Deep Learning

AlejandroMllo

This document serves as a very brief survey of the topics covered in each chapter of the book Deep Learning [1].

Disclaimer: This document is completely extracted from [1], the author does not attribute any ownership over the material.

1 Introduction

- Deep Learning (DL) learns complicated concepts by building them out of simpler ones. DL is the study of models composed of either learned functions or learned concepts.
- Data Representation is key when finding patterns.
- Many AI tasks can be solved by designing the right set of features to extract.
- **Representation Learning:** discovers the mapping from representation to output and the representation itself.
  - Makes use of an *Autoencoder* which is a combination of:
    - Encoder: Converts the input data into a different representation.
    - Decoder: Converts the new representation back into the original format.
- Factors: sources of influence in the model.
- **Depth**, in a DL model, enables the computer to learn a multistep computer program (not all the information in a layer's activations necessarily encodes factors that explain the input). There are two ways to measure it:
  - Based on the number of sequential instructions, or
  - Based on the depth of the graph describing how concepts are related to each other (used by deep probabilistic models).
- DL is not an attempt to simulate the brain. It borrows ideas from different fields (one of which is neuroscience).

2 Linear Algebra

- Scalar: a single number. Denoted by lowercase variable names and italics.
- Vectors: ordered array of numbers. Denoted by lowercase and bold typeface. $\mathbf{x} = \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix}$
- Matrices: ordered 2-D $m \times n$ array of numbers. Denoted by uppercase and bold typeface.

$$\mathbf{A} = \begin{bmatrix} A_{1,1} & \dots & A_{1,n} \\ \vdots & \dots & \vdots \\ A_{1,m} & \dots & A_{m,n} \end{bmatrix}$$

- Tensors: array of numbers arranged on a regular grid with a variable number of axes.
- Transpose: mirror image of a matrix/vector. $(A^\top)_{i,j} = A_{j,i}$.
- It is possible to add $m \times n$ matrices together. $\mathbf{C} = \mathbf{A} + \mathbf{B}$, where $C_{i,j} = A_{i,j} + B_{i,j}$.
- To add a scalar to a matrix or multiply a matrix by a scalar, perform that operation on each element of the matrix.

- Matrix Multiplication: $\mathbf{C} = \mathbf{AB}$ is defined as:

$$C_{i,j} = \sum_k A_{i,k} B_{k,j}$$

  where $\mathbf{A}$ is of shape $m \times n$, $\mathbf{B}$ is of shape $n \times p$ and $\mathbf{C}$ is of shape $m \times p$.
  Matrix product has some useful properties not covered in detail in the book.
- System of linear equations: $\mathbf{Ax} = \mathbf{b}$, where $\mathbf{x}$ is a vector of unknown variables to be solved.
- Identity Matrix ($\boldsymbol{I}_n$): $n \times n$ matrix whose entries along the main diagonal are 1, while all the other entries are zero.
- Matrix Inverse ($\mathbf{A}^{-1}$): $\mathbf{A}^{-1}\mathbf{A} = \boldsymbol{I}_n$. For a matrix to have an inverse, it must be $n \times n$ and all its columns be linearly independent, it means that no column is a linear combination of another column.
- Norm (L^p): function that measures the size of vectors.

$$\|\mathbf{x}\|_p = \left(\sum_i |x_i|^p \right)^{\frac{1}{p}}$$

  where $p \in \mathbb{R}$, $p \geq 1$.
  - The squared L^2 norm is commonly used and can be calculated as $\mathbf{x}^\top \mathbf{x}$.
  - L^∞ norm (max norm): absolute value of the element with the largest magnitude in the vector.
- Frobenius Norm: used to measure the size of a matrix (analogous to the L^2 norm of a vector).

$$\|\mathbf{A}\|_F = \sqrt{\sum_{i,j} A_{i,j}^2}$$

- Symmetric Matrix: $\mathbf{A} = \mathbf{A}^\top$.
- Unit Vector: $\|\mathbf{x}\|_2 = 1$ (unit norm).
- Vectors $\mathbf{x}$ and $\mathbf{y}$ are orthogonal if $\mathbf{x}^\top \mathbf{y} = 0$. If both this vectors have unit norm, then they are called orthonormal.
- Orthogonal Matrix: square matrix whose rows are mutually orthonormal and whose columns are mutually orthonormal.
- An eigenvector of a square matrix $\mathbf{A}$ is a nonzero vector $\mathbf{v}$ such that multiplication by $\mathbf{A}$ alters only the scale of $\mathbf{v}$: $\mathbf{Av} = \lambda\mathbf{v}$. The scalar λ is known as the eigenvalue corresponding to this eigenvector. The eigendecomposition of $\mathbf{A}$ is given by: $\mathbf{A} = \mathbf{V}\text{diag}(\lambda)\mathbf{V}^{-1}$, where $\mathbf{V}$ is the matrix whose columns are each eigenvector of $\mathbf{A}$ and $\text{diag}(\lambda)$ is the diagonal matrix of eigenvalues such that the eigenvalue at $\lambda_{i,i}$ is the one associated with the eigenvector (column) i of $\mathbf{V}$. Eigendecomposition is not defined for every matrix.
- Singular Value Decomposition (SVD): $\mathbf{A} = \mathbf{U}\mathbf{D}\boldsymbol{V}^\top$, where $\mathbf{A}$ is a $m \times n$ matrix. $\mathbf{U}$ ($m \times m$) is composed by the eigenvectors of $\boldsymbol{A}\boldsymbol{A}^\top$; $\mathbf{V}$ ($n \times n$) is composed by the eigenvectors of $\boldsymbol{A}^\top\boldsymbol{A}$; and $\mathbf{D}$ ($m \times n$) is a diagonal matrix composed by the eigenvalues of $\boldsymbol{A}\boldsymbol{A}^\top$ or $\boldsymbol{A}^\top\boldsymbol{A}$.
- Moore-Penrose pseudoinverse: The pseudoinverse of $\mathbf{A}$ is defined as a matrix $\boldsymbol{A}^+ = \boldsymbol{V}\boldsymbol{D}^+\boldsymbol{U}^\top$, where $\mathbf{U}$, $\mathbf{D}$ and $\mathbf{V}$ are the SVD of $\mathbf{A}$, and the pseudoinverse $\boldsymbol{D}^+$ of a diagonal matrix $\mathbf{D}$ is obtained by taking the reciprocal of its nonzero elements then making the transpose of the resulting elements.
- The Trace operator gives the sum of the diagonal entries of a matrix: $\text{Tr}(\mathbf{A}) = \sum_i \boldsymbol{A}_{i,i}$. It has some useful identities to manipulate expressions.
- Determinant ($\det(\mathbf{A})$): function that maps matrices to real scalars. It is equal to the product of all the eigenvalues of a matrix.
- Principal Components Analysis (PCA): ML algorithm that can be derived using only knowledge of basic Linear Algebra. The reader can find a description of the implementation at section 2.12.

3 Probability and Information Theory

- Possible sources of uncertainty:
  1. Inherent stochasticity in the system being modeled.
  2. Incomplete observability.
  3. Incomplete modeling.

- Probability can be seen as the extension of logic to deal with uncertainty.
- Random Variable: variable x (discrete or continuous) that can take on different values randomly.
- Probability Distribution (PD): description of how likely a random variable or a set of random variables is to take on each of its possible states.
- Probability Mass Function (PMF): PD over discrete variables. A PMF, P, maps from a state of a random variable to the probability of that random variable taking on that state. A PMF acting on many variables at the same time is known as a joint probability distribution: $P(\mathrm{x} = x, \mathrm{y} = y)$.
- Probability Density Function (PDF): describes PD of continuous random variables. A PDF, $p(x)$, gives the probability of landing inside an infinitesimal region, not at a specific state.
- Marginal PD: it is the PD of a subset of a set of variables of which we know the PD.
- Conditional Probability: the probability of some event $\mathrm{x} = x$, given that some other event $\mathrm{y} = y$ has happened.

$$P(\mathtt{y} = y \mid \mathtt{x} = x) = \frac{P(\mathtt{y} = y\ ,\ \mathtt{x} = x)}{P(\mathtt{x} = x)},\ \textit{where}\ P(\mathtt{x} = x) > 0.$$

- Chain Rule of Conditional Probabilities: any joint PD over many random variables may be decomposed into conditional distributions over only one variable:

$$P(\mathtt{x}^{(1)}, \ldots, \mathtt{x}^{(n)}) = P(\mathtt{x}^{(1)}) \prod_{i=2}^{n} P(\mathtt{x}^{(i)} \mid \mathtt{x}^{(1)}, \ldots, \mathtt{x}^{(i-1)}).$$

- Two random variables x and y are independent ($\mathrm{x} \perp \mathrm{y}$) if:

$$\forall x \in \mathtt{x}, y \in \mathtt{y}, p(\mathtt{x} = x, \mathtt{y} = y) = p(\mathtt{x} = x)p(\mathtt{y} = y)$$

- Two random variables x and y are conditionally independent given a random variable z ($\mathrm{x} \perp \mathrm{y} | z$) if:

$$\forall x \in \mathtt{x}, y \in \mathtt{y}, z \in \mathtt{z}, p(\mathtt{x} = x, \mathtt{y} = y \mid \mathtt{z} = z) = p(\mathtt{x} = x \mid \mathtt{z} = z)p(\mathtt{y} = y \mid \mathtt{z} = z)$$

- The expectation ($\mathbb{E}_{x \sim P}[f(x)]$), or expected value, of some function $f(x)$ w.r.t. a PD $P(\mathrm{x})$ is the average, or mean value, that f takes on when x is drawn from P.

$$\mathbb{E}_{x \sim P}[f(x)] = \sum_{x} P(x)f(x),\ \textit{for discrete variables.}$$

$$\mathbb{E}_{x \sim p}[f(x)] = \int p(x)f(x)dx,\ \textit{for continuous variables.}$$

- The variance gives a measure of how much the values of a function of a random variable x vary as we sample different values of x from its PD. The square root of the variance is the standard deviation.

$$\mathtt{Var}(f(X)) = \mathbb{E}[(f(x) - \mathbb{E}[f(x)])^2]$$

- The covariance gives some sense of how much two values are linearly related to each other, as well as the scale of these variables. Two variables have zero covariance if they are not linearly dependent.

$$\mathtt{Cov}(f(X), g(y)) = \mathbb{E}[(f(x) - \mathbb{E}[f(x)])(g(y) - \mathbb{E}[g(y)])]$$

- The correlation normalizes the contribution of each variable in order to measure only how much the variables are related.
- The covariance matrix of a random vector $\mathbf{x} \in \mathbb{R}^n$ is an $n \times n$ matrix, such that $\mathrm{Cov}(\mathrm{x})_{i,j} = \mathrm{Cov}(\mathrm{x}_i, \mathrm{x}_j)$. The diagonal elements of the covariance give the variance: $\mathrm{Cov}(\mathrm{x}_i, \mathrm{x}_i) = \mathrm{Var}(\mathrm{x}_i)$.
- The Bernoulli Distribution is a distribution over a single binary random variable. It is controlled by a single parameter $\phi \in [0, 1]$, which gives the probability of the random variable being equal to 1. $P(\mathtt{x} = x) = \phi^x(1 - \phi)^{1-x}$, $\mathbb{E}_{\mathtt{x}}[\mathtt{x}] = \phi$, $\mathtt{Var}_{\mathtt{x}}(\mathtt{x}) = \phi(1 - \phi)$.
- The multinoulli, or categorical, distribution is a distribution over a single discrete variable with k different states, where k is finite. It is parametrized by a vector $\boldsymbol{p} \in [0, 1]^{k-1}$, where p_i gives the probability of the i-th state. The final, k-th state's probability is given by 1 - $1^\top \boldsymbol{p}$.
- Gaussian, normal, distribution: distribution over the real numbers. The parameters $\mu \in \mathbb{R}$ and $\sigma \in (0, \infty)$ control the normal distribution.

$$\mathcal{N}(x; \mu, \sigma^2) = \sqrt{\frac{1}{2\pi\sigma^2}} \exp\left(-\frac{1}{2\sigma^2}(x - \mu)^2\right).$$

To evaluate the PDF efficiently, parametrize the distribution with $\beta \in (0, \infty)$, to control the precision, or inverse variance:

$$\mathcal{N}(x; \mu, \beta^{-1}) = \sqrt{\frac{\beta}{2\pi}} \exp\left(-\frac{1}{2}\beta(x-\mu)^2\right).$$

Multivariate normal distribution: the normal distribution generalized to $\mathbb{R}^n$. It may be parametrized with a positive definite symmetric matrix $\boldsymbol{\Sigma}$:

$$\mathcal{N}(\boldsymbol{x}; \boldsymbol{\mu}, \boldsymbol{\Sigma}) = \sqrt{\frac{1}{(2\pi)^n \det(\boldsymbol{\Sigma})}} \exp\left(-\frac{1}{2}(\boldsymbol{x}-\boldsymbol{\mu})^\top \boldsymbol{\Sigma}^{-1}(\boldsymbol{x}-\boldsymbol{\mu})\right).$$

where $\boldsymbol{\mu}$ is a vector with the mean of the distribution, and $\boldsymbol{\Sigma}$ gives the covariance matrix.
Use a precision matrix $\boldsymbol{\beta}$, to evaluate several times for different values of the parameters:

$$\mathcal{N}(\boldsymbol{x}; \boldsymbol{\mu}, \boldsymbol{\beta}^{-1}) = \sqrt{\frac{\det(\boldsymbol{\beta})}{(2\pi)^n}} \exp\left(-\frac{1}{2}(\boldsymbol{x}-\boldsymbol{\mu})^\top \boldsymbol{\beta}(\boldsymbol{x}-\boldsymbol{\mu})\right).$$

- Exponential distribution: PD with a sharp point at $x = 0$: $p(x; \lambda) = \lambda \mathbf{1}_{x \geq 0} \exp(-\lambda x)$. It uses $\mathbf{1}_{x \geq 0}$ to assign probability zero to all negative values of x.
- Laplace distribution: PD that places a sharp peak of probability mass at an arbitrary point μ: $\text{Laplace}(x; \mu, \gamma) = \frac{1}{2\gamma} \exp\left(-\frac{|x-\mu|}{\gamma}\right)$.
- Dirac delta function: makes possible to specify that all the mass in a PD clusters around a point.
- Mixture distribution: PD made up of several component distributions.
- Latent variable: random variable that is not possible to observe directly.
- Logistic sigmoid:

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

- Softplus function: (smoothed version of $x^+ = \max(0, x)$)

$$\zeta(x) = \log(1 + \exp x)$$

- Bayes' Rule:

$$P(\mathtt{x}|\mathtt{y}) = \frac{P(\mathtt{x})P(\mathtt{y}|\mathtt{x})}{P(\mathtt{y})}$$

  It is possible to compute $P(\mathrm{y})$ as: $P(\mathtt{y}) = \sum_x P(\mathtt{y}|\mathtt{x})P(\mathtt{x})$
- Information theory: quantifies how much information is present in a signal.
  - Basic intuition: learning that an unlikely event has occurred is more informative than learning that a likely event has occurred. Likely events have low (or zero) information content. Less likely events have higher information content. Independents events have additive information content.
  - Self-information of an event $\mathrm{x} = x$: $I(x) = -\log P(x)$ $[nats]$. One nat is the amount of information gained by observing an event of probability $\frac{1}{e}$. (The book uses log as the natural logarithm).
  - Shannon entropy: quantifies the amount of uncertainty in an entire PD. $H(\mathtt{x}) = \mathbb{E}_{x \sim P}[I(x)]$.
  - To measure how different two distributions over the same random variable are, use the Kullback-Leibler (KL) divergence: $D_{KL}(P||Q) = \mathbb{E}_{x \sim P}[\log P(x) - \log Q(x)]$.
  - Cross-entropy: $H(P, Q) = -\mathbb{E}_{x \sim P} \log Q(x)$.
- A Structured probabilistic model, or graphical model, represents the factorization of a PD with a graph (represents the PD as factors). Each node in the graph is a random variable, and an edge connecting two random variables means that the PD is able to represent direct interaction between those two random variables.
  - Directed models represent factorization into conditional PD.

$$p(\mathtt{x}) = \prod_i p(\mathtt{x}_i | PaG(x_i)),$$

    where $\mathrm{PaG}(\mathrm{x}_i)$ represents the parents of x_i.
  - Undirected models represent factorizations into a set of functions. Any set of nodes connected to each other is called a clique. Each clique $C(i)$ is associated with a factor $\phi^{(i)}(C^{(i)})$.

$$p(\mathtt{x}) = \frac{1}{Z} \prod_i \phi^{(i)}(C^{(i)}),$$

    where Z is a normalizing constant defined to be the sum or integral over all states of the product of the ϕ functions.

4 Numerical Computation

- Algorithms that solve mathematical problems by methods that update estimates of the solution via an iterative process.
- Underflow: numbers near zero that are rounded to zero.
- Overflow: numbers with large magnitude are approximated as ∞ or -∞.
- Conditioning: how rapidly a function changes w.r.t. small changes in its inputs.
- Optimization: minimize or maximize an objective function $f(\boldsymbol{x})$ by altering $\boldsymbol{x}$. Sometimes, the value that minimizes or maximizes a function, is denoted with a superscript $*$: $\boldsymbol{x}^* = arg\,min\ f(\boldsymbol{x})$.
- Gradient Descent: reduce $f(x)$ by moving x in small steps with the opposite sign of the derivative. $f(x - \epsilon \texttt{sign}(f'(x))$.
- Critical point: point with zero slope.
- In the context of Deep Learning it is tried to find a value of f that is very low but not necessarily minimal in any formal sense.
- The partial derivative $\frac{\partial}{\partial x_i} f(\boldsymbol{x})$ measures how f changes as only the variable x_i increases at point $\boldsymbol{x}$. The gradient generalizes the notion of derivative to the case where the derivative is w.r.t. a vector: the gradient of f is the vector containing all the partial derivatives, denoted $\nabla_x f(\boldsymbol{x})$.
- The directional derivative in direction $\boldsymbol{u}$ (a unit vector) is the slope of the function f in direction u. The minimization occurs when the gradient points directly uphill, and the negative gradient points directly downhill.
- It is possible to decrease f by moving in the direction of the negative gradient. This is the method of steepest descent, or gradient descent: $\boldsymbol{x}' = \boldsymbol{x} - \epsilon \nabla_x f(\boldsymbol{x})$, where ϵ is the learning rate, a positive scalar determining the size of the step. This method converges when every element of the gradient is zero (or very close to zero). To jump directly to the critical point, solve: $\nabla_x f(\boldsymbol{x}) = 0$.
  Hill climbing is the generalization of this method for discrete spaces.
- If there is the function $\boldsymbol{f} : \mathbb{R}^m \to \mathbb{R}^n$, then the Jacobian matrix $\boldsymbol{J} \in \mathbb{R}^{n \times m}$ of $\boldsymbol{f}$ is defined such that $J_{i,j} = \frac{\partial}{\partial x_j} f(\boldsymbol{x})_i$.
- The Hessian matrix $\boldsymbol{H}(f)(\boldsymbol{x})$ is defined such that:

$$\boldsymbol{H}(f)(\boldsymbol{x})_{i,j} = \frac{\partial^2}{\partial x_i x_j} f(\boldsymbol{x}).$$

  Equivalently, the Hessian is the Jacobian of the gradient. Also, it is symmetric (if the function is continuous at such points). The second derivative in a specific direction represented by a unit vector $\boldsymbol{d}$ is given by $\boldsymbol{d}^\top \boldsymbol{H} \boldsymbol{d}$; when $\boldsymbol{d}$ is an eigenvector of $\boldsymbol{H}$, the second derivative in that direction is the corresponding eigenvalue.
  To the extent that the function to be minimized can be approximated well by a quadratic function, the eigenvalues of the Hessian thus determine the scale of the learning rate.
  Using the eigendecomposition of the Hessian matrix, it is possible to generalize the second derivative test to multiple dimensions.
- Newton's Method: uses a second-order Taylor series expansion to approximate $f(\boldsymbol{x})$ near some point $\boldsymbol{x}^{(0)}$.
- In Deep Learning, sometimes the functions used are restricted to those that are either Lipschitz continuous or have Lipschitz continuous derivatives. A Lipschitz continuous function is a function f whose rate of change is bounded by a Lipschitz constant $\mathcal{L} : \forall \boldsymbol{x}, \forall \boldsymbol{y}, |f(\boldsymbol{x}) - f(\boldsymbol{y})| \leq \mathcal{L} ||\boldsymbol{x} - \boldsymbol{y}||_2$.
- Constrained Optimization: used to find the maximal or minimal value of $f(\boldsymbol{x})$ for values of $\boldsymbol{x}$ in some set $\mathbb{S}$.

5 Machine Learning Basics

- ML Algorithm: algorithm that is able to learn from data.
- Learning is the means of attaining the ability to perform a task.
- Some common ML tasks: classification, classification with missing inputs, regression, transcription, machine translation, structured output, anomaly detection, synthesis and sampling, imputation of missing values, denoising, density estimation or probability mass function estimation.
- Performance measure: quantitative measure (specific to the task) of the abilities of a ML algorithm.
- Unsupervised learning algorithms experience a dataset containing many features, then learn useful properties of the structure of this dataset.

- Supervised learning algorithms experience a dataset containing features, but each example is also associated with a label or target.
- Reinforcement learning algorithms interact with an environment, so there is a feedback loop between the learning system and its experiences.
- Design matrix: matrix containing a different example in each row; each column corresponds to a different feature.
  The vector $\boldsymbol{y}$, provides the label y_i to example i.
- Parameters (weights): values that control the behavior of the system.
- Linear Regression example:
  - Task: to predict y from $\boldsymbol{x}$ by outputting $\hat{y} = \boldsymbol{w}^\top \boldsymbol{x}$.
  - Performance measure: mean square error of the model on the test set.

$$\texttt{MSE}_{\texttt{test}} = \frac{1}{m} \sum_i (\hat{\boldsymbol{y}}^{(\texttt{test})} - \boldsymbol{y}^{(\texttt{test})})_i^2$$

  - The objective is to design an algorithm that will improve the weights $\boldsymbol{w}$ in a way that reduces $\texttt{MSE}_{\texttt{test}}$ when the algorithm is allowed to gain experience by observing a training set $(\boldsymbol{X}^{(\texttt{train})}, \boldsymbol{y}^{(\texttt{train})})$. The way of doing this, it to minimize the MSE on the training set (solve for where its gradient is zero).
  - The idea is to minimize the training error, but measure the performance based on the test error.
  - The intercept term b of an affine function is often called the bias parameter.
- Generalization: the ability to perform well on previously unobserved inputs.
- Generalization/Test error: the expected value of the error on a new input.
- ML assumes that the datasets are independent and identically distributed.
- The factors determining how well a ML algorithm performs are its ability to make the training error small and make the gap between training and test error small.
- Underfitting: the model is not able to obtain a sufficiently low error value on the training set.
- Overfitting: the gap between the training error and test error is too large.
- Capacity: the model's ability to fit a wide variety of functions. One way to change it, is to change the number of input features it has and simultaneously add new parameters associated with those features.
- Hypothesis space: the set of functions that the learning algorithm is allowed to select as being the solution.
- Determine the capacity of a deep learning algorithm is difficult because the effective capacity is limited by the capabilities of the optimization algorithm.
- The no free lunch theorem for ML states that, averaged over all possible data-generating distributions, every classification algorithm has the same error rate when classifying previously unobserved points.
- The behavior of an algorithm is also affected by the specific identity of the functions in its hypothesis space.
- It is possible to regularize a model that learns a function $f(\boldsymbol{x}; \theta)$ by adding a penalty called a regularizer to the cost function. Regularization is any modification made to a learning algorithm that is intended to reduce its generalization error but not its training error.
- Expressing preferences for one function over another is a more general way of controlling a model's capacity than including or excluding members from the hypothesis space.
- Hyperparameters: settings used to control the algorithm's behavior.
- Point estimation: the attempt to provide the single 'best' prediction of some quantity of interest.
- Function estimation: approximatting f with a model or estimate $\hat{f}$.
- The bias of an estimator is defined as $\text{bias}(\hat{\boldsymbol{\theta}_m}) = \mathbb{E}(\hat{\boldsymbol{\theta}_m}) - \boldsymbol{\theta}$. It measures the expected deviation from the true value of the function or parameter.
- The variance of an estimator is simply the variance: $\text{Var}(\hat{\theta})$. It measures the deviation from the expected estimator value that any particular sampling of the data is likely to cause.
  The standard error, denoted $\text{SE}(\hat{\theta})$, is the square root of the variance.
- The generalization error is often estimated computing the sample mean of the error on the test set.
- Desirable estimators are those with small MSE and these are the estimators that manage to keep their bias and variance somewhat in check.

- Consistency: as the number of data points m in the dataset increases, the point estimates converge to the true value of the corresponding parameters. It ensures that the bias induced by the estimator diminishes as the number of data examples grows.
- Maximum Likelihood Estimation: minimizes the dissimilarity between the empirical distribution $\hat{p}_{data}$, defined by the training set and the model distribution, with the degree of dissimilarity between the two measured by the KL divergence. The KL divergence is given by

$$D_{KL}(\hat{p}_{data}||p_{model}) = \mathbb{E}_{x\sim\hat{p}_{data}}[\log \hat{p}_{data}(\boldsymbol{x}) - \log p_{model}(\boldsymbol{x})]$$

  When training the model, it is only needed to minimize $-\mathbb{E}_{x\sim\hat{p}_{data}}[\log \hat{p}_{data}(\boldsymbol{x})]$.
- Bayesian statistics: considers all possible values of $\boldsymbol{\theta}$ when making a prediction. It uses probability to reflect degrees of certainty in states of knowledge.
  - The knowledge of $\boldsymbol{\theta}$ is represented using the prior probability distribution, $p(\boldsymbol{\theta})$.
  - To recover the effect of data on what is believed about $\boldsymbol{\theta}$:

$$p(\boldsymbol{\theta}|x^{(1)}, \cdots, x^{(m)}) = \frac{p(x^{(1)}, \cdots, x^{(m)}|\boldsymbol{\theta})p(\boldsymbol{\theta})}{p(x^{(1)}, \cdots, x^{(m)})}$$

- Support Vector Machine (SVM): supervised learning algorithm that outputs a class identity.
- k-nearest neighbors: family of supervised learning techniques used for classification or regression.
- Principal Component Analysis learns a representation that has lower dimensionality than the original input. It also learns a representation whose elements have no linear correlation with each other. This is a first step toward the criterion of learning representations whose elements are statistically independent. To achieve full independence, a representation learning algorithm must also remove the nonlinear relationships between variables.
- Stochastic Gradient Descent (SGD): nearly all of deep learning is powered by this algorithm.
  - The insight of SGD is that the gradient is an expectation. It can be approximately estimated using a small set of samples.
  - Each step of the algorithm samples a minibatch of examples $\mathbb{B} = \{\boldsymbol{x}^{(1)}, \cdots, \boldsymbol{x}^{(m')}\}$ drawn uniformly from the training set.
  - The estimate of the gradient is formed as:

$$\boldsymbol{g} = \frac{1}{m'}\nabla_{\boldsymbol{\theta}} \sum_{i=1}^{m'} L(\boldsymbol{x}^{(i)}, y^{(i)}, \boldsymbol{\theta})$$

    using examples from the minibatch $\mathbb{B}$.
  - The algorithm then follows the estimated gradient downhill:

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \epsilon \boldsymbol{g}$$

    where ϵ is the learning rate.
- The recipe, of most, deep learning algorithm can be described as: combine a specification of a dataset, a cost function, an optimization procedure and a model.
- The Curse of Dimensionality: many ML algorithms become exceedingly difficult when the number of dimensions in the data is high. The number of possible distinct configuration of a set of variables increases exponentially as the number of variables increases.
- Smoothness prior or local constancy prior: this prior states that the learned function should not change very much within a small region.
- The core idea in deep learning is that is assumes that the data was generated by the composition of factors, or features, potentially at multiple levels in a hierarchy.
- Manifold: connected region. Mathematically, it is a set of points associated with a neighborhood around each point. From any given point, it locally appears to be a Euclidean space. - In ML it tends to be used to designate a connected set of points that can be approximated well by considering only a small number of degrees of freedom, or dimensions, embedded in a higher-dimensional space.
- Manifold learning algorithms assume that most of $\mathbb{R}^n$ consists of invalid inputs, and that interesting inputs occur only along a collection of manifolds containing a small subset of points, with interesting variations in the output of the learned function occurring only along directions that lie on the manifold, or with interesting variations happening only when moving from one manifold to another.

6 Deep Feedforward Networks

- They define a mapping $\boldsymbol{y} = f(\boldsymbol{x}; \boldsymbol{\theta})$ and learn the value of the parameters $\boldsymbol{\theta}$ that result in the best function approximation.
- There are no feedback connections in which outputs of the model are fed back into itself.
- The model is associated with a directed acyclic graph describing how the functions are composed together.
- Output layer: final layer (function) of the network.
- The training data provides noisy, approximate examples of $f^*(\boldsymbol{x})$ evaluated at different training points. Each example $\boldsymbol{x}$ is accompanied by a label $y \approx f^*(\boldsymbol{x})$, that specify what the output layer must do. The learning algorithm must decide how to use the hidden layers to best implement an approximation of f^* (the training data does not show a desired output for each of these layers).
- This networks require to initialize all weights to small random values.
- To apply gradient-based learning, a cost function and output representation must be chosen.
- The total cost of the network will often combine one of the primary cost functions with a regularization term.
- Most modern NN are trained using maximum likelihood.This means that the cost function is simply the negative log-likelihood:
$$J(\boldsymbol{\theta}) = -\mathbb{E}_{x,y\sim\hat{y}_{data}} \log p_{model}(\boldsymbol{y}|\boldsymbol{x})$$
The expansion of the above equation typically yields some terms that do not depend on the model parameters and may be discarded.
- The equivalence between maximum likelihood estimation and minimization of MSE holds regardless of the $f(\boldsymbol{x}; \theta)$ used to predict the mean of the Gaussian.
- The gradient of the cost function must be large and predictable enough to serve as a good guide for the learning algorithm.
- It is possible to view the cost function as a functional. A functional maps from functions to real numbers. It can have its minimum at some specified function.
- The choice of cost function is tightly coupled with the choice of output unit.
- Linear units for Gaussian distributions. The layer produces the mean of a conditional Gaussian distribution: $p(\boldsymbol{y}|\boldsymbol{x}) = \mathcal{N}(\boldsymbol{y}; \hat{\boldsymbol{y}}, \boldsymbol{I})$.
- Sigmoid units for Bernoulli distributions. Useful for predicting the value of a binary variable y.
- Logit: variable (usually denoted by z) defining a probability distribution based on exponentiation and normalization over binary variables.
- Softmax units for Multinoulli distributions. Useful to represent a PD over a discrete variable with n possible values.
- The objective functions that do not use a log to undo the exp of the softmax fail to learn when the argument to the exp becomes very negative.
- In general, NN represent a function $f(\boldsymbol{x}|\boldsymbol{y})$. The outputs of $f(\boldsymbol{x}|\boldsymbol{y}) = \boldsymbol{\omega}$ provide the parameters for a distribution over y. The loss function can then be interpreted as $-\log p(\mathrm{y}, \boldsymbol{\omega}(\boldsymbol{x}))$.
- Gaussian mixture: lets predict real values from a conditional distribution $p(\boldsymbol{y}|\boldsymbol{x})$ that can have several different peaks in $\boldsymbol{y}$ space for the same value of $\boldsymbol{x}$.
- The function used in the context of NN usually have defined left derivatives and defined right derivatives.
- Most hidden units can be described as accepting a vector of inputs $\boldsymbol{x}$, computing an affine transformation $\boldsymbol{z} = \boldsymbol{W}^\top \boldsymbol{x} + \boldsymbol{b}$, and then applying an element-wise nonlinear function $g(\boldsymbol{z})$.
- ReLU are an excellent default choice of hidden unit. They use the activation function $g(\boldsymbol{z}) = \max\{0, z\}$. They cannot learn via gradient-based methods on examples for which their activation is zero.
  Three generalizations of ReLU are based on using a nonzero slope α_i when $z_i < 0 : h_i = g(\boldsymbol{z}, \boldsymbol{\alpha})_i = \max(0, z_i) + \alpha_i \min(0, z_i)$.
  - Absolute value error rectification fixes $\alpha_i = -1$ to obtain $g(z) = |z|$.
  - Leaky ReLU fixes α_i to a small value like 0.01.
  - Parametric ReLU (PReLU) treats α_i as a learnable parameter.

- Maxout units divide $\boldsymbol{z}$ into groups of k values. Each maxout unit then outputs the maximum element of one of these groups: $g(\boldsymbol{z})_i = \max_{j \in \mathbb{G}^{(i)}} z_j$, where $\mathbb{G}^{(i)}$ is the set of indices into the inputs for group i, $\{(i-1)k+1, \cdots, ik\}$. They can resist a phenomenon called catastrophic forgetting, in which NN forget how to perform tasks that they were trained on in the past.
- Logistic Sigmoid activation function: $g(z) = \sigma(z)$. Sigmoidal units saturate across most of their domain. Their use as output units is compatible with the use of gradient-based learning when an appropriate cost function can undo the saturation of the sigmoid in the output layer.
- Hyperbolic Tangent activation function: $g(z) = \tanh(z) = 2\sigma(2z) - 1$. It resembles the identity function more closely, in the sense that $\tanh(0) = 0$ while $\sigma(0) = \frac{1}{2}$.
- It is acceptable for **some** layers of the NN to be purely linear.
- Radial basis function (RBF) unit: $h_i = \exp\left(-\frac{1}{\sigma_i^2}||\boldsymbol{W}_{:,i} - \boldsymbol{x})||^2\right)$. It becomes more active as $\boldsymbol{x}$ approaches a template $\boldsymbol{W}_{:,i}$. Because it saturates to 0 for most $\boldsymbol{x}$, it can be difficult to optimize.
- Softplus units: $g(a) = \zeta(a) = \log(1 + \mathrm{e}^a)$. Its use is generally discouraged.
- Hard tanh unit: $g(a) = \max(-1, \min(1, a))$.
- Architecture of the network: how many units it should have and how these units should be connected to each other.
- Most NN are organized into groups of units called layers. Most NN architectures arrange these layers in a chain structure, with each layer being a function of the layer that preceded it. In these architectures the main considerations are choosing the depth of the network and the width of each layer. Deeper networks generalize better (most of the time).
- Universal Approximation theorem: a feedforward NN with a linear output layer and at least one hidden layer with any "squashing" activation function can approximate any Borel measurable function from one finite-dimensional space to another with any desired nonzero amount of error, provided that the network is given enough hidden units. The derivatives of the feedforward network can also approximate the derivatives of the function arbitrarily well.
- Many architectures build a main layers chain but then add extra architectural features to it, such as skip connections going from layer i to layer $i+2$ or higher. These skip connections make it easier for the gradient to flow from output layers to layers nearer the input.
- During training, forward propagation can continue onward until it produces a scalar cost $J(\boldsymbol{\theta})$. The back-propagation algorithm allows the information from the cost to then flow backward through the network in order to compute the gradient.
- Back-propagation is the method for computing the gradient, while another algorithm, such as stochastic gradient descent, is used to perform learning using this gradient. In learning algorithm, the gradient that is most often required is the gradient of the cost function w.r.t. the parameters: $\boldsymbol{\nabla}_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$.
- The chain rule of calculus is used to compute the derivatives of functions formed by composing other functions whose derivatives are known. Backprop computes the chain rule, with a specific order of operations that is highly efficient.
  Suppose that $\boldsymbol{x} \in \mathbb{R}^m, \boldsymbol{y} \in \mathbb{R}^n, g$ maps from $\mathbb{R}^m$ to $\mathbb{R}^n$, and f maps from $\mathbb{R}^n$ to $\mathbb{R}$. If $\boldsymbol{y} = g(\boldsymbol{x})$ and $z = f(\boldsymbol{y})$, then
  $$\frac{\partial z}{\partial x_i} = \sum_i \frac{\partial z}{\partial y_j} \frac{\partial y_j}{\partial x_i}$$
  In vector notation, this may be equivalently written as
  $$\nabla_{\boldsymbol{x}} z = \left(\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}\right)^\top \nabla_{\boldsymbol{y}} z,$$
  where $\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}$ is the $n \times m$ Jacobian matrix of g.
  So, the gradient of a variable $\boldsymbol{x}$ can be obtained by multiplying a Jacobian matrix $\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}$ by a gradient $\nabla_{\boldsymbol{y}} z$. The backprop algorithm consists of performing such a Jacobian-gradient product for each operation in the graph.
- Algorithms 6.1, 6.2, 6.3, 6.4, 6.5 in the book further enhance understanding of backprop.

7 Regularization for Deep Learning

- Regularization: strategies designed to reduce the test error, possibly at the expense of increased training error. Its goal is to make a model match the true data-generating processes. An effective regularizer is one that makes a profitable trade, reducing variance significantly while not overly increasing the bias.

- Many regularization approaches are based on limiting the capacity of the models by adding a parameter norm penalty $\Omega(\boldsymbol{\theta})$ to the objective function J. The regularized objective function is:

$$\tilde{J}(\boldsymbol{\theta}; \boldsymbol{X}, \boldsymbol{y}) = J(\boldsymbol{\theta}; \boldsymbol{X}, \boldsymbol{y}) + \alpha\Omega(\boldsymbol{\theta})$$

  where $\alpha \in [0, \infty)$ is a hyperparameter that weights the relative contribution of the norm penalty term, Ω, relative to the standard objective function J.
- For NN is typically used a parameter norm penalty Ω that penalizes only the weights of the affine transformation at each layer and leaves the biases unregularized.
- L^2 Parameter regularization (weight decay): this strategy drives the weight closer to the origin by adding the regularization term $\Omega(\boldsymbol{\theta}) = \frac{1}{2}||\boldsymbol{w}||_2^2$ to the objective function.
- L^1 regularization: $\Omega(\boldsymbol{\theta}) = ||\boldsymbol{w}||_1 = \sum_i |w_i|$. It controls the strength of the regularization by scaling the penalty Ω using a positive hyperparameter α. The regularization contribution to the gradient does not scale linearly as in L^2. Its solution tends to be more sparse than L^2's solution.
- It is possible to think of a parameter norm penalty as imposing a constraint on the weights.
- It is possible to use explicit constraints rather than penalties.
- Weight decay will cause gradient descent to quit increasing the magnitude of the weights when the slope of the likelihood is equal to the weight decay coefficient.
- Data set augmentation: generate new $(\boldsymbol{x}, y)$ pairs by transforming the $\boldsymbol{x}$ inputs in the training set.
- NN prove not to be very robust to noise. One way to improve the robustness of NN is simply to train them with random noise applied to their inputs. This is also a form of data augmentation.
- When comparing ML benchmark results, taking the effect of dataset augmentation into account is important.
- Noise applied to the weights can be interpreted as equivalent to a more traditional form of regularization, encouraging stability of the function to be learned.
- Most datasets have some number of mistakes in the y labels. It can be harmful to maximize $\log p(y|\boldsymbol{x})$ when y is a mistake. One way to prevent this is to explicitly model the noise on the labels. For example, assume that for some small constant ϵ, the training set label y is correct with probability $1 - \epsilon$, and otherwise any of the other possible labels might be correct.
- In the paradigm of semi-supervised learning, both unlabeled examples from $P(\mathbf{x})$ and labeled examples from $P(\mathbf{x}, \mathbf{y})$ are used to estimate $P(\mathbf{y} \mid \mathbf{x})$ or predict $\mathbf{y}$ from $\mathbf{x}$. In deep learning, semi-supervised learning refers to learning a representation $\boldsymbol{h} = f(\boldsymbol{x})$. The goal is to learn a representation so that examples from the same class have similar representations.
- Multitask learning is a way to improve generalization by pooling the examples (which can be seen as soft constraints imposed on the parameters) arising out of several tasks.
- Among the factors that explain the variations observed in the data associated with different tasks, some are shared across two or more tasks.
- Epoch: a training iteration over the dataset.
- Early stopping: every time the error on the validation set improves, a copy of the model parameters is stored. When the training algorithm terminates, it returns these parameters, rather than the latest parameters. The algorithms terminates when no parameters have improved over the best recorded validation error for some pre-specified number of iterations. See algorithm 7.1 on the book.
- Parameter sharing: regularization method to force sets of parameters to be equal.
- Any model that has hidden units can be made sparse.
- Bootstrap aggregating (bagging) is a technique for reducing generalization error by training several different models separately, than have all models vote on the output for test examples. This is an example of a general strategy in ML called model averaging. Techniques employing this strategy are known as ensemble methods.
- Dropout trains an ensemble consisting of all subnetworks that can be constructed by removing nonoutput units from an underlying base network. When extremely few labeled training examples are available, dropout is less effective.

8 Optimization for Training Deep Models

- The focus is finding the parameters $\boldsymbol{\theta}$ of a NN that significantly reduce a cost function $J(\boldsymbol{\theta})$.
- Most ML scenarios care about some performance measure P, that is defined w.r.t. the test set and may also be intractable. Then it reduces a different cos function $J(\boldsymbol{\theta})$ in the hope that doing so will improve P.

- Typically, the cost function can be written as an average over the training set, such as

$$J(\boldsymbol{\theta}) = \mathbb{E}_{(\boldsymbol{x},y)\sim\hat{p}_{data}} L(f(\boldsymbol{x};\boldsymbol{\theta}), y),$$

  where L is the per-example loss function, $f(\boldsymbol{x};\boldsymbol{\theta})$ is the predicted output when the input is $\boldsymbol{x}$, and $\hat{p}_{data}$ is the empirical distribution.
  It is preferred to minimize the corresponding objective function where the expectation is taken across the $data-generating\ distribution\ p_{data}$ rather than just over the finite training set:

$$J^*(\boldsymbol{\theta}) = \mathbb{E}_{(\boldsymbol{x},y)\sim p_{data}} L(f(\boldsymbol{x};\boldsymbol{\theta}), y),$$

- When the true distribution $p_{data}(\boldsymbol{x}, y)$ is not known, and only is available a training set of samples, then you have a ML problem. To convert a ML problem back into an optimization problem is to minimize the expected loss on the training set, Empirical risk:

$$\mathbb{E}_{\boldsymbol{x},t\sim\hat{p}_{data}}[L(f(\boldsymbol{x};\boldsymbol{\theta}))] = \frac{1}{m}\sum_{i=1}^{m} L(f(\boldsymbol{x}^{(i)};\boldsymbol{\theta}), y^{(i)}).$$

- Optimization algorithms for ML typically compute each update to the parameters based on an expected value of the cost function estimated using only a subset of the terms of the full cost function.
- Optimization algorithms that use the entire training set are called batch or deterministic gradient methods; those that use a single example at a time are sometimes called stochastic or online methods. Minibatch or minibatch stochastic methods use more than one but fewer than all the training examples; this are commonly used in deep learning.
- Generalization error is often best for a batch size of 1 (this might require a small learning rate).
- Minibatch stochastic gradient descent follows the gradient of the true generalization error as long as no examples are repeated.
- To obtain an unbiased estimator of the exact gradient of the generalization error, sample a minibatch of examples $\boldsymbol{x}^{(1)}, \cdots, \boldsymbol{x}^{(i)}$ with corresponding targets $y^{(i)}$ from the data-generating distribution p_{data}, then computing the gradient of the loss w.r.t. the parameters for that minibatch:

$$\hat{\boldsymbol{g}} = \frac{1}{m}\nabla_{\boldsymbol{\theta}} \sum_{i} L(f(\boldsymbol{x}^{(i)};\boldsymbol{\theta}), y^{(i)}).$$

  Updating $\boldsymbol{\theta}$ in the direction of $\hat{\boldsymbol{g}}$ performs SGD on the generalization error.
- When optimizing a convex function, a good solution is reached if a critical point of any kind is found.
- Local minima can be problematic if they have high cost in comparison to the global minimum. A test can rule out local minima as the problem is plotting the norm of the gradient over time. If the norm of the gradient does not shrink to insignificant size, the problem is neither local minima nor any other kind of critical point.
- Gradient clipping heuristic: when the traditional gradient descent algorithm proposes making a very large step (it is on a cliff), this heuristic intervenes to reduce the step size, making it less likely to go outside the region where the gradient indicates the direction of approximately steepest descent.
- Some optimization problems might be avoided if there exists a region of space connected reasonably directly to a solution by a path that local descent can follow, and if it is possible to initialize learning within that well-behaved region.
- A sufficient condition to guarantee convergence of SGD is that $\sum_{k=1}^{\infty} \epsilon_k = \infty\ and\ \sum_{k=1}^{\infty} \epsilon_k^2 < \infty$, where ϵ_k is the learning rate at iteration k. In practice, it is common to decay the learning rate linearly until iteration τ: $\epsilon_k = (1-\alpha)\epsilon_0 + \alpha\epsilon_\tau$.
- Usually τ may be set to the number of iterations required to make a few hundred passes through the training set, and ϵ_τ should be set to roughly 1 percent the value of ϵ_0.
- For a large enough dataset, SGD may converge to within some fixed tolerance of its final test set error before it has processed the entire training set.
- To study the convergence rate of an optimization algorithm it is common to measure the excess error $J(\boldsymbol{\theta}) - \min_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$, which is the amount by which the current cost function exceeds the minimum possible cost.
- Momentum: this optimization method accelerates learning, especially in the face of high curvature, small but consistent gradients, or noisy gradients. It accumulates an exponentially decaying moving average of past gradients and continues to move in their direction.
  - It introduces a variable $\boldsymbol{v}$ that gives the direction and speed at which the parameters move through parameter space. It is set to an exponentially decaying average of the negative gradient.

  - A hyperparameter $\alpha \in [0, 1)$ determines how quickly the contributions of previous gradients exponentially decay.

$$\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} - \epsilon \nabla_{\boldsymbol{\theta}} \left(\frac{1}{m} \sum_{i=1}^{m} L(\boldsymbol{f}(\boldsymbol{x}^{(i)}, \boldsymbol{\theta}), \boldsymbol{y}^{(i)}) \right)$$
$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}$$

  - See algorithm 8.2 for an example of SGD with momentum.

- Nesterov Momentum: similar to momentum, but here the gradient is evaluated after the current velocity is applied. See algorithm 8.3.
- Training algorithms for deep learning models are usually iteratively and thus require the user to specify some initial point from which to begin the iterations. Some initial points may be beneficial from the viewpoint of optimization but detrimental from the viewpoint of generalization.
- The initial parameters need to "break symmetry" between different units. If two hidden units with the same activation function are connected to the same inputs, then these units must have different initial parameters.
- Typically, the biases for each unit are set to heuristically chosen constants, and the weights are initialized randomly.
- AdaGrad: this algorithm individually adapts the learning rates of all model parameters by scaling them inversely proportional to the square root of the sum of all the historical squared values of the gradient. The parameters with the largest partial derivative of the loss have a correspondingly rapid decrease in their learning rate, while parameters with small partial derivatives have a relatively small decrease in their learning rate. See algorithm 8.4.
- RMSProp: this algorithm modifies AdaGrad to perform better in the nonconvex setting by changing the gradient accumulation function into an exponentially weighted moving average. It uses an exponentially decaying average to discard history from the extreme past so that it can converge rapidly after finding a convex bowl, as if it were an instance of AdaGrad initialized within that bowl. See algorithm 8.5.
- Adam: see algorithm 8.7. Here, first, momentum is incorporated directly as an estimate of the first-order moment (with exponential weighting) of the gradient. Second, it includes bias corrections to the estimates of both the first order moments (the momentum term) and the (uncentered) second-order moments to account for their initialization at the origin.
- Newton's method is an optimization scheme based on using a second-order Taylor series expansion to approximate $J(\boldsymbol{\theta})$ near some point $\boldsymbol{\theta}_0$, ignoring derivatives of higher order. See algorithm 8.8.
- Conjugate gradients is a method to efficiently avoid the calculation of the inverse Hessian by iteratively descending conjugate directions (that is, directions that will not undo progress made in the previous direction). See algorithm 8.9.
- The nonlinear conjugate gradients algorithm includes occasional resets where the method of conjugate gradients is restarted with line search along the unaltered gradient. This is used when it is not known if the objective is quadratic.
- The Broyden-Fletcher-Goldfarb-Shanno (BFGS) algorithm attempts to bring some of the advantages of Newton's method without the computational burden. There is also a limited memory version.
- Batch normalization provides an elegant way of reparametrizing almost any deep network. Let $\boldsymbol{H}$ be a minibatch of activations of the layer to normalize, arranged as a design matrix, with the activations for each example appearing in a row of the matrix. To normalize $\boldsymbol{H}$, replace it with $\boldsymbol{H}' = \frac{\boldsymbol{H}-\boldsymbol{\mu}}{\boldsymbol{\sigma}}$, where $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}$ are vectors containing the mean and standard deviation of each unit, respectively.
- If $f(\boldsymbol{x})$ is minimized w.r.t. a single variable x_i, then minimized w.r.t. another variable x_j, and so on, repeatedly cycling through all variables, at some moment it will arrive at a (local) minimum. This is known as coordinate descent. Block coordinate descent refers to minimizing w.r.t. a subset of the variables simultaneously.
- Polyak averaging consists of averaging several points in the trajectory through parameter space visited by an optimization algorithm. If t operations of gradient descent visit points $\boldsymbol{\theta}^{(1)}, \cdots, \boldsymbol{\theta}^{(t)}$, then the output of the algorithm is $\hat{\boldsymbol{\theta}}^{(t)} = \frac{1}{t} \sum_i \boldsymbol{\theta}^{(i)}$.
- Pretraining: training a simple model on simple tasks before confronting the challenge of training the desired model to perform the desired task.
- Greedy supervised pretraining: pretraining algorithms that break supervised learning problems into other simpler supervised learning problems.
- Many improvements in the optimization of deep models have come from designing the models to be easier to optimize.

- Modern NN have been designed so that their local gradient information corresponds reasonably well to moving toward a distant solution.
- Continuation methods are a family of strategies that can make optimization easier by choosing initial points to ensure that local optimization spends most of its time in well-behaved regions of space. The idea behind them is to construct a series of objective functions over the same parameters. To minimize a cost function $J(\boldsymbol{\theta})$, new cost function $J^{(0)}, \cdots, J^{(n)}$ are constructed (they are designed to be increasingly difficult to minimize).
- Curriculum learning: planning a learning process to begin by learning simple concepts and progress to learning more complex concepts that depend on these simpler concepts.

9 Convolutional Networks

- CNNs are a specialized kind of NN for processing data that has a known grid-like topology. They are simply NN that use convolution (a specialized kind of liner operation) in place of general matrix multiplication in at least one of their layers.
- In its more general form, convolution is an operation on two functions of a real valued argument. The first argument (the first function) to the convolution is often referred to as the input, and the second argument (the second function) as the kernel. The output is sometimes referred to as the feature map. A convolution over functions x and w is denoted with an asterisk: $s(t) = (x * w)(t)$.
- In ML applications, the input is usually a multidimensional array of data, and the kernel is usually a multidimensional array of parameters that are adapted by the learning algorithm. These multidimensional arrays are known as tensors.
- Convolutions can be used over more than one axis at a time. Also they are commutative. Ex: if using a two-dimensional image I as input, then is probable that a two-dimensional kernel K is wanted: $S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(m,n)K(i-m, j-n)$.
- Many NN libraries implement a related function call the cross-correlation, which is the same as convolution but without flipping the kernel: $S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(i+m, j+n)K(m,n)$.
- Convolution enables sparse interactions, parameter sharing and equivariant representations. They also provide a means for working with inputs of variable size.
- Receptive field: the output units in a layer that are input to a unit in other layer.
- Sparse interactions/connectivity/weights: every output unit does not interact with every input unit.
- Parameter sharing: using the same parameter for more than one function in a model. It means that the network has tied weights.
- Equivariance: a function is equivariant if, in the case the input changes, the output changes in the same way. The function f is equivariant to g if: $f(g(x)) = g(f(x))$.
- Stages of a convolutional network's layers:
  1. The layer performs several convolutions in parallel to produce a set of linear activations.
  2. (Detector stage) Each linear activation is run through a nonlinear activation function.
  3. A pooling function is used to modify the output of the layer further.
- A pooling function replaces the output of the net at a certain location with a summary statistic of the nearby outputs. It helps to make the representation approximately invariant to small translations of the input. Its use can be viewed as adding an infinitely strong prior that the function the layer learns must be invariant to small translations.
- Prior probability distribution: PD over the parameters of a model that encodes the beliefs about what models are reasonable, before any data is seen. A weak prior is a prior distribution with high entropy. A strong prior has very low entropy. An infinitely strong prior places zero probability on some parameters and says that these parameter values are completely forbidden, regardless of how much support the data give to those values.
- It is possible to think of the use of convolution as introducing an infinitely strong prior PD over the parameters of a layer.
- Convolution and pooling can cause underfitting.
- Convolution in the context of NN actually means an operation that consists of many applications of convolution in parallel. The objective is that each layer of the network extracts many kinds of features, at many locations.
- Because convolutional networks usually use multichannel convolution, the linear operations they are based on are commutative only if each operation has the same number of output channels as input channels.

- Assume we have a 4-D kernel tensor $\mathbf{K}$ with element $K_{i,j,k,l}$ giving the connection strength between a unit in channel i of the output and a unit in channel j of the input, with an offset of k rows and l columns between the output unit and the input unit. Assume our input consists of observed data $\mathbf{V}$ with element $V_{i,j,k}$ giving the value of the input unit within channel i at row j and column k. Assume our output consists of $\mathbf{Z}$ with the same format as $\mathbf{V}$. If $\mathbf{Z}$ is produced by convolving $\mathbf{K}$ across $\mathbf{V}$ without flipping $\mathbf{K}$, then
$$Z_{i,j,k} = \sum_{l,m,n} V_{l,j+m-1,k+n-1} K_{i,l,m,n},$$
where the summation over l, m and n is over all values for which the tensor indexing operations inside the summation are valid.
- If we want to sample only every s (stride) pixels in each direction in the output, the we can define a downsampled convolution function c such that
$$Z_{i,j,k} = c(\boldsymbol{K}, \boldsymbol{V}, s)_{i,j,k} = \sum_{l,m,n} [V_{l,(j-1)\times s+m,(k-1)\times s+n} K_{i,l,m,n}].$$
- One essential feature of any convolutional network implementation is the ability to implicitly zero pad the input $\mathbf{V}$ to make it wider. Without this feature, the width of the representation shrinks by one pixel less than the kernel width at each layer.
- Unshared convolution: in some cases convolutions are not used, and instead locally connected layers are used. In this case, the adjacency matrix in the graph of the MLP is the same, but every connection has its own weight, specified by a 6-D tensor $\mathbf{W}$. The indices into $\mathbf{W}$ are respectively: i, the output channel; j, the output row; k, the output column; l, the input channel; m, the row offset within the input; and n, the column offset withing the input. The linear part of a locally connected layer is then given by
$$Z_{i,j,k} = \sum_{l,m,n} [V_{l,j+m-1,k+m-1} w_{i,j,k,l,m,n}]$$
This is useful when is known that each feature should be a function of a small part of space, but there is no reason to think that the same feature should occur across all of space.
- Tiled convolution: rather than learning a separate set of weights at every spatial location, we learn a set of kernels that we rotate through as we move through space.
- To perform learning in a CNN, it must be possible to compute the gradient w.r.t. the kernel, given the gradient w.r.t. the outputs.
- Convolution, backprop from output to weights, and backprop from output to inputs are the operations sufficient to compute all the gradients needed to train any depth of feedforward convolutional network, as well as to train convolutional networks with reconstruction functions based on the transpose of convolution.
- CNNs can be used to output a high-dimensional structured object. Typically this object is just a tensor.
- The data used with a CNN usually consists of several channels, each channel being the observation of a different quantity at some point in space or time.
- When the inputs are of different size (only because they contain varying amounts of observations of the same kind of things) convolution is straightforward to apply; the kernel is simply applied a different number of times depending on the size of the input, and the output of the convolution operation scales accordingly.
- Convolution is equivalent to converting both the input and the kernel to the frequency domain using a Fourier transform, performing point-wise multiplication of the two signals, and converting back to the time domain using an inverse Fourier transform.
- When a d-dimensional kernel can be expressed as the outer product of d vectors, one vector per dimension, the kernel is called separable. When the kernel is separable, naive convolution is equivalent to compose d one-dimensional convolutions with each of these vectors. The composed approach is significantly faster.
- The most expensive part of CNN training is learning the features. When performing supervised training with gradient descent, every gradient step requires a complete run of forward propagation and backward propagation through the entire network.
- To obtain convolution kernels without supervised learning, simply initialize them randomly, design them by hand or learn them with an unsupervised criterion.

10 Sequence Modeling: Recurrent and Recursive Nets

- Recurrent neural networks (RNNs) are a family of NN for processing sequential data. They can scale to much longer sequences than unspecialized NN and can also process sequences of variable length.
- RNNs share parameters across different parts of a model to extend and apply the model to examples of different forms and generalize across them. - Each member of the output is produced using the same update rule applied to previous outputs.
- RNNs operate on sequences that contain vectors $\boldsymbol{x}^{(t)}$ with the time step index t ranging from 1 to τ. In practice, they usually operate on minibatches of such sequences, with a different sequence length τ for each member of the minibatch.
- A computational graph is a way to formalize the structure of a set of computations.
- The classical form of a dynamical system $\boldsymbol{s}^{(t)} = f(\boldsymbol{s}^{(t-1)};\ \boldsymbol{\theta})$, where $\boldsymbol{s}^{(t)}$ is the state of the system, is recurrent because the definition of $\boldsymbol{s}$ at time t refers back to the same definition at time $t-1$.
- The network typically learns to use the state as a kind of lossy summary of the task-relevant aspects of the past sequence of inputs up to t.
- It is possible to represent the unfolded recurrence after t steps: $\boldsymbol{h}^{(t)} = g^{(t)}(\boldsymbol{x}^{(t)}, \boldsymbol{x}^{(t-1)}, \cdots, \boldsymbol{x}^{(2)}, \boldsymbol{x}^{(1)}) = f(\boldsymbol{h}^{(t-1)}, \boldsymbol{x}^{(t)}; \boldsymbol{\theta})$, where the function $g^{(t)}$ takes the whole past sequence as input and produces the current state.
- Design patterns for RNNs include:
  - RNN that produce an output at each time step and have recurrent connections between hidden units. Any function computable by a Turing machine can be computed by this network.
    Forward propagation begins with a specification of the initial state $\boldsymbol{h}^{(0)}$. Then, for each time step from $t = 1$ to $t = \tau$, the following updates are applied:
    $$\boldsymbol{a}^{(t)} = \boldsymbol{b} + \boldsymbol{W}\boldsymbol{h}^{(t-1)} + \boldsymbol{U}\boldsymbol{x}^{(t)},$$
    $$\boldsymbol{h}^{(t)} = \tanh \boldsymbol{a}^{(t)},$$
    $$\boldsymbol{o}^{(t)} = \boldsymbol{c} + \boldsymbol{V}\boldsymbol{h}^{(t)},$$
    $$\hat{\boldsymbol{y}}^{(t)} = softmax(\boldsymbol{o}^{(t)}).$$
    The total loss for a given sequence of $\boldsymbol{x}$ values paired with a sequence of $\boldsymbol{y}$ values would the just be the sum of the losses over all the time steps.
  - RNN that produce an output at each time step and have recurrent connections only from the output at one timestep to the hidden units at the next time step. This option is less powerful because it lacks hidden-to-hidden recurrent connections. Training can be parallelized.
  - RNN with recurrent connections between hidden units, that read an entire sequence and then produce a single output.
- Teacher forcing is a procedure that emerges from the maximum likelihood criterion, in which during training the model receives the ground truth output $y^{(t)}$ as input at time $t+1$. Its disadvantage arises if the network is going to be later used in an open-loop mode, with the network outputs fed back as input.
- When using a predictive log-likelihood training objective, the RNN is trained to estimate the conditional distribution of the next sequence element $\boldsymbol{y}^{(t)}$ given the past inputs.
- It is possible to view the RNN as defining a graphical model whose structure is the complete graph, able to represent direct dependencies between any pair of y values. Every past observation $y^{(i)}$ may influence the conditional distribution of some $y^{(t)}$ (for $t > i$), given the previous values.
- The parameter sharing used in RNNs relies on the assumption that the same parameters can be used for different time steps.
- Some common ways of providing an extra input to an RNN are: as an extra input at each time step, or as the initial state $\boldsymbol{h}^{(0)}$, or both.
- Up to now RNNs have a casual structure, meaning that the state at time t captures only information from the past, $\boldsymbol{x}^{(1)}, \cdots, \boldsymbol{x}^{(t-1)}$, and the present input $\boldsymbol{x}^{(t)}$. Also, some models allow information from past $\boldsymbol{y}$ values to affect the current state when the $\boldsymbol{y}$ values are available.
- Many applications need to output a prediction of $\boldsymbol{y}^{(t)}$ that may depend on the whole input sequence. Bidirectional RNNs were invented to address that need. They combine an RNN that moves forward through time, beginning from the start of the sequence, with another RNN that moves backward through time, beginning from the end of the sequence.
- An RNN can be trained to map an input sequence to an output sequence which is not necessarily of the same length.

- The input to an RNN is sometimes called the "context". The objective is to produce a representation of this context, C. C might be a vector or sequence of vectors that summarize the input sequence $\boldsymbol{X} = (\boldsymbol{x}^{(1)}, \cdots, \boldsymbol{x}^{(n_x)})$.
- Encoder-decoder or sequence-to-sequence RNN architecture: it is composed of an encoder RNN that reads the input sequence as well as a decoder RNN that generates the output sequence. The final hidden stat of the encoder RNN is used to compute a generally fixed-size context variable C, which represents a semantic summary of the input sequence and is given as input to the decoder RNN. The two RNNs are trained jointly to maximize the average of $\log P(\boldsymbol{y}^{(1)}, \cdots, \boldsymbol{y}^{(n_y)} | \boldsymbol{x}^{(1)}, \cdots, \boldsymbol{x}^{(n_x)})$ over all the pairs of $\boldsymbol{x}$ and $\boldsymbol{y}$ sequences in the training set. The lenghts of n_x and n_y can vary from each other.
- The computation in most RNNs can be decomposed into three blocks of parameters and associated transformations:
  1. From the input to the hidden state,
  2. From the previous hidden state to the next hidden state, and
  3. From the hidden state to the output.
- Recursive neural networks represent other generalization of RNNs, which is structured as a deep tree, rather than the chain-like structure of RNNs.
- A clear advantage of recursive nets over recurrent nets is that for a sequence of the same length τ, the depth (measured as the number of compositions of nonlinear operations) can be drastically reduced from τ to $O(\log \tau)$.
- Recurrent networks involve the composition of the same function multiple times, once per time step, which can result in extremely nonlinear behavior.
- It is possible to think of the recurrence relation $\boldsymbol{h}^{(t)} = \boldsymbol{W}^\top \boldsymbol{h}^{(t-1)}$ as a very simple RNN lacking of nonlinear activation function, and lacking inputs $\boldsymbol{x}$.
- The recurrent weights mapping from $\boldsymbol{h}^{(t-1)}$ to $\boldsymbol{h}^{(t)}$ and the input weights mapping from $\boldsymbol{x}^{(t)}$ to $\boldsymbol{h}^{(t)}$ are some of the most difficult parameters to learn in a recurrent network. One proposed approach to avoiding this difficulty is to set the recurrent weights such that the recurrent hidden units do a good job of capturing the history of past inputs, and only learn the output weights. This idea is proposed for echo state networks (ESN) or liquid state machines.
- Reservoir computing: denote the fact that the hidden units form a reservoir of temporal features that may capture different aspects of the history of inputs.
- To represent a rich set of histories in the RNN, view the recurrent net as a dynamical system, and set the input and recurrent weights such that the dynamical system is near the edge of stability.
- Everything about backpropagation via repeated matrix multiplication applies equally to forward propagation in a network with no nonlinearity, where the state $\boldsymbol{h}^{(t+1)} = \boldsymbol{h}^{(t)\top} \boldsymbol{W}$.
- One way to deal with long-term dependencies is to design a model that operates at multiple time scales, so that some parts of the model operate at fine-grained time scales and can handle small details, while other parts operate at coarse time scales and transfer information from the distant past to the present more efficiently.
- One way to obtain coarse time scales is to add direct connections from variables in the distant past to variables in the present.
- Another way to obtain paths on which the product of derivatives is close to one is to have units (known as leaky units) with linear self-connections and a weight near one on these connections.
- Leaky units allow the network to accumulate information over a long duration. Once that information has been used, however, it might be useful for the NN to forget the old state.
- Other approach to handling long-term dependencies involves actively removing length-one connections and replacing them with longer connections.
- The most effective sequence models used in practical applications are called gated RNNs. This include long short-term memory (LSTM) and networks based on the gated recurrent unit.
- Gated RNNs are based on the idea of creating paths through time that have derivatives that neither vanish nor explode.
- LSTM model: it uses self-loops to produce paths were the gradient can flow for long durations; also the weight on this self-loop is conditioned on the context, rather than fixed. By making the weight of this self-loop gated (controlled by another hidden unit), the time scale of integration can be changed dynamically.

- Instead of a unit that simply applies an element-wise nonlinearity to the affine transformation of inputs and recurrent units, LSTM recurrent networks have "LSTM cells" that have an internal recurrence (a self-loop), in addition to the outer recurrence of the RNN. Each cell has the same inputs and outputs as an ordinary recurrent network, but also has more parameters and a system of gating units that controls the flow of information. The most important component is the state unit $s_i^{(t)}$, which has a linear self-loop similar to the leaky units. The self-loop weight (or the associated time constant) is controlled by a forget gate unit $f_i^{(t)}$ (for time step t and cell i), which sets the weight via a sigmoid unit. Section 10.10.1 presents the corresponding equations.
- A single gating unit simultaneously controls the forgetting factor and the decision to update the state unit.
- It is often much easier to design a model that is easy to optimize than it is to design a more powerful optimization algorithm.
- Clipping gradients: this is used when the parameter gradient is very large, such that a gradient descent parameter update could throw the parameters very far into a region where the objective function is larger, undoing much of the work that had been done to reach the current solution. This helps to deal with exploding gradients.
- An idea to deal with vanishing gradients is to regularize or constrain the parameters so as to encourage "information flow". In particular, the objective is that the gradient vector $\nabla_{\boldsymbol{h}^{(t)}} L$ being back-propagated maintains its magnitude, even if the loss function only penalizes the output at the end of the sequence.
- NN excel at storing implicit knowledge, but they struggle to memorize facts.

11 Practical Methodology

- Practical design process:
  - Determine your goals.
    - Which error metric to use (usually different from the cost function used to train the model).
    - What is the expected performance.
  - Establish a working end-to-end pipeline as soon as possible, including the estimation of the appropriate performance metrics.
    - Choose the general category of model based on the structure of the data.
    - Optimization algorithm: SGD with momentum with a decaying learning rate; or ADAM.
  - Instrument the system well to determine bottlenecks in performance.
  - Repeatedly make incremental changes based on specific findings from the instrumentation.
- Bayes error defines the minimum error rate that can be achieved.
- Precision: fraction of detections reported by the model that were correct.
- Recall: fraction of true events that were detected by the model.
- To summarize the performance of a classifier, it is possible to convert precision p and recall r into an F-score given by: $F = \frac{2pr}{p+r}$. Another option is to report the total area lying beneath the PR curve.
- In some applications, it is possible for the ML system to refuse to make a decision.
- Coverage: fraction of examples for which the ML system is able to produce a response.
- If performing supervised learning (SL) with fixed-size vectors as input, use a feedforward network with fully connected layers. If the input has known topological structure, use a CNN. In these cases, start by using some kind of piecewise linear unit (ReLUs or their generalizations). If the input or output is a sequence, use a gated recurrent net (LSTM or GRU).
- Batch normalization can have a dramatic effect on optimization performance, especially for CNNs and networks with sigmoidal performance. Sometimes reduces generalization error and allows dropout to be omitted.
- Early stopping should be used almost universally.
- Dropout is an excellent regularizer compatible with many models and training algorithms.
- If the application is in a context where unsupervised learning (UL) is known to be important, then include it in the first end-to-end baseline. Otherwise, only use UL in the first attempt if the task to be solved is unsupervised.
- It is often much better to gather more data than to improve the learning algorithm.

- If performance on the training set is poor, improve the learning algorithm (parameter tunning, increase size of the model, etc). If this does not work, the problem might be the quality of the data.
- If performance on the training set is acceptable, but the test set performance is much worse, then gather more data (an alternative is to reduce the size of the model and/or improve regularization). If it is not possible, improve the learning algorithm itself (but this becomes the domain of researchers).
- Choosing the hyperparameters manually requires understanding what the hyperparameters do and how ML models achieve good generalization. Automatic hyperparameter selection algorithms greatly reduce the need to understand these ideas, but they are often much more computationally costly.
- Effective capacity: the representational capacity of the model, the ability of the learning algorithm to successfully minimize the cost function used to train the model, and the degree to which the cost function and training procedure regularize the model.
- The generalization error typically follows a U-shaped curve when plotter as a function of one of the hyperparameters. Somewhere in the middle lies the optimal model capacity, which achieves the lowest possible generalization error, by adding a medium generalization gap to a medium amount of training error.
- The learning rate is perhaps the most important hyperparameter. It controls the effective capacity of the model, because it is highest when the learning rate is correct. It has a U-shaped curve for training error.
- Tuning the hyperparameters other than the learning rate requires monitoring both training and test error, then adjusting its capacity appropriately.
- Tuning hyperparameters with grid search: for each hyperparameter, the user selects a small finite set of values to explore. The grid search algorithm then trains a model for every joint specification of hyperparameter values in the Cartesian product of the set of values for each individual hyperparameter. The experiment that yields the best validation set error is then chosen as having found the best hyperparameters.
- Another hyperparameter tuning technique is random search, where a PD distribution is defined in a specified search range and according to the characteristics of each hyperparameter.
- Debugging strategies for NN: either we design a case that is so simple that the correct behavior actually can be predicted, or we design a test that exercises one part of the NN implementation in isolation.
- Some debugging tests include: visualize the model in action; visualize the worst mistakes; reason about software using training and test error; fit a tiny dataset; compare back-propagated derivatives to numerical derivatives; monitor histograms of activations and gradient

12 Applications

- Deep learning is based on the idea that a large population of features acting together can exhibit intelligent behavior.
- Nowadays NN are trained mostly using GPUs or the CPUs of many machines networked together.
- Obtaining good performance with GPUs:
  - Most writable memory locations are not cached, so it can actually be faster to compute the same value twice, rather than compute it once and read it back from memory.
  - Multithreaded code.
  - Memory operations can be faster if they can be coalesced. Coalesced reads or writes occur when several threads can each read or write a value that they need simultaneously, as part of a single memory transaction.
  - Make sure that each thread in a group executes the same instructions simultaneously.
- Is better to distribute the workload of training and inference across many machines.
- Asynchronous SGD causes the learning process to be faster overall.
- Model compression: large models learn some function $f(\boldsymbol{x})$, but do so using many more parameters than are necessary for the task. As soon as is possible to fit this function, a training set containing infinitely many examples can be generated, simply by applying f to randomly sampled points $\boldsymbol{x}$. Then the new, smaller model is trained to match $f(\boldsymbol{x})$ on these points.
- Use dynamic structure to accelerate data-processing systems. NN use conditional computation.
- To accelerate inference in a classifier use a cascade of classifiers. This strategy may be applied when the goal is to detect the presence of a rare object (or event).

- Different kinds of preprocessing are applied to both the training and the test set with the goal of putting each example into a more canonical form to reduce the amount of variation that the model needs to account for.
- Dataset augmentation: improve the generalization of a classifier by increasing the size of the training set by adding extra copies of the training examples that have been modified with transformations that do not change the class.
- The task of speech recognition is to map an acoustic signal containing a spoken natural language utterance into the corresponding sequence of words intended by the speaker.
  - Unsupervised pretraining phase considered unnecessary (it did not bring significant improvements).
  - CNNs that replicate weights across time and frequency are commonly used.
- Natural Language Processing (NLP): use of human languages by a computer.
  - Techniques that are specialized for processing sequential data are commonly used.
  - Word-based language models operate on an extremely high-dimensional and sparse discrete space, they define a PD over sequences of tokens in a natural language.
  - Neural Language Models (NLM): class of language model designed to overcome the curse of dimensionality problem for modeling natural language sequences by using a distributed representation of words.
- Machine translation is the task of reading a sentence in one natural language and emitting a sentence with the equivalent meaning in another language. This systems often involve many components.
  - They used attention-based systems that are composed of:
    1. A process that reads raw data.
    2. A list of feature vectors storing the output of the reader.
    3. A process that exploits the content of the memory to sequentially perform a task, at each time step having the ability to put attention on the content of one memory element (or a few, with a different weight).
- Other application involve:
  - Recommender systems: collaborative filtering algorithms, reinforcement learning, etc.
  - Knowledge representation, Reasoning and Question answering.
    - Determine how distributed representations can be trained to capture the relations between two entities.
- Reinforcement learning requires choosing a trade-off between exploration and exploitation. Exploitation refers to taking actions that come from the current, best version of the learned policy. Exploration refers to taking actions specifically to obtain more training data.

13 Linear Factor Models

- Models that use probabilistic inference, many of which use latent variables $\boldsymbol{h}$, with $p_{model}(\boldsymbol{x}) = \mathbb{E}_{\boldsymbol{h}} p_{model}(\boldsymbol{x}|\boldsymbol{h})$.
- They are defined by the use of a stochastic linear decoder function that generates $\boldsymbol{x}$ by adding noise to a linear transformation of $\boldsymbol{h}$.
- Its data-generation process is described as follows:
  - The explanatory factors $\boldsymbol{h}$ are sampled from a distribution $\mathbf{h} \sim p(\boldsymbol{h})$, where $p(\boldsymbol{h})$ is a factorial distribution, with $p(\boldsymbol{h}) = \prod_i p(h_i)$.
  - Next, the real-valued observable variables are sampled given the factors $\boldsymbol{x} = \boldsymbol{W}\boldsymbol{h} + \boldsymbol{b} +$ noise.
- Probabilistic PCA, factor analysis and other linear factor models are special cases of the above equations.
- Independent Component Analysis (ICA) is an approach to modeling linear factors that seeks to separate an observed signal into many underlying signals that are scaled and added together to form the observed data. These signals are intended to be fully independent.
  - This is commonly used to recover low-level signals that have been mixed together.
- Slow Feature Analysis (SFA) is a linear factor model that uses information from time signals to learn invariant features. The underlaying idea is that the important characteristics of scenes change very slowly compared to the individual measurements that make up a description of a scene.

  - It is possible to theoretically predict which features SFA will learn.
- Sparse coding is an unsupervised feature learning and feature extraction mechanism. Strictly speaking, "sparse coding" refers to the process of inferring the value of $\boldsymbol{h}$ in a model, while "sparse modeling" refers to the process of designing and learning the model, but the term "sparse coding" is often used to refer to both.
- Linear factor models including PCA and factor analysis can be interpreted as learning a manifold.

14 Autoencoders

- NN trained to attempt to copy its input to its output. They are designed to be unable to learn to copy perfectly, so it prioritizes to learn useful properties of the data.
- Internally, it has a hidden layer $\boldsymbol{h}$ that describes a code used to represent the input. The network may be viewed as consisting of two parts: an encoder function $\boldsymbol{h} = f(\boldsymbol{x})$ and a decoder that produces a reconstruction $\boldsymbol{r} = g(\boldsymbol{h})$.
- Modern autoencoders have generalized to stochastic mappings $p_{encoder}(\boldsymbol{h}|\boldsymbol{x})$ and $p_{decoder}(\boldsymbol{x}|\boldsymbol{h})$. This means that both the encoder and the decoder are not simple functions but instead involve some noise injection.
- One way to obtain useful features from the autoencoder is to constraint $\boldsymbol{h}$ to have a smaller dimension than $\boldsymbol{x}$. This autoencoders are known as undercomplete.
- The learning process is described simply as minimizing a loss function $L(\boldsymbol{x}, g(f(\boldsymbol{x})))$.
- The autoencoder also fails to learn anything useful if the hidden code is allowed to have dimension greater than (overcomplete) or equal to the input.
- Regularized autoencoders let you train any architecture of autoencoder successfully, choosing the code dimension and the capacity of the encoder and decoder based on the complexity of the distribution to be modeled. Its loss function encourages the model to have other properties: sparsity of the representation, smallness of the derivative of the representation, and robustness to noise or to missing inputs.
- Nearly any generative model with latent variables and equipped with an inference procedure may be viewed as a particular form of autoencoder.
- A sparse autoencoder is an autoencoder whose training criterion involves a sparsity penalty $\Omega(h)$ on the code layer $\boldsymbol{h}$, in addition to the reconstruction error: $L(\boldsymbol{x}, g(f(\boldsymbol{x}))) + \Omega(h)$. They are typically used to learn features for another task, such as classification.
- Training an autoencoder is a way of approximately training a generative model.
- A denoising autoencoder (DAE) minimizes $L(\boldsymbol{x}, g(f(\tilde{\boldsymbol{x}})))$, where $\tilde{\boldsymbol{x}}$ is a copy of $\boldsymbol{x}$ that has been corrupted by some form of noise. DAEs must therefore undo this corruption rather than simply copying their input.
  - The training procedure introduces a corruption process $C(\tilde{\mathtt{x}}|\mathtt{x})$. The autoencoder then learns a reconstruction distribution $p_{reconstruct}(\tilde{\mathtt{x}}|\mathtt{x})$ estimated from training pairs $(\tilde{\boldsymbol{x}}|\boldsymbol{x})$ as follows:
    1. Sample a training example $\boldsymbol{x}$ from the training data.
    2. Sample a corrupted version $\tilde{\boldsymbol{x}}$ from $C(\tilde{\mathtt{x}}|\mathtt{x} = \boldsymbol{x})$.
    3. Use $(\tilde{\boldsymbol{x}}|\boldsymbol{x})$ as a training example for estimating the autoencoder reconstruction distribution $p_{reconstruct}(\boldsymbol{x}|\tilde{\boldsymbol{x}}) = p_{decoder}(\boldsymbol{x}|\boldsymbol{h})$ with $\boldsymbol{h}$ the output of encoder $f(\tilde{\boldsymbol{x}})$ and $p_{decoder}$ typically defined by a decoder $g(\boldsymbol{h})$.
- Another strategy for regularizing an autoencoder is to use a penalty Ω, as in sparse autoencoders, but with a different form of Ω: $\Omega(\boldsymbol{h}, \boldsymbol{x}) = \lambda \sum_i ||\nabla_{\boldsymbol{x}} h_i||^2$. This forces the model to learn a function that does not change much when $\boldsymbol{x}$ changes slightly, and to learn features that capture information about the training distribution.
  This are known as contractive autoencoders (CAE).
- Using deep encoders and decoders offers many advantages. Depth can exponentially reduce the computational cost of representing some functions and the amount of training data needed.
- Autoencoders are just feedforward networks.
- A general strategy for designing the output units and the loss function of a feedforward network is to define an output distribution $p(\boldsymbol{y}|\boldsymbol{x})$ and minimize the negative log-likelihood $-\log p(\boldsymbol{y}|\boldsymbol{x})$.
- In an autoencoder, $\boldsymbol{x}$ is now the target as well as the input.
- Score matching provides a consistent estimator of probability distributions based on encouraging the model to have the same score as the data distribution at every training point $\boldsymbol{x}$. In this context, it is a particular gradient field: $\nabla_{\boldsymbol{x}} \log p(\boldsymbol{x})$.

- Like many other ML algorithms, autoencoders exploit the idea that data concentrates around a low-dimensional manifold or a small set of such manifolds. Autoencoders take this idea further and aim to learn the structure of the manifold.
- By making the reconstruction function sensitive to perturbations of the input around the data points, we cause the autoencoder to recover the manifold structure.
- Nonparametric manifold learning procedures build a nearest neighbor graph in which nodes represent training examples.
- The contractive autoencoder introduces an explicit regularizer on the code $\boldsymbol{h} = f(\boldsymbol{x})$, encouraging the derivatives of f to be as small as possible:

$$\Omega(\boldsymbol{h}) = \lambda \left\| \frac{\partial f(\boldsymbol{x})}{\partial \boldsymbol{x}} \right\|_F^2 .$$

- Predictive sparse decomposition (PSD) is a model that is a hybrid of sparse coding and parametric autoencoders. The model consists of an encoder $f(\boldsymbol{x})$ and a decoder $g(\boldsymbol{h})$ that are both parametric. During training, $\boldsymbol{h}$ is controlled by minimizing: $\|\boldsymbol{x} - g(\boldsymbol{h})\|^2 + \lambda \mid \boldsymbol{h} \mid_1 + \gamma \|\boldsymbol{h} - f(\boldsymbol{x})\|^2$.
  PSD is an example of learned approximate inference.
- Autoencoders have been successfully applied to dimensionality reduction and information retrieval tasks.

15 Representation Learning

- Shared representations are useful to handle multiple modalities or domains, or to transfer learned knowledge to tasks for which few or no examples are given but a task representation exists.
- Generally speaking, a good representation is one that makes a subsequent learning task easier.
- Representation learning provides one way to perform unsupervised and semi-supervised learning.
- Greedy layer-wise unsupervised pretraining: it relies on a single-layer representation learning algorithm, a single-layer autoencoder, a sparse coding model, or another model that learns latent representations. Each layer is pretrained using unsupervised learning, taking the output of the previous layer and producing as output a new representation of the data, whose distribution is hopefully simpler. See algorithm 15.1.
  - It optimizes each piece of the solution independently.
  - It is expected to be more effective when the initial representation is poor.
  - It is likely to be most useful when the function to be learned is extremely complicated.
- Unsupervised pretraining combines two different ideas:
  1. The choice of initial parameters for a deep NN can have a significant regularizing effect on the model.
  2. Learning about the input distribution can help with learning about the mappings from inputs to outputs.
- NN that receive unsupervised pretraining consistently halt in the same region of function space.
- Deep learning techniques based on supervised learning, regularized with dropout or batch normalization, are able to achieve human-level performance on many tasks, but only with extremely large labeled datasets.
- Transfer learning and domain adaptation refer to the situation where what has been learned in one setting is exploited to improve generalization in another setting.
- In transfer learning, the learner must perform two or more different tasks, but it is assumed that many of the factors that explain the variations in distribution P_1 are relevant to the variations that need to be captured for learning P_2.
- In domain adaptation, the task (and the optimal input-to-output mapping) remains the same between each setting, but the input distribution is slightly different.
- Concept drift: form of transfer learning due to gradual changes in the data distribution over time.
- Two extreme forms of transfer learning are:
  - One-shot learning: only one labeled example of the transfer task is given. The representation learns to cleanly separate the underlying classes during the first stage.
  - Zero-shot(data) learning: no labeled examples are given.

- Zero-data learning and zero-shot learning are only possible because additional information has been exploited during training.
- In a zero-data learning scenario, the model is trained to estimate the conditional distribution $p(\boldsymbol{y}|\boldsymbol{x}, T)$, where T is a description (in a way that allows some sort of generalization) of the task that the model is expected to perform.
- Multimodal learning: captures a representation in one modality, a representation in the other, and the relationship (in general a joint distribution) between pairs $(\boldsymbol{x}, \boldsymbol{y})$ consisting of one observation $\boldsymbol{x}$ in one modality and another observation $\boldsymbol{y}$ in the other modality.
- An emerging strategy for unsupervised learning is to modify the definition of which underlying causes are most salient.
- Generative adversarial networks: a generative model is trained to fool a feedforward classifier. The feedforward classifier attempts to recognize all samples from the generative model as being fake and all samples from the training set as being real. In this framework, any structured pattern that the feedforward network can recognize is highly salient.
- A benefit of learning the underlying causal factors, is that if the true generative process has $\mathbf{x}$ as an effect and $\mathbf{y}$ as a cause, then modeling $p(\mathbf{x} \mid \mathbf{y})$ is robust to changes in $p(\mathbf{y})$.
- Distributed representation of concepts: representations composed of many elements that can be set separately from each other. They can use n features with k values to describe k^n different concepts.
- An ideal representation is one that disentangles the underlying causal factors of variation that generated the data, especially those that are relevant to the specific application.
- An adequate regularizer might help the learning algorithm discover features that correspond to underlying factors.
- Some generic regularization strategies:
  - Smoothness: this is the assumption that $f(\boldsymbol{x} + \epsilon \boldsymbol{d}) \approx f(\boldsymbol{x})$ for unit $\boldsymbol{d}$ and small ϵ. This idea is insufficient to overcome the curse of dimensionality.
  - Linearity: the learning algorithm assumes that the relationships between some variables are linear.
  - Multiple explanatory factors: the learning algorithm assumes that the data is generated by multiple underlying explanatory factors, and that most tasks can be solved easily given the state of each of these factors.
  - Causal factors: the model is constructed in such a way that is treats the factors of variation described by the learned representation $\boldsymbol{h}$ as the causes of the observed data $\boldsymbol{x}$, and not vice versa.
  - Depth or a hierarchical organization of explanatory factors: the use of a deep architecture expresses the belief that the task should be accomplished via a multistep program, with each step referring back to the output of the processing accomplished via previous steps.
  - Shared factors across tasks: when many tasks corresponding to different y_i variables sharing the same input $\mathbf{x}$, or when each task is associated with a subset or a function $f^{(i)}(\mathbf{x})$ of a global input $\mathbf{x}$, the assumption is that each y_i is associated with a different subset from a common pool of relevant factors $\mathbf{h}$.
  - Manifolds: probability mass concentrates, and the regions in which it concentrates are locally connected and occupy a tiny volume.
  - Natural clustering: the learning algorithm assume that each connected manifold in the input space may be assigned to a single class.
  - Temporal and spatial coherence: the algorithm assumes that the most important explanatory factors change slowly over time, or at least that it is easier to predict the true underlying explanatory factors than to predict raw observations.
  - Sparsity: most features should presumably not be relevant to describing most inputs. It is therefore reasonable to impose a prior that any feature that can be interpreted as "present" or "absent" should be absent most of the time.
  - Simplicity of factor dependencies: in good high-level representations, the factors are related to each other through simple dependencies.

16 Structured Probabilistic Models for Deep Learning

- A structured probabilistic model or graphical model is a way of describing a probability distribution, using a graph to describe which random variables in the probability distribution interact with each other **directly** (this allows the model to have significantly less parameters and therefore be estimated reliably from less data).
- Deep learning's goal is to be able to understand high-dimensional data with rich structure.
- In a graphical model, each node represents a random variable, and each edge represents a direct interaction. These direct interactions imply other, indirect interactions, but only the direct interactions need to be explicitly modeled.
- Graphical models can be divided into: models based on direct acyclic graphs, and models based on undirected graphs.
- Directed models:
  - A Directed graphical model, is also known as a belief network or Bayesian network.
  - If node a has an arrow pointing to node b, then the distribution over b depends on the value of a (that is, the PD of b is defined via a conditional distribution).
  - A directed graphical model defined on variables $\mathbf{x}$ is defined by a directed acyclic graph $\mathcal{G}$ whose vertices are the random variables in the model, and a set of local conditional probability distributions $p(\mathrm{x}_i|Pa_{\mathcal{G}}(\mathrm{x}_i))$, where $Pa_{\mathcal{G}}(\mathrm{x}_i)$ gives the parents of x_i in $\mathcal{G}$. The PD over $\mathbf{x}$ is given by $p(\mathbf{x}) = \prod_i p(\mathrm{x}_i|Pa_{\mathcal{G}}(\mathrm{x}_i))$.
  - As long as each variable has few parents in the graph, the distribution can be represented with very few parameters.
  - The graph encodes only simplifying assumptions about which variables are conditionally independent from each other. Information that cannot be encoded in the graph, is encoded in the definition of the conditional distribution itself.
- Undirected models:
  - An undirected model is also known as Markov random field (MRF) or Markov network.
  - They are used when the interactions seem to have no intrinsic direction, or to operate in both directions.
  - If two nodes are connected by an edge, then the random variables corresponding to those nodes interact with each other directly.
  - An undirected graphical model is a structured probabilistic model defined on an undirected graph $\mathcal{G}$. For each clique $\mathcal{C}$ in the graph, a factor (constrained to be nonnegative) $\phi(\mathcal{C})$ (also called a clique potential) measures the affinity of the variables in that clique for being in each of their possible joint states. Together they define an unnormalized PD: $\tilde{p}(\mathbf{x}) = \prod_{\mathcal{C}\in\mathcal{G}} \phi(\mathcal{C})$.
  - There is nothing to guarantee that multiplying the cliques together will yield a valid PD.
- To obtain a valid PD from the unnormalized PD, use the corresponding normalized PD (Gibbs distribution): $p(\mathbf{x}) = \frac{1}{Z}\tilde{p}(\mathbf{x})$, where Z (the partition function) is the value that results in the PD summing (for discrete variables) or integrating (for continuous variables) to 1: $Z = \int \tilde{p}(\mathbf{x})d\mathbf{x}$ or $\sum_{\boldsymbol{x}} \tilde{p}(\boldsymbol{x})$.
- In undirected models, it is possible to specify the factors in such a way that Z does not exist. This happens if some of the variables in the model are continuous and the integral of $\tilde{p}$ over their domain diverges.
- A convenient way to enforce the undirected model's assumption that: $\forall\mathbf{x}$, $\tilde{p}(\mathbf{x}) > 0$, is to use an energy-based model (EBM) where $\tilde{p}(\mathbf{x}) = \exp(-E(\mathbf{x}))$, and $E(\mathbf{x})$ is known as the energy function.
- Any distribution of the form given by the previous equation is an example of a Boltzmann distribution.
- Many algorithms that operate on probabilistic models need to compute not $p_{model}(\boldsymbol{x})$ but only $\log \tilde{p}_{model}(\boldsymbol{x})$.
- Separation: conditional independence implied by a graph, there is no requirement that the graph imply all independences that are present. It is said that a set of variables $\mathbb{A}$ is separated from another set of variables $\mathbb{B}$ given a third set of variables $\mathbb{S}$ if the graph structure implies that $\mathbb{A}$ is independent from $\mathbb{B}$ given $\mathbb{S}$. Similar concepts apply to directed models, but in that context they are referred to as d-separation.
- We may choose to use either directed modeling or undirected modeling based on which approach can capture the most independences in the PD or which uses the fewest edges to describe the distribution.

- Only directed models can represent a structure called immortality. It occurs when two random variables a and b are both parents of a third random variable c, and there is no edge directly connecting a and b in either direction.
- To convert a directed model with graph $\mathcal{D}$ into an undirected model, we need to create a new graph $\mathcal{U}$. For every pair of variables x and y, we add an undirected edge connecting x and y to $\mathcal{U}$ if there is a directed edge (in either direction) connecting x and y in $\mathcal{D}$ or if x and y are both parents in $\mathcal{D}$ of a third variable z. The resulting $\mathcal{U}$ is known as a moralized graph.
- A directed graph $\mathcal{D}$ cannot capture all the conditional independences implied by an undirected graph $\mathcal{U}$ if $\mathcal{U}$ contains a loop of length greater than three, unless that loop contains a chord (a connection between any nonconsecutive variables in the sequence defining a loop).
- A factor graph is a graphical representation of an undirected model that consists of a bipartite undirected graph.
- One advantage of directed graphical models is that a simple and efficient procedure called ancestral sampling can produce a sample from the joint distribution represented by the model. The basic idea is to sort the variables x_i in the graph into a topological ordering so that for all i and j, j is greater than i if x_i is a parent of x_j. The variables can then be sampled in this order.
- A good generative model needs to accurately capture the distribution over the observed, or "visible", variables $\mathbf{v}$. Often the different elements of $\mathbf{v}$ are highly dependent on each other. In the context of deep learning, the approach most commonly used to model these dependencies is to introduce several latent or "hidden" variables, $\mathbf{h}$. The model can then capture dependencies between any pair of variables v_i and v_j indirectly, via direct dependencies between v_i and $\mathbf{h}$, and direct dependencies between $\mathbf{h}$ and v_j.
- Roughly, structure learning, is to connect those variables that are tightly coupled and omit edges between other variables.
- Inference problems: we must predict the value of some variables given other variables, or predict the PD over some variables given the value of other variables.
- Deep learning essentially always makes use of the idea of distributed representations. Even shallow models nearly always have a single layer of latent variables. Deep learning models typically have more latent variables than observed variables.
- Models used in deep learning tend to connect each visible unit v_i to many hidden units h_j, so that $\mathbf{h}$ can provide a distributed representation of v_i (and probably several other observed variables too).
- The deep learning approach to graphical modeling is characterized by a marked tolerance of the unknown. The power of the model is increased until it is just barely possible to train or use.

17 Monte Carlo Methods

- Randomize algorithms fall into two rough categories:
  - Las Vegas algorithms, which always return precisely the correct answer (or report that they failed).
  - Monte Carlo algorithms, which return answers with a random amount of error.
- Many technologies used to accomplish ML goals are based on drawing samples from some PD and using these samples to form a Monte Carlo estimate of some desired quantity.
- When a sum or an integral cannot be computed exactly, it is often possible to approximate it using Monte Carlo sampling. The idea is to view the sum or integral as if it were an expectation under some distribution and to approximate the expectation by a corresponding average. Let

$$s = \sum_{\boldsymbol{x}} p(\boldsymbol{x}) f(\boldsymbol{x}) = E_p[f(\mathbf{x})]$$

or

$$s = \int p(\boldsymbol{x}) f(\boldsymbol{x}) d\boldsymbol{x} = E_p[f(\mathbf{x})]$$

be the sum or integral to estimate, rewritten as an expectation, with the constraint that p is a PD (for the sum) or a probability density (for the integral) over random variable $\mathbf{x}$.
It is possible to approximate s by drawing n samples $\boldsymbol{x}^{(1)}, \cdots, \boldsymbol{x}^{(n)}$ from p and then forming the empirical average

$$\hat{s}_n = \frac{1}{n} \sum_{i=1}^{n} f(\boldsymbol{x}^{(i)}).$$

We compute both the empirical average of the $f(\boldsymbol{x}^{(i)})$ and the empirical variance, and then divide the estimated variance by the number of samples n to obtain an estimator of $\mathrm{Var}[\hat{s}_n]$.

- Any Monte Carlo estimator
$$\hat{s}_p = \frac{1}{n} \sum_{i=1, \mathrm{x}^{(i)} \sim p}^{n} f(\boldsymbol{x}^{(i)})$$
can be transformed into an importance sampling estimator
$$\hat{s}_q = \frac{1}{n} \sum_{i=1, \mathrm{x}^{(i)} \sim q}^{n} \frac{p(\boldsymbol{x}^{(i)}) f(\boldsymbol{x}^{(i)})}{q(\boldsymbol{x}^{(i)})}.$$

- The family of algorithms that use Markov chains to perform Monte Carlo estimates is called Markov chain Monte Carlo methods (MCMC). They are used to approximately sample from $p_{model}(\mathbf{x})$ when there is no tractable method for drawing exact samples from the distribution $p_{model}(\mathbf{x})$ or from a good (low variance) importance sampling distribution $q(\mathbf{x})$. In the context of deep learning, this most often happens when $p_{model}(\mathbf{x})$ is represented by an undirected model.
- The most standard, generic guarantees for MCMC techniques are only applicable when the model does not assign zero probability to any state.
- The core idea of a Markov chain is to have a state $\boldsymbol{x}$ that begins as an arbitrary value. Over time, we randomly update $\boldsymbol{x}$. Eventually $\boldsymbol{x}$ becomes (very nearly) a fair sample from $p(\boldsymbol{x})$. Formally, it is defined by a random state $\boldsymbol{x}$ and a transition distribution $T(\boldsymbol{x}'|\boldsymbol{x})$ specifying the probability that a random update will go to state $\boldsymbol{x}'$ if it starts in state $\boldsymbol{x}$. Running the Markov chain means repeatedly updating the state $\boldsymbol{x}$ to a value $\boldsymbol{x}'$ sampled from $T(\boldsymbol{x}'|\boldsymbol{x})$.
- A conceptually simple and effective approach to building a Markov chain that samples from $p_{model}(\boldsymbol{x})$ is to use Gibbs sampling, in which sampling from $T(\boldsymbol{x}'|\boldsymbol{x})$ is accomplished by selecting one variable x_i and sampling it from p_{model} conditioned on its neighbors in the undirected graph $\mathcal{G}$ defining the structure of the EBM. It is also possible to sample several variables at the same time as long as they are conditionally independent given all their neighbors.
- MCMC methods have a tendency to mix poorly.
- Markov chains based on tempered transitions temporarily sample from higher-temperature distributions to mix to different modes, then resume sampling from the unit temperature distribution.
- In parallel tempering, the Markov chain simulates many different states in parallel, at different temperatures. The highest temperature states mix slowly, while the lowest temperature states, at temperature 1, provide accurate samples from the model. The transition operator includes stochastically swapping states between two different temperature levels, so that a sufficiently high-probability sample from a high-temperature slot can jump into a lower temperature slot.
- Depth may help mixing.

18 Confronting the Partition Function

- The partition function used to learn undirected models by maximum likelihood depends on the parameters. The gradient of the log-likelihood w.r.t. the parameters has a term corresponding to the gradient of the partition function: $\nabla \log p(\mathrm{x};\boldsymbol{\theta}) = \nabla_{\boldsymbol{\theta}} \log \tilde{p}(\mathbf{x};\boldsymbol{\theta}) - \nabla_{\boldsymbol{\theta}} \log Z(\boldsymbol{\theta}$. This is a decomposition into the positive phase and negative phase of learning.
- The Monte Carlo approach to learning undirected models provides an intuitive framework to think about the positive phase and the negative phase. In the positive phase, we increase $\log \tilde{p}(\mathbf{x})$ for $\boldsymbol{x}$ drawn from the data. In the negative phase, we decrease the partition function by decreasing $\log \tilde{p}(\mathbf{x})$ drawn from the model distribution.
- The MCMC approach to maximum likelihood tries to balance between two forces, one pushing up (maximizing $\log \tilde{p}$) on the model distribution where the data occurs, and another pushing down (minimizing $\log Z$) on the model distribution where the model samples occur.
- The contrastive divergence (CD, or CD-k to indicate CD with Gibbs steps) algorithm initializes the Markov chain at each step with samples from the data distribution. See algorithm 18.2. This algorithm can fail to suppress spurious modes.
- A spurious mode is a mode that is present in the model distribution but absent in the data distribution.
- Stochastic maximum likelihood (SML) or Persistent contrastive divergence (PCD/PCD-k): solves many of the problems with CD by initializing the Markov chains at each gradient step with their states from the previous gradient step. See algorithm 18.3.
- The pseudolikelihood is based on the observation that conditional probabilities take this ratio-based form and thus can be computed without knowledge of the partition function. Suppose that we partition $\mathbf{x}$ into $\mathbf{a}$, $\mathbf{b}$ and $\mathbf{c}$, where $\mathbf{a}$ contains the variables we want to find the conditional distribution

over, $\mathbf{b}$ contains the variables we want to condition on, and $\mathbf{c}$ contains the variables that are not part of our query:

$$p(\mathtt{a}|) = \frac{p(\mathtt{a}, \mathtt{b})}{p(\mathtt{b})} = \frac{p(\mathtt{a}, \mathtt{b})}{\sum_{\mathtt{a},\mathtt{c}} p(\mathtt{a}, \mathtt{b}, \mathtt{c})} = \frac{\tilde{p}(\mathtt{a}, \mathtt{b})}{\sum_{\mathtt{a},\mathtt{c}} \tilde{p}(\mathtt{a}, \mathtt{b}, \mathtt{c})}$$

- Score matching provides another means of training a model without estimating Z or its derivatives. Its strategy is to minimize the expected squared difference between the derivatives of the model's log density w.r.t. the input and the derivatives of the data's log density w.r.t. the input.
- Like pseudolikelihood, score matching only works when it is possible to evaluate $\log \tilde{p}(\mathbf{x})$ and its derivatives directly.
- Denoising score matching is useful because in practice, we usually do not have access to the true p_{data} but rather only an empirical distribution defined by samples from it.
- In Noise-contrastive estimation (NCE), the probability distribution estimated by the model is represented explicitly as $\log p_{model}(\mathbf{x}) = \log \tilde{p}_{model}(\mathbf{x};\theta) + c$, where c is explicitly introduced as an approximation of $-\log z(\theta)$. Rather than estimating only θ, the NCE procedure treats c as just another parameter and estimates θ and c simultaneously, using the same algorithm for both.

19 Approximate Inference

- In the context of deep learning, we usually have a set of visible variables $\boldsymbol{v}$ and a set of latent variables $\boldsymbol{h}$. The challenge of inference usually refers to the difficult problem of computing $p(\boldsymbol{h}|\boldsymbol{v})$ or taking expectations with respect to it.
- Most graphical models with multiple layers of hidden variables have intractable posterior distributions. Exact inference requires an exponential amount of time in these models.
- Exact inference can be described as an optimization problem. To construct the optimization problem, assume we have a probabilistic model consisting of observed variables $\boldsymbol{v}$ and latent variables $\boldsymbol{h}$. We would like to compute the log-probability of the observed data, $\log p(\boldsymbol{v};\boldsymbol{\theta})$. Sometimes it is difficult to compute $\log p(\boldsymbol{v};\boldsymbol{\theta})$ if it is costly to marginalize out $\boldsymbol{h}$. Instead, we can compute a lower bound $\mathcal{L}(\boldsymbol{v}, \theta, q) = \log p(\boldsymbol{v};\boldsymbol{\theta}) - D_{KL}(q(\boldsymbol{h}|\boldsymbol{v})||p(\boldsymbol{h}|v;\theta))$, where q is an arbitrary PD over $\boldsymbol{h}$.
- Expectation Maximization (EM) algorithm: maximizes a lower bound $\mathcal{L}$. It is an approach to learning with an approximate posterior. It consists of alternating between two steps until convergence: (1) The E-step (expectation step), (2) The M-step (maximization step).
- Maximum a posteriori inference (MAP): an alternative form of inference that computes the single most likely value of the missing variables, rather than to infer the entire distribution over their possible values.
- The core idea behind variational learning is that we can maximize $\mathcal{L}$ over a restricted family of distributions q, which should be chosen so that it is easy to compute $\mathbb{E}_q \log p(\boldsymbol{h}, \boldsymbol{v})$. A typical way to do this is to introduce assumptions about how q factorizes.
- Using approximate inference as part of a learning algorithm affects the learning process, and this in turn affects the accuracy of the inference algorithm.
  Specifically, the training algorithm tends to adapt the model in a way that makes the approximating assumptions underlying the approximate inference algorithm become more true.
- The wake-sleep algorithm draws samples of both $\boldsymbol{h}$ and $\boldsymbol{v}$ from the model distribution. The inference network can then be trained to perform the reverse mapping: predicting which $\boldsymbol{h}$ caused the present $\boldsymbol{v}$.

20 Deep Generative Models

- We define the Boltzmann machine over a d-dimensional binary random vector $\mathbf{x} \in \{0,1\}^d$. It is an energy-based model, meaning we define the joint PD using an energy function: $P(\boldsymbol{x}) = \frac{\exp(-E(\boldsymbol{x}))}{Z}$, where $E(\boldsymbol{x})$ is the energy function, and Z is the partition function that ensures that $\sum_{\boldsymbol{x}} P(x) = 1$. The energy function and the Boltzmann machine is given by $E(x) = -\boldsymbol{x}^\top \boldsymbol{U}\boldsymbol{x} - \boldsymbol{b}^\top \boldsymbol{x}$, where $\boldsymbol{U}$ is the "weight" matrix of the model parameters and $\boldsymbol{b}$ is the vector of bias parameters.
- One interesting property of Boltzmann machines when trained with learning rules based on maximum likelihood is that the update for a particular weight connecting two units depends only on the statistics of those two units, collected under different distributions: $P_{model}(\boldsymbol{v})$ and $\hat{P}_{data}(\boldsymbol{v})P_{model}(\boldsymbol{h}|\boldsymbol{v})$.
- Restricted Boltzmann Machines (RBM) are undirected probabilistic graphical models containing a layer of observable variables and a single layer of latent variables. They can be stacked to form deeper models.

- Deep Belief Networks (DBN) are generative models with several layers of latent variables. The latent variables are typically binary, while the visible units may be binary or real. There are no intralayer connections. A DBN with l hidden layers contains l weight matrices: $\boldsymbol{W}^{(1)}, \ldots, \boldsymbol{W}^{(l)}$. It also contains $l+1$ bias vectors $\boldsymbol{b}^{(0)}, \ldots, \boldsymbol{b}^{(l)}$, with $\boldsymbol{b}^{(0)}$ providing the biases for the visible layer. It has undirected connections in the deepest layer and directed connections pointing downward between all other pairs of consecutive layers.
- Inference in a DBN is intractable because of the explaining away effect within each directed layer and the interaction between the two hidden layers that have undirected connections.
- A Deep Boltzmann Machine (DBM) is a generative model. It is an entirely undirected model, it has several layers of latent variables, and within each layer, each of the variables are mutually independent, conditioned on the variables in the neighboring layers.
  It is an energy-based model, meaning that the joint PD over the model variables is parametrized by an energy function E.
- Extremely high-dimensional inputs place great strain on the computation, memory and statistical requirements of ML models. Replacing matrix multiplication by discrete convolution with a small kernel is the standard way of solving these problems for inputs that have translation invariant spatial or temporal structure.
- Deep convolutional networks require a pooling operation so that the spatial size of each successive layer decreases.
- In the structured output scenario, we wish to train a model that can map from some input $\boldsymbol{x}$ to some output $\boldsymbol{y}$, and the different entries of $\boldsymbol{y}$ are related to each other and must obey some constraints.
- For sequence modeling, the model must estimate a PD over a sequence of variables, $p(\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(\tau)})$. Conditional Boltzmann machines can represent factors of the form $p(\mathbf{x}^{(t)} | \mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(\tau)})$ in order to accomplish this task.
- Traditional NN implement a deterministic transformation of some input variables $\boldsymbol{x}$. When developing generative models, we often wish to extend NN to implement stochastic transformations of $\boldsymbol{x}$. One straightforward way to do this is to augment the NN with extra inputs $\boldsymbol{z}$ that are sampled from some simple PD. The NN can then continue to perform deterministic computation internally, but the function $f(\boldsymbol{x}, \boldsymbol{z})$ will appear stochastic to an observer who does not have access to $\boldsymbol{z}$. Provided that f is continuous and differentiable, we can then compute the gradients necessary for training using back-propagation as usual.
- Sigmoid belief networks are a simple directed graphical model with a specific conditional PD. In general, we can think of it as having a vector of binary states $\boldsymbol{s}$, with each element of the state influenced by its ancestors: $p(s_i) = \sigma(\sum_{j<i} W_{j,i} s_j + b_i)$.
  Its most common structure is one that is divided into many layers, with ancestral sampling proceeding through a series of many hidden layers and then ultimately generating the visible layer.
- A differentiable generator network transforms samples of latent variables $\mathbf{z}$ to samples $\mathbf{x}$ or to distributions over samples $\mathbf{x}$ using a differentiable function $g(\boldsymbol{z}; \boldsymbol{\theta}^{(g)}$, which is typically represented by a NN. This model class includes variational autoencoders, GANs, and techniques that train generator networks in isolation.
- Generator networks are essentially just parametrized computational procedures for generating samples, where the architecture provides the family of possible distributions to sample from and the parameters select a distribution from within that family.
- Generative adversarial networks are based on a game theoretic scenario in which the generator network must compete against an adversary. The generator network directly produces samples $\boldsymbol{x} = g(\boldsymbol{z}; \boldsymbol{\theta}^{(g)}$. Its adversary, the discriminator network, attempts to distinguish between samples drawn from the training data and samples drawn from the generator. The discriminator emits a probability value given by $d(\boldsymbol{x}; \boldsymbol{\theta}^{(d)}$, indicating the probability that $\boldsymbol{x}$ is a real training example rather than a fake sample drawn from the model.
  The learning process requires neither approximate inference nor approximation of a partition function gradient.
- Generative moment matching networks are a form of generative model based on differentiable generator networks. They are trained with a technique called moment matching. The basic idea behind moment matching is to train the generator in such a way that many of the statistics of samples generated by the model are as similar as possible to those of the statistics of the examples in the training set. In this context, a moment is an expectation of different powers of a random variable.
- Auto-regressive networks are directed probabilistic models with no latent random variables. The conditional PD in these models are represented by NN.

- Linear auto-regressive networks: the simplest form of auto-regressive network has no hidden units and no sharing of parameters or features. Each $P(x_i|x_{i-1}, \ldots, x_1$ is parametrized as a linear model. If the variables are continuous, this model is merely another way to formulate a multivariate Gaussian distribution.
- Contractive autoencoders are designed to recover an estimate of the tangent plane of the data manifold. This means that repeated encoding and decoding with injected noise will induce a random walk along the surface of the manifold.
- Generalized denoising autoencoders are specified by a denoising distribution for sampling an estimate of the clean input given the corrupted input. The Markov chain that generates from the estimated distribution consists of:
    1. Starting from the previous step $\boldsymbol{x}$, inject corruption noise, sampling $\tilde{\boldsymbol{x}}$ from $C(\tilde{\boldsymbol{x}}|\boldsymbol{x})$.
    2. Encode $\tilde{\boldsymbol{x}}$ into $\boldsymbol{h} = f(\tilde{\boldsymbol{x}})$.
    3. Decode $\boldsymbol{h}$ to obtain the parameters $\boldsymbol{\omega} = g(\boldsymbol{h})$ of $p(\mathbf{x}|\boldsymbol{\omega} = g(\boldsymbol{h})) = p(\mathbf{x}|\tilde{\boldsymbol{x}})$.
    4. Sample the next state $\boldsymbol{x}$ from $p(\mathbf{x}|\boldsymbol{\omega} = g(\boldsymbol{h})) = p(\mathbf{x}|\tilde{\boldsymbol{x}})$.
- The walk-back training procedure accelerates the convergence of generative training of denoising autoencoders. Instead of performing a one-step encode-decode reconstruction, this procedure consists of alternative multiple stochastic encode-decode steps, initialized at a training example, and penalizing the last probabilistic reconstruction (or all the reconstructions along the way).
- Generative stochastic networks (GSN) are generalizations of denoising autoencoders that include latent variables $\mathbf{h}$ in the generative Markov chain, in addition to visible variables (usually denoted $\mathbf{x}$). A GSN is parametrized by two conditional PD that specify one step of the Markov chain:
    1. $p(\mathbf{x}^{(k)}|\mathbf{h}^{(k)})$ tells how to generate the next visible variable given the current latent state.
    2. $p(\mathbf{h}^{(k)}|\mathbf{h}^{(k-1)}, \mathbf{x}^{(k-1)})$ tells how to update the latent state variable, given the previous latent state and visible variable.
- Training generative models with hidden units is a powerful way to make models understand the world represented in the given training data. By learning a model $p_{model}(\boldsymbol{x})$ and a representation $p_{model}(\boldsymbol{h}|\boldsymbol{x})$, a generative model can provide answers to many inference problems about the relationships between input variables in $\boldsymbol{x}$ and can offer many different ways of representing $\boldsymbol{x}$ by taking expectations of $\boldsymbol{h}$ at different layers of the hierarchy.

References

[1] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.