Chapter 1
Introduction to Machine Learning

I believe that at the end of the century the use of words and general educated opinion will have altered so much that one will be able to speak of machines thinking without expecting to be contradicted.

`Alan Turing, 1950`

Abstract This chapter will introduce some of the building blocks of machine learning. Specifically, it will touch upon

- Difference between machine learning, statistics and deep learning.
- A discussion on bias and variance and how they are related to underfitting and overfitting.
- Different ways to address underfitting and overfitting, including regularization.
- The need for optimization and the gradient descent method.
- Model hyperparameters and different hyperparameters search methods.
- Quantifying and measuring loss functions of a model.

1.1 Machine Learning

Machine learning is a sub-domain of artificial intelligence (AI), which makes a system automatically discover (learn) the statistical structure of the data and convert those representations (patterns) to get closer to the expected output. The process of **learning** is improved by a measure of the feedback, which compares the computed output to the expected output. Unlike *expert systems*, the machine learning system is not explicitly programmed; it automatically searches for patterns within the *hypothesis space* and, uses the feedback signal to correct those patterns.

A. Ghatak, *Deep Learning with R*,
https://doi.org/10.1007/978-981-13-5850-0_1

To enable a machine learning algorithm to work effectively on real-world data, we need to feed the machine a full range of features and possibilities to train on.

A typical machine learning workflow includes the following:

- Training the model on a training data set, tuning the model on a development set and testing the model on an unseen test data set.
- Trying out the above on different yet, appropriate algorithms using proper performance metrics.
- Selecting the most appropriate model.
- Testing the model on real-world data. If the results are not upto the speed, repeat the above by revaluating the data and/or model, possibly with different evaluation metrics.

We define data sets which we use in machine learning as follows:

- Training set: is the data on which we learn the algorithm.
- Development set: is the data on which we tune the hyperparameters of the model.
- Test set: is the data we use to evaluate the performance of our algorithm.
- Real-world set: is the data on which our selected model will be deployed.

Having data sets from different distributions can have different outcomes on the evaluation metrics of some or all of the data sets. The evaluation metrics may also differ if the model does not fit the respective data sets. We will explore these aspects during model evaluation at a later section.

1.1.1 Difference Between Machine Learning and Statistics

In machine learning, we feed labeled data in batches into the machine learning model, and the model incrementally improves its structural parameters by examining the loss, i.e., the difference between the actual and predicted values. This loss is used as a feedback to an optimization algorithm to iteratively adjust the structural parameters. Training a conventional machine learning model, therefore, consists of feeding input data to the model to train the model to learn the "best" structural parameters of the model. While machine learning focusses on predicting future data and evaluation of the model, statistics is focussed on the inference and explanation cum understanding of the phenomenon [10].

While a machine learning model is an algorithm that can learn from data without being explicitly programmed, statistical modeling is a mathematical representation of the relationship between different variables. Machine learning is a sub-domain of AI whereas, statistics is a sub-domain of mathematics.

1.1.2 Difference Between Machine Learning and Deep Learning

Traditional machine learning techniques find it hard to analyze data with complex spatial or sequence dependencies, and those that require analyzing data which need a large amount of feature engineering like problems related to computer vision and speech recognition. Perception or disambiguation is the awareness, understanding, and interpretation of information through the senses. In reference [11], deep learning is proven to be better than conventional machine learning algorithms for these "perceptual" tasks, but not yet proven to be better in other domains as well.

In deep learning, we use a similar procedure as in machine learning, by transforming the input data by a linear combination of the weights and bias through each layer, by a process known as *forward propagation*, which computes the predicted values. A loss function compares the actual and predicted values and computes a distance score between these values, thereby capturing how well the network has done on a batch of input data. This score is then used as a feedback to adjust the weights incrementally toward a direction that will lower the loss score for the current input data through an optimization algorithm. This update is done using a *backpropagation* algorithm, using the chain rule to iteratively compute gradients for every layer.

A training loop consists of a single forward propagation, calculation of the loss score and backpropagation through the layers using an optimizer to incrementally change the weights. Typically, we would need many iterations over many examples of input data to yield weight values that would minimize the loss to optimal values.

A deep learning algorithm can be thought of as a large-scale parametric model, because it has many layers and scales up to a large amount of input data. Below is the summarization of a grayscale image classification deep learning model having 1.2 million parameters. A model to classify color images can have close to 100 million parameters.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_1 (Dense)              (None, 100)               1228900
_________________________________________________________________
dense_2 (Dense)              (None, 50)                5050
_________________________________________________________________
dense_3 (Dense)              (None, 15)                765
_________________________________________________________________
dense_4 (Dense)              (None, 1)                 16
=================================================================
Total params: 1,234,731
Trainable params: 1,234,731
Non-trainable params: 0
_________________________________________________________________
```

1.2 Bias and Variance

The three major sources of error in machine/deep learning models are irreducible error, bias, and variance.

(a) **Irreducible Error**—is the error which cannot be reduced and its occurrence is due to the inherent randomness present in the data.
(b) **Bias Error**—creeps in deep learning when we introduce a simpler model for a data set which is far more "complex". To avoid bias error, we would need to increase the capacity (consider a more complex model) to match up with the complexity present in the data.
(c) **Variance**—on the contrary is present in deep learning models if we consider an overly complex algorithm for a relatively less "complex" task or data set. To avoid variance in models, we would need to increase the size of the data set.

1.3 Bias–Variance Trade-off in Machine Learning

In machine learning, we can define an appropriate trade-off point between bias and variance by discovering the appropriate model complexity with respect to the data; at which point, an increase in bias results in reduction of variance and vice-versa, as shown by the `darkgray` circle in Fig. 1.1. Any model complexity short of this trade-off point will result in an underfitted model and those beyond the trade-off point, will result in an overfitted model.

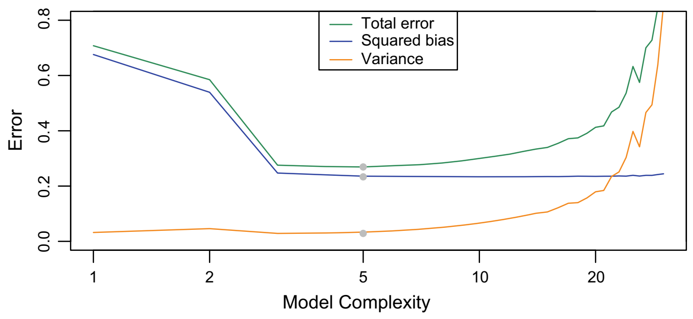

Fig. 1.1 The squared bias keeps decreasing and the variance keeps increasing as the model complexity goes up. The total error starts decreasing till it reaches a point where the model complexity is optimal, and thereafter it starts increasing. The optimal balance between the squared bias and the variance is represented by the dark gray circle

Bayes error, is the lowest possible prediction error that can be achieved and is the same as irreducible error. If we have an exact knowledge of the data distribution, then for a random process, there would still exist some errors. In statistics, the optimal error rate is also known as the Bayes error rate. In machine learning, the Bayes error can be considered as the irreducible error.
In deep learning, since we deal with problems related to human perception and therefore, we consider **Human-Level Error**, as the lowest possible error.

1.4 Addressing Bias and Variance in the Model

One of the important things to understand before we know that our model suffers from bias, variance, or a combination of both bias and variance, is to understand the *optimal error rate*. We have seen earlier that the optimal error is the Bayes error rate. The Bayes error is the best theoretical function for mapping x to y, but it is often difficult to calculate this function.

In deep learning, since we are mostly dealing with problems related to human perceptions like vision, speech, language, etc., an alternative to the Bayes error could be the human-level performance because humans are quite good at tasks related to vision, speech, language, etc. So we can ask this question—how well does our neural network model compare to human-level performance?

Based on the above, we can state the following:

(a) If there exists a large gap between human-level error and the training error, then the model is too simple in relation to human-level performance and it is a bias problem.
(b) If there is a small gap between the training error and human-level error and, a large difference between training error and validation error, it implies that our model has got high variance.
(c) If the training error is less than the human-level error, we cannot say for sure if our model suffers from bias or variance or a combination of both. There are a couple of domains where our model may surpass human-level performance and they include product recommendations, predicting transit time (Google maps), etc.

The actions we can take to address bias and variance are the following:

(a) Bias

 (i) Train a more complex model.
 (ii) Train for a longer duration.
 (iii) Use better optimization algorithms—Momentum, RMSProp, Adam, etc.
 (iv) Use different neural network architectures.
 (v) Carry out better hyperparameter search.

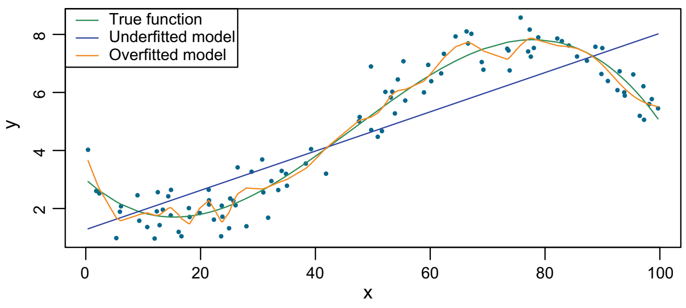

Fig. 1.2 Underfitting and Overfitting: An underfitted model is unable to capture the sinusoidal pattern of the data which is represented by the straight-line linear regression model; an overfitted model has memorized the training data and the noise and is therefore not able to generalize to the data

(b) Variance

- (i) Use more data to train.
- (ii) Use regularization methods, ℓ_2, dropout, etc.
- (iii) Use data augmentation
- (iv) Use different neural network architectures.
- (v) Carry out better hyperparameter search.

1.5 Underfitting and Overfitting

Underfitting occurs when there is high bias present in the model and therefore cannot capture the trend in the data. Overfitting occurs when the model is highly complex resulting in the model capturing the noise present in the data, rather than capturing the trend in the data.

This is illustrated in Fig. 1.2, where true model is a fourth polynomial model represented by the green curve. The overfitted model oscillates wildly and the underfitted model can barely capture the trend in the data set.

1.6 Loss Function

The objective function of every algorithm is to reduce the loss $\mathcal{L}(w)$, which can be represented as

$$\mathcal{L}(w) = Measure\ of\ fit + Measure\ of\ model\ complexity$$

In machine learning, loss is measured by the sum of

- the **fit** of the model.
- the **complexity** of the model.

The measure of the model's fit is determined by

- the *MSE* (Mean Squared Error) for regression,
- *CE* (Classification Error) for classification.

Higher the model complexity, higher is the propensity for the model to capture the noise in the data by ignoring the signal. A measure of the model complexity is determined by

- the sum of the absolute values of the structural parameters of the model (ℓ_1 *regularization*),
- the sum of the squared values of the structural parameters of the model (ℓ_2 *regularization*).

1.7 Regularization

Regularization is the process used to reduce the complexity of the model.

In deep learning, we will use the ℓ_2 regularization technique

$$\ell_2 = w_0^2 + w_1^2 + \cdots + w_n^2 = \sum_{i=0}^{n} w_i^2 = \|w\|_2^2$$

Our objective is to select the model's structural parameters w_i, such that we minimize the loss function $\mathcal{L}(w)$, by using a weight decay regularization parameter λ on the ℓ_2-norm of the parameters such that, it penalizes the model for larger values of the parameters.

$$\mathcal{L}(w) = \mathit{Error\ metric} + \lambda \|w\|_2^2$$

When λ is zero, there is no regularization; values greater than zero force the weights to take a smaller value. When $\lambda = \infty$, the weights become zero.

Since we are trying to minimize the loss function with respect to the weights, we need an optimizing algorithm, to arrive at the optimal value of the weights. One of the optimizing algorithms which we use in machine learning is the gradient descent. Other popular optimization algorithms used in deep learning is discussed in Sect. 5.

Regularization, significantly reduces the variance of a deep learning model, without any substantial increase in its bias.

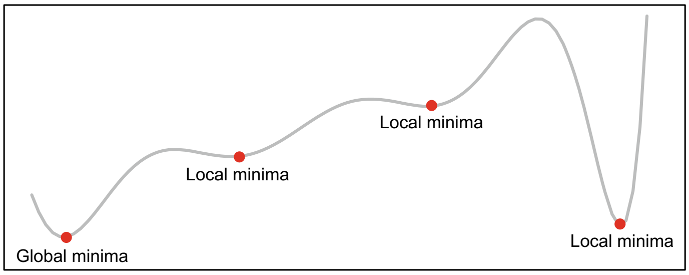

Fig. 1.3 Gradient descent involves reaching the global minima, sometimes traversing through many local minima

1.8 Gradient Descent

Gradient descent is an optimization algorithm used to find the values of weights and biases of the model with an objective to minimize the loss function.

Analogically, this can be visualized as “descending” to the lowest point of a valley (the lowest error value), known as the global minima, as shown in Fig. 1.3. In the process, the descent also has a possibility of getting stuck at a local minima and there are ways to work around and come out of the local minima by selecting the correct learning-rate hyperparameter.

The cost of the model is the sum of all the losses associated with each training example. The gradient of each parameter is then calculated by a process known as **batch gradient descent**. This is the most basic form of gradient descent because we compute the cost as the sum of the gradients of the entire batch of training examples.

In Fig. 1.4, we consider the loss function with respect to one weight (in reality we need to consider the loss function with respect to all weights and biases of the model). To move down the slope, we need to tweak the parameters of the model by calculating the gradient ($\frac{\partial \mathcal{J}}{\partial W}$) of the loss function, which will always point toward the nearest local minima.

Having found the magnitude and direction of the gradient, we now need to nudge the weight by a **hyperparameter** known as the learning rate and then update the value of the new weight (toward the direction of the minima). Mathematically, for one iteration for an observation j and learning rate α, the weight update at time step $(t + 1)$ can be written as follows:

$$\begin{gathered}\text{Repeat till convergence}\\ w_j^{(t+1)} = w_j^{(t)} - \alpha \frac{\partial \mathcal{J}}{\partial w}\end{gathered} \tag{1.8.1}$$

Figure 1.5 is a contour plot showing the steps of a gradient descent optimization for a `sigmoid` activation neural network having an input with two features. In a real model, the inputs may have many features and the loss function may have multiple local minima. The gradients are calculated for all the parameters while iterating over

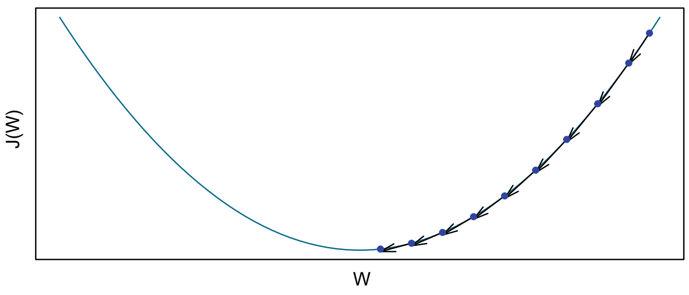

Fig. 1.4 Gradient descent: Rolling down to the minima by updating the weights by the gradient of the loss function

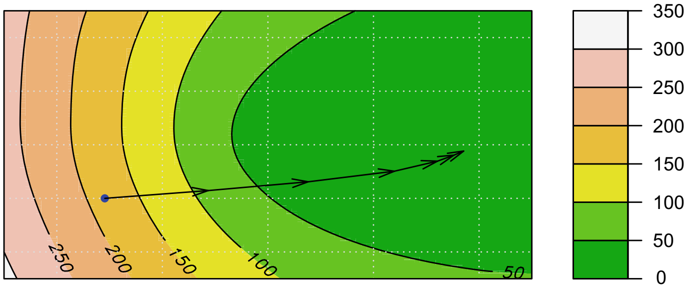

Fig. 1.5 A contour plot showing the cost contours of a sigmoid activation neural network and the cost minimization steps using the gradient descent optimization function

all the training examples. This makes things way harder to visualize, as the plot will have many multiple dimensions.

For large size data sets, batch gradient descent optimization algorithm can take a long time to compute because it considers all the examples in the data to complete one iteration. In such cases, we can consider a subset of the training examples and split our training data into mini-batches. In the **mini-batch gradient descent** method, the parameters are updated based on the current mini-batch and we continue iterating over all the mini-batches till we have seen the entire data set. The process of looking at all the mini-batches is referred to as an **epoch**.

When the mini-batch size is set to one we perform an update on a single training example, and this is a special case of the mini-batch gradient descent known as **stochastic gradient descent**. It is called "stochastic" because it randomly chooses a single example to perform an update till all the examples in the training data set have been seen.

Ideally, in any optimization algorithm, our objective is to find the global minimum of the function, which would represent the best possible parameter values. Depending on where we start in the parameter space, it is likely that we may encounter local minima and saddlepoints along the way. Saddlepoints are a special case of local minima where the derivatives in orthogonal directions are both zero. While dealing

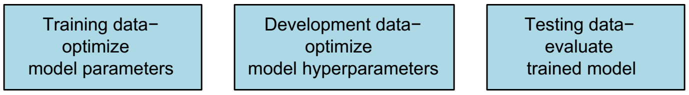

Fig. 1.6 The training process involves optimizing the cost to arrive at the optimal structural parameters using the training data, optimization of the hyperparameters using the development data, and evaluating the final model using the testing data

with high-dimensional spaces, the chances of discovering the global minima is quite low (Fig. 1.3).

There are many approaches to overcome this problem—use a moving average gradient (Momentum), adaptively scaling the learning-rate in each dimension according to the exponentially weighted average of the gradient (RMSProp) and by using a combination of the moving average gradient and adaptive learning rates (Adam). These methods have been discussed in Sects. 5.3.6. Figure 1.6 is a schematic of how the model deals with different sections of the available data.

1.9 Hyperparameter Tuning

During the model training process, our training algorithm handles three categories of data:

- The input training data—is used to configure our model to make accurate predictions from the unseen data.
- Model parameters—are the weights and biases (structural parameters) that our algorithm learns with the input data. These parameters keep changing during model training.
- Hyperparameters—are the variables that govern the training process itself. Before setting up a neural network, we need to decide the number of hidden layers, number of nodes per layer, learning rate, etc. These are configuration variables and are usually constant during a training process.

Structural parameters like the weights and biases are learned from the input dataset, whereas the optimal hyperparameters are obtained by leveraging multiple available datasets.

A hyperparameter is a value required by our model which we really have very little idea about. These values can be learned mostly by trial and error.

While the model (structural) parameters are optimized during the training process by passing the data through a cycle of the training operation, comparing the resulting prediction with the actual value to evaluate the accuracy, and adjusting the parameters till we find the best parameter values; the hyperparameters are "tuned" by running multiple trials during a single training cycle, and the optimal hyperparameters are chosen based on the metrics (accuracy, etc), while keeping the model parameters unchanged. In both the cases, we are modifying the composition of our model to find the best combination of the model parameters and hyperparameters.

1.9.1 Searching for Hyperparameters

In standard supervised learning problems, we try to find the best hypothesis in a given space for a given learning algorithm. In the context of hyperparameter optimization, we are interested in minimizing the validation error of a model parameterized by a vector of weights and biases, with respect to a vector of hyperparameters. In hyperparameter optimization, we try to find a configuration so that the optimized learning algorithm will produce a model that generalizes well to new data. To facilitate this, we can opt for different search strategies:

- A *grid search* is sometimes employed for hyperparameter tuning. With grid search, we build a model for each possible combination of all of the hyperparameter values in the grid, evaluate each model and select the hyperparameters, which give the best results.
- With *random search*, instead of providing a discrete set of values to explore, we consider a statistical distribution of each hyperparameter, and the hyperparameter values are randomly sampled from the distribution.
- A *greedy search* will pick whatever is the most likely first parameter. This may not be a good idea always.
- An *exact search* algorithm as its name signifies, searches for the exact value. Algorithms belonging to this search methodology are `BFS` *(Breadth First Search)* and `DFS` *(Depth First Search).*
- In deep learning for Recurrent Neural Networks, we apply what is known as the *Beam Search.*

We also have the Bayesian optimization technique belonging to a class of `(SMBO)` *Sequential Model-Based Optimization* [8] algorithms, wherein we use the results of our search using a particular method to improve the search for the next method. The hyperparameter metric is the objective function during hyperparameter tuning. Hyperparameter tuning finds the optimal value of this metric (a numeric value), specified by the user. The user needs to specify whether this metric needs to be maximized or minimized.

For most data sets, only a few of the hyperparameters really matter but different hyperparameters are important for different data sets. This makes the grid search a poor choice for configuring algorithms for new data sets. [6]

1.10 Maximum Likelihood Estimation

Maximum likelihood estimation (*MLE*) is a method used to determine parameter values of the model. The parameter values are found such that they maximize the likelihood that the process described by the model produces the data that were actually observed. We would, therefore, pick the parameter values which maximizes the likelihood of our data. This is known as the maximum likelihood estimate. Mathematically, we define it as

$$\arg\max_{\theta} P(y \mid x; \theta) \tag{1.10.1}$$

To arrive at the likelihood over all observations (assuming they are identical and independent of one another, i.e., i.i.d.), we take the product of the probabilities as

$$\arg\max_{\theta} \prod_{i=1}^{n} P(y^{(i)} \mid x^{(i)}; \theta) \tag{1.10.2}$$

We assume that each data point is generated independently of the others, because if the events generating the data are independent, then the total probability of observing all our data is the product of observing each data point individually (the product of the marginal probabilities).

We know that the probability density of observing a single data point generated from a Gaussian distribution is given by

$$P(x; \mu, \sigma) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-(\frac{(x-\mu)^2}{2\sigma^2})} \tag{1.10.3}$$

To obtain the parameters μ and σ^2 of a Gaussian distribution, we need to solve the following maximization problem:

$$\max_{\mu,\sigma^2} log(\mu, \sigma^2;\ x_1, \ldots, x_n)$$

The first-order conditions to obtain a maximum are

$$\frac{\partial}{\partial\mu} log(\mu, \sigma^2;\ x_1, \ldots, x_n) = 0$$

and,

$$\frac{\partial}{\partial\sigma^2} log(\mu, \sigma^2;\ x_1, \ldots, x_n) = 0$$

The partial derivative of the log-likelihood with respect to the mean is

$$\begin{aligned}
\frac{\partial}{\partial\mu}(logP(x;\,\mu,\sigma)) &= \frac{\partial}{\partial\mu}\;log(\mu,\sigma^2;\;x_1,\ldots,x_n) \\
&= \frac{\partial}{\partial\mu} log\left\{\frac{1}{\sqrt{2\pi\sigma^2}}e^{-(\frac{(x-\mu)^2}{2\sigma^2})}\right\} \\
&= \frac{\partial}{\partial\mu}\left\{-\frac{n}{2}ln(2\pi) - \frac{n}{2}ln(\sigma^2) - \frac{1}{2\sigma^2}\sum_{j=1}^{n}(x_j-\mu)^2\right\} \\
&= \frac{1}{\sigma^2}\sum_{j=1}^{n}(x_j-\mu) \\
&= \frac{1}{\sigma^2}\left\{\sum_{j=1}^{n}x_j - n\mu\right\}
\end{aligned} \tag{1.10.4}$$

Solving for μ in Eq. 1.10.4

$$\mu = \frac{1}{n}\sum_{j=1}^{n}x_j \tag{1.10.5}$$

Similarly, the partial derivative of the log-likelihood with respect to the variance is

$$\begin{aligned}
\frac{\partial}{\partial\sigma^2}(logP(x;\,\mu,\sigma)) &= \frac{\partial}{\partial\sigma^2}\;log(\mu,\sigma^2;\;x_1,\ldots,x_n) \\
&= \frac{\partial}{\partial\sigma^2} log\left\{\frac{1}{\sqrt{2\pi\sigma^2}}e^{-(\frac{(x-\mu)^2}{2\sigma^2}}\right\} \\
&= \frac{\partial}{\partial\sigma^2}\left\{-\frac{n}{2}log(2\pi) - \frac{n}{2}log(\sigma^2) - \frac{1}{2\sigma^2}\sum_{j=1}^{n}(x_j-\mu)^2\right\} \\
&= -\frac{n}{2\sigma^2} - \left(\frac{1}{2}\sum_{j=1}^{n}(x_j-\mu)^2\right)\left(-\frac{1}{(\sigma^2)^2}\right) \\
&= \frac{1}{2\sigma^2}\left(\frac{1}{\sigma^2}\sum_{j=1}^{n}(x_j-\mu)^2 - n\right)
\end{aligned} \tag{1.10.6}$$

Assuming $\sigma^2 \neq 0$, we can solve for σ^2

$$\sigma^2 = \frac{1}{n}\sum_{j=1}^{n}(x_j-\hat{\mu})^2 \tag{1.10.7}$$

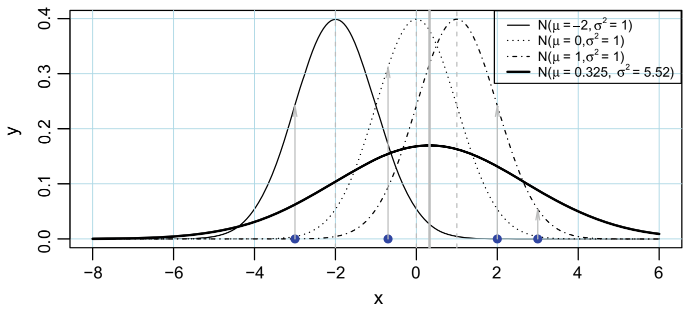

Fig. 1.7 The four data points and the possible Gaussian distributions from which they were drawn. The distributions are normally distributed with ($\mu = -2, \sigma^2 = 1$), ($\mu = 0, \sigma^2 = 1$), and ($\mu = 1, \sigma^2 = 1$). The maximum likelihood estimation predicts that the data points belong to the Gaussian distribution defined with ($\mu = 0.325, \sigma^2 = 5.52$) by maximizing the probability of observing the data

Let us consider four data points on the x-axis. We would want to know which curve was most likely responsible, for our observed data points. We will use *MLE* to find the values of μ and σ^2 to discover the gaussian distribution, that best fits our data (Fig. 1.7).

```r
points <- c(-3, -0.7, 2, 3)
(mu = sum(points)/length(points)) # Refer Eq. 1.10.5
```

```
[1] 0.325
```

```r
(var = (1/length(points))*sum((points-mu)^2)) # ReferEq. 1.10.7
```

```
[1] 5.516875
```

1.11 Quantifying Loss

1.11.1 The Cross-Entropy Loss

When we develop a machine learning model for probabilistic classification, we try to map the model inputs to probabilistic predictions by an iterative method of training the model (by adjusting the model's parameters), so that our predictions are close to the true probabilities.

To arrive at our computed predictions, which are close to the true predictions, we need to reduce the loss (*Mean Squared Error/Classification Error*). In this section, we will try to define this error in terms of the Cross-Entropy(*CE*) Loss. But before that, we will revisit *MLE*, define *Entropy* of a model, its follow up to *Cross-Entropy* and, its relation to the *Kulback–Liebler* Divergence.

In Sect. 1.10, we have seen that the maximum likelihood is that which maximizes the product of the probabilities of a data distribution, which most likely gave rise to our data. Since logarithms reduce potential underflow, due to very small likelihoods, they convert a product into a summation and finally, the natural logarithmic function being a monotonic transformation (if the value on the x-axis increases, the value on the y-axis also increases); it is but natural to apply the *log* function to our likelihood Eq. (1.10.2), to obtain the log-likelihood

$$\begin{aligned} log\ P(y \mid x; \theta) &= log\ \prod_{i=1}^{n} P(y^{(i)} \mid x^{(i)}; \theta) \\ &= \sum_{i}^{n} log\ P(y^{(i)} \mid x^{(i)}; \theta) \end{aligned} \tag{1.11.1}$$

1.11.2 Negative Log-Likelihood

The maximazition of the negative log-likelihood of the estimated data distribution reduces the error of our machine learning model.

In **Linear Regression**, we maximize the log-likelihood (Eq. 1.11.1) of the Gaussian distribution. We may recall that $\theta^T x = \mu$

$$\begin{aligned} log[P(y \mid x; \theta)] &= \sum_{i}^{n} log\ P(y^{(i)} \mid x^{(i)}; \theta) \\ &= \sum_{i}^{n} log \frac{1}{\sigma\sqrt{2\pi}} exp^{-\frac{(y^{(i)} - \theta^T x^{(i)})^2}{2\sigma^2}} \\ &= \sum_{i}^{n} log \frac{1}{\sigma\sqrt{2\pi}} + \sum_{i}^{n}\ log \left(exp^{-\frac{(y^{(i)} - \theta^T x^{(i)})^2}{2\sigma^2}} \right) \\ &= n\ log \frac{1}{\sigma\sqrt{2\pi}} - \frac{1}{2\sigma^2} \sum_{i=1}^{n} (y^{(i)} - \theta^T x^{(i)})^2 \\ &= k_1 - k_2 \sum_{i=1}^{n} (y^{(i)} - \theta^T x^{(i)})^2 \end{aligned} \tag{1.11.2}$$

We can state that minimizing the negative log-likelihood is equivalent to maximizing the likelihood estimation since

$$\arg \underset{x}{max}(x) = \arg \underset{x}{min}(-x)$$

Maximizing Eq. 1.11.2 implies that we need to minimize the mean- squared error between the observed y and the predicted $\hat{y}$; therefore, minimizing the negative log-likelihood of our data with respect to θ is also the same as minimizing the mean squared error.

In **Logistic Regression**, we define $\phi = \frac{1}{1+exp^{-\theta^T x}}$, and the negative log-likelihood can be written as

$$\begin{aligned} -log\ P(y \mid x; \theta) &= -log \prod_{i=1}^{n} (\phi^{(i)})^{y^{(i)}} (1-\phi^{(i)})^{(1-y^{(i)})} \\ &= -\sum_{i=1}^{n} log((\phi^{(i)})^{y^{(i)}} (1-\phi^{(i)})^{(1-y^{(i)})}) \\ &= -\sum_{i=1}^{n} y^{(i)} log(\phi^{(i)}) + (1-y^{(i)}) log(1-\phi^{(i)}) \end{aligned} \tag{1.11.3}$$

In Eq. 1.11.3, minimizing the negative log-likelihood of the data with respect to θ is the same as minimizing the binary log loss (binary cross-entropy, discussed in the following section) between the observed y values and the predicted probabilities thereof.

In a **Multinoulli Distribution**, the negative log-likelihood can be written as

$$\begin{aligned} -log\ P(y \mid x; \theta) &= -log \prod_{i=1}^{n} \prod_{k=1}^{K} \pi_k^{y_k} \\ &= \sum_{i=1}^{n} \sum_{k=1}^{K} y_k log(\pi_k) \end{aligned} \tag{1.11.4}$$

In Eq. 1.11.4 minimizing the negative log-likelihood of the data with respect to θ is the same as minimizing the multi-class log loss (categorical cross-entropy), between the observed y values and the predicted probabilities thereof.

1.11.3 Entropy

Entropy in heat engineering and classical thermodynamics, is a measure of "disorder" based on the second law of thermodynamics, which states that a system's entropy never decreases spontaneously. From the concept of thermodynamics, it is the (log of) number of microstates or microscopic configurations. Intuitively, if the particles inside a system have many possible positions to move around, then the system has high entropy, and if they stay rigid, then the system has low entropy.

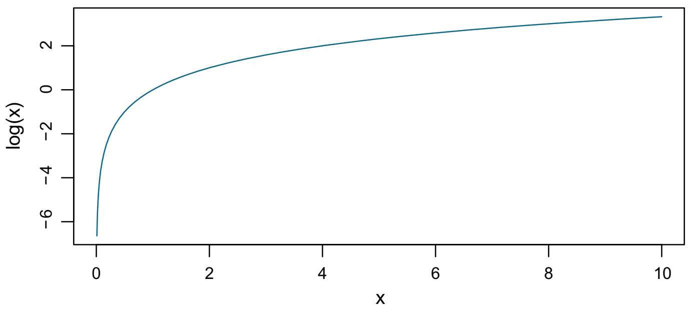

Fig. 1.8 The natural logarithm is a monotonically increasing function

In **Information theory**, entropy is a measure of uncertainty involved in making a prediction. Intuitively, we can describe entropy as how "surprised" would we be of the outcome, after we have made our initial prediction. If we consider an unfair coin that actually turns up a head 99% of the time it is tossed, we would be indeed very surprised if a particular toss turns up a tail. If we can average out the amount of surprise, the mean value of surprise is a measure for how uncertain we are. This measure of uncertainty is called entropy. Entropy therefore defines randomness; it is like describing how unpredictable something is.

If we consider a set of possible events with known probabilities of occurrence being $y_1; y_2; \dots y_n$, as per [9] entropy is a means to find a measure of how much 'choice' is involved in the selection of the event or of how uncertain we are of the outcome. This measure, $H(y_1; y_2; \dots y_n)$ would then be defined by the following properties

- H should be continuous in the y_i.
- If all the y_i are equal, $y_i = \frac{1}{n}$, then H should be a monotonic increasing function of n. The natural logarithm is a monotonically increasing function, implying that if the value on the x-axis increases, the corresponding value on the y-axis also increases (see Fig. 1.8).
- If a choice be broken down into two successive choices, the original H should be the weighted sum of the individual values of H.

The only H satisfying the three above assumptions is of the form

$$H = \sum_{i=1}^{n} y_i log(y_i) \tag{1.11.5}$$

Interpretation of Entropy

If we consider two events with probabilities p and 1-p the entropy H can be written as

$$H = -[p\ log(p) + (1-p)\ log(1-p)] \tag{1.11.6}$$

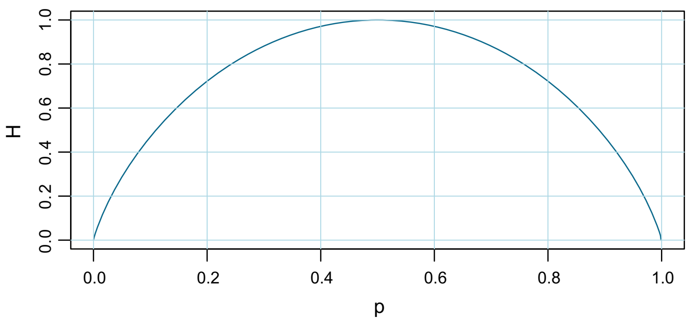

Fig. 1.9 Plot of Entropy with probabilities p and $1 - p$

Entropy can be easily interpreted from Fig. 1.9 as

- When $p = 1$, $H = 0$ implies that we are certain of the outcome and if we are not certain of the outcome, H is positive. Intuitively when $p = 0$, an event never occurs, it cannot contribute to the entropy, as it is never expected to occur and therefore $H = 0$.
- H is maximum when all the p_i are equal, i.e., $\frac{1}{n}$. This is the most uncertain situation.

Entropy is the weighted average of the log probability of the possible events, which measures the uncertainty present in their probability distribution. The higher the entropy, the less certain are we about the value.

1.11.4 Cross-Entropy

Cross-entropy (CE) loss, or log-loss, measures the performance of a classification model whose output has a probability value between 0 and 1. As the predicted probability diverges from the actual value the CE loss consequently increases. Therefore say, predicting a probability of 0.021 when the true observed value is 1 would result in a high CE loss. A perfect model, therefore, should have a CE loss of 0.

How close is the predicted probability distribution to the true distribution, i.e., to find the cross-entropy loss, we use

$$H(y, \hat{y}) = \sum_i y_i log\frac{1}{\hat{y}_i} = -\sum_i y_i log(\hat{y}_i) \tag{1.11.7}$$

where, $\sum_i y_i$ is the true probability distribution and $\sum_i \hat{y}_i$ is the computed probability distribution

Let us go through an example where we have three training items with the following computed outputs and target outputs

Table 1.1 Target values and computed probabilities of a neural network

	Class A	Class B	Class C
Target	0.000	1.000	0.000
Computed probability	0.128	0.719	0.153

For the data in Table 1.1, we can calculate the cross-entropy, using Eq. 1.11.7 as equal to 0.475, and it gives us a measure of how far away is our prediction from the true distribution.

```
H = -(0.0 * log2(0.128) + 1.0 * log2(0.719) + 0.0 * log2(0.153))
H
```

```
[1] 0.4759363
```

In **binary classification**, where the number of output class labels are two, CE is calculated as

$$CE_{Loss} = H(y, \hat{y}) = -(y\ log(\hat{y}) + (1 - y) log(1 - \hat{y}))\ \text{—}\textit{for binary classification} \tag{1.11.8}$$

In **multi-class classification**, where the number of output class labels are more than two, i.e., n-labels, CE is calculated as

$$CE_{Loss} = H(y, \hat{y}) = -\sum_{1}^{n} y_i\ log(\hat{y}_i)\ \text{—}\textit{for multi-class classification} \tag{1.11.9}$$

1.11.5 Kullback–Leibler Divergence

Kullback–Leibler[1] Divergence (KL Divergence) from $\hat{y}$ to y is the difference between CE (Eq. 1.11.7) and entropy (Eq. 1.11.5). It quantifies the additional uncertainty in y introduced by using $\hat{y}$ to approximate y

$$\begin{aligned} KL(y\ ||\ \hat{y}) &= \sum_i y_i\ log \frac{1}{\hat{y}_i} - \sum_i y_i\ log \frac{1}{y_i} \\ &= H(y, \hat{y}) - H(y) \\ &= \sum_i y_i log \frac{y_i}{\hat{y}_i} \end{aligned} \tag{1.11.10}$$

[1]Named after Solomon Kullback and Richard Leibler in 1951.

In information theory, the *KL* Divergence measures the number of bits required on average to encode symbols from y according to $\hat{y}$. The *KL* Divergence is never negative, and it is zero only when y and $\hat{y}$ are the same. Minimizing CE is the same as minimizing the *KL* Divergence from $\hat{y}$ to y.

1.11.6 Summarizing the Measurement of Loss

Let us consider the empirical true distribution to be p and, the predicted distribution (the model we are trying to optimize) to be q.

From the above discussions, we can state that **KL divergence** allows us to measure the difference between two probability distributions.

- The **entropy**, $H(p)$ of a distribution p, gives us an estimate of the uncertainty present in the distribution or, how certain can we be of the outcome.
- The **Cross-Entropy** $H(p, q)$ between two distributions p and q, quantifies the difference between the two probability distributions; i.e., how close is the predicted distribution to the true distribution. In machine learning classification problems, the Cross-Entropy loss, i.e., log-loss, measures the Cross-Entropy between the empirical distribution of the labels (given the inputs) and the distribution predicted by the model. In binary classification, the cross-entropy is proportional to the **negative log- likelihood**, and therefore minimizing the negative log-likelihood is equivalent to maximizing the likelihood.
- The difference, i.e., $KL(p \parallel q)$, measures the average number of extra bits per message, whereas $H(p, q)$ measures the average number of total bits per message.
- If the empirical distribution p is fixed it would be equivalent to say that we are minimizing the *KL* divergence between the empirical distribution and the predicted distribution. As we can see in the expression above, the two are related by the additive term $H(p)$, i,e, the entropy of the empirical distribution. Because p is fixed, $H(p)$ does not change with the parameters of the model, and can be disregarded in the loss function. This may not be true where p may also vary.

1.12 Conclusion

We have touched upon the basic facets, which define a machine learning algorithm. **Machine learning** belongs to the domain of AI and it endeavors to develop *models* (statistical programs) from exposure to training data. The process of training a machine learning algorithm results in a model, and is therefore called a learning algorithm.

Deep learning is another subset of AI, where *models* represent geometric transformations over many different layers.

In both machine learning and deep learning, the real knowledge are the structural parameters, i.e., the weights and biases of the model. The common ground, therefore, is to discover the best set of parameters, which will define the best model.