# Deep learning book Ch. 5- Machine learning basics notes

*James Chuang*

*February 24, 2017*

## Contents

Part 1 of my notes on chapter 5 of the Deep Learning Book on Machine Learning Basics (up to section 5.4).

***Machine learning***: a form of applied statistics with increased emphasis on the use of computers to statistically estimate complicated functions and a decreased emphasis on proving confidence intervals around these functions.

## 5.1 Learning Algorithms

"A computer program is said to learn from experience $E$ with respect to some class of tasks $T$ and performance measure $P$, if its performance at tasks in $T$, as measured by $P$, improves with experience $E$."

### 5.1.1 The Task, $T$

Learning is *not* the task. ***Learning*** is the means of attaining the ability to perform the task.

- E.g., in training a robot to walk, the task is walking.
    - This could be accomplished by writing a program specifying how to walk, or by programming the robot to learn how to walk.

ML tasks are usually described in how the system should process an ***example***- a collection of ***features*** that have been quantitatively measured from some object or event that we want the system to process. An example is typically represented as a vector $\mathbf{x} \in \mathbb{R}^n$, where each entry $x_i$ of the vector is another feature.

Common ML tasks:

- Classification
- Classification with missing inputs
- Regression
- Transcription: unstructured data representation $\rightarrow$ discrete, textual form
- Machine translation
- Structured output: any task where the output is a vector with important relationships between the different elements, e.g. parsing– mapping a natural language sentence into a tree describing its grammatical structure, and tagging nodes of the trees as being verbs, nouns, adverbs, etc.
- Anomaly detection
- Synthesis and sampling: generate new examples similar to the training set

- Imputation of missing values
- Denoising: predict a clean example $\mathbf{x} \in \mathbb{R}^n$ from a corrupted example $\tilde{\mathbf{x}} \in \mathbb{R}^n$
- Density estimation/ probability mass function estimation

#### 5.1.2 The Performance Measure, $P$

To evaluate a ML algorithm, need a quantitative measure of its performance. Usually this performance measure $P$ is specific to the task $T$ being carried out by the system.

For classification, classification with missing inputs, transcription tasks, we often use the ***accuracy*** of the model– the proportion of examples for which the model produces the correct output, or the ***error rate***– the proportion of examples for which the model produces an incorrect output. The error rate is the expected 0-1 loss, i.e. it is 0 for a correctly classified example and 1 otherwise.

For other tasks such as density estimation, 0-1 loss doesn't make sense. Instead, we use a performance metric that gives the model a continuous-valued score for each example, most commonly the average log-probability the model assigns to some examples.

Performance measured must be measured on a ***test set*** of data that the model has not been trained in order to tell how well the model generalizes.

#### 5.1.3 The Experience, $E$

ML algorithms– unsupervised or supervised depending on what kind of experience they are allowed to have during learning.

Most learning algorithms experience an entire ***dataset***– a collection of many examples, aka ***data points***.

***Unsupervised learning***– experience a dataset containing many features, and learn useful properties of the structure of the dataset. (Learn $p(\mathbf{x})$ by observing examples of $\mathbf{x}$).

***Supervised learning***– experience a dataset containing features, where each example has a ***label*** or ***target***. (Learn to predict $\mathbf{y}$ from $\mathbf{x}$, usually by estimating $p(\mathbf{y} \mid \mathbf{x})$ from several examples of $\mathbf{x}$ and $\mathbf{y}$).

Unsupervised learning/supervised learning are not formally defined or completely separate. From the chain rule of probability:

$$\forall \mathbf{x} \in \mathbb{R}^n, \quad p(\mathbf{x}) = \prod_{i=1}^{n} p(x_i \mid x_1, \ldots, x_{i-1})$$

, meaning that the unsupervised problem of modeling $p(\mathbf{x})$ could be split into $n$ supervised learning problems. Alternatively the supervised learning problem of learning $p(y \mid \mathbf{x})$ could be solved using unsupervised learning to learn the joint distribution $p(\mathbf{x}, y)$ and inferring:

$$p(y \mid \mathbf{x}) = \frac{p(\mathbf{x}, y)}{\sum_{y'} p(\mathbf{x}, y')}$$

Some ML algorithms do not just experience a fixed dataset. E.g. ***reinforcement learning*** algorithms interact with an environment, with feedback between the learning system and its experiences (not covered in this book).

Datasets can be commonly described with a ***design matrix*** $\mathbf{X}$, where rows are examples and columns are features. However, this only works for data where each example can be described as a vector of the same size. In supervised learning, the design matrix is paired with a vector of labels $\mathbf{y}$.

### 5.2 Capacity, Overfitting and Underfitting

What separates machine learning from optimization is that we want the ***generalization/test error*** to be low, not just the training error.

- ***Generalization error***
  - the expected value of the error on a new input, where the expectation is across different possible inputs drawn from the distribution of inputs we expect the system to encounter in practice.

To affect test set performance while only observing the training set, need to make some assumptions about how the training and test sets are collected in order to apply statistical learning theory.

The train and test data are generated by a probability distribution over datasets– the ***data generating process***. We make a set of assumptions known as the ***i.i.d assumptions***:

- the examples in each dataset are *independent* from each other
- train and test set are *identically distributed*
    - i.e. drawn from the same probability distribution as each other
    - The same underlying distribution, known as the ***data generating distribution*** $p_{\text{data}}$, generates every train and test example.

In using a ML algorithm, we sample the training set, use it to choose parameters to reduce training set error, then sample the test set. Under this process, expected test error $\geq$ expected training error. Factors determining how well a ML algorithm will perform are its ability to:

- make the training error small
- make the gap between training and test error small
- ***Underfitting***
    - when a model is unable to obtain a sufficiently low training error
- ***Overfitting***
    - when the gap between training and test error is too large

Under- vs. overfitting is controlled by a model's ***capacity*** (flexibility/degrees of freedom, etc.). ML algorithms generally perform best when their capacity is appropriate for the true complexity of the task they need to perform and the amount of training data provided.

- ***Representational capacity***
    - determined by the family of functions to choose from when varying parameters
- ***Effective capacity***
    - the actual capacity of the algorithm, limited by the imperfection of optimization

The most well-known means of quantifying model capacity is the ***Vapnik-Chervonenkis dimension***, or VC dimension. The VC dimension measures the capacity of a binary classifier, and is defined as being the largest possible value of $m$ for which there exists a training set of $m$ different $\mathbf{x}$ points that the classifier can label arbitrarily (See CS229 notes 4 for more).

### 5.3 Hyperparameters and Validation Sets

- ***hyperparameters***
    - parameters which control the behavior of the learning algorithm
    - not adapted by the learning algorithm itself
        - although, possible to design nested learning procedure where one algorithm learns the best hyperparameters for another
    - e.g.:
        - in polynomial regression, the degree of the polynomial is a *capacity* hyperparameter
        - $\lambda$ in weight decay (L2 regularization)
    - hyperparameters may be chosen and not learned because they are difficult to optimize
        - more often, it is not appropriate to learn the hyperparameter on the training set (also see discussion of bias-variance tradeoff in ESL)
            - this applies to all hyperparameters controlling model capacity (aka flexibility, degrees of freedom)
                - if learned on the training set, the hyperparameter would always be set to the maximum capacity, resulting in overfitting
        - to address this, need to set hyperparameters based on a ***validation set*** not observed during training of the algorithm
            - the validation set needs to be distinct from the test set, since no aspects of the model, including hyperparameters, should be determined based on the test set

- therefore, the validation set is a subset of of the training data

#### 5.3.1 Cross-Validation

- dividing a dataset into a fixed training and test set can be problematic if it the resulting test set is small
  - small test set implies statistical uncertainty around the estimated average test error
    - makes it difficult to claim that one algorithm performs better than another on a given task
  - alternatively, can repeating training and testing on randomly chosen subsets of the dataset
    - most commonly, $k$***-fold cross-validation*** is used:
      - split dataset into $k$ disjoint subsets
      - estimate test error by taking the average test error across $i$ trials
        - on trial $i$, the $i$-th subset is used as the test set and the other $i-1$ subsets are used as the training set
    - no unbiased estimators of the variance of these average error estimators exist, but approximations are typically used

### 5.4 Estimators, Bias, and Variance

Statistical concepts such as ***parameter estimation***, ***bias***, and ***variance*** are useful to formally characterize the machine learning concepts of ***generalization***, ***underfitting***, and ***overfitting***.

#### 5.4.1 Point Estimation

- ***point estimation***: the attempt to provide the single "best" prediction of some quantity of interest
  - e.g. a single parameter, a vector of parameters, or even a whole function
  - notation: denote estimates with hats, e.g. $\hat{\theta}$ is an estimate of the true value $\theta$
  - let $\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(m)}$ be a set of $m$ i.i.d data points
    - a ***point estimator*** or ***statistic*** is any function of the data:

$$\hat{\theta}_m = g\left(\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(m)}\right)$$

      - this definition is very general: $g$ is not required to return a value close to the true value $\theta$, or to even return a value limited to the set of allowed values of $\theta$
      - the frequentist perspective: the true parameter value $\theta$ is fixed but unknown, while $\hat{\theta}$ is a function of the data
        - since the data is drawn from a random process, any function of the data is random
          - therefore, $\hat{\theta}$ is a random variable
    - point estimation can also refer to the estimation of the relationship between input and target variables, i.e. ***function estimation***.
- ***function estimation***: trying to predict a variable $\mathbf{y}$ given an input vector $\mathbf{x}$
  - assume there is a function $f(\mathbf{x})$ describing the approximate relationship between $\mathbf{y}$ and $\mathbf{x}$
    - e.g. $\mathbf{y} = f(\mathbf{x}) + \epsilon$, where $\epsilon$ represents the part of $\mathbf{y}$ not predictable from $\mathbf{x}$
  - we are interested in approximating $f$ with a model/estimate $\hat{f}$
    - $\hat{f}$ is really just a point estimator in function space
      - e.g., linear regression can be interpreted as estimating a parameter $\mathbf{w}$ or estimating a function $\hat{f}$ mapping $\mathbf{x}$ to $y$

Now for commonly studied properties of point estimators:

#### 5.4.2 Bias

Def ***bias***:

$$\text{bias}\left(\hat{\theta}_m\right) = \mathsf{E}\left(\hat{\theta}_m\right) - \theta$$

- the expectation is over the data (seen as samples from a random variable)

- an estimator $\hat{\theta}_m$ is:
  - ***unbiased*** if bias$(\hat{\theta}_m) = 0$
    - i.e., E $\left(\hat{\theta}_m\right) = \theta$
  - ***asymptotically unbiased*** if $\lim_{m \to \infty}$ bias $\left(\hat{\theta}_m\right) = 0$
    - i.e., $\lim_{m \to \infty}$ E $\left(\hat{\theta}_m\right)$
- unbiased estimators are clearly desirable, but are not always the "best"
  - we may take an increase in bias if it results in a reduction in variance

### 5.4.3 Variance and Standard Error

How much do we expect an estimator to vary as a function of the data sample?

The ***variance*** of an estimator is simply the variance (...what an amazing insight!):

$$\begin{aligned}\text{Var}\left(\hat{\theta}\right) &= \text{E}\left[\left(\hat{\theta} - \text{E}\left[\hat{\theta}\right]\right)^2\right] \\ &= \text{E}\left[\hat{\theta}^2\right] - \text{E}^2\left[\hat{\theta}\right]\end{aligned}$$

where the random variable is the training set.

Def ***standard error***: SE $\left(\hat{\theta}\right) = \sqrt{\text{Var}\left(\hat{\theta}\right)}$.

Variance or standard error of an estimator provides a measure of how we would expect the estimate computed from data to vary as we independently resample the dataset from the underlying data generating process. It is desirable to have an estimator with low variance.

The ***standard error of the mean***:

$$\text{SE}\left(\hat{\mu}_m\right) = \sqrt{\text{Var}\left[\frac{1}{m}\sum_{i=1}^{m} x^{(i)}\right]} = \frac{\sigma}{\sqrt{m}},$$

where $\sigma^2$ is the true variance of the samples $x^i$. The standard error is often estimated by using an estimate of $\sigma$. Neither the square root of the sample variance nor the square root of the unbiased estimator of the variance provide an unbiased estimate of the standard deviation. Both approaches tend to underestimate the true standard deviation, but are still used in practice. The square root of the unbiased estimator of the variance is less of an underestimate. For large $m$, the approximation is quite reasonable.

- SEM is often useful in ML: the generalization error is often estimated by computing the sample mean of the test set error
  - the accuracy of this estimate is determined by the size of the test set
  - taking advantage of the central limit theorem (i.e., the mean is approximately distributed with a normal distribution), the standard error can be used to compute the probability that the true expectation falls in any chosen interval (think 95% confidence intervals).
    - in ML, it is common to say that an algorithm is better than another if the upper bound of the 95% CI for error of the first algorithm is less than the lower bound of the 95% CI of the other

### 5.4.4 Trading off Bias and Variance to Minimize Mean Squared Error

- bias and variance measure two different sources of error in an estimator
  - bias measures the expected deviation from the true value of the function or parameter
  - variance measures the deviation from the expected estimator value that any particular sampling of the data is likely to cause
- balancing bias and variance when choosing an estimator
  - one way to choose is by cross-validation (empirically, highly successful)

- alternatively, compare the ***mean squared error*** (MSE) of the estimates:

$$\begin{aligned}
\text{MSE} &= \mathsf{E}\left[\left(\hat{\theta}_m - \theta\right)^2\right] \\
&= \mathsf{E}\left[\left(\hat{\theta} - \mathsf{E}\left[\hat{\theta}\right] + \mathsf{E}\left[\hat{\theta}\right] - \theta\right)^2\right] \\
&= \mathsf{E}\left[\left(\hat{\theta} - \mathsf{E}\left[\hat{\theta}\right]\right)^2 + \left(\mathsf{E}\left[\hat{\theta}\right] - \theta\right)^2 + 2\left(\hat{\theta} - \mathsf{E}\left[\hat{\theta}\right]\right)\left(\mathsf{E}\left[\hat{\theta}\right] - \theta\right)\right] \\
&= \mathsf{E}\left[\left(\hat{\theta} - \mathsf{E}\left[\hat{\theta}\right]\right)^2\right] + \mathsf{E}\left[\left(\mathsf{E}\left[\hat{\theta}\right] - \theta\right)^2\right] + 2\mathsf{E}\left[\left(\hat{\theta} - \mathsf{E}\left[\hat{\theta}\right]\right)\left(\mathsf{E}\left[\hat{\theta}\right] - \theta\right)\right] \\
&= \mathsf{E}\left[\left(\hat{\theta} - \mathsf{E}\left[\hat{\theta}\right]\right)^2\right] + \left(\mathsf{E}\left[\hat{\theta}\right] - \theta\right)^2 + 2\left(\mathsf{E}\left[\hat{\theta}\right] - \mathsf{E}\left[\hat{\theta}\right]\right)\left(\mathsf{E}\left[\hat{\theta}\right] - \theta\right) \\
&= \text{Var}\left(\hat{\theta}\right) + \text{Bias}^2\left(\hat{\theta}\right)
\end{aligned}$$

- From the ***bias-variance decomposition*** above, it is clear that the MSE incorporates both the bias and the variance
  - estimators with small MSE therefore manage to keep both their bias and variance somewhat in check
- the ***bias-variance trade-off***: increasing the capacity/flexibility of a model tends to increase variance and decrease bias

#### 5.4.5 Consistency

- we are also concerned with the behaviors of estimators as the amount of training data grows
  - (weak) ***consistency***: as the number of data points $m$ in the dataset grows, the point estimates should converge to the true value of the corresponding parameters. Formally:

$$\lim_{m\to\infty} \hat{\theta}_m \xrightarrow{p} \theta$$

$\xrightarrow{p}$ indicates convergence in probability, i.e.

for any $\epsilon > 0,$

$$P\left(\left|\hat{\theta}_m - \theta\right| > \epsilon\right) \to 0 \quad \text{as} \quad m \to \infty$$

  - strong consistency refers to the *almost sure* convergence of $\hat{\theta}$ to $\theta$
    - *almost sure* convergence of a sequence of random variables $\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \ldots$ to a value $\mathbf{x}$ occurs when:

$$p\left(\lim_{m\to\infty} \mathbf{x}^{(m)} = \mathbf{x}\right) = 1$$

  - consistency ensures that the bias induced by the estimator will diminish as the number of data examples grows
    - the reverse is not true: asymptotic unbiasedness does **not** imply consistency
      - e.g. consider estimating the mean parameter $\mu$ of a normal distribution $\mathcal{N}\left(x; \mu, \sigma^2\right)$ with a dataset consisting of $m$ samples: $\left\{x^{(1)}, \ldots, x^{(m)}\right\}$
      - $x^{(1)}$, the first example of the dataset *could* be used as an estimator: $\hat{\theta} = x^{(1)}$
        - $\mathsf{E}\left[\hat{\theta}_m\right] = \theta$, so the estimator is unbiased no matter how many data points are seen
          - therefore, the estimate is asymptotically unbiased
          - however, it is **not** a consistent estimator, since it is **not** the case that $\hat{\theta}_m \to \theta$ as $m \to \infty$.

### 5.5 Maximum Likelihood Estimation

- consider a set of $m$ examples $\mathbb{X} = \left\{\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(m)}\right\}$ drawn independently from the true but unknown data generating distribution $p_{\text{data}}(\mathbf{x})$.
- let $p_{\text{model}}(\mathbf{x}; \theta)$ be a parametric family of probability distributions over the same space as $p_{\text{data}}(\mathbf{x})$, indexed by $\theta$

- in other words, $p_{\text{model}}(\mathbf{x};\theta)$ maps any configuration $\mathbf{x}$ to a real number estimating the true probability $p_{\text{data}}(\mathbf{x})$.
- the maximum likelihood estimator for $\theta$ is then defined as:

$$\begin{aligned}
\theta_{ML} &= \arg\max_{\theta} p_{\text{model}}(\mathbb{X};\theta) \\
&= \arg\max_{\theta} \prod_{i=1}^{m} p_{\text{model}}(\mathbf{x}^{(i)};\theta) && \text{samples i.i.d.} \\
&= \arg\max_{\theta} \sum_{i=1}^{m} \log p_{\text{model}}(\mathbf{x}^{(i)};\theta) && \text{to avoid underflow} \\
&= \arg\max_{\theta} \frac{1}{m}\sum_{i=1}^{m} \log p_{\text{model}}(\mathbf{x}^{(i)};\theta) && \text{argmax is unchanged by rescaling} \\
&= \arg\max_{\theta} \mathsf{E}_{\mathbf{X}\sim\hat{p}_{\text{data}}} \log p_{\text{model}}(\mathbf{x};\theta)
\end{aligned}$$

- one way to interpret MLE:
  - view it as minimizing the dissimilarity between the empirical distribution $\hat{p}_{\text{data}}$ defined by the training set and the model distribution, measured by the KL divergence:

$$D_{\text{KL}}(\hat{p}_{\text{data}} \parallel p_{\text{model}}) = \mathsf{E}_{\mathbf{X}\sim\hat{p}_{\text{data}}}[\log \hat{p}_{\text{data}}(\mathbf{x}) - \log p_{\text{model}}(\mathbf{x})]$$

  - the left term, $\log \hat{p}_{\text{data}}(\mathbf{x})$, is a function only of the data generating process, and not of the model
    - this means when we train the model to minimize the KL divergence, we need only minimize

$$-\mathsf{E}_{\mathbf{X}\sim\hat{p}_{\text{data}}}[\log p_{\text{model}}(\mathbf{x})]$$

    - , which is the same as the maximization above.
- minimizing the KL divergence corresponds exactly to minimizing the cross-entropy between the distributions.
  - we can thus see MLE as an attempt to make the model distribution match the empirical distribution $\hat{p}_{\text{data}}$, in lieu of matching the true generating distribution $p_{\text{data}}$ which we do not have direct access to.
- Maximum likelihood = minimization of the negative log-likelihood (NLL) = minimization of cross-entropy.

#### 5.5.1 Conditional Log-Likelihood and Mean Squared Error

- MLE can be generalized to the case where the goal is to estimate a conditional probability $P(\mathbf{y} \mid \mathbf{x};\theta)$ in order to predict $\mathbf{y}$ given $\mathbf{x}$ (i.e., the supervised learning situation)
- let:
  - $\mathbf{X}$ represent the input
  - $\mathbf{Y}$ represent the observed targets (labels)
- then, the conditional maximum likelihood estimator is:

$$\theta_{\text{ML}} = \arg\max_{\theta} P(\mathbf{Y} \mid \mathbf{X};\theta)$$

- if the samples are assumed to be i.i.d., then this can be decomposed:

$$\begin{aligned}
\theta_{\text{ML}} &= \arg\max_{\theta} P(\mathbf{Y} \mid \mathbf{X};\theta) \\
&= \arg\max_{\theta} \prod_{i=1}^{m} P\left(\mathbf{y}^{(i)} \mid \mathbf{x}^{(i)};\theta\right) && \text{examples i.i.d} \\
&= \arg\max_{\theta} \sum_{i=1}^{m} \log P\left(\mathbf{y}^{(i)} \mid \mathbf{x}^{(i)};\theta\right) && \text{to avoid underflow}
\end{aligned}$$

- **example**: Linear Regression is maximum likelihood estimation assuming Gaussian conditional distributions:
  - think of the model producing a conditional distribution $p(y \mid \mathbf{x})$

- assume $p(y \mid \mathbf{x})\mathcal{N}\left(y; \hat{y}\left(\mathbf{x}; \mathbf{w}\right), \sigma^2\right)$ (this assumes constant $\sigma^2$)

$$\sum_{i=1}^{m} \log p\left(y^{(i)} \mid \mathbf{x}^{(i)}; \theta\right)$$ conditional log-likelihood, assuming i.i.d examples

$$\sum_{i=1}^{m} \log\left(\frac{1}{\sqrt{2\sigma^2\pi}} \exp \frac{-\left\|\hat{y}^{(i)} - y^{(i)}\right\|^2}{2\sigma^2}\right)$$ def. Gaussian

$$-\sum_{i=1}^{m} \log \sigma - \sum_{i=1}^{m} \log\left(2\pi\right)^{\frac{1}{2}} - \sum_{i=1}^{m} \frac{\|\hat{y}^{(i)} - y^{(i)}\|^2}{2\sigma^2}$$

$$-m \log \sigma - \frac{m}{2} \log\left(2\pi\right) - \sum_{i=1}^{m} \frac{\|\hat{y}^{(i)} - y^{(i)}\|^2}{2\sigma^2}$$

- only the last term depends on the training data
  - comparing this to MSE:

$$\text{MSE}_{\text{train}} = \frac{1}{m}\sum_{i=1}^{m}\left\|\hat{y}^{(i)} - y^{(i)}\right\|^2$$

  - , we see that minimizing MSE gives the same solution as maximizing the log-likelihood above

#### 5.5.2 Properties of Maximum Likelihood

- main appeal of the maximum likelihood estimator:
  - it can be shown to be the best estimator asymptotically, in terms of its rate of convergence as the number of examples $m \to \infty$
- under appropriate conditions, MLE has the property of consistency. Conditions:
  - the true distribution $p_{\text{data}}$ must lie within the model family $p_{\text{model}}(\cdot; \theta)$
    - otherwise, no estimator can recover $p_{\text{data}}$
  - the true distribution $p_{\text{data}}$ must correspond to exactly one value of $\theta$
    - otherwise, MLE can recover the correct $p_{\text{data}}$, but will not be able to determine which value of $\theta$ was used by the data generating process
- def ***statistic efficiency***: for a fixed number of examples $m$, an estimator with greater statistic efficiency will obtain a lower generalization error than an estimator with less statistic efficiency
  - statistical efficiency is typically studied in the *parametric* case, where the goal is to identify the true parameter (as opposed to a function)
  - one way to measure how close we are to a true parameter is by the expected mean squared error:
    - the squared difference between the estimated and true parameter values, where the expectation is over $m$ training examples from the data generating distribution
    - parametric MSE decreases as $m$ increases
    - for $m$ large, the ***Cramér-Rao bound*** shows that no consistent estimator has a lower MSE than MLE
- for these reasons (consistency and efficiency), MLE is often the preferred estimator for machine learning
  - when the number of examples is small enough to yield overfitting, regularization can be used to bias MLE such that it has less variance

### 5.6 Bayesian Statistics

- ***frequentist statistics***: estimate a single value of $\theta$, then make all predictions based on that estimate
  - the true parameter $\theta$ is fixed but unknown
  - the estimate $\hat{\theta}$ is a random variable (because it is a function of a random dataset)
- ***Bayesian statistics***: consider all possible values of $\theta$ when making a prediction
  - use probability to reflect degrees of certainty of states of knowledge
  - dataset is directly observed, and so is not random

- the true parameter $\theta$ is unknown or uncertain, and is therefore a random variable
- before observing data, represent knowledge of $\theta$ using the ***prior probability distribution***, $p(\theta)$ (aka ***the prior***)
  - the prior is usually chosen to have high entropy to reflect a high degree of uncertainty in the value of $\theta$ before observing any data
    - e.g. uniform distribution, Gaussian
    - many priors instead reflect a preference for "simpler" solutions
- consider a set of data samples $\left\{x^{(1)}, \ldots, x^{(m)}\right\}$
  - the data affects our belief about $\theta$ by combining the data likelihood $p\left(x^{(1)}, \ldots, x^{(m)} \mid \theta\right)$ with the prior using Bayes' rule:

$$p\left(\theta \mid x^{(1)}, \ldots, x^{(m)}\right) = \frac{p\left(x^{(1)}, \ldots, x^{(m)} \mid \theta\right) p(\theta)}{p\left(x^{(1)}, \ldots, x^{(m)}\right)}$$

$$\text{posterior} = \text{evidence} \cdot \text{prior}$$

- the prior usually begins as a relatively uniform or Gaussian distribution with high entropy
  - observation of the data causes the posterior to lose entropy and concentrate around a few highly likely values of the parameters
- two important differences relative to maximum likelihood estimation
  - whereas MLE makes predictions using a point estimate of $\theta$, the Bayesian approach makes predictions using a full distribution over $\theta$
    - e.g., after observing $m$ examples, the predicted distribution over the next data sample $x^{(m+1)}$, is given by

$$p\left(x^{(m+1)} \mid x^{(1)}, \ldots, x^{(m)}\right) = \int p\left(x^{m+1} \mid \theta\right) p\left(\theta \mid x^{(1)}, \ldots, x^{(m)}\right) d\theta$$

    - here, each value of $\theta$ with positive probability density contributes to the prediction of the next sample, with the contribution weighted by the posterior density itself
      - whereas MLE addresses the uncertainty in a given point estimate of $\theta$ by evaluating its variance, the Bayesian approach deals with uncertainty in the estimator by integrating over the uncertainty
        - this tends to protect well against overfitting
  - second difference: the prior has an influence by shifting probability density towards regions of the parameter space that are preferred *a priori*
    - in practice, the prior often expresses a preference for models that are simpler or more smooth
- Bayesian methods typically generalize much better when training data is limited, but typically suffer from high computational cost when the number of training examples is large

### 5.6.1 Maximum A Posteriori (MAP) Estimation

- it is often desirable to have a single point estimate, even when using Bayesian methods
  - one common reason: most operations involving the Bayesian posterior for most interesting models are intractable
  - can still gain some of the benefit of the Bayesian approach (e.g. allowing the prior to influence the choice of point estimate)
- ***maximum a posteriori*** (MAP) point estimation:
  - choose the point of maximal posterior probability/probability density:

$$\begin{aligned}\theta_{\text{MAP}} &= \arg\max_{\theta} p(\theta \mid \mathbf{x}) \\ &= \arg\max_{\theta} \log p(\mathbf{x} \mid \theta) + \log p(\theta) && \text{by Bayes' rule} \\ &= \text{log-likelihood term} + \text{prior distribution term}\end{aligned}$$

  - as with full Bayesian inference, MAP Bayesian inference has the advantage of leveraging information that is brought in by the prior and cannot be found in the training data
    - this reduces the variance in the MAP point estimate (compared to the ML estimate), but at the cost of increased bias
  - many regularization strategies, e.g. ML learning with weight decay, can be interpreted as making the MAP approximation to Bayesian inference

        - this applies when the regularization consists of adding an extra term to the objective function that corresponds to $\log p(\theta)$
        - MAP Bayesian inference provides a straightforward way to design complicated yet interpretable regularization terms

## 5.7 Supervised Learning Algorithms

***supervised learning***: associate some input with some output, given a training set of example inputs $\mathbf{x}$ and outputs $\mathbf{y}$

### 5.7.1 Probabilistic Supervised Learning

- most supervised learning algorithms in this book are based on estimating $p(y \mid \mathbf{x})$
    - this can be done simply by using ML estimation to find the best parameter vector $\theta$ for a parametric family of distributions $p(y \mid \mathbf{x}; \theta)$
        - previously, we have seen that linear regression corresponds to the family

$$p(y \mid \mathbf{x}; \theta) = \mathcal{N}\left(y; \theta^T \mathbf{x}, \mathbf{I}\right).$$

        - this can be generalized to the classification scenario by defining a different family of probability distributions
            - for two-class classification, only need to specify the probability of one of the classes
            - distributions over binary variables must always have mean between 0 and 1 (to be a valid probability)
                - one solution is ***logistic regression***: squash the output of the linear function into the interval (0,1) and interpret that value as a probability:

$$p(y = 1 \mid \mathbf{x}; \theta) = \sigma\left(\theta^T \mathbf{x}\right)$$

                - there is no closed form solution for the optimal weights in logistic regression
                    - therefore, they are usually found by maximizing the log-likelihood (minimizing the negative log-likelihood) by gradient descent
        - same strategy can be applied to essentially any supervised learning problem: write down a parametric family of conditional probability distributions over the right kind of input and output variables

### 5.7.2 Support Vector Machines

- A discriminative model driven by a linear function $\mathbf{w}^T\mathbf{x}$.
    - $\mathbf{w}^T\mathbf{x} > 0 \rightarrow$ predict positive class
    - $\mathbf{w}^T\mathbf{x} < 0 \rightarrow$ predict negative class
- The ***kernel trick***: many ML algorithms can be written exclusively in terms of dot products between examples, e.g.

$$\mathbf{w}^T\mathbf{x} = \sum_{i=1}^{m} \alpha_i \mathbf{x}^T \mathbf{x}^{(i)}$$

- here, $\mathbf{x}^{(i)}$ is a training example and $\alpha$ is a vector of coefficients
    - rewriting the learning algorithm this way allows us to replace $\mathbf{x}$ by the output of a given feature function $\phi(\mathbf{x})$ and the dot product with a ***kernel*** function $k(\mathbf{x}, \mathbf{x}^{(i)}) = \phi(\mathbf{x}) \cdot \phi(\mathbf{x}^{(i)})$
    - analogous to the dot product case, we can make predictions using the function

$$f(\mathbf{x}) = \sum_i \alpha_i k\left(\mathbf{x}, \mathbf{x}^{(i)}\right)$$

    - this function is nonlinear with respect to $\mathbf{x}$, but the relationship between $\phi(\mathbf{x})$ and $f(\mathbf{x})$ is linear
        - also, the relationship between $\alpha$ and $f(\mathbf{x})$ is linear
    - the kernel-based function is exactly equivalent to preprocessing the data by applying $\phi(\mathbf{x})$ to all inputs, then learning a linear model in the new transformed space.

- the kernel trick is powerful for two reasons:
    - first, it allows us to learn models that are nonlinear as a function of $\mathbf{x}$ using convex optimization techniques that are guaranteed to converge efficiently
        - this is possible because we consider $\phi$ fixed and optimize only $\alpha$, i.e., the optimization algorithm can view the decision function as being linear in a different space
    - second, the kernel function $k$ often admits an implementation that is significantly more computationally efficient than naively constructing two $\phi(\mathbf{x})$ vectors and explicitly taking their dot product.
        - in some cases, $\phi(\mathbf{x})$ can even be infinite dimensional, which would result in an infinite computational cost for the naive, explicit approach
        - in many cases, $k(\mathbf{x}, \mathbf{x}')$ is a nonlinear, tractable function of $\mathbf{X}$ even when $\phi(\mathbf{x})$ is intractable
            - for example, construct a feature mapping $\phi(x)$ over the non-negative integers $x$
            - suppose that this mapping returns a vector containing $x$ ones followed by infinitely many zeros
            - the kernel function $k(x, x^{(i)}) = \min(x, x^{(i)})$ is exactly equivalent to the corresponding infinite-dimensional dot product.

#### 5.7.3 Other Simple Supervised Learning Algorithms

- $k$-nearest neighbors: non-parametric algorithm for classification or regression
    - because it is non-parametric, can achieve very high capacity
        - however, has high computational cost and may generalize poorly to small datasets
    - a weakness: kNN cannot learn that one feature is more discriminative than another:
        - imagine a regression task with $\mathbf{x} \in \mathbb{R}^{100}$ drawn from an isotropic Gaussian, where only a single variable $x_1$ is relevant to the output
            - the nearest neighbors of most points will be determined by the large number of features $x_2$ through $x_{100}$, not the lone feature $x_1$
            - thus, the output on small training sets will essentially be random
- decision trees: see ESL Chapter 9

### 5.8 Unsupervised Learning Algorithms

- ***unsupervised learning***: experience features without labels
    - distinction between supervised and unsupervised algorithms is not rigidly defined, because there is no objective test for distinguishing whether a value is a feature or a target
    - informally, unsupervised learning refers to most attempts to extract information from a distribution that do not require human labor to label examples
    - e.g.:
        - density estimation
        - learning to draw samples from a distribution
        - denoising data from some distribution
        - finding a manifold that the data lie near
        - clustering data
    - a classic unsupervised learning task: find the "best" representation of the data
        - "best" generally means a representation that preserves as much information about $\mathbf{x}$ while obeying some penalty or constraint aimed at keeping the representation *simpler* or more accessible than $\mathbf{x}$ itself
            - three common (but not mutually exclusive) ways to define a *simpler* representation:
                1. lower dimensional representations
                    - compress as much information about $x$ as possible in a smaller representation
                2. sparse representations
                    - embed the dataset into a representation whose entries are mostly zeros for most inputs
                    - typically requires increasing the dimensionality of the representation, s.t. the representation becoming mostly zeros does not discard too much information

        - results in an overall structure of the representation that tends to distribute data along the axes of the representation space
    3. independent representations
        - attempt to *disentangle* the sources of variation underlying the data distribution s.t. the dimensions of the representation are statistically independent

#### 5.8.1 Principal Components Analysis

- an unsupervised learning algorithm that learns a lower dimensionality representation of the data such that the elements of the representation have no linear correlation with each other
    - a first step toward the criterion of learning representations whose elements are statistically independent (full independence would also require removing nonlinear relationships)
- PCA learns an orthogonal, linear transformation of the data that projects an input $\mathbf{x}$ to a representation $\mathbf{z}$
    - can be used as a simple and effective dimensionality reduction method that preserves as much of the information in the data as possible (measured by least-squares reconstruction error)
- how PCA decorrelates the original data representation $\mathbf{X}$:
    - consider the $m \times n$-dimensional design matrix $\mathbf{X}$
        - assume that the data has a mean of zero, i.e. $\mathrm{E}[\mathbf{x}] = \mathbf{0}$
            - if this is not the case, the data can easily be centered by subtracting the mean from all samples
        - the unbiased sample covariance matrix associated with $\mathbf{X}$:

$$\mathrm{Var}[\mathbf{x}] = \frac{1}{m-1}\mathbf{X}^T\mathbf{X}$$

        - PCA will find a representation (by linear transformation) $\mathbf{z} = \mathbf{x}^T\mathbf{W}$ where $\mathrm{Var}[\mathbf{z}]$ is diagonal
        - previously, we saw that the principal components of a design matrix $\mathbf{X}$ are given by the eigenvectors of $\mathbf{X}^T\mathbf{X}$, i.e.:

$$\mathbf{X}^T\mathbf{X} = \mathbf{W}\boldsymbol{\Lambda}\mathbf{W}^T$$

        - an alternative derivation of the principal components by the singular value decomposition which shows that the principal components are the right singular vectors of $\mathbf{X}$:
            - let $\mathbf{W}$ be the right singular vectors in the decomposition $\mathbf{X} = \mathbf{U}\boldsymbol{\Sigma}\mathbf{W}^T$
            - then recover the original eigenvector equation with $\mathbf{W}$ as the eigenvector basis:

$$\begin{aligned}
\mathbf{X}^T\mathbf{X} &= \left(\mathbf{U}\boldsymbol{\Sigma}\mathbf{W}^T\right)^T\mathbf{U}\boldsymbol{\Sigma}\mathbf{W}^T \\
&= \mathbf{W}\left(\mathbf{U}\boldsymbol{\Sigma}\right)^T\mathbf{U}\boldsymbol{\Sigma}\mathbf{W}^T \\
&= \mathbf{W}\boldsymbol{\Sigma}^T\mathbf{U}^T\mathbf{U}\boldsymbol{\Sigma}\mathbf{W}^T \qquad \mathbf{U}^T\mathbf{U} = \mathbf{I} \text{ by def of SVD} \\
&= \mathbf{W}\boldsymbol{\Sigma}^2\mathbf{W}^T
\end{aligned}$$

            - The SVD is helpful to show that PCA results in a diagonal $\mathrm{Var}[\mathbf{z}]$:

$$\begin{aligned}
\mathrm{Var}[\mathbf{x}] &= \frac{1}{m-1}\mathbf{X}^T\mathbf{X} \\
&= \frac{1}{m-1}\mathbf{W}\boldsymbol{\Sigma}^2\mathbf{W}^T
\end{aligned}$$

            - If we take $\mathbf{z} = \mathbf{x}^T\mathbf{W}$, we can ensure that the covariance of $\mathbf{z}$ is diagonal as required:

$$\begin{aligned}
\mathrm{Var}[\mathbf{z}] &= \frac{1}{m-1}\mathbf{Z}^T\mathbf{Z} \\
&= \frac{1}{m-1}\mathbf{W}^T\mathbf{X}^T\mathbf{X}\mathbf{W} \\
&= \frac{1}{m-1}\mathbf{W}^T\mathbf{W}\boldsymbol{\Sigma}^2\mathbf{W}^T\mathbf{W} \\
&= \frac{1}{m-1}\boldsymbol{\Sigma}^2 \qquad \mathbf{W}^T\mathbf{W} = \mathbf{I} \text{ by def. of SVD}
\end{aligned}$$

        - i.e., when we project the data $\mathbf{x}$ to $\mathbf{z}$ via the linear transformation $\mathbf{W}$, the resulting representation has a diagonal covariance matrix (given by $\boldsymbol{\Sigma}^2$), which implies that the individual elements of $\mathbf{z}$ are mutually uncorrelated
            - this property means that PCA is an example of a representation that attempts to ***disentangle the unknown factors of variation*** underlying the data
            - disentangling feature dependencies that are more complicated require more than what can be done with a simple linear transformation

#### 5.8.2 $k$-means Clustering

- $k$-means clustering: divide the training set into $k$ different clusters of examples that are near each other
    - think of the algorithm as providing a $k$-dimensional one-hot code vector $\mathbf{h}$ represented an input $\mathbf{x}$
        - if $\mathbf{x}$ belongs to cluster $i$, then $h_i = 1$ and all other entries of representation $\mathbf{h}$ are zero
        - this is an extreme example of a sparse representation:
            - it loses many of the benefits of a distributed representation, but still confers some statistical advantages (e.g., naturally conveys the idea that all examples in the same cluster are similar to each other)
            - has the computational advantage that the entire representation may be captured by a single integer
            - later, will develop more flexible sparse representations where more than one entry can be non-zero for each input $\mathbf{x}$
    - the $k$-means algorithm:
        - initialize $k$ centroids $\left\{\mu^{(1)}, \ldots, \mu^{(k)}\right\}$ to different values
        - alternate between:
            - assign each training example to cluster $i$, where $i$ is the nearest centroid $\mu^{(i)}$
            - update each centroid $\mu^{(i)}$ to the mean of all training examples $x^j$ assigned to cluster $i$
    - there is no single criterion that measures how well a clustering of the data corresponds to the real world
        - can measure certain properties such as the average Euclidean distance from a cluster centroid to the members of the clusters
            - however, do not know how well the cluster assignments correspond to properties of the real world
                - example: cluster red cars, grey cars, red trucks, grey trucks
            - because of this, we may prefer a distributed representation to a one-hot representation
                - having many attributes reduces the burden on the algorithm to guess which single attribute we care about

### 5.9 Stochastic Gradient Descent

- ***Stochastic gradient descent*** (SGD): an extension of gradient descent that powers nearly all of deep learning.
- the recurring problem: large training sets are necessary for good generalization, but large training sets are also more computationally expensive.
- cost functions often decompose as a sum over training examples of some per-example loss function
    - e.g. the negative conditional log-likelihood of the training data can be written as

$$J(\theta) = \mathbb{E}_{\mathbf{x}, y \sim \hat{p}_{\text{data}}} L(\mathbf{x}, y, \theta) = \frac{1}{m} \sum_{i=1}^{m} L\left(\mathbf{x}^{(i)}, y^{(i)}, \theta\right)$$

    - here $L$ is the per-example loss $L(\mathbf{x}, y, \theta) = -\log p(y \mid \mathbf{x}; \theta)$.
- for these additive cost functions, gradient descent requires computing

$$\nabla_\theta J(\theta) = \frac{1}{m} \sum_{i=1}^{m} \nabla_\theta L\left(\mathbf{x}^{(i)}, y^{(i)}, \theta\right)$$

- the computational cost of this operation is $O(m)$
    - as the training set size grows to billions of examples, the time to take a single gradient step becomes prohibitively long
- the insight of stochastic gradient descent: the gradient is an expectation, which can be estimated using a small set of samples

    - specifically, on each step of the algorithm, we can sample a ***minibatch*** of examples $\mathbb{B} = \left\{\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{\left(m'\right)}\right\}$ drawn uniformly from the training set
    - minibatch size $m'$ is typically chosen to be a relatively small number of examples, ranging from 1 to a few hundred
        - crucially, $m'$ is usually held fixed as the training set size $m$ grows
        - thus, we may fit a training set with billions of examples using updates computed on only a hundred examples.
- the estimate of the gradient is formed as

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

- using examples from the minibatch $\mathbb{B}$. SGD then follows the estimated gradient downhill, with learning rate $\epsilon$:

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

### 5.10 Building an ML algorithm

Nearly all deep learning algorithms combine specification of a ***dataset***, a ***cost function***, an ***optimization procedure***, and a ***model***. E.g. linear regression:

- dataset: $\mathbf{X}$ and $\mathbf{y}$
- cost function: $J(\mathbf{w}, b) = -\mathsf{E}_{\mathbf{X}, y \sim \hat{p}_{\text{data}}} \log p_{\text{model}}\left(y \mid \mathbf{x}\right)$
- model: $p_{\text{model}}(y \mid \mathbf{x}) = \mathcal{N}(y; \mathbf{x}^T \mathbf{w} + b, 1)$
- optimization algorithm: solve for zero gradient with normal equations

The cost function typically includes at least one term that causes the learning process to perform statistical estimation. The most common cost function is the negative log-likelihood (minimizing NLL = maximum likelihood estimation).

The cost function may also include additional terms, such as regularization terms like weight decay (aka L2, or ridge regression).

If the model is nonlinear, then the cost function usually cannot be optimized in closed form. This requires an iterative numerical optimization procedure, e.g. gradient descent.

In some cases, the cost function may be a function that we cannot actually evaluate, for computational reasons. In these cases, we can still approximately minimize the function using iterative numerical optimization if we have some way of approximating its gradients.

### 5.11 Challenges Motivating Deep Learning

- traditional ML algorithms failed to generalize well on AI tasks such as speech or object recognition
- generalizing to new examples becomes exponentially more difficult when working with high-dimensional data
    - mechanisms used to achieve generalization in traditional ML are insufficient to learn complicated functions in high-dimensional space
    - these spaces also often impose high computational costs
- hence, the motivation for deep learning:

#### 5.11.1 The Curse of Dimensionality

- the ***curse of dimensionality***: many ML problems become exceedingly difficult when the number of dimensions in the data is high (see ESL Ch. 2.5 for a more rigorous explanation)
    - of particular concern: the number of possible distinct configurations of a set of variables increases exponentially with the number of variables

### 5.11.2 Local Constancy and Smoothness Regularization

- in order to generalize well, ML algorithms need to be guided by prior beliefs about the kind of function they should learn
  - previously, have seen these priors incorporated as explicit beliefs in the form of probability distributions over parameters of the model
  - more informally, may also discuss prior beliefs as directly influencing the *function* itself and only indirectly acting on the parameters via their effect on the function
  - additionally, informally discuss prior beliefs as being expressed implicitly, by choosing algorithms that are biased toward choosing some class of functions over another, even though these biases may not be expressed (or be possible to express) in terms of a probability distribution representing our degree of belief in various functions
- among the most widely used of the "implicit" priors: the ***smoothness/local constancy prior***:
  - the function we learn should not change very much within a small region
  - simple ML algorithms rely exclusively on this prior to generalize well
    - as a result, they fail to scale to the statistical challenges involved in solving AI-level tasks
    - deep learning introduces additional (explicit and implicit) priors in order to reduce generalization error on sophisticated tasks
- why the smoothness prior alone is insufficient for AI-level tasks
  - there are many different ways to implicitly or explicitly express the smoothness/local constancy prior
  - they all encourage the learning process to learn a function $f^*$ that satisfies the condition:

$$f^*(\mathbf{x}) \approx f^*(\mathbf{x}+\epsilon)$$

    - for most configurations $\mathbf{x}$ and small changes $\epsilon$
      - i.e., if we know a good answer for an input $\mathbf{x}$, then that answer is probably good in the neighborhood of $\mathbf{x}$
        - if we have several good answers in some neighborhood we would combine them (by some form of averaging or interpolation) to produce an answer that agrees with as many of them as possible
    - an extreme example of the local constancy approach: $k$NN learning algorithms
    - more kernel machines interpolate between training set outputs associated with nearby training examples
      - an important class of kernels: ***local kernels***
        - $k(\mathbf{u},\mathbf{v})$ large when $\mathbf{u}=\mathbf{v}$ and decreases as $\mathbf{u}$ and $\mathbf{v}$ grow further apart from each other
        - i.e., a similarity function performing template matching between a test example $\mathbf{x}$ and the training examples $\mathbf{x}^{(i)}$
    - in general, local constancy methods require $O(k)$ examples to distinguish $O(k)$ regions in input space
    - assuming local constancy will not allow a learner to represent a complex function that has many more regions to be distinguished than the number of training examples
      - e.g., imagine a checkerboard
    - the smoothness assumption and the associated non-parametric learning algorithms work well as long as there are enough examples for the learning algorithm to observe high points on most peaks and low points in most valleys of the true function to be learned
      - this is generally true when the function is smooth enough and varies in few dimensions
      - in high dimensions, even a very smooth function can change smoothly but in a different way along each dimension
        - additionally, if the function behaves differently in different regions, it can become extremely complicated to describe with a set of training examples
  - how to generalize well for these complicated functions
    - key insight: a very large number of regions (e.g. $O\left(2^k\right)$), can be defined with $O(k)$ examples *as long as we introduce some dependencies between the regions via additional assumptions about the underlying data generating distribution*
      - many DL algorithms provide implicit or explicit assumptions that are reasonable for a broad range of AI tasks to capture these advantages
    - other approaches make stronger, task-specific assumptions
      - e.g. could solve the checkerboard task by assuming the target function is periodic
        - usually, we do not include such strong, task-specific assumptions into neural networks s.t. they can generalize to a much wider variety of structures
          - AI tasks have structure too complex to be limited to simple, manually encoded properties

- the core idea in deep learning:
  - assumption that the data was generated by the ***composition of factors/features***, potentially at multiple hierarchical levels
  - these mild assumptions allow an exponential gain in the relationship between the number of examples and the number of regions that can be distinguished

### 5.11.3 Manifold Learning

- ***manifold***: a connected region
  - mathematically, a set of points, associated with a neighborhood around each point
  - from any given point, the manifold locally appears to be a Euclidean space
    - e.g. we experience the world as a 2-D plane, but it is in fact a spherical manifold in 3-D space. ("but is it really?"" -flat earth society)
    - the definition of a neighborhood around each point implies the existence of transformations that can be applied to move on the manifold from one position to a neighboring one
      - e.g. walking north, south, east, west
  - in ML, "manifold" tends to be used more loosely to designate a connected set of points that can be approximated well by only considering a small number of degrees of freedom, or dimensions, embedded in a higher-dimensional space
    - each dimension corresponds to a local direction of variation
    - in ML, the dimensionality of the manifold is allowed to vary from one point to another
      - this often happens when a manifold intersects itself
        - e.g. a figure eight is a 1-D manifold in most places, but 2-D at the intersection in the center
  - many ML problems seem hopeless if we expect the algorithm to learn functions with interesting variations across all of $\mathbb{R}^n$
    - ***manifold learning*** algorithms surmount this by assuming that most of $\mathbb{R}^n$ consists of invalid inputs, and that interesting variations in the output of the learned function occur only in directions that lie on the manifold, or with interesting variations happening only when moving from one manifold to another
      - the assumption that the data lie along a low-dimensional manifold may not always be correct or useful
        - it can be argued that in the context of AI tasks, such as those that involve processing images, sounds, or text, the manifold assumption is at least approximately correct
          - first argument: the probability distribution over images, text strings, and sounds that occur in real life is highly concentrated
            - uniform noise essentially never resembles structured inputs from these domains
          - however, concentrated probability distributions are not sufficient to show that the data lies on a reasonably small number of manifolds- must also show that the examples encountered as connected to each other by other examples, with each example surrounded by other highly similar examples that may be reached by applying transformations to traverse the manifold
            - second argument: we can imagine such neighborhoods and transformations- e.g. for images, think of dimming or brightening the lights, moving or rotating objects, altering the colors on the surfaces of objects, etc.
              " ...the manifold of images of human faces may not be connected to the manifold of images of cat faces" - lmao
        - when the data lies on a low-dimensional manifold, it can be natural for machine learning algorithms to represent the data in terms of coordinates on the manifold, rather than in terms of the coordinates in $\mathbb{R}^n$
          - extracting these manifold coordinates it challenging, but shows promise