Sage Research Methods Foundations

Neural Networks

Contributors: Wenshuo Liu

Pub. Date: 2020

Product: Sage Research Methods Foundations

DOI: https://doi.org/10.4135/9781526421036888177

Methods: Convolutional neural networks, Neural networks, Machine learning

Disciplines: Anthropology, Business and Management, Communication and Media Studies, Computer Science, Counseling and Psychotherapy, Criminology and Criminal Justice, Economics, Education, Engineering, Geography, Health, History, Marketing, Mathematics, Medicine, Nursing, Political Science and International Relations, Psychology, Social Policy and Public Policy, Science, Social Work, Sociology, Technology

Access Date: February 4, 2025

Publisher: SAGE Publications Ltd

City: London

Online ISBN: 9781529749274

Abstract

Neural networks (also called artificial neural networks) refers to a class of machine learning methods that was developed in multiple fields including statistics and artificial intelligence. This entry discusses several topics with regard to neural networks in three sections. The first section introduces the model and theories of the quintessential neural network model, feed-forward neural networks, under the framework of supervised learning. It focuses on the training and regularization of neural networks models given their uniqueness among other machine leaning models. The second section discusses the modern developments of neural networks that lead to its successful application in the various domains of artificial intelligence. Such developments are grouped into three categories: architecture design, methodology refinements, and advances in software and hardware. The third section briefly illustrates neural networks applied in the other two major paradigms of machine learning: unsupervised learning and reinforcement learning.

Introduction

In a common setting, a machine leaning task seeks a predictive model that can "learn" from a collection of training data and predicts future observations. Neural networks—also known as artificial neural networks (ANN)—achieve such a goal with a structure that was vaguely inspired by the biological communication between neurons in animal brains, which also serves as the origin of the terminology. In the 2010s, a number of notable advances in both methodology and computational technology made it possible to implement neural networks that have tens or even hundreds of layers. The learning methods utilizing ANN with many layers gained a new name—*deep learning* (Goodfellow et al., 2016; LeCun, Bengio, & Hinton, 2015) and had groundbreaking successful applications in many "human-mastered" domains such as computer vision, natural language processing, speech recognition, and game playing. Successful applications led to active research on neural networks in both academia and industry in the 2010s.

This entry introduces different aspects of neural networks in three sections. The first section introduces the underlying statistical model and theories of neural networks. It starts with the definition of a supervised learning problem, in which a learning method aims to find an approximated function between the predictive variables and the outcome, from the training data. Neural networks are one of the methods proposed to solve such a problem. The section elaborates the architecture and statistical model of feed-forward neural networks, which is the quintessential neural network model. From the statistical modeling point of view, such a model can be considered a parametric model with an unreasonably large number of parameters, which is prone to the issue of overfitting. Thus, the section spends considerable space to explain the special features of parameter optimization, or model training of neural networks, with extra highlight on regularization schemes. This section ends with a lists of potential limitations of neural networks.

The second section demonstrates how the modern developments of neural networks make it successful and influential. The underlying model of neural networks was proposed in the 1980s. However, well before the 2010s, neural networks did not show much advantages over other machine learning methods, such as linear/logistic regressions and decision tree–based methods (Hastie, Tibshirani, & Friedman, 2016). It was a series of developments on both methodology and computational technology that lead to groundbreaking applications of neural networks in the problems that are almost intractable for other machine learning methods. This section elaborates the details of those developments, which are grouped into three categories. First, the architecture design is illustrated by two prototypical examples: the convolutional neural networks (CNNs), specifically designed for image data, show superior performance in many computer vision tasks such as image classification, segmentation, and localization; the recurrent neural networks (RNNs), specifically designed for sequence data, are widely utilized in language-related applications such as machine translation and speech recognition. Second, a series of methodological refinements are discussed, including rectified linear unit (ReLU) as a new activation function, dropout as a new regularization and batch normalization to avoid vanishing or exploding gradients. Lastly, since neural networks rely heavily on computation, the software and hardware advances that ensure the feasibility of neural networks are presented.

The third section discusses a few implementations of neural networks beyond the supervised learning scenario. Many data-driven problems are not limited to predictive modeling. Neural networks also achieve state-of-the-art standards in many of them. This section briefly introduces utilization of neural networks under two other problem settings: unsupervised learning and reinforcement learning.

Models and Theories of Neural Networks

People seek to efficiently and effectively extract useful information or knowledge from data and have developed and continue to develop tools to support and automate this task. Because of the advances of computer technology and the growing availability of large and complex data resources, many new data-mining methods make use of both statistical modeling and computational algorithms to look for patterns and inference directly from the data, with minimal or even no explicit instructions. Design and study of such methods has become an active research area, called *machine learning* (Bishop, 2011; Hastie, Tibshirani, & Friedman, 2016). Neural networks are one of the machine learning methods. To explain the mechanism of neural networks in a concrete manner, this entry first introduces *supervised learning*, a simple and prevalent machine learning task, where neural networks are most commonly applied.

Supervised Learning

As a specific machine learning task, supervised learning looks for the predictive relation between a class of feature variables $X = \left(X_1,\ X_2,\ \cdots\right)$ and some outcome Y, from a collection of sample pairs of $(X,\ Y)$, called *training data*. As a simple example, one can approximately infer the *WEIGHT* of a person from his or her (*HEIGHT*, *GENDER*), given the (*HEIGHT*, *GENDER*, *WEIGHT*) of a group of people. In a complex problem, the feature variables X can be the brightness of each pixel from an image (very high-dimensional), and the outcome Y can be the object to be detected in the image. The corresponding training data would be a set of images with or without such an object.

A commonly used approach for this problem, called *parametric modeling*, is to assume a mathematic function $Y = f(X,\ \theta)$, with parameters $\theta = \left(\theta_1,\ \theta_2,\ \cdots\right)$ and try to find the best set of θ that make the predicted outcome $Y_{predict} = f\left(X_{train},\ \theta\right)$ as close as possible to the true outcome Y, on the training data set. For example, we can assume a person's *WEIGHT* is a linear function of his or her *HEIGHT* and *GENDER*,

$$WEIGHT = \theta_1 \times HEIGHT + \theta_2 \times GENDER$$

To find the best values of θ_1 and θ_2, we can minimize the mean square error (MSE),

$$\sum_{i \text{ over the training samples}} \left(\theta_1 \times HEIGHT_i + \theta_2 \times GENDER_i - WEIGHT_i\right)^2$$

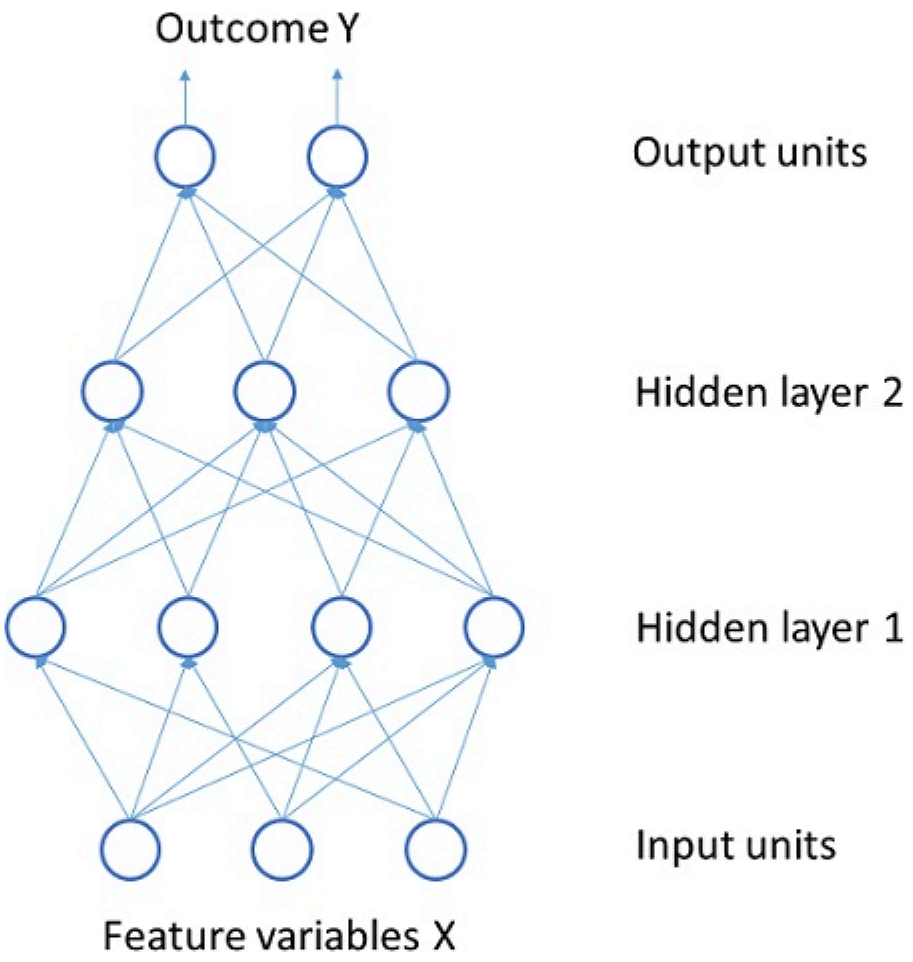
with respect to θ_1 and θ_2. MSE is a specific form of *loss functions*, which measures the difference between the predicted and the true outcome, taking different forms depending on the data type of the outcome. Algorithms to minimize the loss functions, called *optimization* (Boyd and Vandenberghe, 2004), are well studied in computer sciences. The process of looking for the best set of parameters are usually called *model training*. The modeling approach presented in this example is called *linear regression*, which makes strong and simple assumptions that the feature variables are additive and linear in the prediction of the outcome. In a complex problem—for example, the object detection from images—linear regression will evidently fail for ignoring the interactions and nonlinearity of the feature variables. A more flexible model is desired for such cases, with feed-forward neural networks being a good example.

On the contrary of supervised learning, unsupervised learning refers to the machine learning task of finding underlying patterns of data without preexisting labels, such as principle component analysis and clustering. More details on unsupervised learning are presented later in the Neural Networks Beyond Supervised Learning section.

Feed-Forward Neural Networks

Feed-forward neural networks, or multilayer perceptron (MLP), is the quintessential model of neural networks. Inspired by the biological communication between neurons of animal brains, MLP takes the form of a layer-wised graph, with basic elements of nodes (also called units or neurons) and directional edges. Figure 1 illustrates an example of such an architecture. The idea of feed-forward neural networks is to approximate a complex function from the feature variables to the outcome $Y = f(X, \theta)$, by a collection of simple functions in a nested form. Each node, playing the role of a computing unit, takes the outputs of nodes from the last layer and does a simple operation and then pass the results to the next layer. The nodes of the first layer, called *input units*, takes the values of the feature variables X; the nodes of the final layer, called *output units*, output the outcome Y (which can be multidimensional). The simple operation on each node is designed to be elementary, with a derivative that is easy to calculate (for reasons to be explained later). But, it has to be nonlinear because nested linear functions are still linear. In practice, the operation of each unit usually takes the form of a weighted summation followed by a nonlinear transformation, with the weights being the parameters of the model.

Figure 1. An example of the architecture of a feed-forward neural network, with two hidden layers.



Speaking in a mathematical language, using $y_{i,\,j}$ to represent the output of the *i*th node in the *j*th layer, each node does the following calculation

$$y_{i,\,j} = g\left(\sum_k w_{k,\,j-1} \times y_{k,\,j-1}\right)$$

Here $w_{i,\,j}$ are the weights of the last layer's outputs, and $g(\cdot)$ is the nonlinear transformation, called *activation function*. Two popular forms of activation functions are *sigmoid function*$g(x) = 1 \Big/ \left(1 + e^{-x}\right)$ and ReLU $g(x) = \max(0,\ x)$. The sigmoid function smoothly maps values from $(-\infty,\ \infty)$ to $(0,\ 1)$, representing the activation of a neuron (node) by the increasing input signal. However, recently, researchers found that ReLU, as an alternative of the sigmoid function, can speed up the training of neural networks since its derivative is simple. It was mathematically proven (Cybenko, 1989; Hornik, Stinchcombe, & White, 1989) that a feed-forward neural network with sufficient hidden units can approximate an adequately large family of complicated functions (almost any function in practice).

Feed-forward neural networks follow the strategy of making the least assumptions on the feature–outcome relation, and letting the model learn it solely from the data. The graphical structure of feed-forward neural networks enables learning of underlying patterns directly from the data, and autonomous capture of interactions and nonlinearity of the feature variables. However, neural networks usually require a larger sample size of the training data, compared to other machine learning methods—the training data need to be sufficiently large (typically at the scale of 10,000) to show its pattern.

The depth (number of layers) and width (number of nodes of a layer) of a feed-forward neural network are the choice of the machine learning practitioner. Furthermore, a node in the network is not necessarily connected to every node from the last layer. In fact, determination of the depth, width, and connectivity, called *architecture design*, is sophisticated and crucial for the model performance. A good architecture design should purposefully fit the specific data type or the substantive problem. This topic is discussed in more detail in the Modern Development of Neural Networks section.

Model Training of Neural Networks

Feed-forward neural networks achieve model flexibility by aggregating many simple models, resulting in a large number of parameters to be optimized. The model training (i.e., minimization of the loss function with respect to all the parameters) is methodologically difficult and computationally expensive.

Gradient-Based Algorithm and Back-Propagation

To minimize a multivariate, differentiable function, gradient descent is a widely used algorithm. The gradient of a function is the vector composed of the derivatives of the function with respective to each of the variables $\nabla L(\theta) = \left(\frac{\partial L}{\partial \theta_1}, \ \frac{\partial L}{\partial \theta_2}, \ \cdots \right)$. It represents the direction of the steepest ascent or descent of the function (think about the direction perpendicular to the contour on a map). The gradient decent algorithm proposed to repeatedly go along the gradient direction with a properly chosen step size, starting from an initial guess of the optimal point, until the value of the function does not decrease any more. Gradient descent has many variants. For example, *stochastic gradient descent* (SGD) is the algorithm that divides the training data into small batches and calculates the gradient of the loss function using only one batch of data for each optimizing step of moving toward the gradient direction. It is especially effective when training data have a huge sample size. SGD is the most popular algorithm for training neural network models.

To minimize the loss function of a neural network by SGD, the gradient of the loss function must be calculated efficiently. For such a purpose, the *back-propagation* algorithm is proposed (Rumelhart, Hinton, & Williams, 1986). Making use of the chain rule of derivation, back-propagation calculates the derivatives of the network weights in a backward fashion: It starts with calculating the derivatives of the weights from the last layer and reuses them when calculating the derivatives of the weights from the previous layers, until reaching the first layer. When making predictions, the values go through the network forwardly; while calculating the gradient, the derivatives propagate backwardly.

Overfitting Issue and Regularization

The super flexibility of feed-forward neural networks causes a problem called *overfitting*: when overly trained, the model may fit to the noise instead of the signal. In such a case, the model fits particularly to the training data very well but generalizes poorly to new observations. Overfitting is a general problem for all machine learning methods and is usually amended by a scheme called *regularization*. Regularization aims to artificially reduce the unnecessary flexibility of a model. The most popular way to achieve such a goal is *shrinkage* of parameters: adding a penalization to every large parameters so that the model function behaves more regularly. Besides shrinkage, many other regularization schemes are developed for neural networks, with an incomplete listed of examples as follows:

- *Data augmentation* aims to add artificial randomness that are irrelevant to the learning task to the training data. As an example, when training a model that can detect “cats” in images, one may want to randomly rotate, enlarge, and shift the training images of cats. By doing so, one not only acquires more training data but also explicitly “tells” the model that orientation, size, and location are not correct features of “cats.”
- *Early stopping* is an ad hoc but practically useful scheme (Prechelt, 2012). It inherits the spirit of

cross-validation, a common practice in machine learning. Cross-validation refers to the method that randomly puts aside a small portion of training data and uses it solely for evaluating the model fitness. Similarly, the idea of early stopping is to randomly split the training data into a training set (a large proportion) and validation set (a small proportion) and monitor the loss function on the validation set while training the model on the training set. This loss function on the validation set is a good approximation of the generalization error. Thus, when it starts to increase, it is a sign of overfitting, and the training should stop.

- *Dropout* is a scheme designed uniquely for neural networks (Srivastava et al., 2014). It is discussed in later sections as one of the modern improvements in methodology of neural networks.
- *Multitask learning* is also a scheme particularly appropriate for neural networks. A feed-forward neural network can have multiple output units, meaning it can model multiple outcomes simultaneously. In fact, learning multiple tasks together enable parameter sharing and therefore alleviates overfitting. For example, in a learning task to recognize handwriting of digits 0–9, it is wiser to train one neural network with 10 output units that predict the 10 digits, instead of training a separate neural network for each of the digits. This is because the handwritten digits share many features, such as lines, corners and contours, and shared lower layers can more efficiently extract these features.

Limitations of Neural Networks

The theoretical perspectives of neural networks are concluded with the following list of limitations:

- *Lack of a rigorous theoretical foundation*. Training of neural networks posts an ultra-high-dimensional, nonconvex optimization problem, and it is hard to justify that SGD gives a rightful solution to it. Although discussed in a few works (Choromanska et al., 2014; Dauphin et al., n.d.), this question is far from conclusive from a theoretical perspective. In practice, there are also many questions that people do not fully understand, for example, why neural networks performs better in certain tasks than nonparametric models such as kernel methods.
- *Difficult interpretation*. In general, neural networks serves as a black box prediction model. Its parameters, the weights of the connections between nodes, do not have a clear interpretation. This is a big shortage compared with other machine learning methods. For instance, linear regression has coefficients that are substantively meaningful; decision tree–based methods can at least rank the

feature variables in terms of importance. Lack of interpretability brings hardship to model diagnosis for neural networks. Without much intuitive guidance, designing neural networks generally can only rely on exhaustive experiments. Difficult interpretation of neural networks also becomes an obstacle for dissemination in applications. In some tasks such as automatic diagnosis of diseases, people hesitate to believe the prediction by neural networks. For fatal tasks, such as autonomous driving, applications of neural networks are highly debatable. Some recent work has started to emphasize and improve the interpretability of neural networks (Gilpin et al., 2018).

- *Heavy computation*. Neural networks have a vast number of free parameters, and its training requires immense matrix computation and gradient calculation. As a result, neural networks began to proliferate only in the 2010s when software for graphical computation and hardware for large-scale parallel computation became available. These advances of technology and their influence on neural networks are discussed in the next section. Even equipped with such modern tools, training of some complicated neural networks on large data set still takes days or even weeks, consuming a large amount of computational resources (graphics processing unit [GPU] and memory) as well. This drawback becomes even more evident when neural networks have many hyperparameters (e.g., number of layers, number of nodes in each layer, strength of dropout) that need to be chosen to find the best model. The process of finding the best set of hyperparameters, called *hyperparameter tuning*, consumes a lot of time and computational resources. There are also studies on fast hyperparameter tuning, with Bayesian optimization (Snoek, Larochelle, & Adams, 2012; Snoek et al., 2015) being a widely used method.

Modern Development of Neural Networks

As mentioned earlier, neural networks became popular among machine learning methods only in the 2010s because a series of developments in both methodology and technology made it stand out in many learning tasks. This section of the entry discusses the modern developments of neural networks from three aspects: architecture design, refinement of methodology, and software and hardware advances.

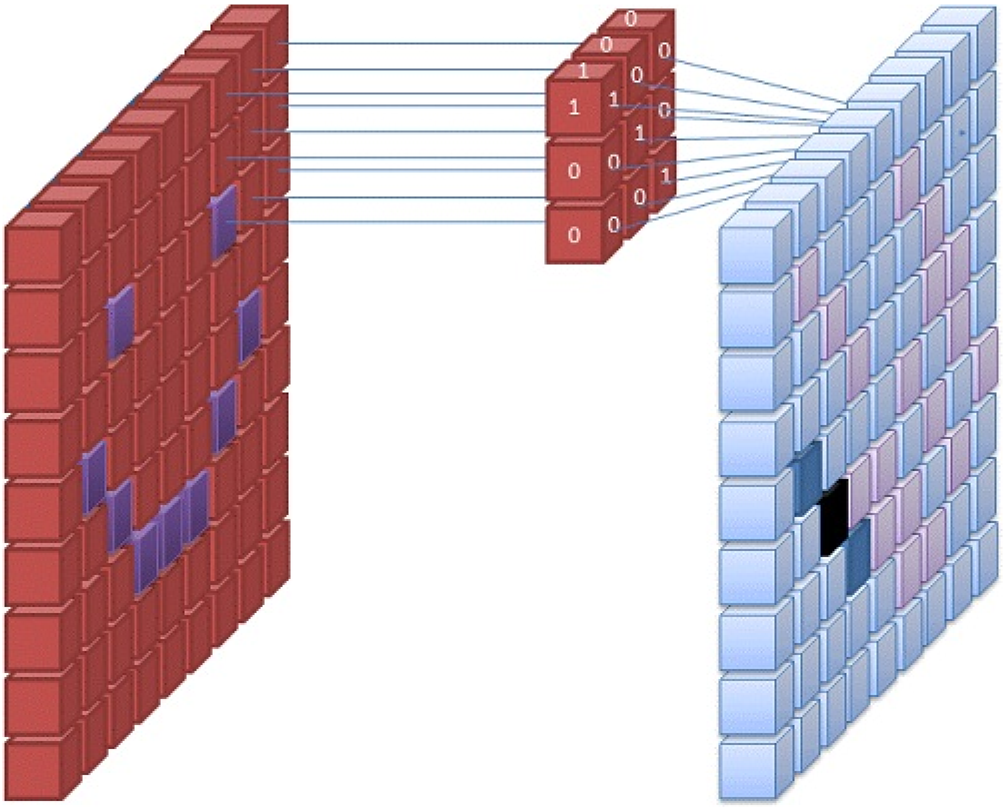
Architecture Design

Architecture design refers to the decision of the graphical structure of a neural network. For the feed-forward neural networks, it includes determination of the depth (number of layers), width (number of nodes in a layer), and connectivity between nodes. As will be explained in the following subsections, an architecture can be more sophisticated and critical for model performance. A good architecture design should incorporate the intuition of the substantive question or data type into the network structure and therefore enable parameter sharing, simplifying the model without loss of necessary flexibility. These strategic statements are demonstrated here by two prototypical examples: CNNs and RNNs.

Convolutional Neural Networks

CNNs are the networks specially designed for image data. An image from the data prospective can be seen as one (for grayscale image) or three (for color image encoded by RGB [red, green, and blue]) matrices, whose elements are the color intensity of each pixel. To accommodate such data structure, a convolution layer organizes the hidden nodes in a 2-dimensional form, called *feature maps* and tailors the connection between nodes in the form of filters to capture only local patterns. Figure 2 illustrates how a convolution layer with a 3*3 filter works: to get the value of the destination pixel, one takes the 3*3 area around the corresponding pixel on the source feature map, position-wisely multiplies with the filter matrix, and sums over the products. Such an operation is a discretized convolution from a mathematical point of view. The elements of the filter matrix are the weights (parameters) of the convolution layer to be trained. Comparing with a fully connected layer (where each node connects all nodes from the last layer), a convolution layer forces many weights to be zero and restricts many nonzero weights to be the same. By doing so, a convolution layer effectively reduces the number of free parameters and therefore simplifies the model, without damaging the 2-dimensional pattern of image data. It has proved to be a successful architecture design in many image-related learning tasks.

Figure 2. A schematic graph for convolution layer.

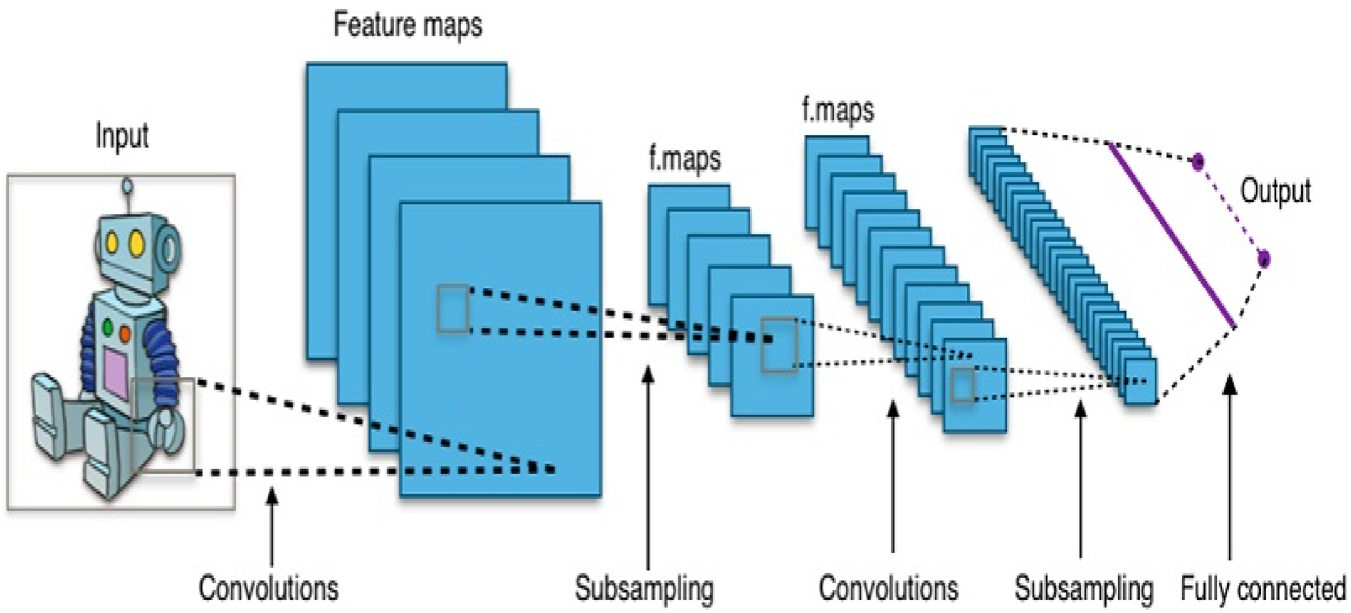


Source: Wikimedia Commons/Cecbur, https://commons.wikimedia.org/wiki/File: Convolutional_Neural_Network_NeuralNetworkFilter.gif. Licensed under Creative Commons Attribution-Share Alike 4.0 International license, https://creativecommons.org/licenses/by-sa/4.0/deed.en

In a CNN, convolution layers are usually stacked so that high-level features can be extracted by capturing features from previous feature maps. Pooling layers, also known as subsampling layers, are usually used in between convolutional layers to reduce computational cost and ensure the stability of the model over translational errors. Figure 3 shows an example of a typical CNN architecture for natural object detection. The feature maps of the first layer usually capture low-level features like lines and corners, while the deeper feature maps may catch high-level features like the digital number. In the latest applications, CNNs can easily achieve depth over 100 layers (He et al., 2015; Huang et al., 2016; Szegedy et al., 2015), extracting sophisticated features and performing outstandingly in image-based tasks such as object detection and localization

(Redmon & Farhadi, 2016; Redmon et al. 2015; Zhou et al., 2015).

Figure 3. An example of a convolutional neural network for natural object detection.



Source: Wikimedia Commons/Aphex 34, https://commons.wikimedia.org/wiki/File:Typical_cnn.png. Licensed under Creative Commons Attribution-Share Alike 4.0 International license, https://creativecommons.org/licenses/by-sa/4.0/deed.en

It is worth mentioning that image data challenges played an important role in the promotion of neural networks. The Large Scale Visual Recognition Challenge (ILSVRC) hosted by ImageNet, is considered one of, if not the, most impactful of them. ImageNet collected over 1 million links to Internet images for 1,000 object classes as a resourceful training data set for image-related learning tasks. ImageNet also hosted an annual challenge requesting data scientists to build machine learning models to correctly classify unseen images into the 1,000 classes. Since 2012, the winners of ILSVRC have been dominated by CNNs. The series of winning models (AlexNet, VGG, GoogLeNet, and ResNet) are publicly released and implemented in many programing languages. They have become off-the-shelf tools for image-related applications and have been widely used in further research and industrial products.

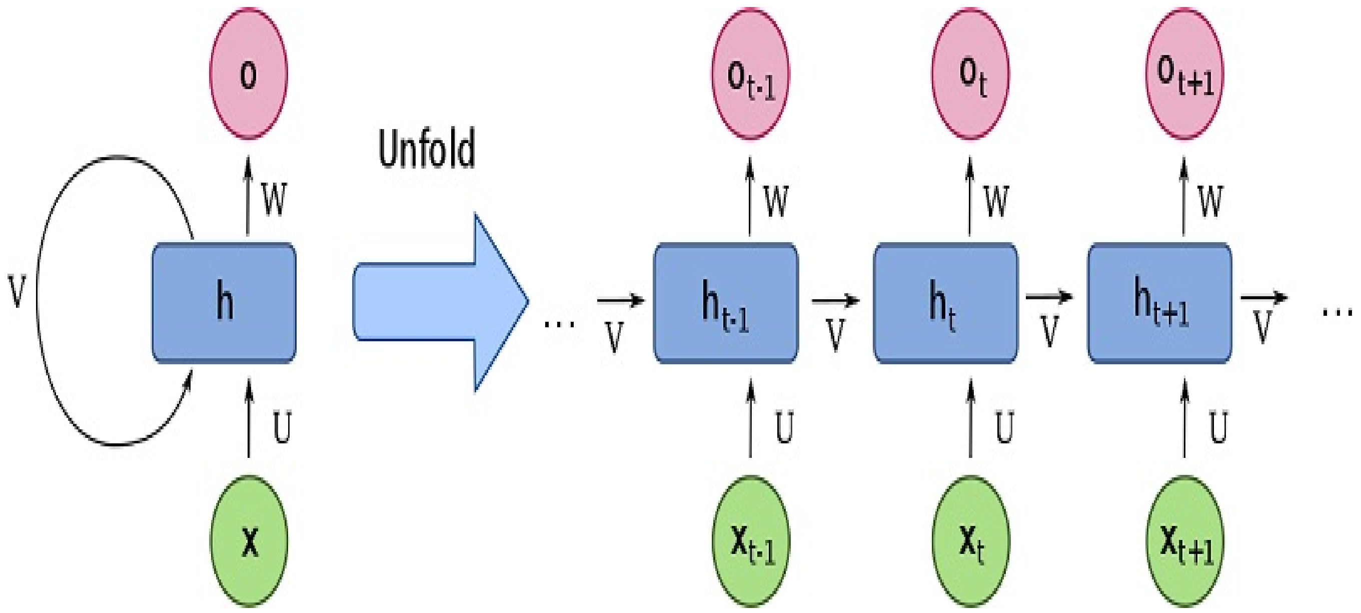
Computer vision using CNNs has an impact beyond engineering. For example, in medical sciences, such techniques are used in automated disease detection from medical images (Lundervold & Lundervold, 2019). In social sciences, CNN-based computer vision techniques are used on Google Street View images to estimate socioeconomic characteristics of geographical regions (Gebru et al., 2017).

Recurrent Neural Networks

RNNs are designed to model sequence data. Compared to other popular longitudinal models such as the Markov model, it has the advantage of preserving long-term memories. It is widely used in language-related learning tasks, such as natural language processing, machine translation, and speech recognition.

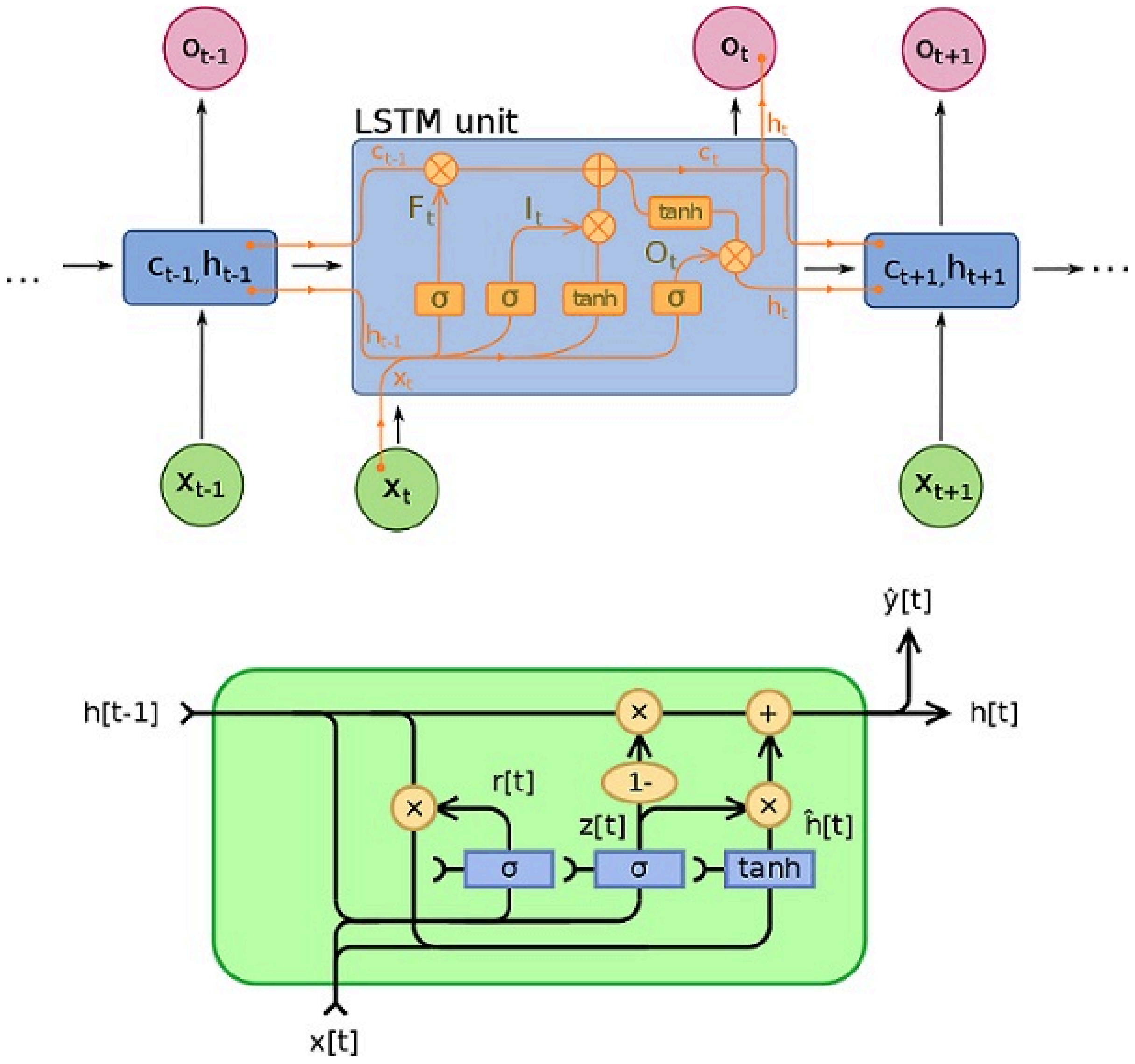
In some learning tasks, the feature variables form a sequence, that is, they have an intrinsic order and variable length, naturally labeled by time: $X = \left(X_1,\ X_2,\ \cdots,\ X_t,\ \cdots\right)$. The feature component at a particular time X_t can still be multidimensional. Some typical examples include words in a sentence and the medical history of a patient. To capture the longitudinal correlation of the data, RNNs feature an iterative chain structure consisting of the recurrent units, as shown in Figure 4. The unit at time t takes in the feature variable at that time and passes some information to the unit at the next moment. The unit can either have an output at every moment or have an overall output at the last moment, depending on the specific application. The recurrent unit has an internal architecture that calculates a hidden state vector (the h_t in Figure 4) and passes to the next moment, in addition to the output. The hidden state is a latent representation of the information that propagates along time, such as the semantic meaning of a sentence up to a particular word or the health condition of a patient at a certain visit. The calculations within a recurrent unit are implemented by a collection of linear models with activation functions, just like those in feed-forward neural networks. Figure 4 represents the simplest recurrent unit architecture, called a vanilla RNN. The two most popular architectures in practice are long short-term memory (LSTM) and gated recurrent unit (GRU). Figure 5 shows them. RNNs can have variable number of units to incorporate data with different lengths, but the parameters in all the units are forced to be identical, fulfill parameter sharing, and therefore reduce the risk of overfitting.

Figure 4. A schematic plot of the basic RNN architecture.



Source: Wikimedia Commons/François Deloche, https://commons.wikimedia.org/wiki/File:Recurrent_neural_network_unfold.svg. Licensed under Creative Commons Attribution-Share Alike 4.0 International license, https://creativecommons.org/licenses/by-sa/4.0/deed.en

Figure 5. Upper panel: the architecture of long short-term memory (LSTM). Lower panel: the architecture of the gated recurrent unite (GRU).



Source: Upper panel: Wikimedia Commons/François Deloche, https://commons.wikimedia.org/wiki/File:Long_Short-Term_Memory.svg; licensed under Creative Commons Attribution-Share Alike 4.0 International license, https://creativecommons.org/licenses/by-sa/4.0/deed.en. Lower panel: Wikimedia Commons/Jeblad, https://commons.wikimedia.org/wiki/File:Gated_Recurrent_Unit,_base_type.svg; licensed under Creative Commons Attribution-Share Alike 4.0 International license, https://creativecommons.org/licenses/by-sa/4.0/deed.en

The vanilla RNN unit calculates the hidden state vector (the information to pass to the next moment) and the output as two separate functions of the input variable X_t and the hidden state from the last moment h_{t-1} , in the form of a linear model followed by an activation function. Such structure can in principle keep track of arbitrarily long-term dependency of the sequence data. But the implementation suffers from vanishing or exploding gradients when trained with back-propagation. LSTM (Hochreiter & Schmidhuber, 1997) was proposed to overcome such an issue. LSTM keeps track of a cell state in addition to the hidden state and controls the flow of information by three "regulator" usually called gates: an input gate, an output gate, and a forget gate (I_t, O_t, and F_t respectively in Figure 5). These gates are also implemented by linear functions with activations. Intuitively, they "decide" which information to be passed to the next moment and which information to be forgotten, by their function parameters. Through such an architecture, LSTM allows some of the gradients to flow unchanged and therefore partially solved the problem of vanishing gradients. GRU (Cho et al., 2014), proposed much later, is designed to simplify the LSTM. GRU has fewer free parameters than LSTM and performs similarly in many tasks. But some recent research (Weiss, Goldberg, & Yahav, 2018) shows that GRU may be less capable compared with LSTM.

Since RNNs units deliver information to the units of the next moment, RNNs have effectively a "deep" architecture. In practice, LSTM and GRU units can also be "stacked" to form an even deeper architecture that can capture intricate longitudinal dependence in the data. As an example, in the research by Ilya Sutskever, Oriol Vinyals, and Quoc V Le (2014), a multilayer LSTM achieved state-of-the-art performance in the translation between French and English.

Because of its capability of capturing longitudinal patterns, RNNs are widely applied in natural language processing and therefore influence language-related studies. As an example, GRU-based deep neural networks are used in detection of hate speech on Twitter (Zhang, Robinson, & Tepper, 2018).

CNNs and RNNs are the two most famous and successful architectures of neural networks. However, a new architectures called *attention mechanism* has shown effectiveness in various task domains. Inspired by the fact that a human focuses on only a small but informative area when looking at objects, the attention mechanism is designed to learn extra weights representing how much attention should be paid to each node of the last layer. As an example, some recently proposed attention-based neural networks (Devlin et al., 2018; Vaswani et al., 2017) set up new benchmarks in many natural language processing tasks.

Methodology Refinements

Although research on the theoretical foundation faces considerable difficulties, there are still a series of improvements in methodology making neural network a practically useful machine learning tool. Each of these improvements may be incremental, but accumulation of them has dramatically boosted the performance of neural networks. Most of them have become common practice in model building. The following subsections provide an incomplete list of methodological improvements of neural networks.

Evolution of Activation Functions

As previously mentioned, an activation function is introduced after the linear model for each neural network node, to provide nonlinearity to the model. In the early stage of neural networks, activation functions are usually chosen to be smooth, such as the logistic sigmoid function $g(x) = 1 \Big/ \left(1 + e^{-x}\right)$ originating from probabilistic intuition or the hyperbolic tangent function $\tanh(x) = \left(e^{x} - e^{-x}\right) \Big/ \left(e^{x} + e^{-x}\right)$. It was found in 2011 (Glorot, Bordes, & Bengio, n.d.) that a brutally simple choice, the ReLU $g(x) = \max(0, \ x)$ enables better training of deep neural networks. ReLU was actually proposed much earlier (Hahnloser et al., 2000) by biological motivation: Brain neurons can be active (e.g., emitting signal corresponding to the input signal) or inactive (e.g., not responsive at all). Mathematically, ReLU makes back-propagation more efficient because (1) it has a slope that does not saturate like sigmoid function, remedying the vanishing gradient problem; (2) it is sparsely activated, causing many nodes to have zero output, reducing the amount of necessary calculation. ReLU has become the standard choice for activation function for almost all neural network design. Some variants of ReLU, such as leaky ReLU (Maas, 2013), were proposed to solve the issue that a ReLU neuron can be stuck at inactive and always output zero.

New Regularization Methods

As mentioned earlier, regularization is particularly important for neural networks because very flexible models are more prone to overfitting. Beside the regularization methods that are widely used in all statistical learning such as penalization on the parameters, dropout is a regularization technique that is designed uniquely for neural networks. Dropout follows the idea of randomly removing some of the nodes so the network is reduced to a simpler one. When dropout is applied to the model training, some nodes are removed with a probability p (representing the dropout strength). The incoming and outgoing edges of the removed nodes are also disconnected, leaving only a reduced network being trained on the data in that SGD step. The nodes to be removed are resampled for every SGD step. In such a way, an artificial random noise is injected into the training process, preventing the model from fitting particular to the training data. It also considerably improves training speed.

During testing, all nodes are restored, but their outputs are multiplied by a factor of p. Thus, the output of each nodes has the same statistical expectation as in the training stage. The final network is effectively a summation of many randomly generated reduced networks. The error from each reduced networks is likely to cancel each other in the summation, leading to a better prediction accuracy.

Endeavors for Greater Depth

Both intuitively and practically, neural networks achieve better performance with greater depth in general because deeper networks are able to learn higher level, compositional features. However, implementation of networks with a lot of layers faces many technical obstacles. For example, the output of each node tends to have a bad scale—it either vanishes or explodes during the training process by back-propagation, due to the chain rule of derivation. Besides, when the network is too deep, it is difficult for the information to propagate back to the early layers through gradients. To fix these problems, researchers proposed many technical approaches. Batch normalization and skip connection are the most influential examples of such approaches.

Batch normalization, proposed in 2015 (Ioffe & Szegedy, 2015), follows the idea of enforcing a desired scale of the node output by incorporating normalization into the network architecture. Specifically, it normalizes the node outputs for each training batch and introduces the normalization constants as extra parameters to train. This approach improves training speed by enabling a larger learning rate. It also leads to more robust training, by lowering the requirement on weight initialization. In addition, batch normalization partly plays the role of regularization and reduces the necessity of using dropout. Batch normalization is widely used in deep neural networks, especially for deep CNNs, and has shown to be effective in practice. However, the mechanism behind its effectiveness are still under discussion. Researchers propose different reasons why batch normalization improves training efficiency, such as smoothing objective function (Santurkar et al., 2018) or achieving length–direction decoupling (Kohler et al., 2018).

Skip connections are proposed in ResNet (He et al., 2015), in which some of the layers are jumped over by “short-cuts”: the input of those layers are added to the output to be the final output. The purpose of skip connections is too ensure a better back-propagation of the gradients. Looking at it from another angle, skip connections also enable feature reusing: The low-level features now have a chance to be directly passed to higher level instead of going through many layers. In later network design, skip connection becomes a common practice. In DenseNet (Huang et al., 2016), the authors even added skip connections between all the layers, maximizing feature reusing.

Software and Hardware Advances

One important reason that neural network proliferated in the 2010s is the advances of computer technology, in both software and hardware, because neural networks rely on heavy computations. The availability of more powerful computational units, such as GPUs, makes it possible to train deep, sophisticatedly designed neural networks on large-scale training data sets within a reasonable period of time. The feasibility and effectiveness attracts both human power and research funding to be devoted to the study of neural networks, leading to its successful development. This subsection introduces the advances of software and hardware that promote the improvement of neural networks.

Graphical Computation

Neural networks typically have thousands of parameters to train. Modern deep neural networks have much more, sometimes even over a million parameters. Although designed to be efficient, the back-propagation algorithm requires computing the gradient (i.e., the derivatives of the loss function with respect to all the parameters), which is a heavy computational burden. The traditional way of analytically deriving all these derivatives is apparently intractable. However, numerical differentiation is inefficient for not reusing the analytical expressions and also suffers from round-off errors. To overcome these difficulties, modern programing libraries for deep learning usually make use of the automatic symbolic differentiation, sometimes also called graphical computation.

A neural network model can be considered as a complicated function with the outcome being the dependent variable and the features being the independent variables. No matter how complicated it is, the function can be decomposed into a sequence of elementary arithmetic operation (e.g., addition, multiplication) and elementary functions (e.g., power, exponential). In such a way, the calculation of a function can be symbolically represented as a computational graph, with each node being either an input value or an elementary operation. In a computer program, the computational graphs are implemented as a data structure. With a computational graph and a set of input values to the graph, a generic graph evaluation engine can efficiently carry out the calculation of the function.

A more important utility of computational graphs is to speed up the calculation of derivatives in back-propagation. There are basically two approaches for it. One approach, called *symbol-to-number*, takes a computational graph and a set of numerical values for the inputs, and returns the numerical gradient at those values. It is used in programing libraries like Torch (Collobert & Kavukcuoglu, 2011) and Caffe (Jia et al., 2014). Another approach (*symbol-to-symbol*) adds additional nodes to a computational graph to provide a symbolic description of the desired derivatives. It is used in Theano (Bastien et al., 2012; Bergstra et al., 2010) and Tensorflow (Abadi et al., 2015). Computer scientists have devoted great efforts to developing these programing libraries, with the aim that all the derivatives are calculated in an automatic and efficient way. Owing to them, researchers in neural networks can focus on architecture and methodologies without worrying about implementations.

A big family of deep learning software has been developed and evolved. Because the maintenance of a software requires a lot of work, the most popular deep learning software eventually become the ones actively maintained by large companies. Up until 2019, the most popular deep learning software are Tensorflow maintained by Google and PyTorch mentained by Facebook.

Large-Scale Parallel Computation

The development of deep learning software is in parallel with the development of hardware. It turns out one of the best ways to accelerate computation is large-scale parallelization, both for neural networks and in a general sense. It is achieved by GPUs.

GPUs are specialized hardware originally designed for computer graphs and image processing (where the name comes from). They have a highly parallel structure that enables them to perform a large amount of small computational tasks simultaneously. In combination with algorithms that process large blocks of data in parallel, GPUs can be much more efficient than general-purpose central processing units. The training and inference of neural network models involves a great number of matrix calculations, which can be decomposed into small pieces of computational tasks in parallel and therefore are typical use cases for GPU acceleration. Generally speaking, using GPUs can accelerate the training of deep neural network models by a factor of 10, as shown in many benchmarking works (Shi et al., 2016; Wang, Wei, & Brooks, 2019). It also enables the application that requires a fast, online inference, such as self-driving vehicles. The popularity of GPU usage in the artificial intelligence (AI) industry actually enhances the evolution of GPUs in the direction that further facilitates deep neural networks. As of 2019, most of the GPUs that support neural network training are manufactured by Nvidia, which continuously provides special libraries for deep learning, such as CUDA tool kit and CuDNN. Another major GPU manufacturer, AMD, is also developing platforms and open-source environments to support neural network training and inference. Moreover, Google developed its own tensor processing unit, another AI accelerator that only supports Tensorflow.

Neural Networks Beyond Supervised Learning

The previous sections introduced various aspects of neural networks in the scenario of supervised learning, where a neural network serves as a prediction model for the outcome of interest given the feature variables. As a matter of fact, neural networks also make differences in many machine learning tasks beyond the supervised learning framework. This section discusses neural networks in the other two main categories of machine learning: unsupervised learning and reinforcement learning.

Neural Networks in Unsupervised Learning

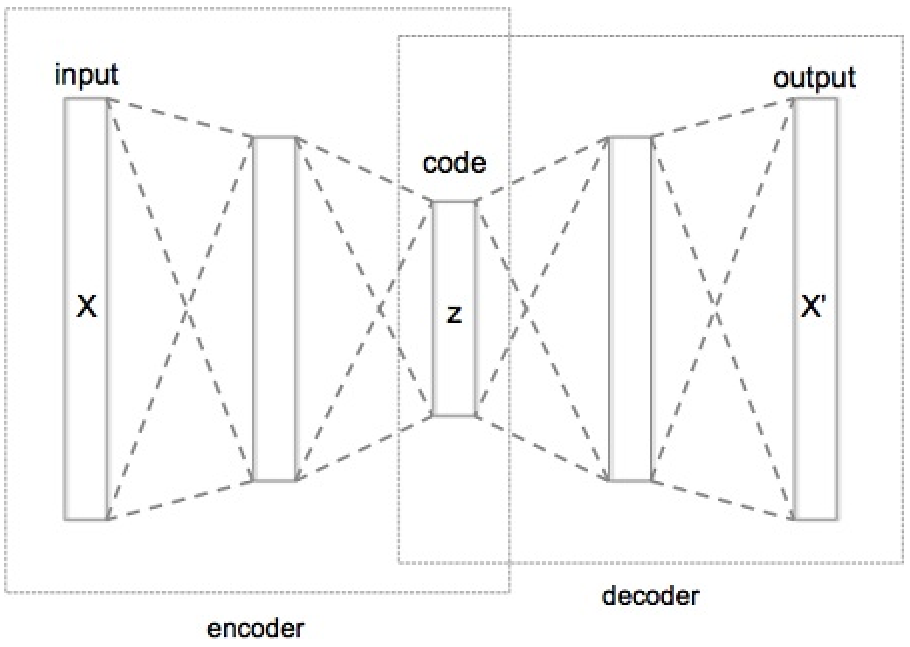
Unsupervised learning refers to the machine learning task aiming for capturing the underlying patterns in data without existing labels (outcome). The difference between supervised and unsupervised learning can be illustrated by the following simple example. Given a bunch of images of either a dog or a cat, and the labels "dog" or "cat" for all images, to build a model that can predict the label on a new image is a supervised learning problem. Given a bunch of images of either a dog or a cat, without the labels, to group the images into two categories (in some problems the number of categories may be unknown) is an unsupervised learning problem, actually a clustering problem in this particular case. It can be seen that unsupervised problems are in general more difficult.

From the perspective of statistics, the central problem of unsupervised learning is to estimate the joint distribution of the feature variables $P\left(x_1,\ x_2,\ \cdots\right)$. As a comparison, supervised learning seeks the conditional probability distribution $P(y \mid x_1,\ x_2,\ \cdots)$ or $P(x_1,\ x_2,\ \cdots \mid y)$, depending on the specific modeling strategy. Again, when the feature variables have a very high dimension and interact in a complicated way (e.g., the pixel values of an image), without the availability of labels *y*, unsupervised learning presents much hardship. Unlike supervised learning that has been utilized a lot in industry, most unsupervised learning problems are still under active research. This subsection lists a few successful neural networks in unsupervised learning problems.

Autoencoder

The purpose of an autoencoder is to compress data into low-dimension "codes" that maximally preserve the information in the data. It is a way to achieve dimension reduction. A simple autoencoder takes the form of a feed-forward neural network, consisting of an encoder and a decoder, as shown in Figure 6. The encoder usually has hidden layers with decreasing widths to fulfill the goal of compressing the input data *x* into a low-dimensional code *z* (sometimes called a bottleneck feature). The decoder has an inverse structure of the encoder, aiming to reconstruct the input *x* from the code *z*. The idea behind autoencoder is that some components of the input *x* are not informative (noise), thus a shorter code *z* is sufficient to be uncompressed into some data x' that closely matches the original data *x*. In the training of an autoencoder, the loss function is usually chosen to be a distance measure of the similarity between the reconstructed data and the original data. Since the code *z* usually represents the original data *x* in an "untangled" way, performing downstream tasks, such as classification or clustering, on *z* usually achieves better results than performing directly on *x*, and more efficiently as well.

Figure 6. Schematic structure of a simple autoencoder with three hidden layers.



Source: Wikimedia Commons/Chervinskii, https://commons.wikimedia.org/wiki/File:Autoencoder_structure.png; licensed under Creative Commons Attribution-Share Alike 4.0 International license, https://creativecommons.org/licenses/by-sa/4.0/deed.en

Autoencoder has many variants, including denoising autoencoder (Vincent et al., 2010), contractive autoencoder (Rifai et al., 2011), and variational autoencoder (VAE; Kingma & Welling, 2013), which have better performance in practice than the feed-forward autoencoder introduced before. Among them, VAE is still widely used as a generative model with good application in generating images.

Generative Models

Generative models aims to obtain an implicit joint distribution of the data. Specifically speaking, such a distribution of the data cannot be written out in any analytical form, but a model or mechanism can generate an arbitrary number of data samples. For example, a good generative model trained on a collection of human face images, should be able to generate new faces with various facial features and expressions that are not close to any of the training images. The VAE mentioned previously can play the role of a generative model. Another famous example of generative models is deep belief network (Hinton, 2009). The remainder of this subsection focuses on the generative adversarial network (GAN; Goodfellow et al., 2014), which has been the most influential generative model.

GAN is creatively designed as two neural networks that contest with each other: A network called generator attempts to generate data samples that resembles the training data. Another network called discriminator tries to distinguish the generated data from the true training data. The two networks are training together so that the generator produces better data samples while the discriminator becomes more skilled at identifying synthetic samples. In the end, even a very skillful discriminator cannot tell the difference between the generated sample and the true samples. Then, the generator becomes a good generative model.

GAN is originally famous for generating images. A GAN trained on photographs can generate new photographs with high resolution and quality that look authentic to human observers (Radford, Metz, & Chintala, 2015; Wang et al., 2018). Lately, GAN found application in various fields ranging from completion of artworks to improving texture quality in video games. It is also a hot research area.

Neural Networks in Reinforcement Learning

Reinforcement learning (Sutton & Barto, 2018) is formulated as a problem of choosing a series of actions (sometimes called policy) based on the environment to maximize a cumulative reward. A typical example is chess playing, where the pieces' location on the board is the environment, the players' moves are the actions, and the final winning is the reward. In this example, reinforcement learning is apparently difficult for many reasons. For instance, there is a large number of candidate moves for a player at each step, making the search for optimal action hard; the immediate reward at a certain step is not well defined since the cumulative reward is the final wining. These two difficulties actually represent the focuses of the study of reinforcement learning: Optimal action search seeks a balance between *exploration* of unseen moves and *exploitation* of the current best practice. Modeling of the reward function is addressed by algorithms such as Q-learning (Watkins & Dayan, 1992).

Deep neural networks are effective in modeling reward function and policy choice. One of the most famous successes of neural networks in reinforcement learning is the Go playing program AlphaGo developed by Google DeepMind (Silver et al., 2016), which beat the human world champion by a large margin. Because of its generality, reinforcement learning is studied in many disciplines and has wide applications.

Further Readings

Deng, L., & Yu, D. (2014). Deep learning: Methods and applications. *Foundations and Trends® in Signal Processing*, 7, 197–387. doi:10.1561/2000000039

ImageNet. Retrieved from http://www.image-net.org/

ImageNet Large Scale Visual Recognition Competition. Retrieved from http://image-net.org/challenges/LSVRC/

Goodfellow, I., Bengio, Y., Courville, A., & Bengio, Y. (2016). *Deep learning* (Vol. 1). Cambridge, MA: MIT press.

LeCun, Y., Bengio, Y., & Hinton, G. (2015, May). Deep learning. *Nature*, 521, 436–444.

References

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., … Zheng, X. (2015). *TensorFlow: Large-scale machine learning on heterogeneous distributed systems*. Berkeley, CA: USENIX.

Bastien, F., Lamblin, P., Pascanu, R., Bergstra, J., Goodfellow, I., Bergeron, A., … Bengio, Y. (2012, November). Theano: New features and speed improvements. Paper presented at the Deep Learning and Unsupervised Feature Learning NIPS Workshop, 2012. Retrieved from https://arxiv.org/pdf/1211.5590

Bergstra, J., Breuleux, O., Bastien, F., Lamblin, P., Pascanu, R., Desjardins, G., … Bengio, Y. (2010). Theano: A CPU and GPU math compiler in python. In *Proceedings of the 9th Python in Science Conference*, June 28 to July 3, Austin, TX, pp. 3–10.

Bishop, C. M. (2011). *Pattern recognition and machine learning*. New York, NY: Springer.

Boyd, S., & Vandenberghe, L. (2004). *Convex optimization, with corrections 2008* (1st ed.). Cambridge, England: Cambridge University Press.

Cho, K., van Merrienboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., & Bengio, Y. (2014, June). Learning phrase representations using RNN encoder-decoder for statistical machine translation. ArXiv:1406.1078 [Cs, Stat]. Retrieved from http://arxiv.org/abs/1406.1078

Choromanska, A, Henaff, M., Mathieu, M., Ben Arous, G., & LeCun, Y. (2014, November). The loss surfaces of multilayer networks. ArXiv:1412.0233 [Cs]. Retrieved from http://arxiv.org/abs/1412.0233

Collobert, R., & Kavukcuoglu, K. (2011). Torch7: A matlab-like environment for machine learning. Paper presented at BigLearn, NIPS Workshop.

Cybenko, G. (1989). Approximation by superpositions of a sigmoidal function. *MCSS*, 2, 303–314. doi:10.1007/BF02551274

Dauphin, Y. N., Pascanu, R., Gulcehre, C., Cho, K., Ganguli, S., & Bengio, Y. (n.d.). *Identifying and attacking the saddle point problem in high-dimensional non-convex optimization* (Vol. 9). Cambridge, MA: MIT Press.

Devlin, J., Chang, M. W., Lee, K., & Toutanova, K. (2018, October). BERT: Pre-training of deep bidirectional transformers for language understanding. ArXiv:1810.04805 [Cs]. Retrieved from http://arxiv.org/abs/1810.04805

Gebru, T., Krause, J., Wang, Y., Chen, D., Deng, J., Lieberman Aiden, E., & Fei-Fei, L. (2017). Using deep learning and Google street view to estimate the demographic makeup of neighborhoods across the United States. *Proceedings of the National Academy of Sciences*, 114, 13108–13113. doi:10.1073/pnas.1700035114

Gilpin, L. H., Bau, D., Yuan, B. Z., Bajwa, A., Specter, M., & Kagal, L. (2018, May). Explaining explanations: An overview of interpretability of machine learning. ArXiv:1806.00069 [Cs, Stat]. Retrieved from http://arxiv.org/abs/1806.00069

Glorot, X., Bordes, A., & Bengio, Y. (n.d.). Deep sparse rectifier neural networks. Journal of Machine Learning Research, 9.

Goodfellow, I., Bengio, Y., Courville, A., & Bengio, Y. (2016). *Deep learning* (Vol. 1). Cambridge, MA: MIT Press.

Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., … Bengio, Y. (2014). Generative adversarial nets. In Z.Ghahramani, M.Welling, C.Cortes, N. D.Lawrence, & K. Q.Weinberger (Eds.), Advances in neural information processing systems27 (pp. 2672–2680). Curran Associates. Retrieved from http://papers.nips.cc/paper/5423-generative-adversarial-nets.pdf

Hahnloser, R. H. R., Sarpeshkar, R., Mahowald, M. A., Douglas, R. J., & Sebastian Seung, H. (2000). Digital selection and analogue amplification coexist in a cortex-inspired silicon circuit. *Nature*, 405, 947. doi:10.1038/35016072

Hastie, T., Tibshirani, R., & Friedman, J. (2016). *The elements of statistical learning: Data mining, inference, and prediction* (2nd ed.). New York, NY: Springer.

He, K., Zhang, X., Ren, S., & Sun, J. (2015, December). Deep residual learning for image recognition. ArXiv:1512.03385 [Cs]. Retrieved from http://arxiv.org/abs/1512.03385

Hinton, G. E. (2009). Deep belief networks. *Scholarpedia*, 4, 5947. doi:10.4249/scholarpedia.5947

Hochreiter, S., & Schmidhuber, J. (1997). Long short-term memory. *Neural Computation*, 9, 1735–1780. doi:10.1162/neco.1997.9.8.1735

Hornik, K., Stinchcombe, M., & White, H. (1989). Multilayer feedforward networks are universal approximators. *Neural Networks*, 2, 359–366. doi:10.1016/0893-6080(89)90020-8

Huang, G., Liu, Z., van der Maaten, L., & Weinberger, K. Q. (2016, August). Densely connected convolutional networks. ArXiv:1608.06993 [Cs]. Retrieved from http://arxiv.org/abs/1608.06993

Ioffe, S., & Szegedy, C. (2015, February). Batch normalization: Accelerating deep network training by reducing internal covariate shift. ArXiv:1502.03167 [Cs]. Retrieved from http://arxiv.org/abs/1502.03167

Jia, Y., Shelhamer, E., Donahue, J., Karayev, S., Long, J., Girshick, R., … Darrell, T. (2014). Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM International Conference on Multimedia*, 675–678. MM ‘14. New York, NY: ACM. doi:10.1145/2647868.2654889

Kingma, D. P., & Welling, M. (2013, December). Auto-encoding variational bayes. ArXiv:1312.6114 [Cs, Stat]. Retrieved from http://arxiv.org/abs/1312.6114

Kohler, J., Daneshmand, H., Lucchi, A., Zhou, M., Neymeyr, K., & Hofmann, T. (2018, May). Exponential convergence rates for batch normalization: The power of length-direction decoupling in non-convex optimization. ArXiv:1805.10694 [Cs, Stat]. Retrieved from http://arxiv.org/abs/1805.10694

LeCun, Y., Bengio, Y., & Hinton, G. (2015). Deep learning. *Nature*, 521, 436.

Lundervold, A. S., & Lundervold, A. (2019). An overview of deep learning in medical imaging focusing on MRI [Special issue: Deep Learning in Medical Physics]. *Zeitschrift Für Medizinische Physik*, 29, 102–127.

doi:10.1016/j.zemedi.2018.11.002

Maas, A. L. (2013). *Rectifier nonlinearities improve neural network acoustic models*. Stanford, CA: Stanford University.

Prechelt, L. (2012). Early stopping—But when? In M.Grégoire, G. B.Orr, & K.-R.Müller (Eds.), *Neural networks: Tricks of the trade* (2nd ed., pp. 53–67). Lecture Notes in Computer Science. Berlin, Heidelberg: Springer. doi:10.1007/978-3-642-35289-8_5

Radford, A., Metz, L., & Chintala, S. (2015, November). Unsupervised representation learning with deep convolutional generative adversarial networks. ArXiv:1511.06434 [Cs]. Retrieved from http://arxiv.org/abs/1511.06434

Redmon, J., Divvala, S., Girshick, R., & Farhadi, A. (2015, June). You only look once: Unified, real-time object detection. ArXiv:1506.02640 [Cs]. Retrieved from http://arxiv.org/abs/1506.02640

Redmon, J., & Farhadi, A. (2016, December). YOLO9000: Better, faster, stronger. ArXiv:1612.08242 [Cs]. Retrieved from http://arxiv.org/abs/1612.08242

Rifai, S., Vincent, P., Muller, X., Glorot, X., & Bengio, Y. (2011). Contractive auto-encoders: Explicit invariance during feature extraction. In Proceedings of the 28th International Conference on International Conference on Machine Learning, 833–840. ICML'11. USA: Omnipress. Retrieved from http://dl.acm.org/citation.cfm?id=3104482.3104587

Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1986). Learning representations by back-propagating errors. *Nature*, 323, 533. doi:10.1038/323533a0

Santurkar, S., Tsipras, D., Ilyas, A., & Madry, A. (2018, May). How does batch normalization help optimization?ArXiv:1805.11604 [Cs, Stat]. Retrieved from http://arxiv.org/abs/1805.11604

Shi, S., Wang, Q., Xu, P., & Chu, X. (2016, August). Benchmarking state-of-the-art deep learning software tools. ArXiv:1608.07249 [Cs]. Retrieved from http://arxiv.org/abs/1608.07249

Silver, D., Huang, A., Maddison, C. L., Guez, A., Sifre, L., van den Driessche, G., … Hassabis, D. (2016). Mastering the game of go with deep neural networks and tree search. *Nature*, 529, 484–489. doi:10.1038/nature16961

Snoek, J., Larochelle, H., & Adams, R. P. (2012, June). Practical bayesian optimization of machine learning algorithms. ArXiv:1206.2944 [Cs, Stat]. Retrieved from http://arxiv.org/abs/1206.2944

Snoek, J., Rippel, O., Swersky, K., Kiros, R., Satish, N., Sundaram, N., … Adams, R. P. (2015). Scalable Bayesian optimization using deep neural networks. In Proceedings of the 32nd International Conference on International Conference on Machine Learning, 37, 2171–2180. ICML'15. JMLR.org. Retrieved from http://dl.acm.org/citation.cfm?id=3045118.3045349

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014). Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15, 1929–1958.

Sutskever, I., Vinyals, O., & Le, Q. V. (2014). Sequence to sequence learning with neural networks. In Z.Ghahramani, M.Welling, C.Cortes, N. D.Lawrence, & K. Q.Weinberger (Eds.), Advances in neural information processing systems27 (pp. 3104–3112). Curran Associates. Retrieved from http://papers.nips.cc/paper/5346-sequence-to-sequence-learning-with-neural-networks.pdf

Sutton, R. S., & Barto, A. G. (2018). *Reinforcement learning: An introduction* (2nd ed.). Cambridge, MA: A Bradford Book.

Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., … Rabinovich, A. (2015). Going deeper with convolutions. Computer vision and pattern recognition (CVPR). Retrieved from http://arxiv.org/abs/1409.4842

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, I., Gomez, A. N., … Polosukhin, I. (2017). Attention is all you need. In I.Guyon, U. V.Luxburg, S.Bengio, H.Wallach, R.Fergus, S.Vishwanathan, & R.Garnett (Eds.), Advances in neural information processing systems 30 (pp. 5998–6008). Curran Associates. Retrieved from http://papers.nips.cc/paper/7181-attention-is-all-you-need.pdf

Vincent, P., Larochelle, H., Lajoie, I., Bengio, Y., & Manzagol, P. A. (2010). Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion. *Journal of Machine Learning Research*, 11, 3371–3408.

Wang, T.-C., Liu, M. Y., Zhu, J. Y., Tao, A., Kautz, J., & Catanzaro, B. (2018). High-resolution image synthesis and semantic manipulation with conditional GANs. 8798–8807. Retrieved from http://openaccess.thecvf.com/content_cvpr_2018/html/Wang_High-Resolution_Image_Synthesis_CVPR_2018_paper.html

Wang, Y. E., Wei, G. Y., & Brooks, D. (2019, July). Benchmarking TPU, GPU, and CPU platforms for deep learning. ArXiv:1907.10701 [Cs, Stat]. Retrieved from http://arxiv.org/abs/1907.10701

Watkins, C. J. C. H., & Dayan, P. (1992). Q-Learning. *Machine Learning*, 8, 279–292. doi:10.1007/BF00992698

Weiss, G., Goldberg, Y., & Yahav, E. (2018, May). On the practical computational power of finite precision RNNs for language recognition. ArXiv:1805.04908 [Cs, Stat]. Retrieved from http://arxiv.org/abs/1805.04908

Zhang, Z., Robinson, D., & Tepper, J. (2018). Detecting hate speech on twitter using a convolution-GRU based deep neural network. In G.Aldo, R.Navigli, M.-E.Vidal, P.Hitzler, R.Troncy, L.Hollink, A.Tordai, & M.Alam (Eds.), *Lecture Notes in Computer Science: The semantic web* (pp. 745–760). Berlin, Germany: Springer International.

Zhou, B., Khosla, A., Lapedriza, A., Oliva, A., & Torralba, A. (2015, December). Learning deep features for discriminative localization. ArXiv:1512.04150 [Cs]. Retrieved from http://arxiv.org/abs/1512.04150

https://doi.org/10.4135/9781526421036888177