partial}{\partial \mathbf{z}_i}\|\mathbf{x}_i - \mathbf{W}\mathbf{z}_i\|_2^2 = 2\mathbf{z}_i - 2\mathbf{W}^\top \mathbf{x}_i.$$

Setting the derivative equal to zero and solving gives:

$$\mathbf{z}_i = \mathbf{W}^\top \mathbf{x}_i.$$

(b) **Step 2: Fix Z and solve for W:** Minimizing $\|\mathbf{X} - \mathbf{Z}\mathbf{W}^\top\|_2^2$ is equivalent to the linear regression least squares problem, with:

$$\mathbf{W} = \mathbf{X}\mathbf{Z}^\top(\mathbf{Z}\mathbf{Z}^\top)^{-1}.$$

In practice, it is rare to use this algorithm directly. Instead, principal components can be computed efficiently using the **singular value decomposition (SVD)** of the matrix $\mathbf{X}$.

2. **Efficient Approach:** In practice, PCA is computed using Singular Value Decomposition (SVD):

$$X = U\Sigma V^\top, \quad W = V_{[:,1:K]},\ Z = XW.$$

9.1.3 Probabilistic PCA

Theorem 9.1.4 (Probabilistic PCA)**.** *Probabilistic PCA reformulates PCA as a probabilistic generative model:*

$$p(z_i) = \mathcal{N}(z_i|0, I), \quad p(x_i|z_i) = \mathcal{N}(x_i|Wz_i + \mu, \sigma^2 I),$$

where:

- $z_i \in \mathbb{R}^K$ *is the latent variable.*
- $x_i \in \mathbb{R}^D$ *is the observed data, with* $K < D$*.*
- $W \in \mathbb{R}^{D\times K}$ *maps the latent space to the observed space.*
- $\mu \in \mathbb{R}^D$ *is the mean of the distribution.*
- $\sigma^2 I$ *represents isotropic Gaussian noise.*

The model has a tractable posterior distribution:

$$p(z_i|x_i) = \mathcal{N}(z_i|M^{-1}W^\top(x_i - \mu), \sigma^2 M^{-1}),$$

where $M = W^\top W + \sigma^2 I$*.*

Remark (Marginal Likelihood). *The marginal likelihood of x_i is:*

$$p(x_i) = \int p(x_i|z_i)p(z_i)dz_i,$$

which is tractable and results in a Gaussian distribution. This enables efficient computation of likelihoods.

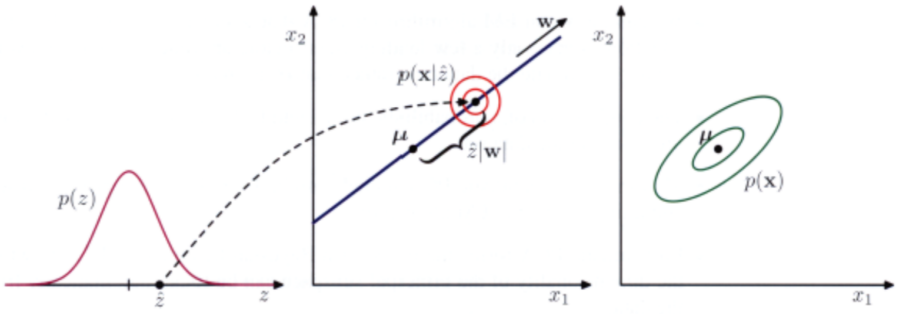


Figure 9.3: Probabilistic PCA from a generative viewpoint. The leftmost plot shows $p(z)$, the middle plot shows $p(x|z)$, and the rightmost plot shows the marginal density $p(x)$.

Generative Viewpoint

- Probabilistic PCA can be visualized as follows:
  - A $K = 1$-dimensional latent space is sampled ($p(z)$).
  - The latent variable z is projected into a $D = 2$-dimensional space using $x = Wz + \mu$.
  - Gaussian noise with variance σ^2 is added.
- The resulting marginal density $p(x)$ over the observed space is Gaussian.

9.1.4 Variational Autoencoders (VAEs)

Definition 9.1.5 (VAE Model). A VAE models data x with latent variables z using:

$$p_\theta(x, z) = p_\theta(x \mid z)p(z),$$

where $p(z)$ is typically a Gaussian prior $\mathcal{N}(0, I)$, and $p_\theta(x \mid z)$ is a Gaussian likelihood modeled by a neural network.

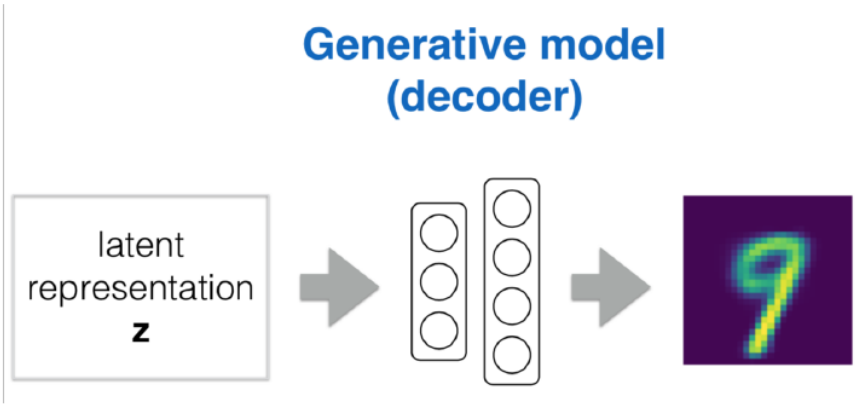


Figure 9.4: Variational Auto-Encoder Example.

Remark (Challenge the Marginal Likelihood)**.** *In this model, the marginal likelihood $p(X)$ is completely intractable:*

$$\log p(X) = \sum_{i=1}^{N} \log p(x_i) = \sum_{i=1}^{N} \log \int p(x_i \mid z_i, \theta) p(z_i)\, dz_i.$$

- *In the linear model, we had closed forms for both the marginal likelihood terms $p(x_i)$ and the posterior distribution $p(z_i \mid x_i)$.*
- *However, if we have a nonlinear f_θ in $p(x_i \mid z_i) = \mathcal{N}(x_i \mid f_\theta(z_i), \sigma^2 I)$, neither of these are the case.*

9.1.5 Amortized Inference Networks

We approach this problem similarly to how we have used *variational Bayes* elsewhere.

Main difference: Instead of learning a single posterior approximating $p(Z|X)$, we will employ **amortized inference**, by learning an *inference network*.

- For any input data $\mathbf{x}_i$, we will learn a "one data-point posterior" $q_\phi(\mathbf{z}_i|\mathbf{x}_i)$.
- Inference entails finding a value of ϕ that makes $q_\phi(\mathbf{z}_i|\mathbf{x}_i)$ "close" to the true posterior $p_\theta(\mathbf{z}_i|\mathbf{x}_i)$.

This is analogous to finding the projection matrix from $\mathbf{X}$ to $\mathbf{Z}$ in the linear case, where the same projection happens for each $\mathbf{x}_i$:

$$\mathbf{z}_i = \mathbf{W}^\top \mathbf{x}_i, \quad \mathbf{Z} = \mathbf{X}\mathbf{W}.$$

A typical choice for $q_\phi(\mathbf{z}|\mathbf{x})$ is to use a Gaussian distribution whose mean and variance are output by another deep network:

$$q_\phi(\mathbf{z}|\mathbf{x}) = \mathcal{N}(\mathbf{z}|\mu(\mathbf{x}), \text{diag}(\sigma(\mathbf{x})^2)),$$

where:

- $\mu : \mathbb{R}^D \to \mathbb{R}^K$: A deep network that outputs the posterior mean.
- $\sigma : \mathbb{R}^D \to \mathbb{R}^K$: A deep network that outputs the posterior standard deviation.

Practical Note: In practice, the network for σ typically outputs the log of the standard deviation, which is then exponentiated to ensure positivity.

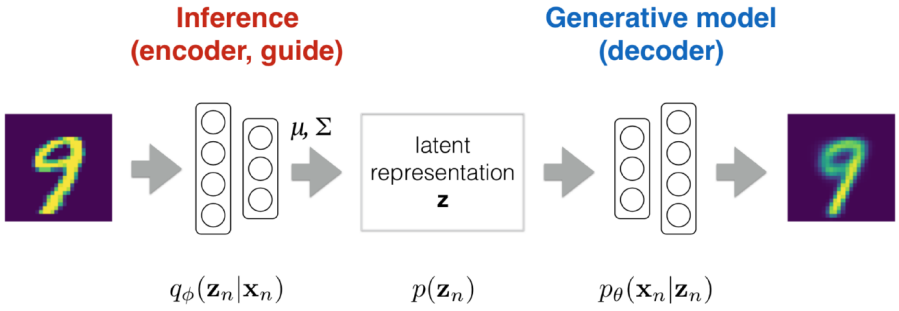


Figure 9.5: High-level diagram: autoencoder interpretation.

- **Encoder (Inference Network):** The encoder approximates $q_\phi(z \mid x)$ and maps input data x to the latent space z.
- **Decoder (Generative Model):** The decoder models $p_\theta(x \mid z)$ and reconstructs x from z.
- **Latent Representation:** The latent space z captures the essential features of the data.

Definition 9.1.6 (Inference Network)**.** The inference network $q_\phi(z \mid x)$ is parameterized as:

$$q_\phi(z \mid x) = \mathcal{N}(z \mid \mu(x), \text{diag}(\sigma^2(x))),$$

where:

- $\mu(x) : \mathbb{R}^D \to \mathbb{R}^K$ is a deep network that outputs the posterior mean.
- $\sigma(x) : \mathbb{R}^D \to \mathbb{R}^K$ is a deep network that outputs the posterior standard deviation.

Remark (Enforcing Positivity)**.** *In practice, the network for $\sigma(x)$ outputs $\log \sigma(x)$, which is exponentiated to ensure positivity for the standard deviation.*

Theorem 9.1.7 (Analogy to Linear Models)**.** *Amortized inference can be viewed as a nonlinear generalization of linear projection in classical latent variable models:*

$$z_i = W^\top x_i \quad \textit{and} \quad Z = XW,$$

where the same projection is applied for each data point x_i in the linear case.

9.1.6 The Kullback-Leibler (KL) Divergence and the Per-Datapoint ELBO

Definition 9.1.8 (Kullback-Leibler Divergence for Inference Networks)**.** To fit the inference network, we minimize the Kullback-Leibler (KL) divergence between the approximate posterior $q_\phi(z \mid x)$ and the true posterior $p_\theta(z \mid x)$:

$$D_{\mathrm{KL}}(q_\phi(z \mid x) \| p_\theta(z \mid x)) = \int q_\phi(z \mid x) \log \frac{q_\phi(z \mid x)}{p_\theta(z \mid x)} dz.$$

Remark (Key Differences from Parameter Inference)**.** *This KL divergence differs from those used in parameter inference:*

- *It is performed over the latent representation z, not the network parameters θ.*
- *The posterior approximation $q_\phi(z \mid x)$ explicitly takes the data x as input.*

9.1.7 Derivation of the Per-Datapoint ELBO

Rearranging the KL divergence leads to a tractable optimization objective:

$$D_{\mathrm{KL}}(q_\phi(z \mid x) \| p_\theta(z \mid x)) = \int q_\phi(z \mid x) \log \frac{q_\phi(z \mid x)}{p_\theta(z \mid x)} dz.$$

Breaking down the terms:

$$D_{\mathrm{KL}}(q_\phi(z \mid x) \| p_\theta(z \mid x)) = \int q_\phi(z \mid x) \log \frac{p_\theta(x, z)}{q_\phi(z \mid x)} dz - \log p_\theta(x),$$

we arrive at:

$$-\mathbb{E}_{q_\phi(z|x)} \left[\log \frac{p_\theta(x, z)}{q_\phi(z \mid x)} \right] + \log p_\theta(x).$$

Definition 9.1.9 (Per-Datapoint ELBO)**.** The per-datapoint Evidence Lower Bound (ELBO) is given by:

$$\mathcal{L}(x; \phi, \theta) = \mathbb{E}_{q_\phi(z|x)} \left[\log \frac{p_\theta(x, z)}{q_\phi(z \mid x)} \right].$$

This is tractable as it includes the joint distribution $p_\theta(x, z)$, rather than the intractable posterior $p_\theta(z \mid x)$ or marginal likelihood $p_\theta(x)$.

9.1.8 Connecting the Per-Datapoint ELBO to the Full ELBO

By re-arranging, we recover:

$$\log p_\theta(x) = \mathcal{L}(x;\phi,\theta) + D_{\mathrm{KL}}(q_\phi(z \mid x) \| p_\theta(z \mid x)).$$

Summing over all data points, we derive the full ELBO:

$$\log p_\theta(X) = \sum_{i=1}^{N} \log p_\theta(x_i) \geq \sum_{i=1}^{N} \mathcal{L}(x_i;\phi,\theta) = \mathcal{L}(X;\phi,\theta).$$

Remark (Relation to the Full ELBO)**.** *The per-datapoint ELBO is naturally extended to the full dataset:*

$$\mathcal{L}(X;\phi,\theta) = \sum_{i=1}^{N} \mathcal{L}(x_i;\phi,\theta).$$

9.1.9 The Evidence Lower Bound (ELBO)

Definition 9.1.10 (Evidence Lower Bound (ELBO))**.** The Evidence Lower Bound (ELBO) is a tractable approximation used to maximize the marginal likelihood $\log p_\theta(x)$ by separating it into two components:

$$\log p_\theta(x) = \mathcal{L}(x;\phi,\theta) + D_{\mathrm{KL}}(q_\phi(z \mid x) \| p_\theta(z \mid x)),$$

where:

- $\mathcal{L}(x;\phi,\theta)$ is the ELBO, an optimization objective.
- $D_{\mathrm{KL}}(q_\phi(z \mid x) \| p_\theta(z \mid x))$ is the KL divergence between the approximate posterior and the true posterior.

Remark (Maximizing the ELBO)**.** *Maximizing $\mathcal{L}(x;\phi,\theta)$ serves as a surrogate to maximize $\log p_\theta(x)$, as:*

$$\mathcal{L}(x;\phi,\theta) \leq \log p_\theta(x).$$

This is equivalent to maximizing the marginal likelihood and minimizing the KL divergence.

Theorem 9.1.11 (Decomposition of the ELBO)**.** *The ELBO can be decomposed into:*

$$\mathcal{L}(x;\phi,\theta) = \mathbb{E}_{q_\phi(z|x)}\left[\log p_\theta(x \mid z)\right] - D_{\mathrm{KL}}(q_\phi(z \mid x) \| p(z)),$$

where:

- *The first term $\mathbb{E}_{q_\phi(z|x)}\left[\log p_\theta(x \mid z)\right]$ is the reconstruction term.*
- *The second term $D_{\mathrm{KL}}(q_\phi(z \mid x) \| p(z))$ is the regularization term that penalizes you from taking a data point and sending it to be very far from 0.*

9.1.10 Optimizing the ELBO

So far, we have seen that maximizing the ELBO

$$\mathcal{L}(x;\phi,\theta) = \mathbb{E}_{q_\phi(z|x)}\left[\log \frac{p_\theta(x,z)}{q_\phi(z|x)}\right] = \mathbb{E}_{q_\phi(z|x)}[\log p_\theta(x|z)] - D_{KL}(q_\phi(z|x)\|p(z)),$$

is sensible for estimating both:

- the generative parameters θ, and
- the inference network parameters ϕ.

Let's look at the **reconstruction term** in the ELBO, $\mathbb{E}_{q_\phi(z|x)}[\log p_\theta(x|z)]$:

- Gradients ∇_θ: not a problem,

$$\nabla_\theta \mathbb{E}_{q_\phi(z|x)}[\log p_\theta(x|z)] = \mathbb{E}_{q_\phi(z|x)}[\nabla_\theta \log p_\theta(x|z)].$$

- Gradients ∇_ϕ: **reparameterization trick**.

Definition 9.1.12 (Reparameterization Trick)**.** Essentially, we need to ensure we can reparameterize the encoder distribution $q_\phi(z|x)$. Define a function $g_\phi(\epsilon, x)$ and distribution $p(\epsilon)$ such that if:

$$\epsilon \sim p(\epsilon), \quad \widetilde{z} = g_\phi(\epsilon, x),$$

then $\widetilde{z} \sim q_\phi(z|x)$.

For example, a Gaussian encoder:

$$q_\phi(z|x) = \mathcal{N}(z|\mu(x), \mathrm{diag}(\sigma(x)^2)),$$

would have:

$$p(\epsilon) = \mathcal{N}(0, \mathbf{I}),$$
$$\widetilde{z} = g_\phi(\epsilon, x) = \mu(x) + \sigma(x) \odot \epsilon.$$

Theorem 9.1.13 (Gradient Computation)**.** *We can compute the gradient using automatic differentiation (`auto-diff`):*

$$\nabla_{\phi,\theta}\mathbb{E}_{q_\phi(z|x)}\left[\log p_\theta(x \mid z)\right] = \nabla_{\phi,\theta}\mathbb{E}_{p(\epsilon)}\left[\log p_\theta(x \mid \widetilde{z} = g_\phi(\epsilon, x))\right] = \mathbb{E}_{p(\epsilon)}\left[\nabla_{\phi,\theta} \log p_\theta(x \mid \widetilde{z} = g_\phi(\epsilon, x))\right].$$

If $p_\theta(x \mid z)$ is Gaussian, e.g., $p_\theta(x \mid z) = \mathcal{N}(x \mid f_\theta(z), \sigma^2 I)$, then the gradient involves differentiating the squared error loss:

$$\log p_\theta(x \mid \widetilde{z} = g_\phi(\epsilon, x)) = -\frac{1}{2\sigma^2}\|x - f_\theta(g_\phi(\epsilon, x))\|_2^2 + \textit{const}.$$

This recovers the (probabilistic) PCA setting if both f_θ and g_ϕ are linear functions.

9.1.11 Mini-batch Gradient Estimation

The full likelihood can be approximated using mini-batches:

$$\log p_\theta(X) = \sum_{i=1}^{N} \log p_\theta(x_i) \geq \sum_{i=1}^{N} \mathcal{L}(x_i; \phi, \theta).$$

Using a mini-batch estimator:

$$\sum_{i=1}^{N} \mathcal{L}(x_i; \phi, \theta) \approx \frac{N}{M} \sum_{j=1}^{M} \mathcal{L}(x_j; \phi, \theta),$$

where x_j are M random samples from the dataset.

9.1.12 Summary of Variational Autoencoders

- A Variational Autoencoder (VAE) is a deep generative model for data x, with a latent variable z.

- The generative model is parameterized as:

$$p_\theta(X, Z) = \prod_{i=1}^{N} p_\theta(x_i \mid z_i) p(z_i).$$

- The model is trained by maximizing the ELBO:

$$q_\phi(z_i \mid x_i) \approx p_\theta(z_i \mid x_i).$$

- The optimization involves both the generative parameters θ and the inference network parameters ϕ.

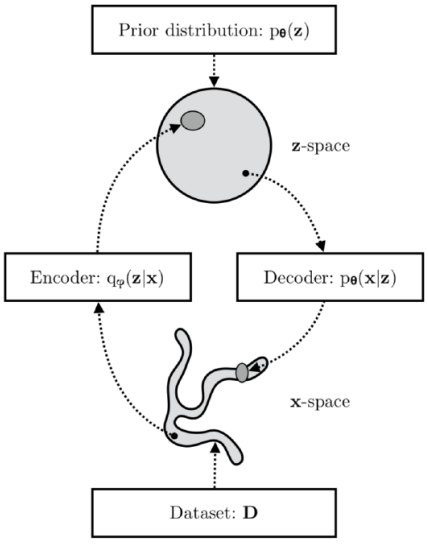
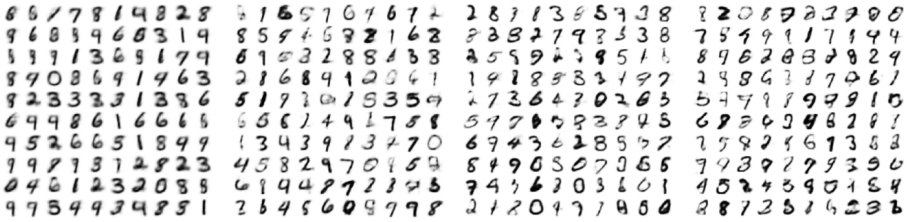


Figure 9.6: Variational Autoencoder Illustration.

9.2 Applications of Variational Autoencoders (VAEs)

Variational autoencoders (VAEs) have been applied to a variety of tasks across different datasets and domains, showcasing their versatility and effectiveness. Below, we outline some key examples:



(a) 2-D latent space (b) 5-D latent space (c) 10-D latent space (d) 20-D latent space

Figure 9.7: MNIST: Latent space comparison

9.2.1 MNIST: The "Hello World" of Deep Generative Models

- Dataset: MNIST, consisting of handwritten digit images.
- **Encoder**: Feed-forward network with a Gaussian posterior.

- **Decoder**: Feed-forward network with a Bernoulli or Gaussian likelihood.
- **Latent Space**: Typically 2-dimensional, allowing for visualization and manipulation. The latent space should be set properly (in 2-D, over-regularized and too simple to capture variations; in 20-D, capture too many variations)
- Observations:
  - The plot demonstrates the mean output of the decoder for varying latent space values.
  - Small latent spaces may limit expressive power, while larger ones increase flexibility.
  - Not use inference at all.

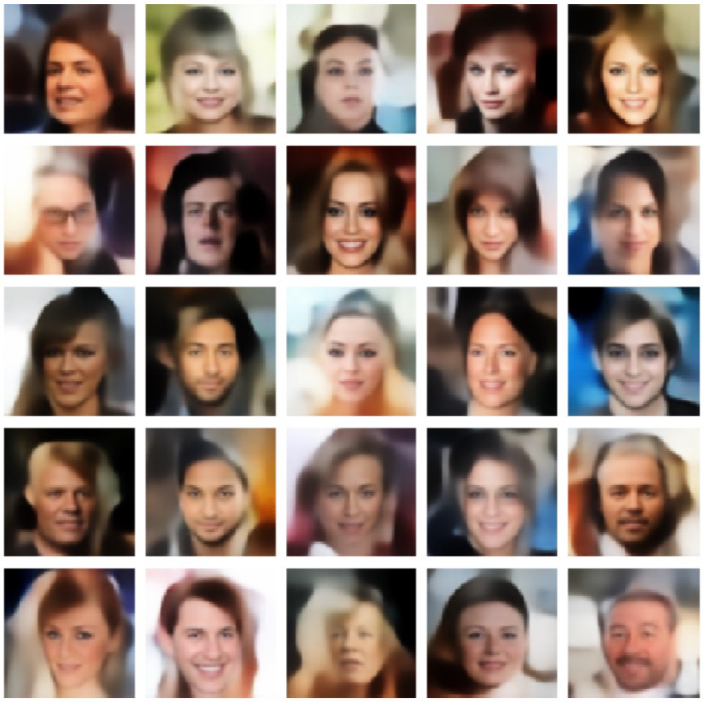


Figure 9.8: Convolutional Network Generating Faces.

9.2.2 Convolutional Networks for CelebA Dataset

- Dataset: CelebA, containing facial images.
- **Encoder**: Fully convolutional network with a Gaussian posterior.

- **Decoder**: Fully convolutional network with a Gaussian likelihood.

- Notes: Modern architectures and likelihood models can generate photorealistic images from the latent space.

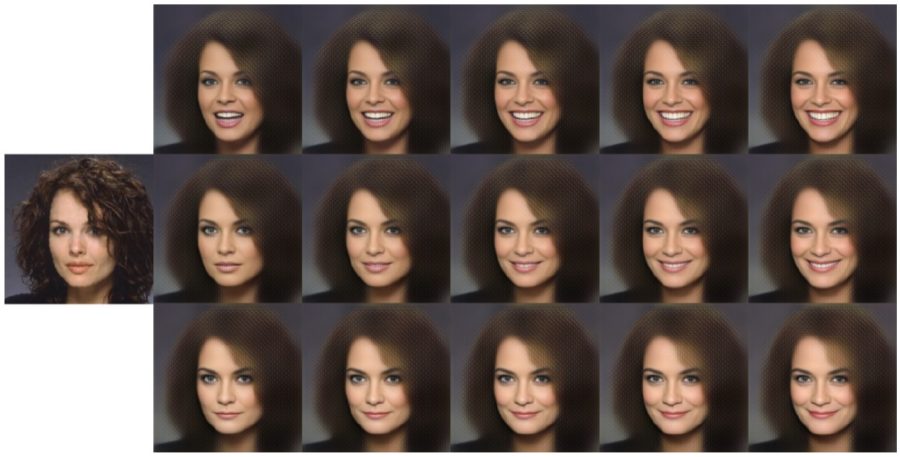


Figure 9.9: Perturbations in latent space.

9.2.3 Latent Space Perturbations

- Application: Identifying and manipulating latent space directions corresponding to interpretable properties.

- Example: Modifying facial attributes such as smile intensity, gender, or hairstyle by altering specific latent variables.

- Reference: Demonstrations by White (2016) showcase latent space navigation for controlled image manipulation.

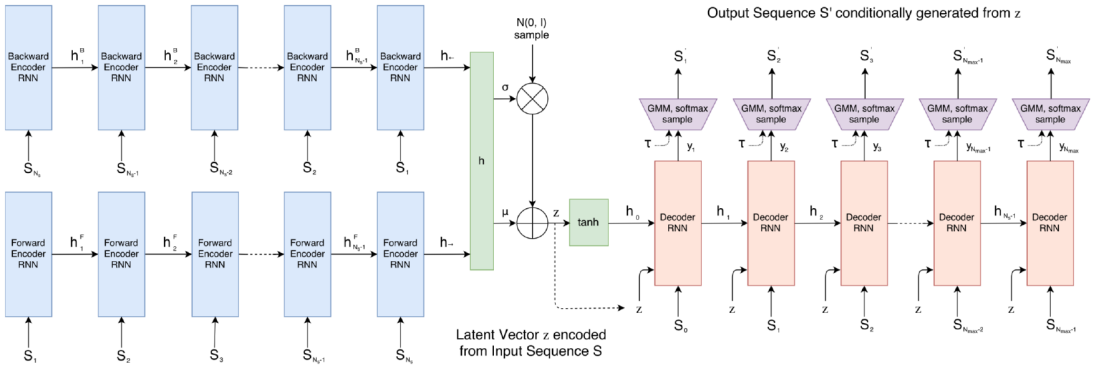


Figure 9.10: Sequence model: SketchRNN.

Figure 9.11: Reconstruction, generation, laten space manipulation.

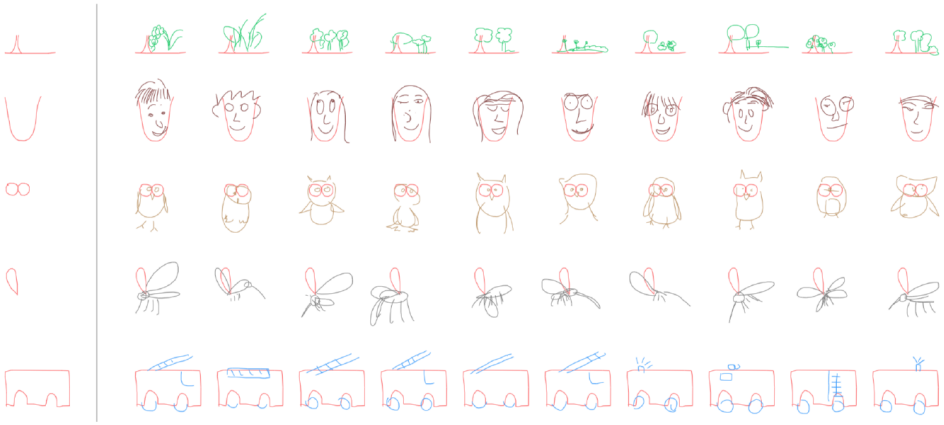


Figure 9.12: Automatic Completion of partial sketches.

9.2.4 SketchRNN: A Sequence Model

- Task: Generating sketch sequences conditioned on input sequences.

- **Encoder**: Bi-directional LSTM producing a Gaussian posterior.

- **Decoder**: LSTM predicting parameters of a distribution over five-tuple outputs (Δx, Δy, pen up, pen down, stop).

- Application: Captures temporal dependencies and produces coherent sequences based on the latent representation.

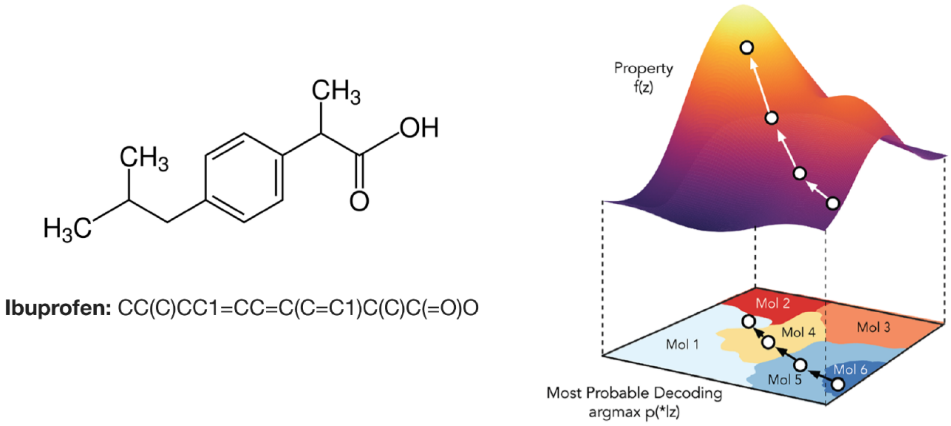


Figure 9.13: Sequence model for Molecules.

9.2.5 Sequence Models for Molecules

- Task: Modeling molecules as discrete objects represented by strings in a formal language (e.g., SMILES strings).
- **Encoder and Decoder**: Generate molecular representations in a continuous latent space.
- Uses:
  - Perform discrete optimization by transitioning to a continuous latent space.
  - Generate novel molecules or predict molecular properties efficiently.

9.3 Semi-Supervised Learning with VAEs

Definition 9.3.1 (Semi-Supervised Learning)**.** Semi-supervised learning is a machine learning paradigm that combines a small amount of labeled data with a large amount of unlabeled data during training. It leverages the unlabeled data to improve learning accuracy over using only the labeled data.

9.3.1 Incorporating Labels into Variational Autoencoders

Remark. *While standard VAEs are trained on unlabeled data x, it is beneficial to incorporate labeled pairs (x_i, y_i) when available to improve model performance and enable tasks such as classification.*

Definition 9.3.2 (Joint Generative Model)**.** We define a generative model over the joint distribution of data x, labels y, and latent variables z:

$$p_\theta(x, y, z) = p_\theta(x \mid y, z)\, p(z)\, p(y).$$

Remark. *This factorization mirrors the real-world causal process: first, a label y is chosen according to $p(y)$, then a latent variable z is sampled from $p(z)$, and finally, data x is generated conditioned on y and z.*

9.3.2 Generative Model and Inference Networks

Generative Model

- **Prior over** z: $p(z) = \mathcal{N}(z \mid 0, I)$.
- **Prior over** y: $p(y) = \text{Categorical}(y \mid \pi)$, where π is the prior class probability vector.
- **Likelihood**: $p_\theta(x \mid y, z) = f_\theta(x; y, z)$, where f_θ is a neural network parameterized by θ.

Inference Networks

- **Approximate Posterior over** y:

  $$q_\phi(y \mid x) = \text{Categorical}(y \mid \pi_\phi(x)),$$

  where $\pi_\phi(x)$ is modeled by a neural network.

- **Approximate Posterior over** z:

  $$q_\phi(z \mid x, y) = \mathcal{N}(z \mid \mu_\phi(x, y), \text{diag}(\sigma_\phi(x, y)^2)),$$

  with $\mu_\phi(x, y)$ and $\sigma_\phi(x, y)$ parameterized by neural networks.

Remark. *Inference proceeds by first inferring y from x, then inferring z conditioned on both x and y. This reflects the dependence structure in the generative model.*

9.3.3 Supervised and Unsupervised ELBOs

Supervised ELBO

Definition 9.3.3 (Supervised ELBO)**.** For labeled data (x_i, y_i), the Evidence Lower Bound (ELBO) is:

$$\mathcal{L}(x_i, y_i; \phi, \theta) = \mathbb{E}_{q_\phi(z|x_i,y_i)} \left[\log p_\theta(x_i, y_i, z) - \log q_\phi(z \mid x_i, y_i)\right].$$

Remark. *This ELBO is similar to that of a standard VAE but includes the label y_i as observed data.*

Unsupervised ELBO

Definition 9.3.4 (Unsupervised ELBO)**.** For unlabeled data x_j, the ELBO is:

$$\mathcal{U}(x_j;\phi,\theta) = \mathbb{E}_{q_\phi(y,z|x_j)}\left[\log p_\theta(x_j,y,z) - \log q_\phi(y \mid x_j) - \log q_\phi(z \mid x_j,y)\right],$$

where $q_\phi(y,z \mid x_j) = q_\phi(y \mid x_j)q_\phi(z \mid x_j,y)$.

Remark. *Since y is unknown for unlabeled data, we need to infer it alongside z. The expectation is taken over both y and z.*

9.3.4 Semi-Supervised Objective Function

Definition 9.3.5 (Overall Objective Function)**.** The overall objective combines the ELBOs for labeled and unlabeled data:

$$\mathcal{J} = \sum_{i=1}^{N_s} \mathcal{L}(x_i,y_i;\phi,\theta) + \sum_{j=1}^{N_u} \mathcal{U}(x_j;\phi,\theta),$$

where N_s is the number of labeled samples and N_u is the number of unlabeled samples.

Remark (Additional Classification Objective)**.** *The current supervised terms do not include the quantity for a classifier, which means the classifier network is only estimated using the unsupervised data x_i. To directly improve the classifier $q_\phi(y \mid x)$, an additional supervised classification term can be added:*

$$\widetilde{\mathcal{J}} = \mathcal{J} + \alpha \sum_{i=1}^{N_s} \log q_\phi(y_i \mid x_i),$$

where α is a weighting hyperparameter.

9.3.5 Check-in and Practical Applications

Remark (Key Questions for VAEs)**.**

- ***Question:*** *Compared to the previous VAE, what do we think the latent space z will look like?*
- ***Question:*** *What is the difference between training this model when fully supervised, and learning ten independent VAEs (one per class)?*

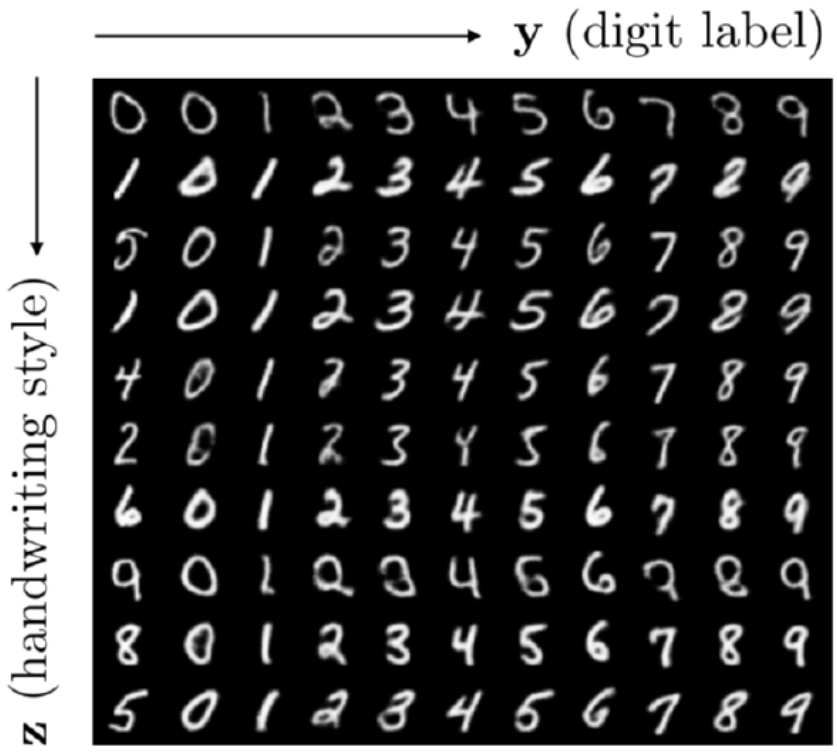


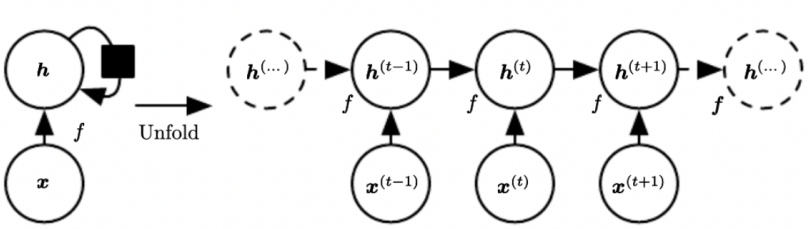
Figure 9.14: Style transfer, or visual analogy.

Style Transfer and Visual Analogies

- Holding the random variable z constant while modifying y transfers *style* across classes.
- Each row in the visualization corresponds to a fixed value of z_i, and columns show the mean $p_\theta(x \mid y_k, z_i)$ for $k = 0, \ldots, 9$.
- This method demonstrates the capability of VAEs to disentangle style (z) from content (y).

Street View House Numbers (SVHN)

- Similar style transfer approach applied to the SVHN dataset.
- The first column shows real data, while the rest display label-conditional reconstructions for digits $0 - 9$.



$$\mathbf{h}_t = f_\theta(\mathbf{x}_t, \mathbf{h}_{t-1})$$

][H]

Figure 9.15: Unfolding Recurrent Neural Networks

How Good is the Classifier?

Remark.
- ***Kingma2014*** *reported a 3.33% test error on MNIST with only 100 labels (10 labels per class), reduced to 2.18% with 3000 labels.*
- ***Siddharth2017*** *achieved a 1.57% test error using 3000 labels with different network architectures, comparable to results on the full dataset.*
- *The MNIST dataset contains a total of 60,000 examples. The Street View House Numbers dataset shows similar performance trends for classification.*

9.4 Deep Generative Models for Sequence Data

9.4.1 Recurrent Neural Networks (RNNs) Architectures

Definition 9.4.1 (Recurrent Neural Network)**.** A Recurrent Neural Network models sequential data by updating its hidden state h_t based on the current input x_t and the previous hidden state h_{t-1}:

$$h_t = f_\theta(x_t, h_{t-1}),$$

where f_θ is a parameterized function, often a nonlinear transformation.

Theorem 9.4.2 (Unfolding Recurrent Neural Networks)**.** *The first diagram illustrates the structure of a Recurrent Neural Network (RNN) and its temporal unfolding over time. At each time step t, the*

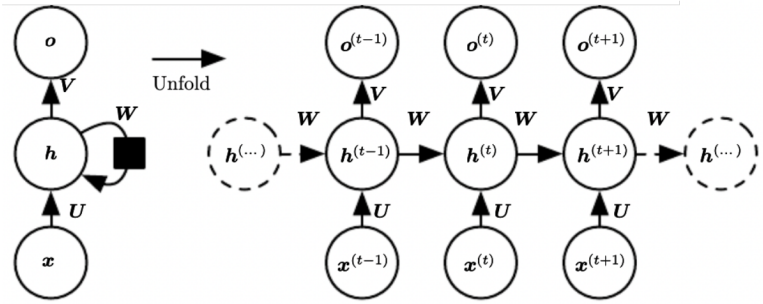
RNN processes an input $\mathbf{x}_t$ and computes a hidden state $\mathbf{h}_t$ based on the current input and the previous hidden state $\mathbf{h}_{t-1}$. Mathematically, this is expressed as:

$$\mathbf{h}_t = f_\theta(\mathbf{x}_t, \mathbf{h}_{t-1}),$$

where f_θ is a parameterized nonlinear function (often implemented as a neural network layer).

Intuition:

- *The hidden state $\mathbf{h}_t$ serves as a compact summary of all the inputs received up to time t. It captures both the immediate input $\mathbf{x}_t$ and the temporal dependencies encoded in $\mathbf{h}_{t-1}$.*
- *The "unfolding" of the RNN demonstrates how the computation graph spans across time steps, emphasizing that the hidden state $\mathbf{h}_t$ is recursively updated based on previous states and inputs.*
- *This recurrent mechanism enables the RNN to model sequential data and capture patterns that unfold over time, such as text, speech, or time series data.*



$$\mathbf{a}_t = \mathbf{b} + \mathbf{W}\mathbf{h}_{t-1} + \mathbf{U}\mathbf{x}_t \qquad \mathbf{h}_t = \tanh(\mathbf{a}_t) \qquad \mathbf{o}_t = \mathbf{c} + \mathbf{V}\mathbf{h}_t$$

Figure 9.16: Temporal Unfolding RNNs

Theorem 9.4.3 (Temporal Unfolding RNNs)**.** *The basic RNN equations can be written as:*

$$\begin{aligned} a_t &= b + Wh_{t-1} + Ux_t, \\ h_t &= \tanh(a_t), \\ o_t &= c + Vh_t, \\ \hat{y}_t &= \textit{softmax}(o_t), \end{aligned}$$

where a_t is the activation, o_t is the output, and $\hat{y}_t$ is the predicted probability distribution over possible outputs.

Intuition:

- *The looped diagram on the left indicates that the RNN processes inputs sequentially, maintaining a hidden state $\mathbf{h}_t$ that evolves over time.*
- *Unfolding the RNN (right side of the diagram) makes the temporal dependencies explicit, showing how the hidden states $\mathbf{h}_t$ are recursively updated across time steps based on $\mathbf{x}_t$ and $\mathbf{h}_{t-1}$.*
- *The dashed circles indicate that the temporal sequence could extend indefinitely, making RNNs particularly well-suited for processing variable-length sequential data.*
- *This mechanism allows the RNN to model both short-term and long-term dependencies in sequential data, such as text, time series, or speech.*

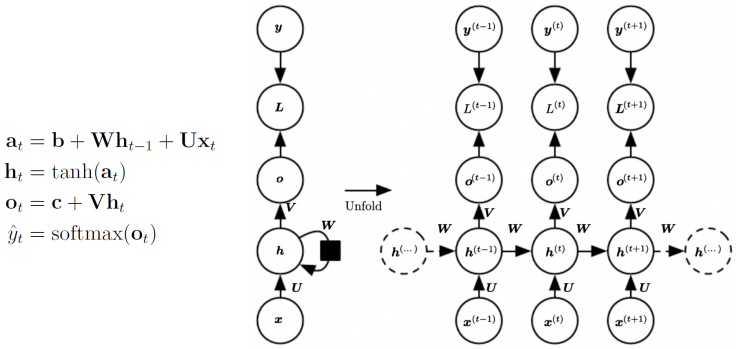


Figure 9.17: RNNs Architecture with Computation Nodes

Theorem 9.4.4 (RNNs Architecture with Computation Nodes)**.** ***Mathematical Formulation:*** *The operations within the RNN are defined as:*

$$\begin{aligned}
\mathbf{a}_t &= \mathbf{b} + \mathbf{W}\mathbf{h}_{t-1} + \mathbf{U}\mathbf{x}_t, \\
\mathbf{h}_t &= \tanh(\mathbf{a}_t), \\
\mathbf{o}_t &= \mathbf{c} + \mathbf{V}\mathbf{h}_t, \\
\hat{\mathbf{y}}_t &= \textit{softmax}(\mathbf{o}_t),
\end{aligned}$$

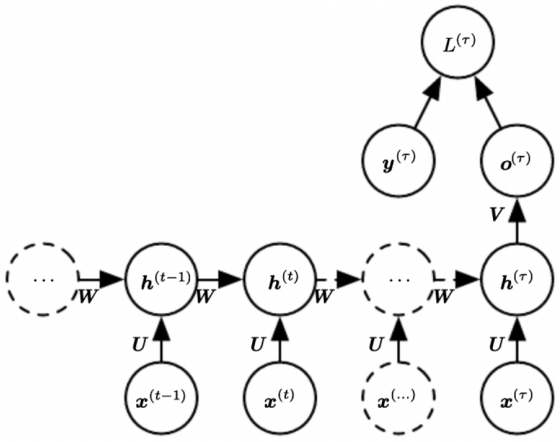
where:

- $\mathbf{a}_t$ *is the intermediate activation at time* t*,*
- $\mathbf{h}_t$ *is the hidden state at time* t*,*
- $\mathbf{o}_t$ *is the output at time* t*,*
- $\hat{\mathbf{y}}_t$ *is the predicted probability distribution over possible outputs,*

- $\mathbf{b}$, $\mathbf{c}$ *are bias vectors,*
- $\mathbf{W}$, $\mathbf{U}$, *and* $\mathbf{V}$ *are weight matrices.*

Intuition:

- *The left portion of the diagram depicts the compact representation of the RNN, showcasing how the input* $\mathbf{x}_t$ *is processed through a series of transformations to produce the output* $\hat{\mathbf{y}}_t$.
- *The temporal unfolding (right side) explicitly demonstrates how hidden states* $\mathbf{h}_t$ *are recursively updated and reused across time steps, capturing temporal dependencies in the input sequence.*
- *The use of non-linear activation (*$\tanh$*) in updating the hidden state allows the network to model complex patterns in sequential data.*
- *The softmax layer at the output ensures that the model generates a valid probability distribution for predictions.*
- *Dashed circles at the beginning and end indicate the potential for indefinitely long sequences, a key strength of RNNs in handling variable-length data.*



$$\mathbf{h}_\tau = f_\theta(\mathbf{x}_1, \ldots, \mathbf{x}_\tau) \qquad \mathbf{y} = g_\theta(\mathbf{h}_\tau)$$

Figure 9.18: Summarizing an entire sequence.

Theorem 9.4.5 (Summarizing an Entire Sequence)**.** *This plot represents the process of summarizing a sequence into a single vector representation and subsequently producing an output.*

Mathematical Formulation:

$$\mathbf{h}_T = f_\theta(\mathbf{x}_1, \ldots, \mathbf{x}_T),$$
$$\mathbf{y} = g_\theta(\mathbf{h}_T).$$

Details:

- *f_θ: A recurrent function that iteratively computes hidden states, where each hidden state $\mathbf{h}_t$ is updated using:*
$$\mathbf{h}_t = f_\theta(\mathbf{x}_t, \mathbf{h}_{t-1}),$$
with $\mathbf{x}_t$ being the input at time t, and $\mathbf{h}_{t-1}$ the hidden state from the previous time step.
- *g_θ: A function (often a feedforward neural network) that maps the final hidden state $\mathbf{h}_T$ to the output $\mathbf{y}$.*
- *$\mathbf{W}, \mathbf{U}$: Weight matrices governing the transformations within f_θ.*
- *The sequence is processed from $t = 1$ to T, and the final hidden state $\mathbf{h}_T$ summarizes the information from the entire input sequence.*

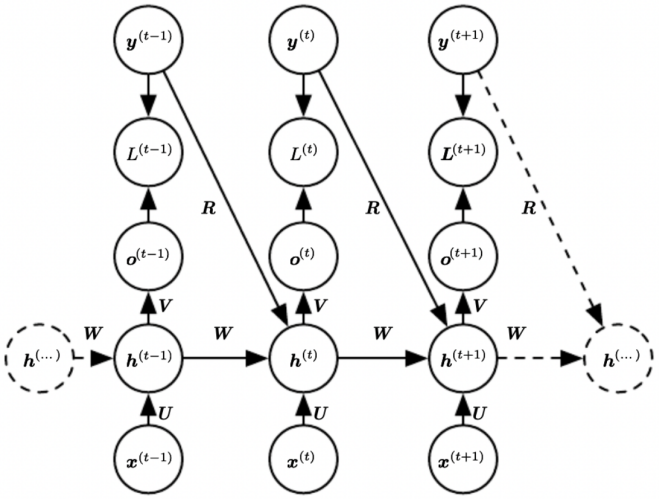


Figure 9.19: Conditioning on output values

Theorem 9.4.6 (Conditioning on Output Values)**.** *This diagram illustrates a sequence model where the hidden states $\mathbf{h}_t$ are influenced by both the input sequence $\mathbf{x}_t$ and the output sequence $\mathbf{y}_t$. This setup is often used in autoregressive models for sequence prediction.*

Mathematical Formulation:

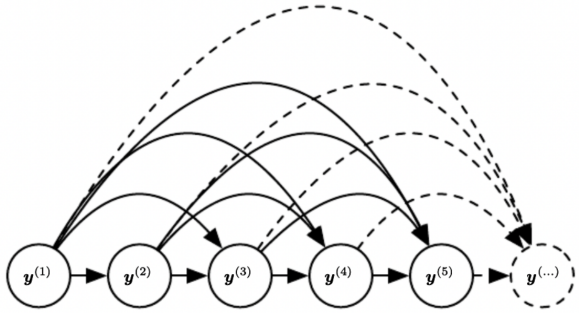
$$\begin{aligned}
\mathbf{a}_t &= \mathbf{b} + \mathbf{W}\mathbf{h}_{t-1} + \mathbf{U}\mathbf{x}_t + \mathbf{R}\mathbf{y}_{t-1}, \\
\mathbf{h}_t &= \tanh(\mathbf{a}_t), \\
\mathbf{o}_t &= \mathbf{c} + \mathbf{V}\mathbf{h}_t, \\
\hat{\mathbf{y}}_t &= \textit{softmax}(\mathbf{o}_t),
\end{aligned}$$

where:

- $\mathbf{h}_t$*: Hidden state at time* t*, encoding the combined influence of* $\mathbf{x}_t$*,* $\mathbf{h}_{t-1}$*, and* $\mathbf{y}_{t-1}$*.*
- $\mathbf{W}, \mathbf{U}, \mathbf{R}$*: Weight matrices for the previous hidden state, current input, and previous output, respectively.*
- $\mathbf{V}$*: Weight matrix mapping the hidden state to the logits* $\mathbf{o}_t$*.*
- $\hat{\mathbf{y}}_t$*: Predicted output at time* t*, computed using a softmax activation on the logits* $\mathbf{o}_t$*.*

Details:

- *The inclusion of* $\mathbf{R}\mathbf{y}_{t-1}$ *ensures that the model conditions on the output* $\mathbf{y}_{t-1}$ *from the previous time step, introducing an autoregressive dependency.*
- *The process unfolds iteratively across time, where each time step's output* $\hat{\mathbf{y}}_t$ *is influenced by the inputs, hidden states, and outputs from previous steps.*
- *This model is particularly useful for tasks such as language modeling, where predicting the next word requires knowledge of both past words and the sequential structure.*



$$p(y_1, \ldots, y_\tau) = p(y_1)p(y_2|y_1)p(y_3|y_1, y_2) \ldots p(y_\tau|y_1, \ldots, y_{\tau-1})$$

Figure 9.20: As a probability distribution over outputs

Theorem 9.4.7 (Probability Distribution Over Outputs)**.** *This diagram represents a sequential model that defines a joint probability distribution over the sequence of outputs* $\mathbf{y}_1, \mathbf{y}_2, \ldots, \mathbf{y}_T$.

Mathematical Formulation:

$$p(\mathbf{y}_1, \mathbf{y}_2, \ldots, \mathbf{y}_T) = p(\mathbf{y}_1)p(\mathbf{y}_2 \mid \mathbf{y}_1)p(\mathbf{y}_3 \mid \mathbf{y}_1, \mathbf{y}_2) \cdots p(\mathbf{y}_T \mid \mathbf{y}_1, \mathbf{y}_2, \ldots, \mathbf{y}_{T-1}).$$

Details:

- *The model decomposes the joint probability distribution into a product of conditional probabilities, leveraging the chain rule of probability.*
- *Each term* $p(\mathbf{y}_t \mid \mathbf{y}_1, \mathbf{y}_2, \ldots, \mathbf{y}_{t-1})$ *represents the probability of the current output* $\mathbf{y}_t$*, conditioned on all previous outputs.*
- *The dashed arrows in the diagram indicate dependencies between the output variables, illustrating that* $\mathbf{y}_t$ *depends on all preceding outputs* $\mathbf{y}_1, \mathbf{y}_2, \ldots, \mathbf{y}_{t-1}$.
- *This formulation is commonly used in autoregressive models, such as language models and sequence prediction tasks, where the output at each step is conditioned on the entire history of previous outputs.*

Applications:

- *Language modeling, where* $\mathbf{y}_t$ *represents the next word in a sentence.*
- *Time series forecasting, where* $\mathbf{y}_t$ *represents the predicted value at a future time step.*
- *Sequence generation tasks, such as text generation or music composition.*

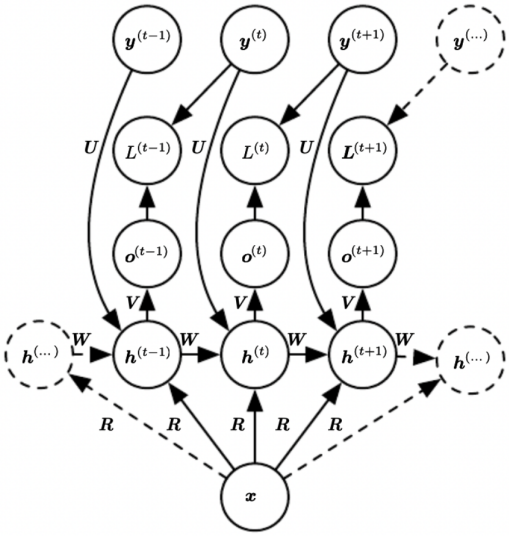


Figure 9.21: Conditioning on 'context'.

Theorem 9.4.8 (Conditioning on Context)**.** *This diagram extends the structure of recurrent neural networks (RNNs) to include conditioning on an external context vector* $\mathbf{x}$*.*

Mathematical Formulation:

$$\begin{aligned}
\mathbf{h}^{(t)} &= f_\theta(\mathbf{h}^{(t-1)}, \mathbf{y}^{(t-1)}, \mathbf{x}), \\
\mathbf{L}^{(t)} &= W\mathbf{h}^{(t)}, \\
\mathbf{o}^{(t)} &= V\mathbf{L}^{(t)} + c, \\
\mathbf{y}^{(t)} &= \mathrm{softmax}(\mathbf{o}^{(t)}).
\end{aligned}$$

Details:

- *The hidden state* $\mathbf{h}^{(t)}$ *at each time step* t *is computed as a function of the previous hidden state* $\mathbf{h}^{(t-1)}$*, the previous output* $\mathbf{y}^{(t-1)}$*, and an additional context vector* $\mathbf{x}$*.*
- $\mathbf{x}$ *serves as a global input that influences all time steps, providing external information or context to the recurrent computation.*
- *The dashed arrows represent the dependency of the hidden states* $\mathbf{h}$ *on the context* $\mathbf{x}$*, illustrating that the context vector is shared across all time steps.*
- *The output* $\mathbf{y}^{(t)}$ *is computed via a softmax operation, ensuring a valid probability distribution over the output classes.*

Applications:

- *Conditional sequence generation, where the context* $\mathbf{x}$ *provides additional information, such as a prompt or metadata.*
- *Machine translation, where* $\mathbf{x}$ *could represent the encoded source sentence.*
- *Text-to-speech models, where* $\mathbf{x}$ *encodes features of the input text or speaker characteristics.*

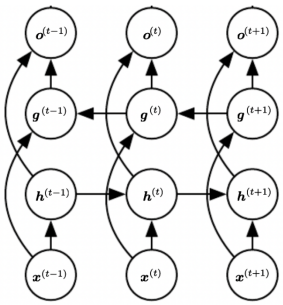


Figure 9.22: Bidirectional RNNs

Theorem 9.4.9 (Bidirectional Recurrent Neural Networks)**.** *Bidirectional Recurrent Neural Networks (BiRNNs) are a modification of standard RNNs, where the sequence is processed in both forward and backward directions to capture dependencies from both past and future contexts.*

Mathematical Formulation:

$$\begin{aligned}
\overrightarrow{\mathbf{h}}^{(t)} &= f_\theta\left(\overrightarrow{\mathbf{h}}^{(t-1)}, \mathbf{x}^{(t)}\right), \\
\overleftarrow{\mathbf{h}}^{(t)} &= f_\theta\left(\overleftarrow{\mathbf{h}}^{(t+1)}, \mathbf{x}^{(t)}\right), \\
\mathbf{g}^{(t)} &= g_\phi\left(\overrightarrow{\mathbf{h}}^{(t)}, \overleftarrow{\mathbf{h}}^{(t)}\right), \\
\mathbf{o}^{(t)} &= h_\psi\left(\mathbf{g}^{(t)}\right).
\end{aligned}$$

Details:

- *The forward hidden state* $\overrightarrow{\mathbf{h}}^{(t)}$ *processes the input sequence* $\{\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \dots\}$ *in the standard left-to-right direction.*
- *The backward hidden state* $\overleftarrow{\mathbf{h}}^{(t)}$ *processes the sequence in the reverse direction* $\{\mathbf{x}^{(T)}, \mathbf{x}^{(T-1)}, \dots\}$*, capturing information from future steps.*

- *The outputs $\mathbf{o}^{(t)}$ at each time step t depend on the combined information from both $\overrightarrow{\mathbf{h}}^{(t)}$ and $\overleftarrow{\mathbf{h}}^{(t)}$, enabling a richer representation.*

Applications:

- *Speech recognition: Captures both prior and future context for transcribing pre-recorded speech.*
- *Text processing: Useful for tasks like named entity recognition, where context from both directions improves prediction accuracy.*
- *Time-series prediction: Incorporates dependencies across the entire sequence for enhanced modeling.*

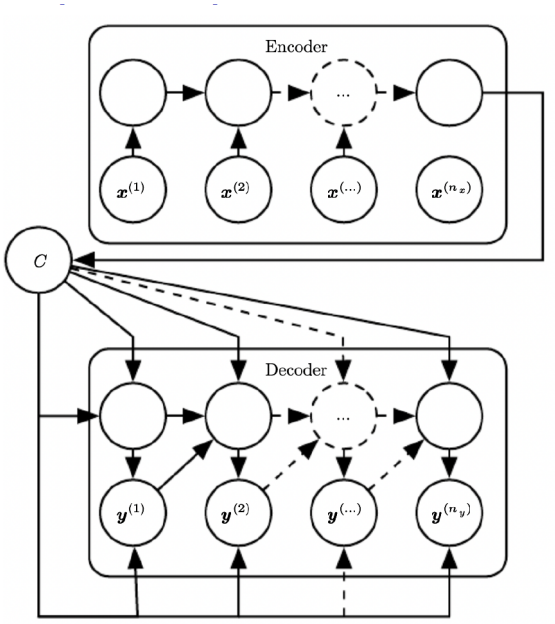


Figure 9.23:

Theorem 9.4.10 (Sequence-to-Sequence (Seq2Seq) Model)**.** *The Sequence-to-Sequence (Seq2Seq) model is designed to map input sequences of one length to output sequences of possibly different lengths. It consists of two primary components: an encoder and a decoder.*

Architecture:

- ***Encoder:*** *Processes the input sequence $\mathbf{x} = \{x^{(1)}, x^{(2)}, \ldots, x^{(n_x)}\}$ and encodes it into a fixed-length context vector $\mathbf{c}$:*

$$\mathbf{c} = f_{enc}\left(\mathbf{x}\right),$$

where f_{enc} is typically a recurrent neural network (RNN) or transformer architecture.

- ***Decoder:*** *Generates the output sequence $\mathbf{y} = \{y^{(1)}, y^{(2)}, \ldots, y^{(n_y)}\}$ conditioned on the context vector $\mathbf{c}$ and its own previous outputs:*

$$y^{(t)} = f_{dec}\left(y^{(t-1)}, \mathbf{c}\right), \quad t = 1, \ldots, n_y.$$

Key Features:

- *Allows for flexible input and output sequence lengths $n_x \neq n_y$.*
- *The context vector $\mathbf{c}$ serves as a bottleneck, summarizing the entire input sequence into a fixed-size representation.*
- *Decoding is autoregressive, meaning each output token depends on previously generated tokens.*

Mathematical Formulation:

$$\begin{aligned}
\textit{Encoder:} \quad & \mathbf{h}_{enc}^{(t)} = f_{enc}\left(\mathbf{h}_{enc}^{(t-1)}, x^{(t)}\right), \quad t = 1, \ldots, n_x, \\
\textit{Context Vector:} \quad & \mathbf{c} = \mathbf{h}_{enc}^{(n_x)}, \\
\textit{Decoder:} \quad & \mathbf{h}_{dec}^{(t)} = f_{dec}\left(\mathbf{h}_{dec}^{(t-1)}, y^{(t-1)}, \mathbf{c}\right), \quad t = 1, \ldots, n_y, \\
\textit{Output:} \quad & y^{(t)} = g_{out}\left(\mathbf{h}_{dec}^{(t)}\right).
\end{aligned}$$

Applications:

- *Machine translation (e.g., translating English sentences into French).*
- *Text summarization (e.g., generating concise summaries of long documents).*
- *Speech recognition and synthesis.*

9.4.2 Encoder-Decoder Architecture

Definition 9.4.11 (Encoder-Decoder Architecture)**.** The encoder-decoder framework uses a bidirectional RNN as an encoder to extract a context representation from the input sequence. A unidirectional RNN decoder generates the output sequence, allowing generative treatment of the decoder.

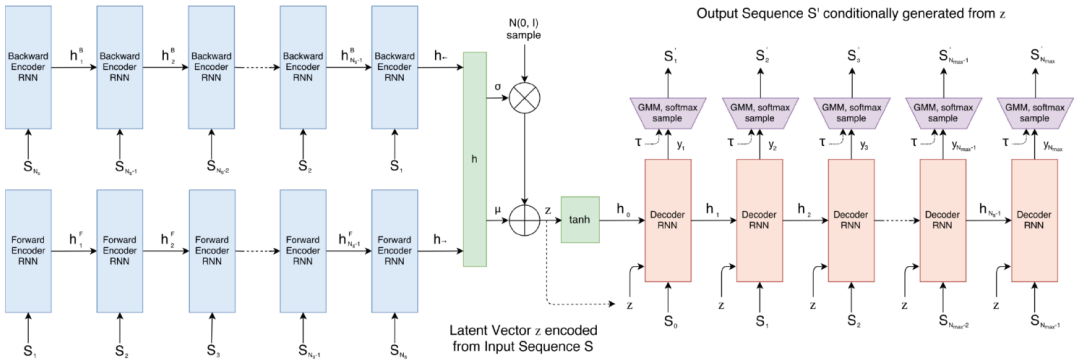


Figure 9.24: Encoder / Decoder Architecture

Remark (Applications)**.** *Encoder-decoder models are effective for tasks where input and output sequences have different lengths, such as:*

- *Machine translation,*
- *Speech recognition,*
- *Text summarization.*

9.4.3 Dealing with Long Sequences

Challenges in Long Sequence Modeling: When modeling long sequences, recurrent neural networks (RNNs) often face two major issues:

- **Vanishing gradients:** Gradients tend to shrink during backpropagation through time, leading to difficulties in learning long-range dependencies.
- **Exploding gradients:** Gradients may grow exponentially, making optimization unstable and leading to divergence.

Solution: Gated Networks (LSTMs and GRUs) To address these challenges, gated architectures like Long Short-Term Memory (LSTM) networks and Gated Recurrent Units (GRUs) were introduced:

- **LSTM:** Introduces gates (input, forget, and output gates) to control the flow of information through time, retaining long-term dependencies.

- **GRU:** A simplified version of LSTM with fewer parameters, using update and reset gates. The GRU update equations are as follows:

$$z_t = \sigma(W_z h_{t-1} + V_z x_t), \quad r_t = \sigma(W_r h_{t-1} + V_r x_t),$$
$$\widetilde{h}_t = \tanh(W(r_t \odot h_{t-1}) + V x_t),$$
$$h_t = (1 - z_t) \odot h_{t-1} + z_t \odot \widetilde{h}_t.$$

and this 'gating' behavior controls the flow of the hidden state through time. These models are better at capturing long-term dependencies than 'vanilla' RNNs.

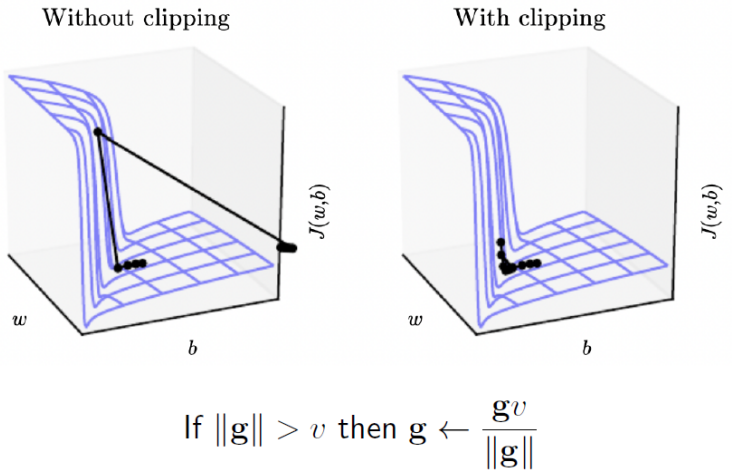


Figure 9.25: Gradient Clipping.

Theorem 9.4.12 (Gradient Clipping)**.** *Let* $\mathbf{g}$ *denote the gradient vector during backpropagation. To stabilize training, we enforce a constraint:*

$$\textit{If } \|\mathbf{g}\| > v, \quad \mathbf{g} \leftarrow \frac{\mathbf{g}v}{\|\mathbf{g}\|}$$

where v *is a pre-defined threshold. This prevents the gradients from exceeding a certain magnitude while maintaining their direction.*

Remark (Summary of RNNs:)**.**

- *RNNs are parameter-efficient due to weight sharing across timesteps.*
- *They are suitable for modeling sequence data with mostly local dependencies.*
- *High-level structures like sequence-to-sequence models enable applications such as machine translation.*

9.4.4 Transformers and Attention Mechanisms

On a lot of NLP tasks these days, these recurrent models are now outperformed by models based on "attention". The most famous family of these models are transformer networks. These networks use "positional embeddings" for each location in the sequence to allow learning what (potentially long-range) dependencies are important.

Definition 9.4.13 (Attention Mechanism)**.** Attention mechanisms compute the importance of each element in a sequence relative to others. The scaled dot-product attention is given by:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right)V,$$

where Q, K, and V are query, key, and value matrices, respectively.

Additionally, a general formulation for attention includes:

$$w^\top h(x) = \sum_i w_i h_i(x),$$

and in the context of parameterized weights:

$$w(x)^\top h(x) = \sum_i w_i(x) h_i(x).$$

For scaled dot-product attention, given:

$$q \in \mathbb{R}^b \to Q \circ \mathbb{R}^{D\times L}, \quad K \in \mathbb{R}^{M\times D}, \quad V \circ \mathbb{R},$$

the resulting attention computation is:

$$\text{Att}(q, K, V) = \text{softmax}\left(\frac{Kq}{\sqrt{D}}\right)V,$$

where the computed weights $w \in \mathbb{R}^M$ are applied to the values V to generate the final representation in $\mathbb{R}^H$.

Remark (Transformers)**.** *Transformers replace RNNs with **attention mechanisms**, enabling parallel computation and improved handling of long-range dependencies. Positional encoding is added to the input sequence to capture order information:*

$$PE(pos, 2i) = \sin\left(\frac{pos}{10000^{2i/d}}\right),$$
$$PE(pos, 2i+1) = \cos\left(\frac{pos}{10000^{2i/d}}\right).$$

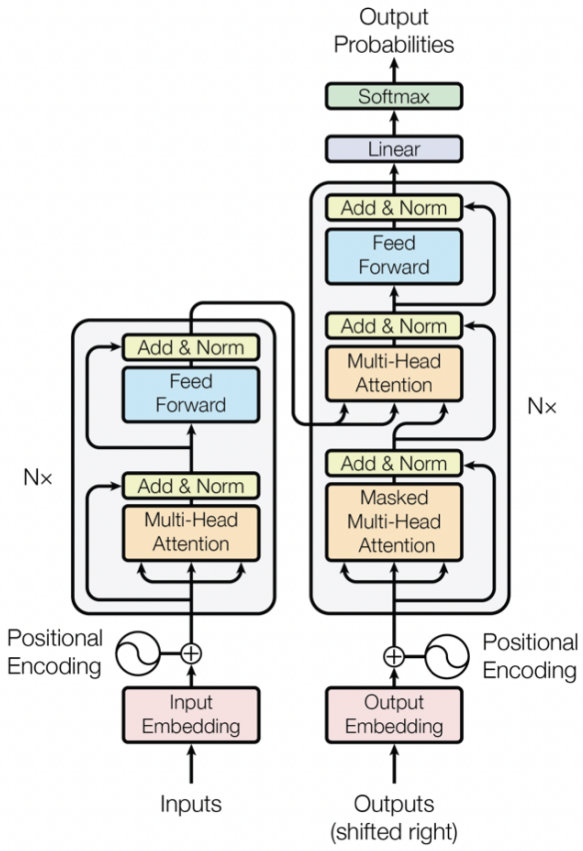


Figure 9.26: Transformer (Sequence model).

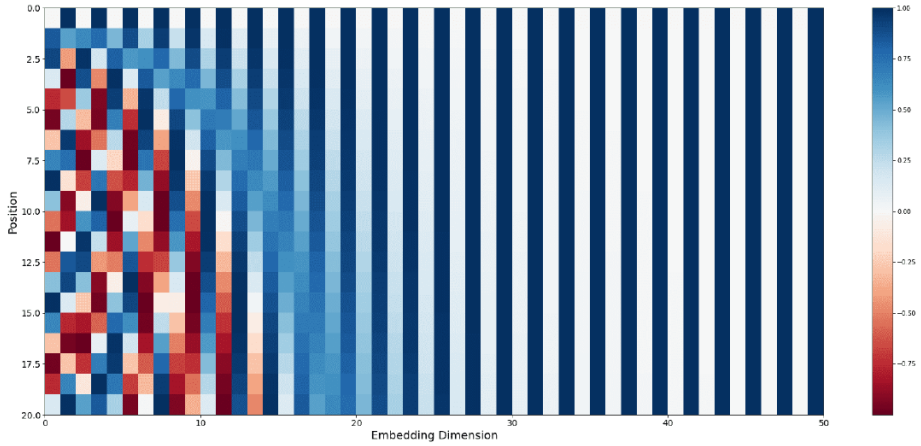


Figure 9.27: Positional Encoding.

- **Generative Pre-trained Transformers (GPT):** Auto-regressive models for generating sequences, trained left-to-right. Like a enidirectional RNN.
- **Bidirectional Encoder Representations (BERT):** Trained on a "masked language model" objective, where e.g. 20 per cent of the tokens are randomly removed, and the network aims to recover them given the rest of the sequence.

9.4.5 Summary of RNNs

- RNNs are efficient for modeling sequential data due to parameter sharing.
- They excel in applications with mostly local dependencies.
- Transformers now dominate many NLP tasks due to their superior handling of long-range dependencies.

9.5 Deep Generative Models Part II: Beyond VAEs

9.5.1 Improving Variational Autoencoders (VAEs)

- **Improving inference over z given x:**
  - Use $q(z|x)$ as an importance sampling proposal. This leads to methods like Importance Weighted Autoencoders (IWAE; Burda et al., 2015).

  - Initialize Markov Chain Monte Carlo (MCMC) methods using $q(z|x)$ (e.g., Hoffman, 2015).
  - Use a more expressive inference model $q(z|x)$, such as replacing the factorized Gaussian assumption with more complex distributions.

- **Improving the prior over z:**
  - Use mixture models as priors, allowing for multimodality in the latent space.
  - Employ autoregressive models as priors to capture sequential dependencies in z.

- **Disentanglement:**
  - Beta-VAE (Higgins et al., 2017): Encourages axis-aligned features in the latent space through a β parameter controlling the trade-off between reconstruction and KL divergence.
  - Total Correlation VAE (Chen et al., 2018): Minimizes the dependency between latent dimensions to improve disentanglement.

9.5.2 Alternative Objectives: The "other" KL divergence

Definition 9.5.1 (The ELBO Objective)**.** The Evidence Lower Bound (ELBO) minimizes:

$$\text{ELBO} = \mathbb{E}_{q_\phi(z|x)}[\log p_\theta(x|z)] - D_{\text{KL}}(q_\phi(z|x) \parallel p(z)),$$

providing a lower bound on $\log p(x)$.

Definition 9.5.2 (The "Other" KL Divergence)**.** An alternative way to motivate this is to directly minimize this KL divergence with respect to θ:

$$\min_\theta D_{\text{KL}}(q_\phi(z|x) \parallel p_\theta(z|x)) = \min_\theta \int q_\phi(z|x) \log \frac{q_\phi(z|x)}{p_\theta(z|x)}\, dz,$$

which simplifies to:

$$\min_\theta \mathbb{E}_{q_\phi(z|x)}[\log q_\phi(z|x)] - \mathbb{E}_{q_\phi(z|x)}[\log p_\theta(z|x)] = \max_\theta \mathbb{E}_{q_\phi(z|x)}[\log p_\theta(z|x)].$$

That is, minimizing this KL divergence is equivalent to maximizing the expected log posterior, under the approximate posterior.

This suggests an alternative approach for estimating ϕ: we find it by minimizing the other KL divergence:

$$\min_\phi D_{\text{KL}}(p_\theta(z|x) \parallel q_\phi(z|x)) = \min_\phi \int p_\theta(z|x) \log \frac{p_\theta(z|x)}{q_\phi(z|x)}\, dz,$$

which simplifies to:

$$\max_{\phi} \mathbb{E}_{p_\theta(z|x)}[\log q_\phi(z|x)].$$

This is an expectation under the current generative model, where we maximize the expected log-likelihood of the inference network.

This can be evaluated on real data $x_1, \ldots, x_N$, or on synthetic data $x \sim p_\theta(x)$. For the latter, note that:

$$\mathbb{E}_{p_\theta(x)}[D_{\mathrm{KL}}(p_\theta(z|x) \parallel q_\phi(z|x))] = \mathbb{E}_{p_\theta(x,z)}\left[\log \frac{p_\theta(z|x)}{q_\phi(z|x)}\right].$$

Definition 9.5.3 (Wake-Sleep Algorithm)**.** An iterative algorithm with two phases:

- **Wake Phase:** Update generative parameters θ using samples from $q_\phi(z|x)$, minimizing $D_{\mathrm{KL}}(q_\phi(z|x) \parallel p_\theta(z|x))$.
- **Sleep Phase:** Update inference parameters ϕ using synthetic data sampled from $p_\theta(x, z)$, minimizing $D_{\mathrm{KL}}(p_\theta(z|x) \parallel q_\phi(z|x))$.

Remark (Reweighted Wake-Sleep (Bornschein & Bengio, 2015))**.** *Instead of minimizing $\mathbb{E}_{p_{data}(x)}[D_{\mathrm{KL}}(p_\theta \parallel q_\phi)]$, let $\widetilde{p}(x)$ denote the data distribution, and minimize:*

$$\mathbb{E}_{\widetilde{p}(x)}[D_{\mathrm{KL}}(p_\theta(z|x) \parallel q_\phi(z|x))] = \mathbb{E}_{\widetilde{p}(x)}\left[\mathbb{E}_{p_\theta(z|x)}[\log p_\theta(z|x) - \log q_\phi(z|x)]\right] = \mathbb{E}_{\widetilde{p}(x)}[\mathbb{E}_{p_\theta(z|x)}[-\log q_\phi(z|x)]] + \textit{const}.$$

The inner expectation over $p_\theta(z|x)$ is tricky (it's over the posterior), but can be handled by importance sampling, using $q_\phi(z|x)$ as a proposal distribution with importance weights:

$$w(z) = \frac{p_\theta(x, z)}{q_\phi(z|x)}.$$

This is sometimes called the "wake-wake" algorithm, as it uses the data for each update.

Remark (Advantages and Disadvantages of Wake-Sleep)**.**

- ***Advantages:***
  - *Discrete latent variables z pose no problem.*
  - *Wake-Sleep (and re-weighted wake-sleep) sample from one distribution to compute gradient estimates of another. There is no need to apply the reparameterization trick:*

$$\nabla_\phi \mathbb{E}_{p_{data}(x)} \mathbb{E}_{p_\theta(z|x)}[\log q_\phi(z|x)] = \mathbb{E}_{p_{data}(x)} \mathbb{E}_{p_\theta(z|x)}[\nabla_\phi \log q_\phi(z|x)].$$

- ***Disadvantages:***
  - *There is no single objective and no guarantee of convergence to a fixed point.*
  - *Wake-sleep can suffer if the "dream" samples from $p_\theta(x, z)$ are far from the real data, particularly early in training.*
- *Due to importance sampling, "wake" gradient estimates for ∇_ϕ are biased, so multiple samples are required.*

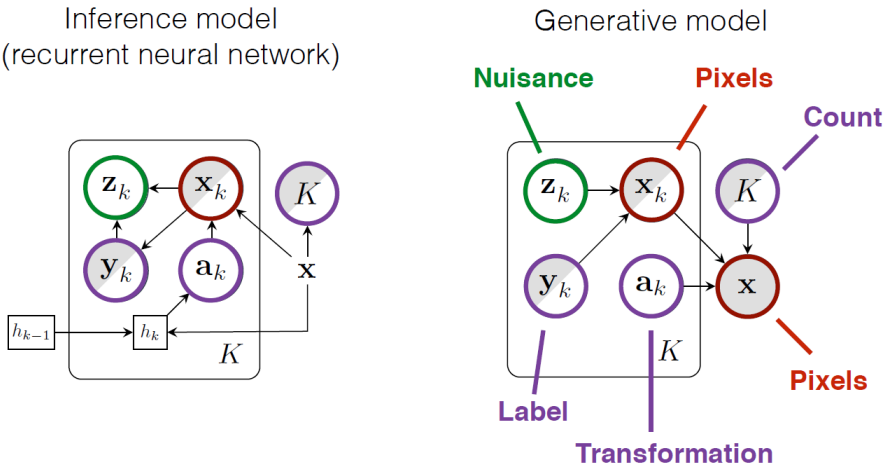


Figure 9.28: Inference and Generative Models.

9.5.3 Larger Compositional Models

Inference and Generative Models

- **Inference Model (Recurrent Neural Network):**
  - The model predicts the number of objects K in the image and their properties (e.g., transformations a_k, latent variables z_k for style, and y_k for label).
  - Hidden states h_{k-1} from the previous step are passed forward to predict K and x iteratively.
- **Generative Model:**
  - Decomposes the image into multiple components:
    - *Nuisance variables (z_k):* Represent style or noise in the data.
    - *Labels (y_k):* Represent the class (e.g., digits in an image).
    - *Transformations (a_k):* Specify affine transformations applied to the objects.
  - The number of objects K determines the total count.
  - Pixels x are reconstructed as a combination of all transformed components.

Inference: Counting and Locating

- **Counting:** The inference model predicts K, the number of objects in the image.

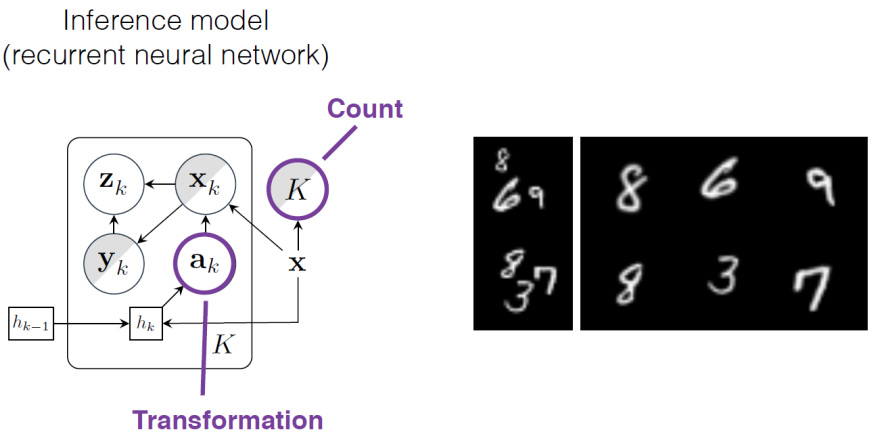


Figure 9.29: Inference: Counting and Locating.

- **Locating:** The transformations a_k allow the model to predict the position and orientation of each object.

- Example: For handwritten digits, the model counts and localizes individual digits within an image (e.g., 8, 6, 9, 8, 3, 7).

9.5.4 Pyro: Deep Probabilistic Programming

- Pyro extends PyTorch to add probabilistic modeling capabilities.

- Users write two programs:

  - A **model**, which defines the generative process.
  - A **guide**, which defines the variational inference process, both operating over the same random variables.

- Automatically computes and optimizes the ELBO for inference.

- Aims to make building deep probabilistic models easier and more efficient.

9.5.5 Other approaches to deep generative models

Generative Adversarial Networks (GANs)

Definition 9.5.4 (GAN Framework)**.** Unlike a Variational Autoencoder (VAE), a Generative Adversarial Network (GAN) generates data *exactly*:

$$z \sim p(z), \quad x = f_\theta(z).$$

This is an implicit or *likelihood-free* generative model, where the data generation process does not rely on explicit likelihood maximization or the Evidence Lower Bound (ELBO).

The GAN framework consists of two components:

- **Generator (G):** Takes a random noise vector z from a simple prior $p(z)$ and generates samples x.
- **Discriminator (D):** Acts as a binary classifier to distinguish between real data (from $p_{\text{data}}(x)$) and generated data (from $p_G(x)$).

The training objective is formulated as a two-player minimax game:

$$\min_G \max_D \mathbb{E}_{p_{\text{data}}(x)}[\log D(x)] + \mathbb{E}_{p_Z(z)}[\log(1 - D(G(z)))].$$

Remark. ***Training Strategy:***

- *The discriminator $D(x)$ learns to maximize its ability to classify real data as 1 and generated data as 0.*
- *The generator $G(z)$ learns to minimize the discriminator's ability to distinguish between real and generated data by improving the quality of generated samples.*

Remark. ***Advantages:***

- *GANs produce **high-quality, realistic samples**, often indistinguishable from real data.*
- *They are highly flexible and applicable to a wide range of data modalities (e.g., images, audio, and text).*

Disadvantages:

- ***Mode Collapse:** The generator may produce limited diversity in samples, failing to represent the full data distribution.*
- ***Training Instability:** The adversarial nature of GANs can lead to difficulties in optimization, requiring careful tuning of learning rates, architectures, and regularization techniques.*

Normalizing Flows

Definition 9.5.5 (Normalizing Flow)**.** A normalizing flow generates data *exactly*:

$$z \sim p(z), \quad x = f_\theta(z).$$

The key difference is that f_θ is restricted to be *bijective*, meaning f_θ is an invertible, one-to-one function. This imposes the condition that:

$$z \in \mathbb{R}^D, \quad x \in \mathbb{R}^D.$$

The likelihood can be computed directly, without requiring the Evidence Lower Bound (ELBO), by maximizing:

$$\log p_X(x) = \log p_Z(f_\theta^{-1}(x)) + \log \left| \det \frac{\partial f_\theta^{-1}(x)}{\partial x} \right|.$$

Remark. ***Training Process:***

- *The bijective property ensures tractability of the log-likelihood computation.*
- *Normalizing flows can be trained directly via maximum likelihood estimation.*

Remark. ***Advantages:***

- *Tractable training objectives, enabling gradient-based optimization.*
- *Guarantees invertibility, allowing exact likelihood evaluation.*

Disadvantages:

- *Requires specially designed layers to ensure bijectivity and efficient computation of the Jacobian determinant.*
- *Cannot perform dimensionality reduction, as z and x must have the same dimensionality.*

9.5.6 Amortized Inference without deep generative models

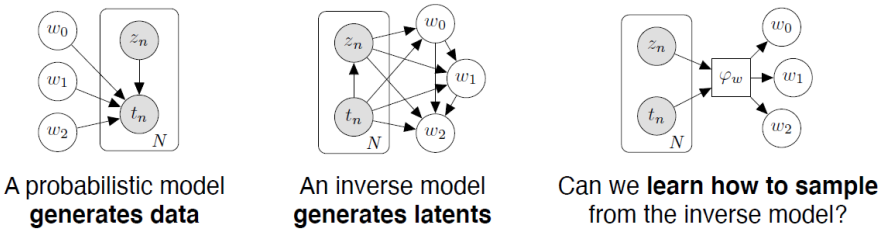


Figure 9.30: Learning Importance Sampling Proposals.

Learning Importance Sampling Proposals

Amortized inference seeks to approximate the target distribution $\pi(x) = p(x|y)$ by learning a mapping from the data to the target. The goal is to use an approximating family $q(x|\lambda)$, where λ is parameterized by a learned function of η and y, $\lambda = \varphi(\eta, y)$. This approach averages over all possible datasets to minimize the following objective:

$$\arg\min_{\eta} \mathbb{E}_{p(y)}\big[D_{\mathrm{KL}}(\pi(x) \parallel q(x|\varphi(\eta, y)))\big].$$

Algorithm: "Sleep" Step from Wake-Sleep

The "sleep" step in the wake-sleep algorithm focuses on learning to invert the generative model before seeing actual data. The new objective function for the upper-level parameters η is defined as:

$$J(\eta) = \int D_{\mathrm{KL}}(\pi(x|y) \parallel q(x|\varphi(\eta, y)))p(y)\, dy.$$

Expanding this, we have:

$$J(\eta) = \int p(y) \int p(x|y) \log \left[\frac{p(x|y)}{q(x|\varphi(\eta, y))}\right] dxdy,$$

which simplifies to:

$$J(\eta) = \mathbb{E}_{p(x,y)}[-\log q(x|\varphi(\eta, y))] + \text{const.}$$

The gradient for $J(\eta)$ is tractable and can be expressed as:

$$\nabla_\eta J(\eta) = \mathbb{E}_{p(x,y)}[-\nabla_\eta \log q(x|\varphi(\eta, y))].$$

Key Points:

- The expectation is taken over all data we might observe.
- Samples can be drawn from the joint distribution $p(x, y)$ to approximate the gradient.
- The algorithm can train entirely offline.

Application: Non-Conjugate Polynomial Regression

The effectiveness of amortized inference can be demonstrated with non-conjugate polynomial regression tasks. By employing neural networks for the inference model and importance-weighted proposals, we observe the following:

- Posterior distributions are approximated across multiple datasets (as shown in the plots).

- Neural network-based proposals provide flexible and scalable posterior estimation.
- Incorporating importance sampling helps refine the learned proposals.

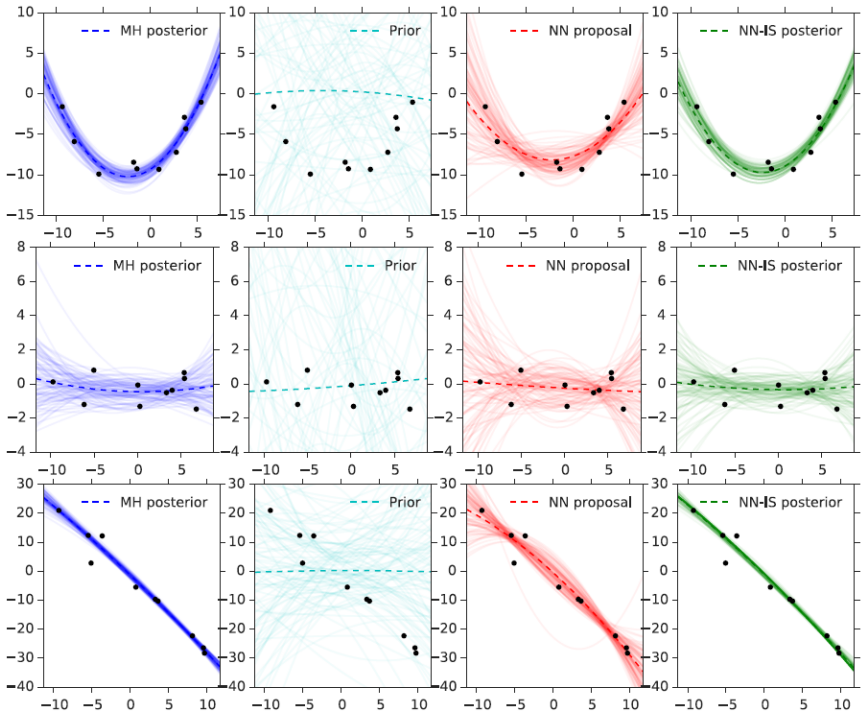


Figure 9.31: Non-conjugate polynomial regression.

The visualizations compare:

- **NN Posterior:** Posteriors learned using neural network inference models.
- **Prior:** Initial assumptions about the data distribution.
- **NN Proposal:** Importance sampling proposals derived from neural networks.
- **IWAE Posterior:** Improved posterior estimations using importance-weighted autoencoders.

9.5.7 Applications and Extensions

Example 9.5.6 (Multi-MNIST)**.** A framework for generating and recognizing multiple digits within an image:

- Combines generative and recognition models with a stochastic sequence generator.
- Handles variable numbers of latent variables K per image.
- Uses spatial transformer networks for image alignment.

Example 9.5.7 (Disentangled Representations)**.** Separates latent variables into interpretable components, such as:

- Label-dependent variables (y).
- Nuisance variables (z).

Remark. *Hybrid generative models, combining structured graphical models and unstructured random variables, enable flexible semi-supervised learning.*

Chapter 10

Gaussian Process and Active Learning

10.1 Gaussian Processes (GPs) and Related Concepts

10.1.1 Gaussian Processes

Definition 10.1.1 (Gaussian Process)**.** A Gaussian Process (GP) is a collection of random variables, such that any finite subset of them has a joint Gaussian distribution. Formally, if $f(x)$ represents the function values at inputs $x_1, x_2, \ldots, x_n$, then:

$$f(x_1), f(x_2), \ldots, f(x_n) \sim \mathcal{N}(\mu, \Sigma),$$

where $\mu = \mathbb{E}[f(x)]$ is the mean function and Σ is the covariance matrix. f is the latent (underlying) true function we are modeling.

- **Covariance Function (Kernel):** The covariance function $k(x, x')$ defines the entries of Σ, such that:
$$\Sigma_{ij} = k(x_i, x_j) = \text{Cov}(f(x_i), f(x_j)).$$
- **Key Assumption:** The kernel must be symmetric and positive semi-definite (PSD) for K to be a valid covariance matrix.

10.1.2 Gaussian Process Prior: Weight-Space Perspective

Setup

We begin with the function representation:

$$f_i = \omega^T \phi(x_i),$$

where:

- $x_i \in \mathbb{R}^D$ is the input,
- $f_i \in \mathbb{R}$ is the output,
- $\phi : \mathbb{R}^D \to \mathbb{R}^H$ is a feature map that maps inputs to an H-dimensional feature space,
- $\omega \sim \mathcal{N}(0, \Sigma)$ is a weight vector drawn from a Gaussian distribution with mean 0 and covariance Σ.

Joint Representation for Multiple Inputs

Given N inputs $x_1, x_2, \ldots, x_N$, we can represent the function values as a vector:

$$\mathbf{f} = \begin{bmatrix} f_1 \\ f_2 \\ \vdots \\ f_N \end{bmatrix} = \Phi\omega,$$

where:

- $\mathbf{f} \in \mathbb{R}^N$ is the vector of function values,
- $\Phi \in \mathbb{R}^{N \times H}$ is the feature matrix, where each row is $\phi(x_i)^T$,
- $\omega \in \mathbb{R}^H$ is the weight vector.

Question: What is the Prior Distribution of f?

To determine $p(\mathbf{f})$, we calculate its mean and covariance:

Mean of f

$$\mathbb{E}[\mathbf{f}] = \mathbb{E}[\Phi\omega] = \Phi\mathbb{E}[\omega].$$

Since $\omega \sim \mathcal{N}(0, \Sigma)$, we have $\mathbb{E}[\omega] = 0$. Thus:

$$\mathbb{E}[\mathbf{f}] = \Phi \cdot 0 = 0.$$

Covariance of $\mathbf{f}$

The covariance of $\mathbf{f}$ is:

$$\text{Cov}(\mathbf{f}) = \mathbb{E}[\mathbf{f}\mathbf{f}^T] - \mathbb{E}[\mathbf{f}]\mathbb{E}[\mathbf{f}]^T.$$

Since $\mathbb{E}[\mathbf{f}] = 0$, this simplifies to:

$$\text{Cov}(\mathbf{f}) = \mathbb{E}[\mathbf{f}\mathbf{f}^T].$$

Substituting $\mathbf{f} = \Phi\omega$:

$$\text{Cov}(\mathbf{f}) = \mathbb{E}[(\Phi\omega)(\Phi\omega)^T].$$

Expanding:

$$\text{Cov}(\mathbf{f}) = \Phi\mathbb{E}[\omega\omega^T]\Phi^T.$$

Since $\omega \sim \mathcal{N}(0, \Sigma)$, we know $\mathbb{E}[\omega\omega^T] = \Sigma$. Thus:

$$\text{Cov}(\mathbf{f}) = \Phi\Sigma\Phi^T.$$

Resulting Distribution of $\mathbf{f}$

The mean and covariance give the prior distribution of $\mathbf{f}$:

$$\mathbf{f} \sim \mathcal{N}(0, K),$$

where $K = \Phi\Sigma\Phi^T$ is the covariance matrix.

10.1.3 Kernels and the Kernel Trick

Definition 10.1.2 (Covariance Matrix of Gaussian Processes)**.** The covariance matrix K for a Gaussian Process is defined as:

$$K = \Phi\Sigma\Phi^T = \text{Cov}(f),$$

where K_{ij} is the covariance between $f(x_i)$ and $f(x_j)$. Each entry of K is given by:

$$K_{ij} = \text{Cov}(f(x_i), f(x_j)) = \phi(x_i)^T\Sigma\phi(x_j) = k(x_i, x_j),$$

where $k(x_i, x_j)$ is the kernel function. The function values f follow the distribution:

$$f \sim \mathcal{N}(0, K),$$

where K is the covariance matrix.

Definition 10.1.3 (Kernel Trick)**.** The kernel trick enables the use of kernel functions $k(x, x')$ without explicitly computing the feature mapping $\phi(x)$. A kernel function $k(x, x')$ can be used if the corresponding covariance matrix K is positive semi-definite (PSD). That is:

$$k(x, x') \text{ must satisfy } K \geq 0 \text{ for all valid inputs.}$$

Example 10.1.4 (Squared Exponential Kernel)**.** A commonly used kernel function is the squared exponential (RBF) kernel:

$$k(x, x') = \exp\left(-\frac{\|x - x'\|^2}{2\ell^2}\right),$$

where $\ell > 0$ is the length scale parameter. This kernel is guaranteed to produce a PSD covariance matrix K, making it valid for use in Gaussian Processes.

10.1.4 Conditioning on Data

Suppose we observe some data $D = \{X, y\}$ and want to predict $f(x_*)$ at a new input x_*. The predictive distribution is:

$$p(f(x_*)|X, y, x_*) = \mathcal{N}(\mu_*, \sigma_*^2),$$

where:

$$\mu_* = k(x_*, X)K^{-1}y,$$
$$\sigma_*^2 = k(x_*, x_*) - k(x_*, X)K^{-1}k(X, x_*).$$

where K is $k(X, X)$.

10.1.5 Noisy Data

When observations y are corrupted with Gaussian noise, the likelihood becomes:

$$y \sim \mathcal{N}(f(x), \sigma_n^2).$$

The covariance matrix is modified to include noise:

$$K_y = K + \sigma_n^2 I.$$

10.2 Infinite-width Deep Networks

10.2.1 Feed-forward Networks as Gaussian Processes

Consider a single-layer feed-forward neural network:

$$f(x) = a + \sum_{h=1}^{H} \omega_h g(b + v_h^\top x),$$

where:

- $a \sim \mathcal{N}(0, \sigma_a^2)$ is a scalar bias term,

- $\omega_h \sim \mathcal{N}(0, \sigma_\omega^2)$ are the weights for the hidden layer,
- $b \sim p(b)$ is a scalar bias for the activation function,
- $v_h \sim p(v)$ are the weights mapping the input $x \in \mathbb{R}^D$ to the hidden layer,
- $g(\cdot)$ is a non-linear activation function (e.g., `sigmoid`, `tanh`).

Theorem 10.2.1 (Moments of $f(x)$)**.**

- ***Mean of $f(x)$:***

$$\mathbb{E}[f(x)] = \mathbb{E}[a] + \sum_{h=1}^{H} \mathbb{E}[\omega_h]\mathbb{E}[g(b + v_h^\top x)] = 0,$$

since the biases and weights have zero mean.

- ***Covariance of $f(x)$:***

$$\mathbb{E}[f(x)f(x')] = \sigma_a^2 + \sum_{h=1}^{H} \sigma_\omega^2 \, \mathbb{E}_{p(b),p(v)}[g(b + v^\top x)g(b + v^\top x')],$$

where the covariance depends on the expectation of the activation functions g over the prior distributions of b and v.

10.2.2 Infinite-Width Limit and the Central Limit Theorem (CLT)

When the width $H \to \infty$, the function $f(x)$ converges to a Gaussian process due to the Central Limit Theorem (CLT). Specifically:

$$\text{Cov}(f(x), f(x')) = \sigma_a^2 + \sigma_\omega^2 \, \mathbb{E}_{p(b),p(v)}[g(b + v^\top x)g(b + v^\top x')].$$

Here: - The covariance kernel is determined by the activation function g and the prior distributions $p(b)$ and $p(v)$.

If g is bounded (e.g., `sigmoid`, `tanh`), the result holds.

10.2.3 More Layers: Deep Gaussian Processes

For a multi-layer network, the recursion for layer ℓ is:

$$\begin{aligned} z^{\ell-1}(x) &\sim \mathcal{GP}(0, k^{\ell-1}(x, x')), \\ h^{\ell-1} &= g(z^{\ell-1}), \\ z^\ell &= b^\ell + \sum_{j=1}^{H} \omega_j^\ell h_j^{\ell-1}, \quad \omega_j^\ell \sim \mathcal{N}(0, \frac{\sigma_\omega^2}{H}). \end{aligned}$$

The covariance of $z^\ell(x)$ and $z^\ell(x')$ is:

$$\mathbb{E}[z^\ell(x)z^\ell(x')] = \sigma_b^2 + \sigma_\omega^2 \, \mathbb{E}_{p(z^{\ell-1})}[g(z^{\ell-1}(x))g(z^{\ell-1}(x'))].$$

- Each layer transforms the Gaussian process into a new process, stacking covariance functions.

Proving that the resulting process is still a GP for deep networks requires further work.

10.2.4 Simplified Result for the GP Prior

Let $f_\theta(x)$ represent a neural network with parameters sampled from the prior $p(\theta)$. Then:

- Mean function:
$$m(x) = \mathbb{E}_{p(\theta)}[f_\theta(x)].$$

- Covariance function:
$$k(x, x') = \mathbb{E}_{p(\theta)}\left[(f_\theta(x) - m(x))(f_\theta(x') - m(x'))\right].$$

Using these, we can define a Gaussian Process:

$$f(x) \sim \mathcal{GP}(m(x), k(x, x')).$$

10.2.5 Interpretation of the Results

1. **Finite Width:** - For small widths H, the neural network does not exactly resemble a GP but may approximate one for large enough H.

2. **Infinite Width:** - As $H \to \infty$, the network behaves as a Gaussian process due to the aggregation of independent random variables (CLT).

This formulation bridges the connection between infinite-width neural networks and Gaussian Processes, where the kernel is defined implicitly by the architecture and activation function.

Active Learning

Definition 10.2.2 (Active Learning)**.** Active learning is a machine learning paradigm where the model iteratively selects the most informative data points to label, minimizing the number of labeled examples required to achieve high performance.

The goal is to determine which data points $x \in \mathcal{X}$ should be labeled next to maximize model improvement.

Entropy and Information Measures

Definition 10.2.3 (Entropy)**.** The entropy of a random variable X is defined as:

$$H(X) = -\sum_{x \in \mathcal{X}} p(x) \log p(x),$$

where $p(x)$ is the probability mass function of X. Entropy quantifies the uncertainty in the distribution of X.

Definition 10.2.4 (Mutual Information)**.** Mutual information between two random variables X and Y is given by:

$$I(X;Y) = H(X) - H(X|Y),$$

where $H(X|Y)$ is the conditional entropy:

$$H(X|Y) = -\sum_{x \in \mathcal{X}, y \in \mathcal{Y}} p(x,y) \log p(x|y).$$

Mutual information measures how much knowing Y reduces the uncertainty about X.

Query Selection Strategies

Definition 10.2.5 (Uncertainty Sampling)**.** The model queries the label for the example it is most uncertain about. For example, in classification tasks:

$$x^* = \arg\min_{x \in \mathcal{X}} \max_{y \in \mathcal{Y}} p(y|x),$$

where $p(y|x)$ is the predicted probability of class y for input x.

Definition 10.2.6 (Bayesian Active Learning by Disagreement (BALD))**.** BALD selects points that maximize the mutual information between the model's parameters and the output y:

$$x^* = \arg\max_{x \in \mathcal{X}} I(y; \theta | x, D),$$

where D is the current labeled dataset, and θ represents the model parameters.

Definition 10.2.7 (Bayesian Active Learning by Disagreement (BALD))**.** BALD selects data points that maximize the mutual information between the predicted output y and the model's parameters θ, given the current labeled dataset D. Formally, the acquisition function is:

$$x^* = \arg\max_{x \in \mathcal{X}} I(y; \theta \mid x, D),$$

where $I(y; \theta \mid x, D)$ is the mutual information.

10.2.6 Mathematics in BALD

The mutual information $I(y; \theta \mid x, D)$ is expressed as:

$$I(y; \theta \mid x, D) = H[y \mid x, D] - \mathbb{E}_{p(\theta|D)} \left[H[y \mid x, \theta] \right],$$

where:

- $H[y \mid x, D]$: Predictive entropy, which quantifies the uncertainty of the model about y after observing D,
- $\mathbb{E}_{p(\theta|D)} \left[H[y \mid x, \theta] \right]$: Expected entropy of y under the posterior distribution of the model parameters.

Detailed Formulation

1. **Predictive Distribution:** The predictive distribution is computed as:

$$p(y \mid x, D) = \int p(y \mid x, \theta) p(\theta \mid D) \, d\theta,$$

where $p(\theta \mid D)$ is the posterior over the model parameters given the data D.

2. **Predictive Entropy:** The predictive entropy is:

$$H[y \mid x, D] = -\sum_{y \in \mathcal{Y}} p(y \mid x, D) \log p(y \mid x, D).$$

3. **Expected Entropy:** The expected entropy of y under the parameter posterior is:

$$\mathbb{E}_{p(\theta|D)} \left[H[y \mid x, \theta] \right] = \int p(\theta \mid D) \left(-\sum_{y \in \mathcal{Y}} p(y \mid x, \theta) \log p(y \mid x, \theta) \right) d\theta.$$

Key Insight

BALD actively queries points where the model's disagreement is highest, which corresponds to maximizing the mutual information. This is effective in situations where uncertainty in the model parameters contributes significantly to the uncertainty in predictions.

Remark. *The mutual information $I(y; \theta \mid x, D)$ captures the reduction in uncertainty about the parameters θ after observing the label y for input x. This is why BALD effectively selects points that lead to the greatest model improvement.*

10.2.7 Practical Example: Bayesian Neural Networks

In a Bayesian neural network, the posterior predictive distribution can be approximated using Monte Carlo samples from the posterior:

$$p(y|x, D) \approx \frac{1}{T} \sum_{t=1}^{T} p(y|x, \theta_t),$$

where $\{\theta_t\}_{t=1}^{T}$ are samples from the posterior $p(\theta|D)$.

The BALD acquisition function becomes:

$$x^* = \arg\max_{x \in \mathcal{X}} \left(-\sum_{y \in \mathcal{Y}} \left(\frac{1}{T} \sum_{t=1}^{T} p(y|x, \theta_t) \right) \log \left(\frac{1}{T} \sum_{t=1}^{T} p(y|x, \theta_t) \right) \right) - \frac{1}{T} \sum_{t=1}^{T} \sum_{y \in \mathcal{Y}} p(y|x, \theta_t) \log p(y|x, \theta_t).$$

10.3 Bayesian Optimization

Objective

Definition 10.3.1 (Bayesian Optimization Problem)**.** Let $f(x)$ be an unknown objective function. The goal is to find the global optimal:

$$x^* = \arg\max_x f(x),$$

where evaluating $f(x)$ is expensive or impractical (e.g., hyperparameter tuning, oil drilling).

- Use a probabilistic model, $p(y \mid x, \mathcal{D}_n)$, as a surrogate for $f(x)$.
- Iteratively select new x to maximize the acquisition function $\alpha(x)$.

Procedure

1. Get an initial sample $\mathcal{D}_n = \{(x_i, y_i)\}_{i=1}^{n}$.

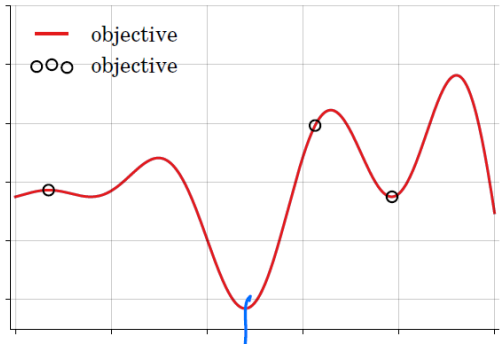


Figure 10.1: Initialize Sample and find lowest point.

2. Fit a probabilistic model to the data:

$$p(y \mid x, \mathcal{D}_n).$$

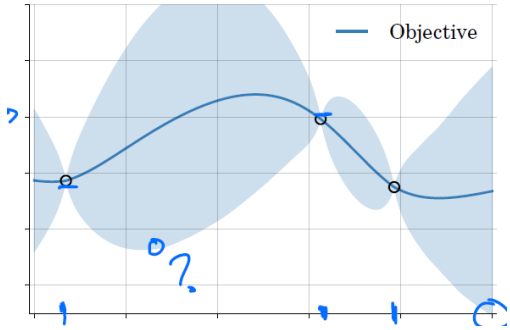


Figure 10.2:

3. Select the next point using the acquisition function:

$$\alpha(x) = \mathbb{E}_{p(y|x,\mathcal{D}_n)}[U(y \mid x, \mathcal{D}_n)],$$

where U is a utility function.

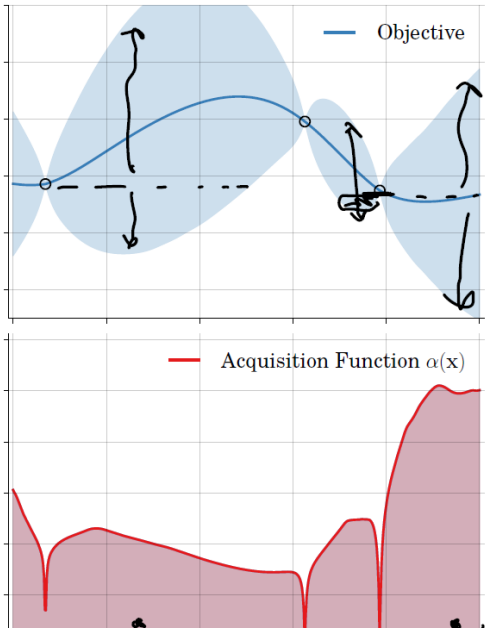


Figure 10.3: Select data collection strategy.

4. Optimize $\alpha(x)$ to determine the next x.

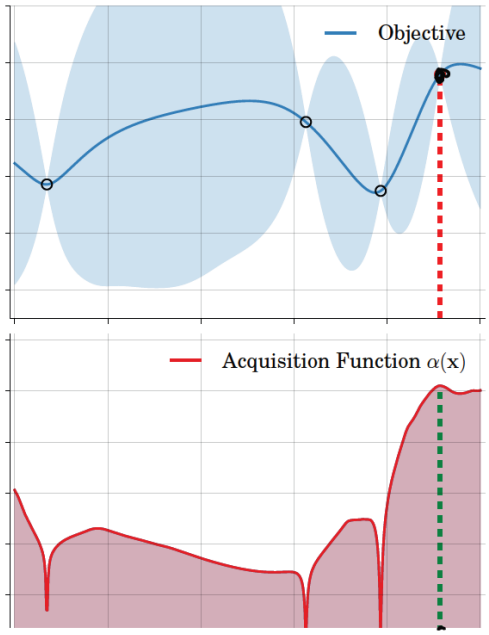


Figure 10.4: Optimize aquisition function.

5. Collect data $y = f(x)$ and update the model.

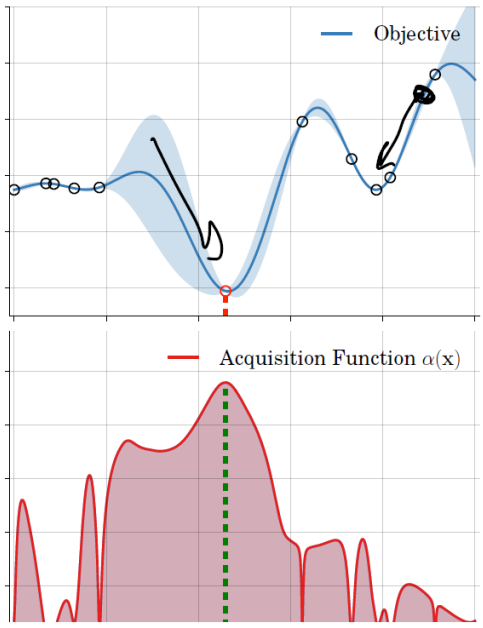


Figure 10.5: Repeat until convergence.

6. Repeat until convergence.

10.3.1 Acquisition Functions

Definition 10.3.2 (Expected Improvement (EI))**.** The expected improvement is defined as:

$$\alpha(x) = \mathbb{E}[\max(0, f(x) - f_{\text{best}})],$$

where f_{best} is the current best observation.

Definition 10.3.3 (Predictive Entropy Search (PES))**.** PES maximizes the expected reduction in the entropy of the posterior distribution of x^*:

$$\alpha(x) = H(f \mid \mathcal{D}_n) - \mathbb{E}_{p(y|x,\mathcal{D}_n)}[H(f \mid \mathcal{D}_n \cup \{(x, y)\})].$$

10.3.2 Surrogate Models

- **Gaussian Processes (GPs):** Popular due to their flexibility and closed-form expressions for posterior updates.

- **Bayesian Neural Networks (BNNs):** Suitable for high-dimensional settings but computationally expensive. However, Bayesian optimization struggle with scalability in high-dimensional feature spaces.

10.3.3 Key Challenges

- High-dimensional input spaces make modeling and optimization difficult.
- Expensive evaluations of $f(x)$ necessitate efficient sampling strategies.
- Balance between exploration (unknown regions) and exploitation (promising regions).