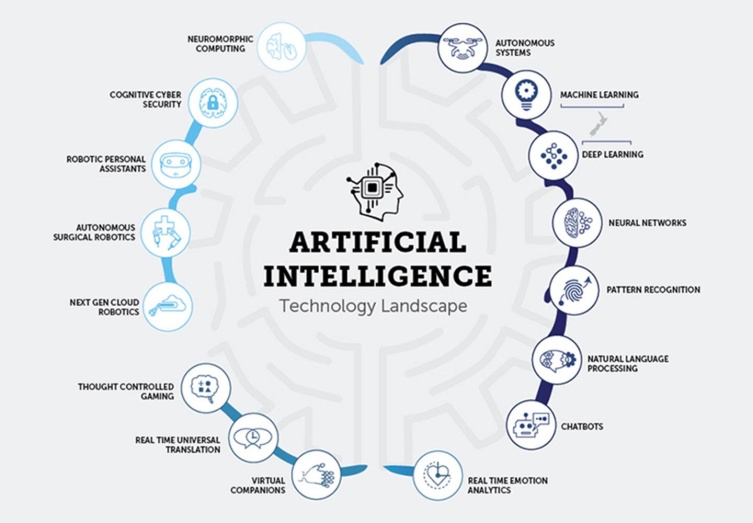
***UNIT I: [Text Book 2]***

* **Fundamentals of Deep Learning:** 4
  + **Artificial Intelligence, 4**
  + **History of Machine learning: 14**
    - **Probabilistic Modeling, 14**
    - **Early Neural Networks, 14**
    - **Kernel Methods, 15**
    - **Decision Trees, 16**
    - **Random forests and Gradient Boosting Machines, 16**
* **Fundamentals of Machine Learning: 93**
  + **Four Branches of Machine Learning, 94**
    - Supervised Learning 94
    - Unsupervised Learning 94
    - Self Supervised Learning 94
    - Reinforcement Learning 95
  + **Evaluating Machine learning Models, 97**
    - Training, Validation, and Test Sets 97
  + **Overfitting and Underfitting 104**
    - Reducing the Network’s Size 104
    - Adding Weight Regularization 107
    - Adding Dropout 109

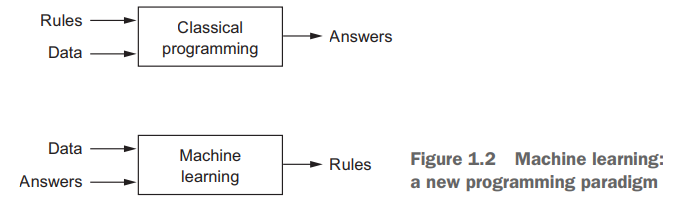
**Fundamentals of Deep Learning**

**Artificial intelligence** is the simulation of human intelligence processes by machines, especially computer systems. Specific applications of AI include expert systems, natural language processing, speech recognition and machine vision.

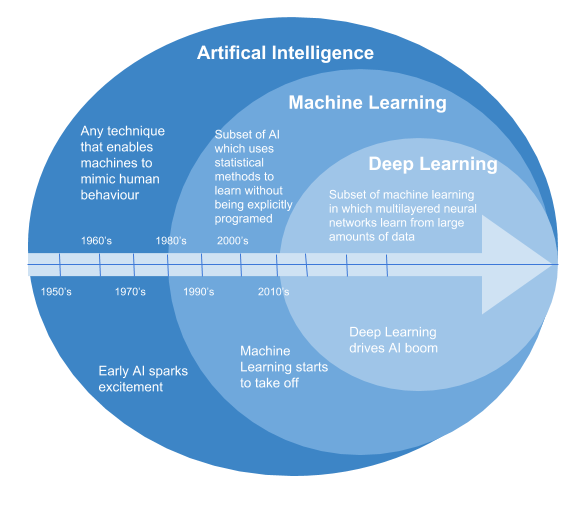


**Machine learning** is a branch of artificial intelligence (AI) and computer science which focuses on the use of data and algorithms to imitate the way that humans learn, gradually improving its accuracy.

With machine learning, humans input data as well as the answers expected from the data, and outcome the rules. These rules can then be applied to new data to produce original answers. A machine-learning system is trained rather than explicitly programmed.

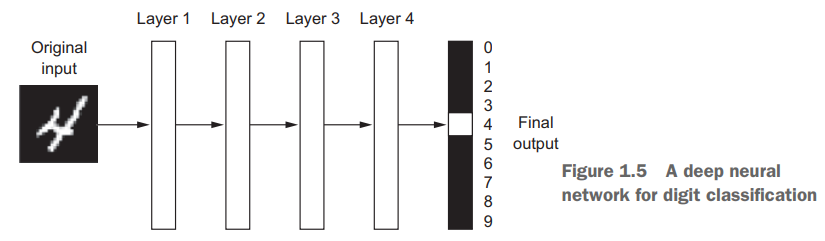


**Deep learning** is a method in artificial intelligence (AI) that teaches computers to process data in a way that is inspired by the human brain. Deep learning models can recognize complex patterns in pictures, text, sounds, and other data to produce accurate insights and predictions.

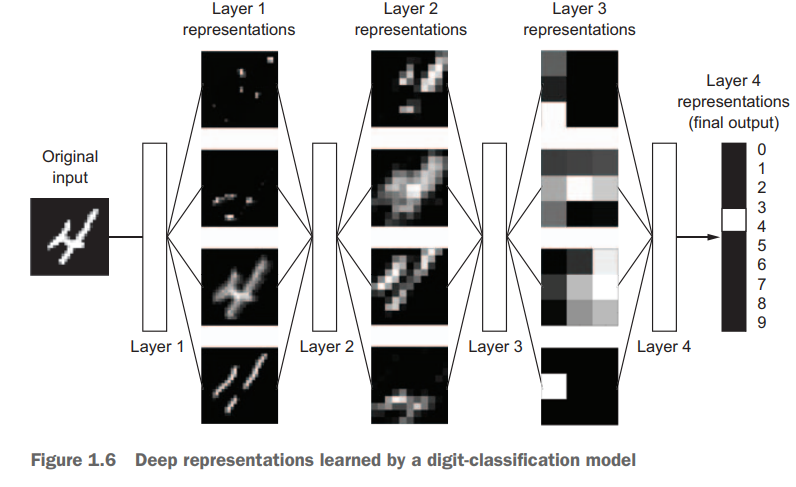


Deep learning is a specific subfield of machine learning: a new take on learning representations from data that puts an emphasis on learning successive layers of increasingly meaningful representations. The deep in deep learning isn’t a reference to any kind of deeper understanding achieved by the approach; rather, it stands for this idea of successive layers of representations. How many layers contribute to a model of the data is called the depth of the model. Other appropriate names for the field could have been layered representations learning and hierarchical representations learning. Modern deep learning often involves tens or even hundreds of successive layers of representations— and they’re all learned automatically from exposure to training data. Meanwhile, other approaches to machine learning tend to focus on learning only one or two layers of representations of the data; hence, they’re sometimes called shallow learning.

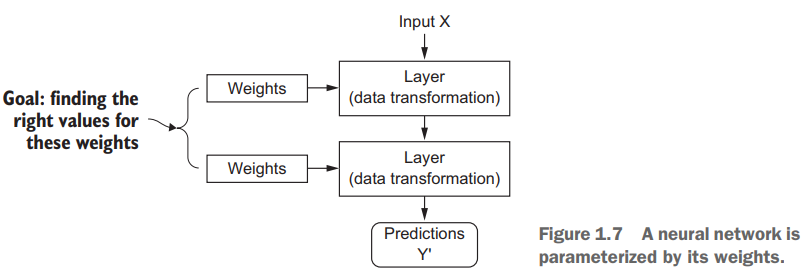
What do the representations learned by a deep-learning algorithm look like? Let’s examine how a network several layers deep (see figure 1.5) transforms an image of a digit in order to recognize what digit it is



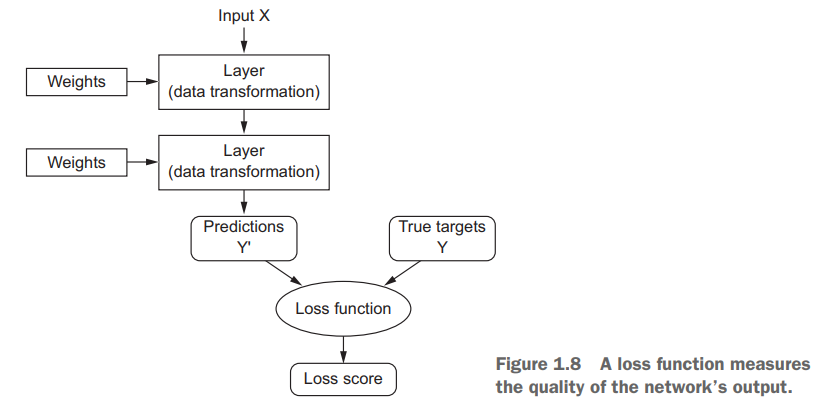
As you can see in figure 1.6, the network transforms the digit image into representations that are increasingly different from the original image and increasingly informative about the final result. You can think of a deep network as a multistage information-distillation operation, where information goes through successive filters and comes out increasingly purified (that is, useful with regard to some task)



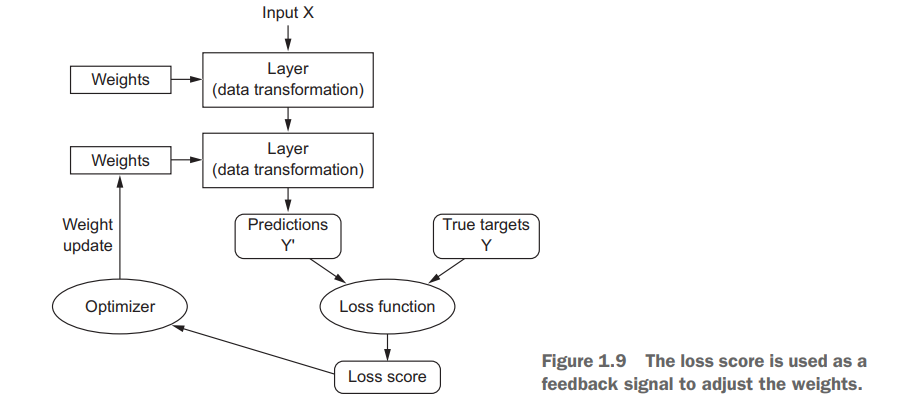
The specification of what a layer does to its input data is stored in the layer’s weights, which in essence are a bunch of numbers. In technical terms, we’d say that the transformation implemented by a layer is parameterized by its weights (see figure 1.7). (Weights are also sometimes called the parameters of a layer.) In this context, learning means finding a set of values for the weights of all layers in a network, such that the network will correctly map example inputs to their associated targets. But here’s the thing: a deep neural network can contain tens of millions of parameters. Finding the correct value for all of them may seem like a daunting task, especially given that modifying the value of one parameter will affect the behavior of all the others!



To control something, first you need to be able to observe it. To control the output of a neural network, you need to be able to measure how far this output is from what you expected. This is the job of the loss function of the network, also called the objective function. The loss function takes the predictions of the network and the true target (what you wanted the network to output) and computes a distance score, capturing how well the network has done on this specific example (see figure 1.8).



The fundamental trick in deep learning is to use this score as a feedback signal to adjust the value of the weights a little, in a direction that will lower the loss score for the current example (see figure 1.9). This adjustment is the job of the optimizer, which implements what’s called the Backpropagation algorithm: the central algorithm in deep learning.



Initially, the weights of the network are assigned random values, so the network merely implements a series of random transformations. Naturally, its output is far from what it should ideally be, and the loss score is accordingly very high. But with every example the network processes, the weights are adjusted a little in the correct direction and the loss score decreases. This is the training loop, which, repeated a sufficient number of times (typically tens of iterations over thousands of examples), yields weight values that minimize the loss function. A network with a minimal loss is one for which the outputs are as close as they can be to the targets: a trained network.

In particular, deep learning has achieved the following breakthroughs, all in historically difficult areas of machine learning:

* Near-human-level image classification.
* Near-human-level speech recognition
* Near-human-level handwriting transcription
* Improved machine translation
* Improved text-to-speech conversion
* Digital assistants such as Google Now and Amazon Alexa
* Near-human-level autonomous driving
* Improved ad targeting, as used by Google, Baidu, and Bing
* Improved search results on the web
* Ability to answer natural-language questions
* Superhuman Go playing

**History of Machine learning: Probabilistic Modeling,**

Probabilistic modeling is the application of the principles of statistics to data analysis. It was one of the earliest forms of machine learning, and it’s still widely used to this day. One of the best-known algorithms in this category is the Naive Bayes algorithm.

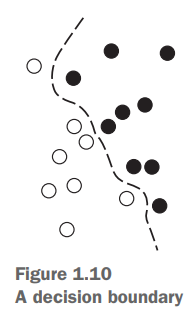
Naive Bayes is a type of machine-learning classifier based on applying Bayes’ theorem while assuming that the features in the input data are all independent (a strong, or “naive” assumption, which is where the name comes from). This form of data analysis predates computers and was applied by hand decades before its first computer implementation (most likely dating back to the 1950s). Bayes’ theorem and the foundations of statistics date back to the eighteenth century, and these are all you need to start using Naive Bayes classifiers.

**Early Neural Networks**

The first successful practical application of neural nets came in 1989 from Bell Labs, when Yann LeCun combined the earlier ideas of convolutional neural networks and backpropagation, and applied them to the problem of classifying handwritten digits. The resulting network, dubbed LeNet, was used by the United States Postal Service in the 1990s to automate the reading of ZIP codes on mail envelopes.

**Kernel Methods,**

Kernel methods are a group of classification algorithms, the best known of which is the support vector machine (SVM). The modern formulation of an SVM was developed by Vladimir Vapnik and Corinna Cortes in the early 1990s at Bell Labs and published in 1995. SVMs aim at solving classification problems by finding good decision boundaries (see figure 1.10) between two sets of points belonging to two different categories. A decision boundary can be thought of as a line or surface separating your training data into two spaces corresponding to two categories. To classify new data points, you just need to check which side of the decision boundary they fall on.

****

SVMs proceed to find these boundaries in two steps:

**1** The data is mapped to a new high-dimensional representation where the decision boundary can be expressed as a hyperplane (if the data was twodimensional, as in figure 1.10, a hyperplane would be a straight line).

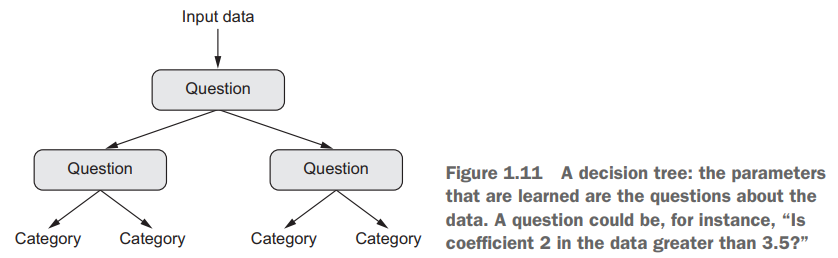
**2** A good decision boundary (a separation hyperplane) is computed by trying to maximize the distance between the hyperplane and the closest data points from each class, a step called maximizing the margin. This allows the boundary to generalize well to new samples outside of the training dataset.

The technique of mapping data to a high-dimensional representation where a classification problem becomes simpler may look good on paper, but in practice it’s often computationally intractable. That’s where the kernel trick comes in (the key idea that kernel methods are named after). Here’s the gist of it: to find good decision hyperplanes in the new representation space, you don’t have to explicitly compute the coordinates of your points in the new space; you just need to compute the distance between pairs of points in that space, which can be done efficiently using a kernel function.

A kernel function is a computationally tractable operation that maps any two points in your initial space to the distance between these points in your target representation space, completely bypassing the explicit computation of the new representation. Kernel functions are typically crafted by hand rather than learned from data—in the case of an SVM, only the separation hyperplane is learned.

**Decision Trees, Random forests and Gradient Boosting Machines**

Decision trees are flowchart-like structures that let you classify input data points or predict output values given inputs (see figure 1.11). They’re easy to visualize and interpret.

****

Businesses use these supervised machine learning techniques like Decision trees to make better decisions and make more profit. Decision trees have been around for a long time and also known to suffer from bias and variance. You will have a large bias with simple trees and a large variance with complex trees.

***Ensemble methods***, which combines several decision trees to produce better predictive performance than utilizing a single decision tree. The main principle behind the ensemble model is that a group of weak learners come together to form a strong learner.

Let’s talk about few techniques to perform ensemble decision trees:

1. Bagging

2. Boosting

**Bagging***(Bootstrap Aggregation) is used when our goal is to reduce the variance of a decision tree. Here idea is to create several subsets of data from training sample chosen randomly*with replacement*. Now, each collection of subset data is used to train their decision trees. As a result, we end up with an ensemble of different models. Average of all the predictions from different trees are used which is more robust than a single decision tree.*

***Random Forest*** is an extension over bagging. It takes one extra step where in addition to taking the random subset of data, it also takes the random selection of features rather than using all features to grow trees. When you have many random trees. It’s called Random Forest .

Let’s look at the steps taken to implement Random forest:

**1**. Suppose there are N observations and M features in training data set. First, a sample from training data set is taken randomly with replacement.

**2**. A subset of M features are selected randomly and whichever feature gives the best split is used to split the node iteratively.

**3**. The tree is grown to the largest.

**4**. Above steps are repeated and prediction is given based on the aggregation of predictions from n number of trees.

**Advantages of using Random Forest technique:**

* Handles higher dimensionality data very well.
* Handles missing values and maintains accuracy for missing data.

**Disadvantages of using Random Forest technique:**

* Since final prediction is based on the mean predictions from subset trees, it won’t give precise values for the regression model.

***Boosting****is another ensemble technique to create a collection of predictors. In this technique, learners are learned sequentially with early learners fitting simple models to the data and then analyzing data for errors. In other words, we fit consecutive trees (random sample) and at every step, the goal is to solve for net error from the prior tree.*

When an input is misclassified by a hypothesis, its weight is increased so that next hypothesis is more likely to classify it correctly. By combining the whole set at the end converts weak learners into better performing model.

**Gradient Boosting** is an extension over boosting method.

Gradient Boosting= Gradient Descent + Boosting.

It uses gradient descent algorithm which can optimize any differentiable loss function. An ensemble of trees are built one by one and individual trees are summed sequentially. Next tree tries to recover the loss (difference between actual and predicted values).

**Advantages of using Gradient Boosting technique:**

* Supports different loss function.
* Works well with interactions.

**Disadvantages of using Gradient Boosting technique:**

* Prone to over-fitting.
* Requires careful tuning of different hyper-parameters

**Fundamentals of Machine Learning**

* + **Four Branches of Machine Learning, 94**
    - **Supervised Learning 94**

It consists of learning to map input data to known targets (also called annotations), given a set of examples (often annotated by humans). Generally, almost all applications of deep learning that are in the spotlight these days belong in this category, such as optical character recognition, speech recognition, image classification, and language translation. Although supervised learning mostly consists of classification and regression, there are more exotic variants as well, including the following (with examples):

* Sequence generation—Given a picture, predict a caption describing it. Sequence generation can sometimes be reformulated as a series of classification problems (such as repeatedly predicting a word or token in a sequence).
* Syntax tree prediction—Given a sentence, predict its decomposition into a syntax tree.
* Object detection—Given a picture, draw a bounding box around certain objectsϒ inside the picture. This can also be expressed as a classification problem (given many candidate bounding boxes, classify the contents of each one) or as a joint classification and regression problem, where the bounding-box coordinates are predicted via vector regression.
* Image segmentation—Given a picture, draw a pixel-level mask on a specific object.
  + - **Unsupervised Learning 94**

This branch of machine learning consists of finding interesting transformations of the input data without the help of any targets, for the purposes of data visualization, data compression, or data denoising, or to better understand the correlations present in the data at hand. Unsupervised learning is the bread and butter of data analytics, and it’s often a necessary step in better understanding a dataset before attempting to solve a supervised-learning problem. Dimensionality reduction and clustering are well-known categories of unsupervised learning.

* + - **Self Supervised Learning 94**

Self-supervised learning is supervised learning without human-annotated labels. There are still labels involved (because the learning has to be supervised by something), but they’re generated from the input data, typically using a heuristic algorithm.

For instance, autoencoders are a well-known instance of self-supervised learning, where the generated targets are the input, unmodified. In the same way, trying to predict the next frame in a video, given past frames, or the next word in a text, given previous words, are instances of self-supervised learning (temporally supervised learning, in this case: supervision comes from future input data). Note that the distinction between supervised, self-supervised, and unsupervised learning can be blurry sometimes—these categories are more of a continuum without solid borders. Self-supervised learning can be reinterpreted as either supervised or unsupervised learning, depending on whether you pay attention to the learning mechanism or to the context of its application.

* + - **Reinforcement Learning 95**

In reinforcement learning, an agent receives information about its environment and learns to choose actions that will maximize some reward. For instance, a neural network that “looks” at a videogame screen and outputs game actions in order to maximize its score can be trained via reinforcement learning.

Currently, reinforcement learning is mostly a research area and hasn’t yet had significant practical successes beyond games. In time, however, we expect to see reinforcement learning take over an increasingly large range of real-world applications: self-driving cars, robotics, resource management, education, and so on.

**Classification and regression glossary**

Classification and regression involve many specialized terms.

* **Sample or input**—One data point that goes into your model.
* **Prediction or output**—What comes out of your model.
* **Target**—The truth. What your model should ideally have predicted, according to an external source of data.
* **Prediction error or loss value**—A measure of the distance between your model’s prediction and the target.
* **Classes**—A set of possible labels to choose from in a classification problem. For example, when classifying cat and dog pictures, “dog” and “cat” are the two classes.
* **Label**—A specific instance of a class annotation in a classification problem. For instance, if picture #1234 is annotated as containing the class “dog,” then “dog” is a label of picture #1234.
* **Ground-truth or annotations**—All targets for a dataset, typically collected by humans.
* **Binary classification**—A classification task where each input sample shouldϒ be categorized into two exclusive categories.
* **Multiclass classification**—A classification task where each input sample should be categorized into more than two categories: for instance, classifying handwritten digits.
* **Multilabel classification**—A classification task where each input sample can be assigned multiple labels. For instance, a given image may contain both a cat and a dog and should be annotated both with the “cat” label and the “dog” label. The number of labels per image is usually variable.
* **Scalar regression**—A task where the target is a continuous scalar value. Predicting house prices is a good example: the different target prices form a continuous space.
* **Vector regression**—A task where the target is a set of continuous values: for example, a continuous vector. If you’re doing regression against multiple values (such as the coordinates of a bounding box in an image), then you’re doing vector regression.
* **Mini-batch or batch**—A small set of samples (typically between 8 and 128) that are processed simultaneously by the model. The number of samples is often a power of 2, to facilitate memory allocation on GPU. When training, a mini-batch is used to compute a single gradient-descent update applied to the weights of the model.

**Evaluating Machine learning Models, 97**

we split the data into a training set, a validation set, and a test set. The reason not to evaluate the models on the same data they were trained on quickly became evident: after just a few epochs, the model began to overfit. That is, their performance on never-before-seen data started stalling (or worsening) compared to their performance on the training data—which always improves as training progresses. In machine learning, the goal is to achieve models that generalize—that perform well on never-before-seen data—and overfitting is the central obstacle. You can only control that which you can observe, so it’s crucial to be able to reliably measure the generalization power of your model.

* + - **Training, Validation, and Test Sets** 97

Evaluating a model always boils down to splitting the available data into three sets: training, validation, and test. You train on the training data and evaluate your model on the validation data. Once your model is ready for prime time, you test it one final time on the test data.

The reason is that developing a model always involves tuning its configuration: for example, choosing the number of layers or the size of the layers (called the hyperparameters of the model, to distinguish them from the parameters, which are the network’s weights). You do this tuning by using as a feedback signal the performance of the model on the validation data. In essence, this tuning is a form of learning: a search for a good configuration in some parameter space. As a result, tuning the configuration of the model based on its performance on the validation set can quickly result in overfitting to the validation set, even though your model is never directly trained on it.

Central to this phenomenon is the notion of information leaks. Every time you tune a hyperparameter of your model based on the model’s performance on the validation set, some information about the validation data leaks into the model. If you do this only once, for one parameter, then very few bits of information will leak, and your validation set will remain reliable to evaluate the model. But if you repeat this many times—running one experiment, evaluating on the validation set, and modifying your model as a result—then you’ll leak an increasingly significant amount of information about the validation set into the model.

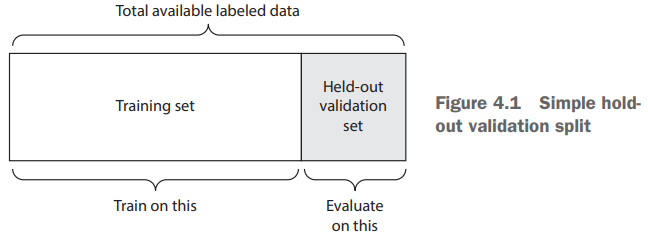
At the end of the day, you’ll end up with a model that performs artificially well on the validation data, because that’s what you optimized it for. You care about performance on completely new data, not the validation data, so you need to use a completely different, never-before-seen dataset to evaluate the model: the test dataset. Your model shouldn’t have had access to any information about the test set, even indirectly.

If anything about the model has been tuned based on test set performance, then your measure of generalization will be flawed. Splitting your data into training, validation, and test sets may seem straightforward, but there are a few advanced ways to do it that can come in handy when little data is available. Let’s see three classic evaluation recipes:

* Simple hold-out validation,
* Kfold validation, and
* Iterated K-fold validation with shuffling

**SIMPLE HOLD-OUT VALIDATION**

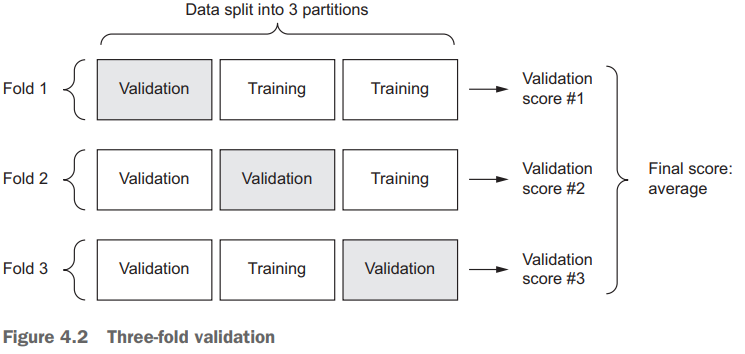
Set apart some fraction of your data as your test set. Train on the remaining data, and evaluate on the test set. As you saw in the previous sections, in order to prevent information leaks, you shouldn’t tune your model based on the test set, and therefore you should also reserve a validation set. Schematically, hold-out validation looks like figure 4.1. The following listing shows a simple implementation.



This is the simplest evaluation protocol, and it suffers from one flaw: if little data is available, then your validation and test sets may contain too few samples to be statistically representative of the data at hand. This is easy to recognize: if different random shuffling rounds of the data before splitting end up yielding very different measures of model performance, then you’re having this issue. K-fold validation and iterated K-fold validation are two ways to address this.

**K-FOLD VALIDATION**

With this approach, you split your data into K partitions of equal size. For each partition i, train a model on the remaining K – 1 partitions, and evaluate it on partition i. Your final score is then the averages of the K scores obtained. This method is helpful when the performance of your model shows significant variance based on your traintest split. Like hold-out validation, this method doesn’t exempt you from using a distinct validation set for model calibration. Schematically, K-fold cross-validation looks like figure 4.2. Listing 4.2 shows a simple implementation.

****

**ITERATED K-FOLD VALIDATION WITH SHUFFLING**

This one is for situations in which you have relatively little data available and you need to evaluate your model as precisely as possible. I’ve found it to be extremely helpful in Kaggle competitions. It consists of applying K-fold validation multiple times, shuffling the data every time before splitting it K ways. The final score is the average of the scores obtained at each run of K-fold validation. Note that you end up training and evaluating P × K models (where P is the number of iterations you use), which can very expensive.

* + **Overfitting and Underfitting 104**

O**verfitting, underfitting, and bias-variance tradeoff** are foundational concepts in machine learning. They are important because they explain the state of a model based on their performance. The best way to understand these terms is to see them as a tradeoff between the **bias** and the **variance** of the model. Let's understand the phenomenon of overfitting and underfitting.

* **Overfitting** occurs when a statistical model or machine learning algorithm **captures the noise** of the data. Intuitively, overfitting occurs when the model or the algorithm fits the data too well. Specifically, overfitting occurs if the model or algorithm shows **low bias** but **high variance**. Overfitting is often a result of an excessively complicated model, and it can be prevented by fitting multiple models and using validation or cross-validation to compare their predictive accuracies on test data.
* **Underfitting** occurs when a statistical model or machine learning algorithm **cannot capture the underlying trend** of the data. Intuitively, underfitting occurs when the model or the algorithm does not fit the data well enough. Specifically, underfitting occurs if the model or algorithm shows **low variance** but **high bias**. Underfitting is often a result of an excessively simple model.
* Both overfitting and underfitting lead to **poor predictions** on new data sets.
* Well, let's understand the Bias and variance in simpler terms. (**Very Simpler Terms!**)

**What is Bias?**

* Bias is the difference between the average prediction of our model and the correct value which we are trying to predict. A model with high bias pays very little attention to the training data and oversimplifies the model.
* **Simple definition:** “Resulted Error from Training Data!”

**What is a Variance?**

* Variance is the variability of model prediction for a given data point or a value that tells us the spread of our data. A model with high variance pays a lot of attention to training data and does not generalize on the data which it hasn’t seen before.
* **Simple definition:** “Resulted Error from Test Data!”
* Well, to understand the concepts more clear and better, I have divided concepts into Two parts, Bias and variance in the case of **Regression** as well as **Classification** models.
* Considering **Regression** models:

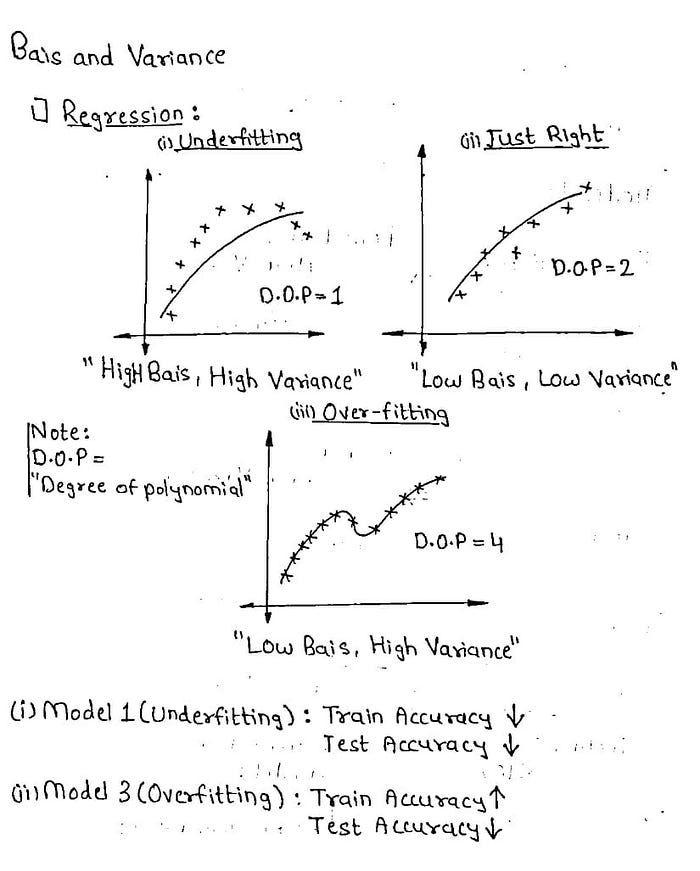


Figure 1: Bias and Variance for Regression Model

* We can see clearly that the Model-1 and Model-3 are **Underfitting** and  **Overfitting**  respectively.
* Model-1 has not captured the trends properly, or the model is too simple, hence it's obvious that the training and test accuracy will be hampered!
* As we discussed earlier, **“Bias is Error resulted from Training set, while Variance is error resulted from Test set!”**. The Model-1 will have less train and test accuracy, I.e. Will have High Bias(**High Training error**) and High Variance(**High Testing error**).
* Similarly, for Model-3, The model has trained too good on training data, the reason it fails for testing data(**Low test accuracy**). Since the training accuracy for Model-3 is High and Test accuracy is low, Model-3 will have Low Bias( **Low Training error**) and High Variance(**High Testing error**).
* Considering Model-2, As the Model-2 is in the “**Just Right**” condition, the model has trained well on training as well as a test set respectively. The reason, model has High training accuracy (**Low Bias-low training error**) and High testing accuracy( **Low Variance-low testing error**).
* Now, Let’s consider the condition for **Classification** models, Please have a look at the explained image below!

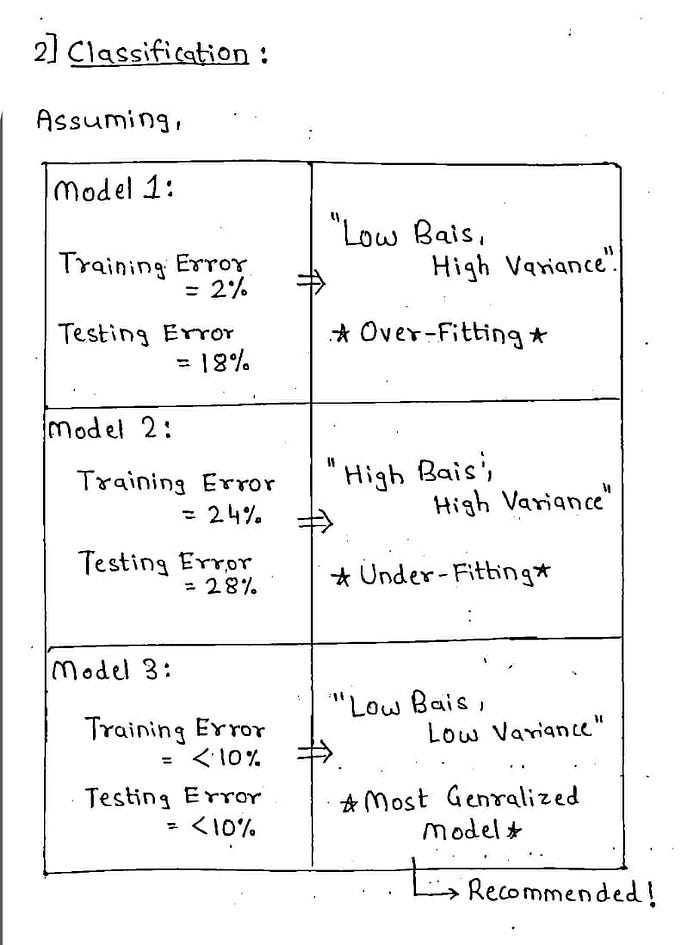


Figure 2: Bias and Variance for Classification Model

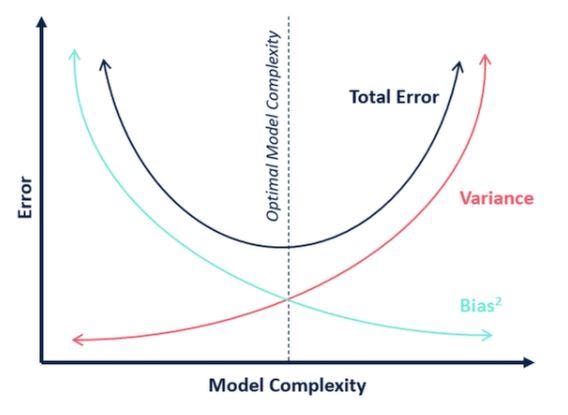
* Here we have 3 Models, which have the following training and testing errors.
* As we can see our Classification Model-1 has a Low Training error(2%), while has a high testing error(18%). As explained the concepts earlier we can conclude the model is having Low Bias(Low training error) and High Variance(High testing error), i.e. the **Model-1 is clearly Overfitting**!
* Similarly, We can conclude our Classification Model-2 as clearly an underfitting model. Coming towards Model-3, This model shall be considered as the **Most Generalized or Most Recommended** model to train on!
* Well, this was the explanation for Underfitting, Overfitting, Bais, and Variance for Regression and Classification Models respectively!
* We are done with the explanation part, now let's have a look at the **graphical plotting**of these concepts. Please have a look at the figure below!
* 

Figure 3: Bias and Variance Trade-off

* Considering Figure 3, the **dotted** line which passes through the points, are the points for which we should design our model, Which would be the “**Most Generalized Model**”.

**Overfitting can be overcome by:**

* + - Reducing the Network’s Size 104
    - Adding Weight Regularization 107
    - Adding Dropout 109

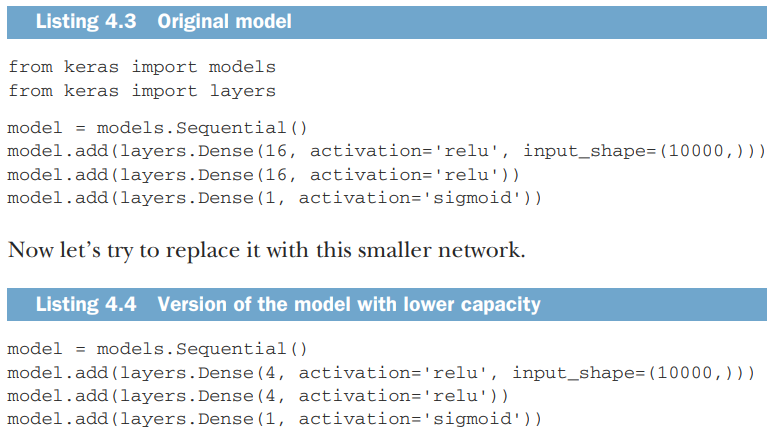
**Reducing the Network’s Size**

The simplest way to prevent overfitting is to reduce the size of the model: the number of learnable parameters in the model (which is determined by the number of layers and the number of units per layer). In deep learning, the number of learnable parameters in a model is often referred to as the model’s capacity. Intuitively, a model with more parameters has more memorization capacity and therefore can easily learn a perfect dictionary-like mapping between training samples and their targets—a mapping without any generalization power. For instance, a model with 500,000 binary parameters could easily be made to learn the class of every digit in the MNIST training set: we’d need only 10 binary parameters for each of the 50,000 digits. But such a model would be useless for classifying new digit samples.

Always keep this in mind: deeplearning models tend to be good at fitting to the training data, but the real challenge is generalization, not fitting. On the other hand, if the network has limited memorization resources, it won’t be able to learn this mapping as easily; thus, in order to minimize its loss, it will have to resort to learning compressed representations that have predictive power regarding the targets—precisely the type of representations we’re interested in.

At the same time, keep in mind that you should use models that have enough parameters that they don’t underfit: your model shouldn’t be starved for memorization resources. There is a compromise to be found between too much capacity and not enough capacity. Unfortunately, there is no magical formula to determine the right number of layers or the right size for each layer. You must evaluate an array of different architectures (on your validation set, not on your test set, of course) in order to find the correct model size for your data. The general workflow to find an appropriate model size is to start with relatively few layers and parameters, and increase the size of the layers or add new layers until you see diminishing returns with regard to validation loss.

Eg:



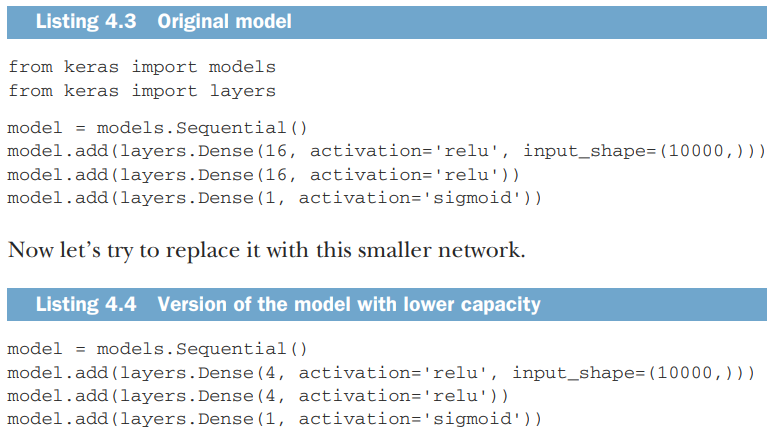
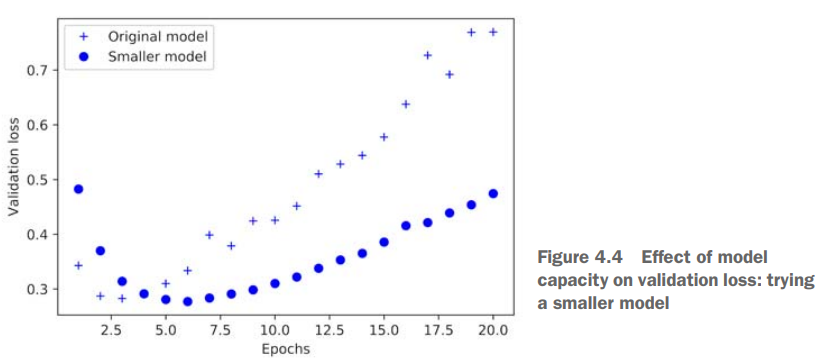


Figure 4.4 shows a comparison of the validation losses of the original network and the smaller network. The dots are the validation loss values of the smaller network, and the crosses are the initial network (remember, a lower validation loss signals a better model).

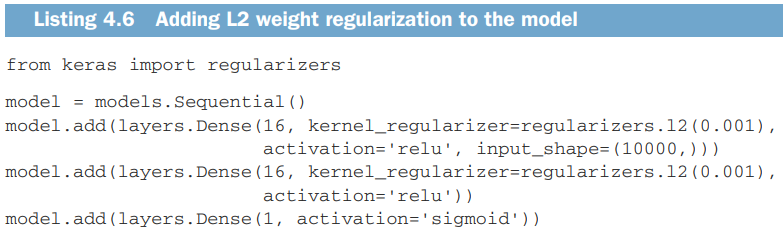


As you can see, the smaller network starts overfitting later than the reference network (after six epochs rather than four), and its performance degrades more slowly once it starts overfitting.

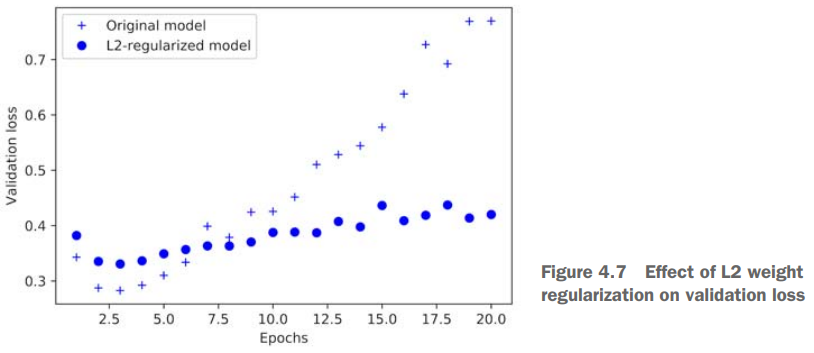
**Adding Weight Regularization**

a common way to mitigate overfitting is to put constraints on the complexity of a network by forcing its weights to take only small values, which makes the distribution of weight values more regular. This is called weight regularization, and it’s done by adding to the loss function of the network a cost associated with having large weights. This cost comes in two flavors: L1 regularization—The cost added is proportional to the absolute value of theϒ weight coefficients (the L1 norm of the weights). L2 regularization—The cost added is proportional to the square of the value of theϒ weight coefficients (the L2 norm of the weights). L2 regularization is also called weight decay in the context of neural networks. Don’t let the different name confuse you: weight decay is mathematically the same as L2 regularization.

In Keras, weight regularization is added by passing weight regularizer instances to layers as keyword arguments. Let’s add L2 weight regularization to the movie-review classification network

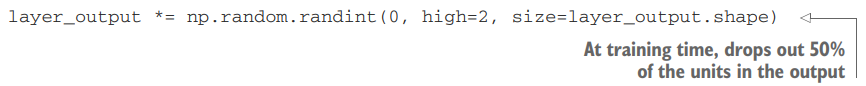


l2(0.001) means every coefficient in the weight matrix of the layer will add 0.001 \* weight\_coefficient\_value to the total loss of the network. Note that because this penalty is only added at training time, the loss for this network will be much higher at training than at test time. Figure 4.7 shows the impact of the L2 regularization penalty. As you can see, the model with L2 regularization (dots) has become much more resistant to overfitting than the reference model (crosses), even though both models have the same number of parameters.



**Adding Dropout**

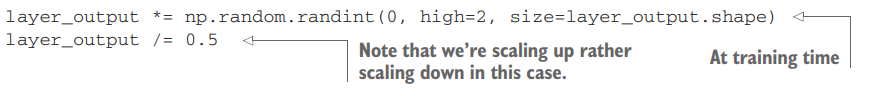
Dropout is one of the most effective and most commonly used regularization techniques for neural networks, developed by Geoff Hinton and his students at the University of Toronto. Dropout, applied to a layer, consists of randomly dropping out (setting to zero) a number of output features of the layer during training. Let’s say a given layer would normally return a vector [0.2, 0.5, 1.3, 0.8, 1.1] for a given input sample during training. After applying dropout, this vector will have a few zero entries distributed at random: for example, [0, 0.5, 1.3, 0, 1.1]. The dropout rate is the fraction of the features that are zeroed out; it’s usually set between 0.2 and 0.5. At test time, no units are dropped out; instead, the layer’s output values are scaled down by a factor equal to the dropout rate, to balance for the fact that more units are active than at training time. Consider a Numpy matrix containing the output of a layer, layer\_output, of shape (batch\_size, features). At training time, we zero out at random a fraction of the values in the matrix:

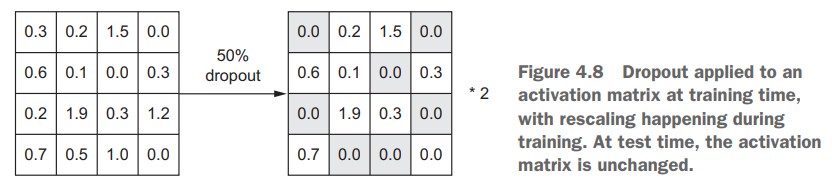


At test time, we scale down the output by the dropout rate. Here, we scale by 0.5 (because we previously dropped half the units):



Note that this process can be implemented by doing both operations at training time and leaving the output unchanged at test time, which is often the way it’s implemented in practice (see figure 4.8):





This technique may seem strange and arbitrary. Why would this help reduce overfitting? Hinton says he was inspired by, among other things, a fraud-prevention mechanism used by banks. In his own words, “I went to my bank. The tellers kept changing and I asked one of them why. He said he didn’t know but they got moved around a lot.

I figured it must be because it would require cooperation between employees to successfully defraud the bank. This made me realize that randomly removing a different subset of neurons on each example would prevent conspiracies and thus reduce overfitting.”1 The core idea is that introducing noise in the output values of a layer can break up happenstance patterns that aren’t significant (what Hinton refers to as conspiracies), which the network will start memorizing if no noise is present. In Keras, you can introduce dropout in a network via the Dropout layer, which is applied to the output of the layer right before it: model.add(layers.Dropout(0.5)) Let’s add two Dropout layers in the IMDB network to see how well they do at reducing overfitting.

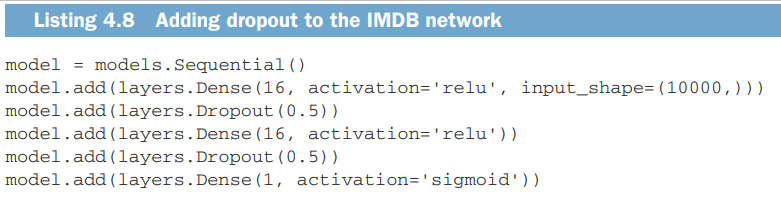


Figure 4.9 shows a plot of the results. Again, this is a clear improvement over the reference network

