

Part VI

DEEP LEARNING METHODS

32

FOUNDATIONS OF DEEP LEARNING

32 FOUNDATIONS OF DEEP LEARNING [1401](#)

 32.1 Neural network fundamentals [1402](#)

 32.1.1 From machine learning to deep learning [1402](#)

 32.1.2 Neurons and neural networks [1403](#)

 32.1.2.1 Artificial neurons [1403](#)

 32.1.2.2 Artificial neural networks [1406](#)

 32.1.3 Universal approximation [1408](#)

 32.1.4 Training via backpropagation [1409](#)

 32.2 Feed-forward neural network [1415](#)

 32.2.1 Regression and classification [1415](#)

 32.2.1.1 Linear regression and classification [1415](#)

 32.2.1.2 Nonlinear extension [1418](#)

 32.2.1.3 Radial basis function network [1420](#)

 32.2.2 Image classification [1422](#)

 32.2.3 Recommender systems [1424](#)

 32.2.4 Approximating numerical partial differential equations [1425](#)

32.1 Neural network fundamentals

32.1.1 From machine learning to deep learning

Over the past decade, one of the most groundbreaking advancement in statistical learning is deep learning[1]. Deep learning methods have achieved substantial success in tracking many complex machine learning problems arising in domains of computer vision, speech, natural language processing, and artificial intelligence, in which traditional machine learning methods, like linear models, tree methods, ensemble methods, kernel tricks fall short.

In our previous chapters, we have seen that the success of traditional machine learning methods rely heavily on human expert crafted features (i.e., feature engineering). By contrast, deep learning methods utilize a layer-by-layer neural structure and its universal approximation power to extract features from data. For domains like vision, speech, reinforcement learning where data is abundant or feature engineering is challenges, deep learning methods win over traditional machine learning methods by a large margin. The ability to extract important features by learning from data even enables end-to-end learning.

One may wonder if deep learning methods will make traditional machine learning methods, such as XGBoost, obsolete. Here we compare deep learning methods and traditional machine learning method to draw insights for this question.

First, they have different requirements on the data quantity and data types. Data is playing arguably the most important role in machine learning problems. Deep learning methods and traditional machine learning methods have different requirements on the data. Deep learning methods, due to its overparameterization nature, usually requires far more training data than traditional machine learning methods. Deep learning methods are usually not suitable for discrete-value data or data with missing entries and require substantial effort to convert or complete data. On the other hand, methods like XGBoost can directly handle discrete-value data and handle missing data issue naturally.

Second, deep learning model selection and training is much more difficult than traditional machine learning methods. Designing a network for complex problems usually requires substantial trial and error. A performing architecture in one domain is usually problem specific and not generalizable to problems in other domains. Even given a promising architecture, training it to work reliably still requires heavily on tremendous experience. By contrast, traditional machine learning methods have quite standard procedure in data prepossessing, feature engineering, and training strategies, thus offering better generalization ability.

Third, deep learning methods generally lack interpretability than traditional machine learning methods, especially tree methods. Lack of model interpretability hinders the application of deep learning methods in real-world domains like finance and medicine. On the other hand, clear model interpretability in traditional methods often allows us to improve models and data usage in a more strategic way.

Fourth concern is computation cost. Deep learning methods require neural networks with much more parameters and more training data, which make deep learning method far more computationally demanding.

In the following section, we will cover the most important aspects on understanding and applying deep learning methods.

32.1.2 Neurons and neural networks

32.1.2.1 Artificial neurons

Artificial neuron [Figure 32.1.1] is the fundamental unit of an artificial neural network. Similar to the working mechanism of biological neurons, artificial neurons receive multiple inputs, aggregate and process them, and produce outputs that would be fed into other neurons.

Formally, we denote the input by a vector $x = (x_1, \dots, x_d) \in \mathbb{R}^d$. The input aggregation process leads to an intermediate result z that can be represented by $z = w^T x + b, w \in \mathbb{R}^d, b \in \mathbb{R}$, where w is a d -dimensional weight vector and b is a bias term.

After aggregation and before the final output, the intermediate z will pass through an activation function $g : \mathbb{R} \rightarrow \mathbb{R}$ to yield the final output of $g(z)$. In general, activation functions are nonlinear functions, and it turns out that the nonlinearity is the crucial foundation for the universal approximation property of an artificial neural network.

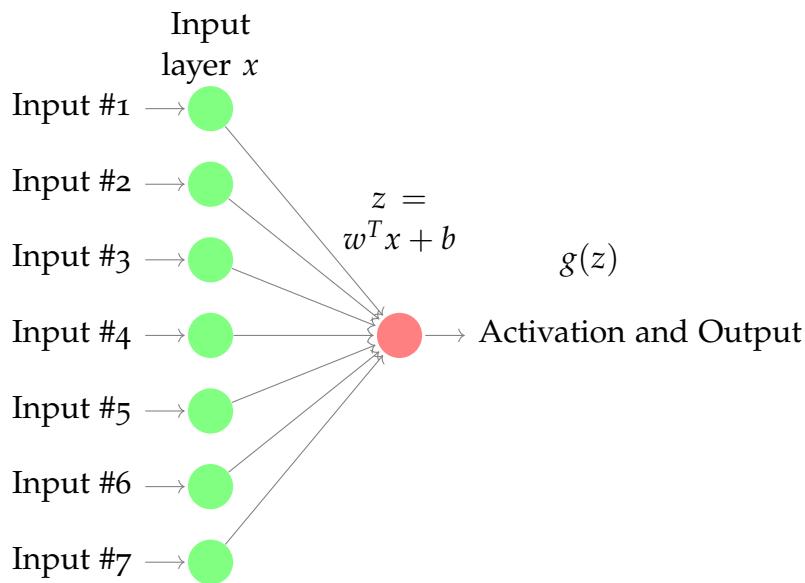


Figure 32.1.1: Scheme of an artificial neuron.

Common activation functions include:

Definition 32.1.1 (activation functions and their derivatives).

- (*sigmoid activation function*)

$$g(z) = \frac{1}{1 + \exp(-z)}, \frac{d}{dz}g(z) = g(z)(1 - g(z)).$$

- (*tanh activation function*)

$$g(z) = \tanh(z), \frac{d}{dz}g(z) = 1 - g(z)^2.$$

- (*ReLU activation function*)

$$g(z) = \max(0, z), \frac{d}{dz}g(z) = \begin{cases} 0, & \text{if } z < 0 \\ 1, & \text{if } z \geq 0 \end{cases}.$$

- (*leaky ReLU activation function*)

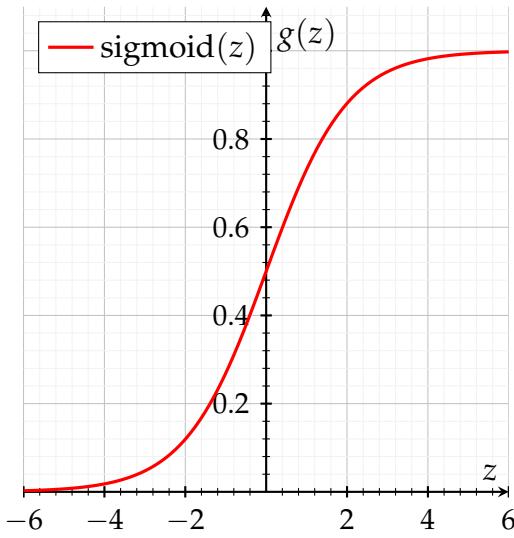
$$g(z) = \max(\gamma z, z), \frac{d}{dz}g(z) = \begin{cases} \gamma, & \text{if } z < 0 \\ 1, & \text{if } z \geq 0 \end{cases},$$

where γ is the leaky parameter usually taking small values like 0.01.

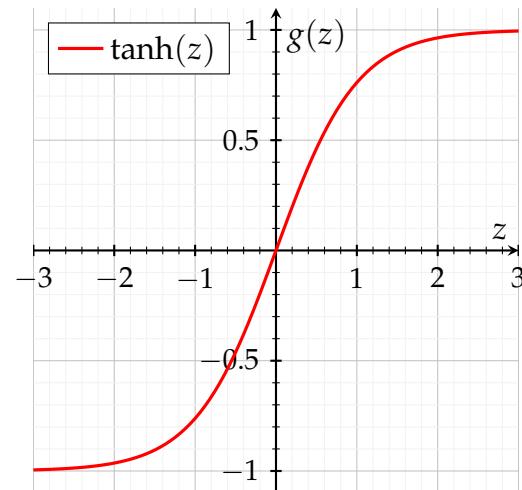
- (radial basis function)

$$g(x) = \exp\left(-\frac{\|x - c\|^2}{\sigma^2}\right),$$

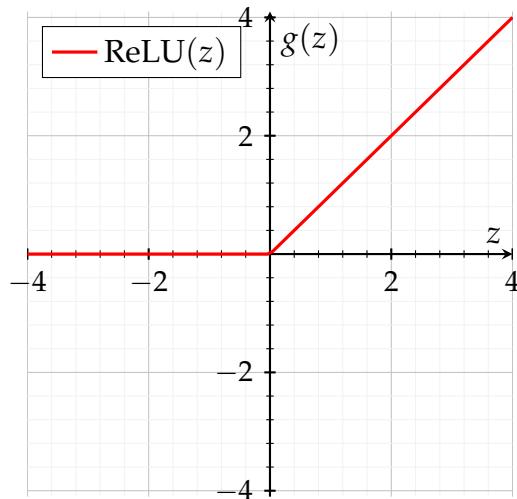
where $c \in \mathbb{R}^D$ and σ are the function parameters. Note that radial basis function directly takes $x \in \mathbb{R}^D$ as input. For its application in functional approximation, see [subsubsection 32.2.1.3](#).



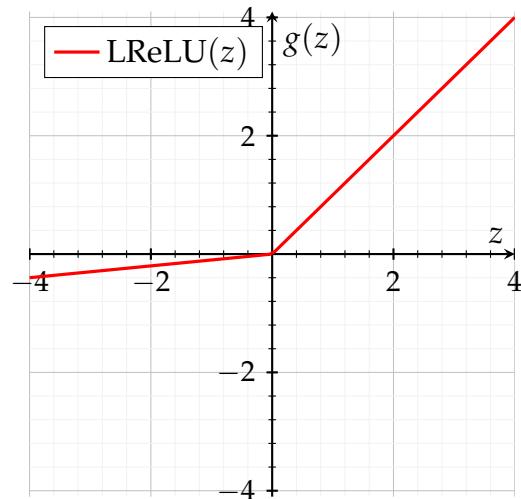
(a) Sigmoid activation function.



(b) tanh activation function.



(c) ReLU activation function.



(d) Leaky ReLU activation function.

Figure 32.1.2: Common activation functions in artificial neural networks.

32.1.2.2 Artificial neural networks

Assembling and stacking multiple neurons together gives a neural network. One relative simple neural network is the feed-forward neural network, which is showed in [Figure 32.1.3](#).

In feed-forward neural networks, neurons are arranged into layer structures. Inputs are fed into the every neuron in first layer; the generated output from he first layer neurons are then fed into the next layer; the process propagates towards final outputs. The inputs are called **input layer**, the final outputs are called **output layer**, and other middle layers called **hidden layers**.

By stacking and connecting neurons in different ways, the resulting artificial neural networks can have various architectures, including more hidden layers (i.e., deep neural networks, [Figure 32.1.4](#)), and more output units in the output layer [[Figure 32.1.5](#)].

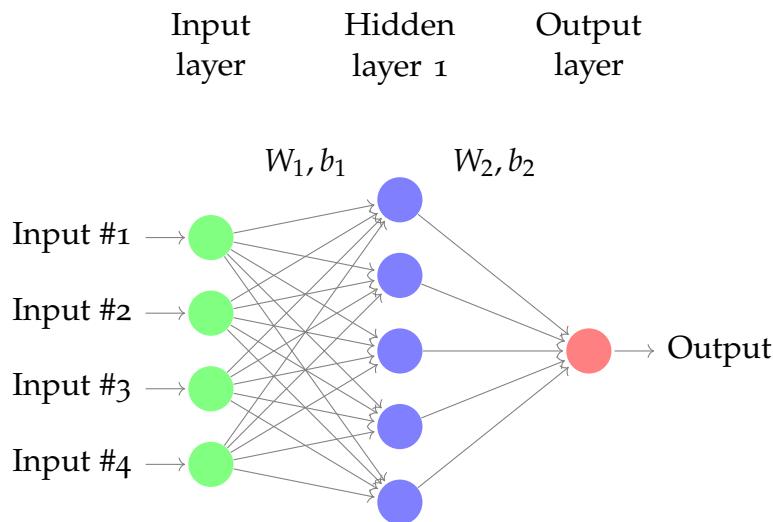


Figure 32.1.3: Scheme for an artificial neural network.

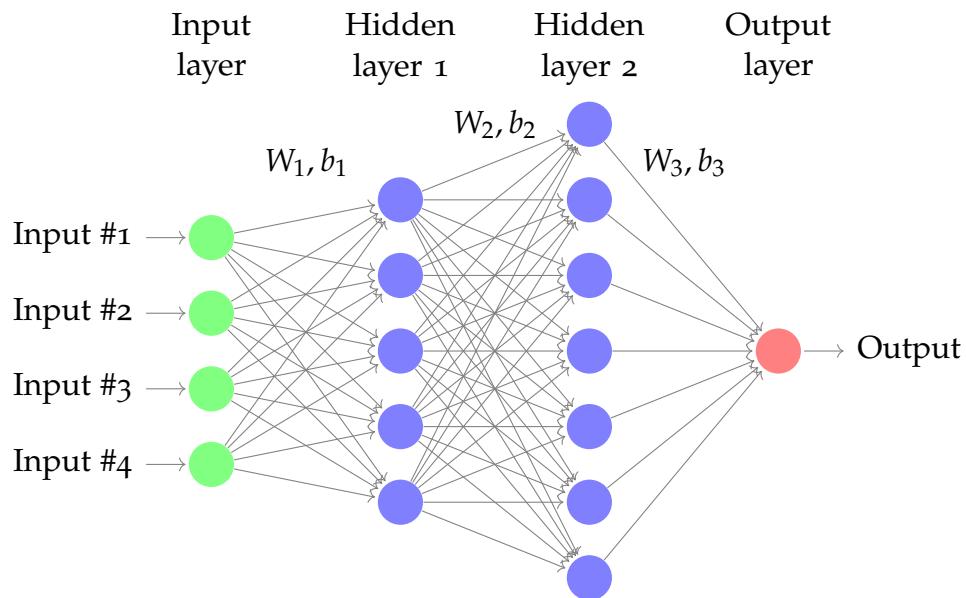


Figure 32.1.4: A four-layer feed-forward neural network.

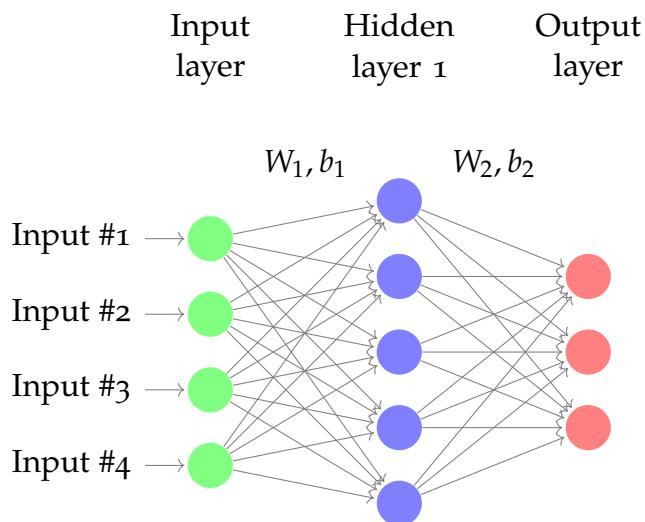


Figure 32.1.5: A three-layer feed-forward neural network with three output units.

Depending on the applications, the activation functions in the output layer can have different choices, including

- identity function for regression on unconstrained outputs (i.e., target $\in \mathbb{R}$)
- sigmoid function for regression on constrained outputs $\in [0, 1]$.
- softmax function to generate probability for C class classification.

We have following summary:

Definition 32.1.2 (output functions and derivatives). Let $z \in \mathbb{R}^H$ denote the *input vector* (after activation) to the output layer. Let $y \in \mathbb{R}^O$ denote the *output vector* of the output layer and y_i be its individual component. Then

- (Identity function)

$$y_i = g(z_i) = z_i, \frac{d}{dt}g(t) = 1.$$

- (Sigmoid function)

$$y_i = g(z_i) = \frac{1}{1 + \exp(-z_i)}, \frac{d}{dt}g(t) = g(t)(1 - g(t)).$$

- (Softmax function) Softmax function, $g(x)$, is a function mapping from $\mathbb{R}^H \rightarrow [0, 1]^C$. The parameter associated with $g(x)$ is a weighting matrix $W = [w_1, w_2, \dots, w_C] \in \mathbb{R}^{H \times C}$ such that

$$z_i = g_i(x) = \frac{\exp(x^T w_i)}{\sum_{k=1}^K \exp(x^T w_i)}, i = 1, 2, \dots, C.$$

32.1.3 Universal approximation

One of most critical properties of a neural network is its universal approximation power to arbitrary continuous functions, provided that there are enough neurons in the hidden layer [Theorem 32.1.1]. Many machine learning problems boil down to seeking an function mapping that minimize a specified loss function. The universal approximation capability of neural networks make them an ideal tool for a broad range of applications if model parameters can be properly estimated. For machine learning problems of different complexity, we can adjust accordingly the size of function space a neural network can represent by adjusting the number of neurons and the number of layers. The flexibility and the approximation power of neural networks make them an indispensable tool to tackle many challenging real-world machine learning problems.

Theorem 32.1.1 (Universal Approximation Theorem). [2][3] Let $\phi(\cdot)$ be a non-constant, bounded, and monotonically-increasing continuous function. Let I_m denote the m -dimensional unit hypercube $[0, 1]^m$. The space of continuous functions on I_m is denoted by $C(I_m)$. Then, given any $\epsilon > 0$ and any function $f \in C(I_m)$, there exist an integer N , real constants $v_i, b_i \in \mathbb{R}$ and real vectors $w_i \in \mathbb{R}^m$, where $i = 1, 2, \dots, N$, such that we may define:

$$F(x) = \sum_{i=1}^N v_i \psi(w_i^T x + b_i),$$

as an approximate realization of the function f ; $|F(x) - f(x)| < \epsilon$ for all $x \in I_m$. In other words, functions $F(x)$ of the form

$$\sum_{i=1}^N v_i \psi(w_i^T x + b_i)$$

are dense in $C(I_m)$.

32.1.4 Training via backpropagation

Neural networks provide a compact, structural parameterized representation of complex continuous mappings. What is more important, we can use simple gradient descent methods to gradually improve the approximation accuracy for a function mapping of interest. Although at first glance one might think its gradient computation can be quite complex considering multiple layer structures, it turns out that its layer-by-layer structure offers a systematic and structural way to calculate gradient with ease [4].

Definition 32.1.3 (forward process). Consider a four-layer feed-forward neural network ([Figure 32.1.6](#)). Denote input as x_0 , and parameters $W_1, b_1, W_2, b_2, W_3, b_3$, and activation functions g_1, g_2, g_3 . Then the forward process consists of the following procedures:

$$\begin{aligned} x_1 &= g_1(W_1 x_0 + b_1) \\ x_2 &= g_2(W_2 x_1 + b_2) \\ x_3 &= g_3(W_3 x_2 + b_3) \end{aligned}$$

Then the network output is $y = g(x_3)$.

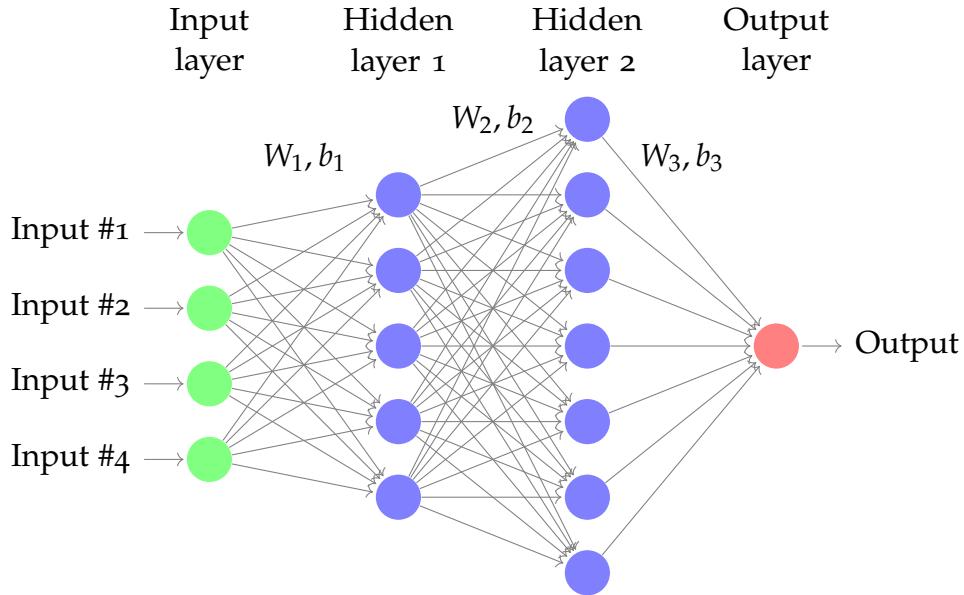


Figure 32.1.6: A four-layer feed-forward neural network.

Lemma 32.1.1 (gradient calculation via backpropagation for a four-layer feed-forward neural network). Consider a four-layer feed-forward neural network. Denote input as x_0 , and parameters $W_1, b_1, W_2, b_2, W_3, b_3$, and activation functions g_1, g_2, g_3, g_4 .

Denote the forward feeding result as x_1, x_2, x_3 .

Consider the loss function on the output x_3 given by

$$E = \frac{1}{2} \|g_4(x_3) - t\|^2,$$

where t is the target value. Then we have

- $\frac{\partial E}{\partial W_3} = \delta_3 x_2^T, \frac{\partial E}{\partial b_3} = \delta_3,$

where $\delta_3 = g'_4(x_3 - t) \odot g'_3(W_3 x_2 + b_3)$.

- $\frac{\partial E}{\partial W_2} = \delta_2 x_1^T, \frac{\partial E}{\partial b_2} = \delta_2,$

where $\delta_2 = W_3^T \delta_3 \odot g'_2(W_2 x_1 + b_2)$.

- $\frac{\partial E}{\partial W_1} = \delta_1 x_0^T, \frac{\partial E}{\partial b_1} = \delta_1,$

where $\delta_1 = W_2^T \delta_2 \odot g'_1(W_1 x_0 + b_1)$.

Proof. (1) (a)

$$\begin{aligned}\frac{\partial E}{\partial W_3} &= g'_4(x_3 - t) \frac{\partial x_3}{\partial W_3} \\ &= [g'_4(x_3 - t) \odot g'_3(W_3 x_2 + b_3)] \frac{\partial W_3 x_2}{\partial W_3} \\ &= [g'_4(x_3 - t) \odot g'_3(W_3 x_2 + b_3)] x_2^T\end{aligned}$$

(b)

$$\begin{aligned}\frac{\partial E}{\partial b_3} &= g'_4(x_3 - t) \frac{\partial x_3}{\partial b_3} \\ &= [g'_4(x_3 - t) \odot g'_3(W_3 x_2 + b_3)] \frac{\partial b_3}{\partial b_3} \\ &= [g'_4(x_3 - t) \odot g'_3(W_3 x_2 + b_3)]\end{aligned}$$

(2) (a)

$$\begin{aligned}\frac{\partial E}{\partial W_2} &= (x_3 - t) \frac{\partial x_3}{\partial W_2} \\ &= [(x_3 - t) \odot g'_3(W_3 x_2 + b_3)] \frac{\partial W_3 x_2}{\partial W_2} \\ &= \delta_3 \frac{\partial W_3 x_2}{\partial W_2} \\ &= W_3^T \delta_3 \frac{\partial x_2}{\partial W_2} \\ &= W_3^T \delta_3 \odot g'_2(W_2 x_1 + b_2) \frac{\partial W_2 x_1}{\partial W_2} \\ &= \delta_2 x_1^T\end{aligned}$$

(b)

$$\begin{aligned}\frac{\partial E}{\partial b_2} &= (x_3 - t) \frac{\partial x_3}{\partial b_2} \\ &= [(x_3 - t) \odot g'_3(W_3 x_2 + b_3)] \frac{\partial W_3 x_2}{\partial b_2} \\ &= \delta_3 \frac{\partial W_3 x_2}{\partial b_2} \\ &= W_3^T \delta_3 \frac{\partial x_2}{\partial b_2} \\ &= [W_3^T \delta_3 \odot g'_2(W_2 x_1 + b_2)] \frac{\partial b_2}{\partial b_2} \\ &= \delta_2\end{aligned}$$

(3) (a)

$$\begin{aligned}
 \frac{\partial E}{\partial W_1} &= (x_3 - t) \frac{\partial x_3}{\partial W_1} \\
 &= [(x_3 - t) \odot g'_3(W_3 x_2 + b_3)] \frac{\partial W_3 x_2}{\partial W_1} \\
 &= \delta_3 \frac{\partial W_3 x_2}{\partial W_1} \\
 &= W_3^T \delta_3 \frac{\partial x_2}{\partial W_1} \\
 &= W_3^T \delta_3 \odot g'_2(W_2 x_1 + b_2) \frac{\partial W_2 x_1}{\partial W_1} \\
 &= \delta_2 \frac{\partial W_2 x_1}{\partial W_1} \\
 &= [W_3^T \delta_2 \odot g'_1(W_1 x_0 + b_1)] \frac{\partial W_1 x_0}{\partial W_1} \\
 &= \delta_1 x_0^T
 \end{aligned}$$

(b)

$$\begin{aligned}
 \frac{\partial E}{\partial W_1} &= (x_3 - t) \frac{\partial x_3}{\partial W_1} \\
 &= [(x_3 - t) \odot g'_3(W_3 x_2 + b_3)] \frac{\partial W_3 x_2}{\partial W_1} \\
 &= \delta_3 \frac{\partial W_3 x_2}{\partial W_1} \\
 &= W_3^T \delta_3 \frac{\partial x_2}{\partial W_1} \\
 &= W_3^T \delta_3 \odot g'_2(W_2 x_1 + b_2) \frac{\partial W_2 x_1}{\partial W_1} \\
 &= \delta_2 \frac{\partial W_2 x_1}{\partial W_1} \\
 &= [W_3^T \delta_2 \odot g'_1(W_1 x_0 + b_1)] \frac{\partial b_1}{\partial b_1} \\
 &= \delta_1
 \end{aligned}$$

□

Remark 32.1.1 (gradients when weights and biases are zero). It is curious to know the gradients when weights W and biases b are zero. Take a three-layer neural network as example. We have

- Forward results $x_1, x_2, x_3 = 0$.
- $\delta_3 = g'_4(-t) \odot g'_3(0)$ and thus

$$\frac{\partial E}{\partial W_3} = 0, \frac{\partial E}{\partial b_3} = \delta_3.$$

- $\delta_2 = 0$ and thus

$$\frac{\partial E}{\partial W_2} = 0, \frac{\partial E}{\partial b_2} = 0.$$

- $\delta_1 = 0$ and thus

$$\frac{\partial E}{\partial W_1} = 0, \frac{\partial E}{\partial b_1} = 0.$$

Theorem 32.1.2 (gradient calculation via backpropagation for a multiple-layer feed forward neural network).^a

Consider a multi-layer feed forward neural network with L layers. Denote input as x_0 , and parameters $W_1, b_1, W_2, b_2, \dots, W_L, b_L$, and activation functions g_1, g_2, \dots, g_L .

Denote the forward feeding result as x_1, x_2, \dots, x_L .

Consider the loss function on the output x_3 given by

$$E = f(x_3).$$

Then it follows that

$$\frac{\partial E}{\partial W_i} = \delta_i x_{i-1}^T, \frac{\partial E}{\partial b_i} = \delta_i, i = 1, 2, \dots, L,$$

where

$$\begin{aligned}\delta_L &= \frac{\partial E}{\partial x_L} \odot g'_L(W_L x_{L-1} + b_L) \\ \delta_i &= W_{i+1}^T \delta_{i+1} \odot g'_L(W_i x_{i-1} + b_i), i = 1, 2, \dots, L-1.\end{aligned}$$

^a also see [link](#)

Remark 32.1.2 (vanishing gradient issues). In training a neural network via gradient methods, one common hurdle is that, in some cases, the gradients could be vanishingly small, significantly slowing down the learning process.

One cause for vanishing gradients is the usage of some activation functions such as the hyperbolic tangent function that are saturate for large inputs and weights. For example, the sigmoid activation function $\sigma(x)$ has gradient given by

$$\sigma'(x) = \sigma(x)(1 - \sigma(x)).$$

Clearly, when $\sigma(x)$ approaches 0 or 1, $\sigma'(x)$ approaches 0. Another cause is associated with the deep architecture for neural networks. As the number of layers exceeds 20 layers, the gradient calculated via chain rule [Theorem 32.1.2] could involve multiplication of multiple small numbers.

Methods to address vanishing gradient issues include:

- Use non-saturate activations, such as ReLU and leaky ReLU.
- Design short-cut connections between layer. The most successful example is ResNet [subsection 34.2.4](#).

32.2 Feed-forward neural network

32.2.1 Regression and classification

32.2.1.1 Linear regression and classification

Although feed-forward multi-layer neural networks, also known as MLPs, are mainly used as approximators for complex mappings, here we show that MLPs are also flexible enough to model linear regression and classifications (logistic regression). The architecture for linear models consists of only the input and output layers [Figure 32.2.1]. The number of neurons in the input layer specifies the number of input features, where the number of neurons in the output layer specifies the number of output labels. Particularly, one output neuron for single output linear regression and binary classification and multiple output neurons for multiple output linear regression and multi-class classification. For regression, the loss function is usually mean squared error, whereas for classification, the loss function is cross entropy loss on the Sigmoid or Softmax of the output.

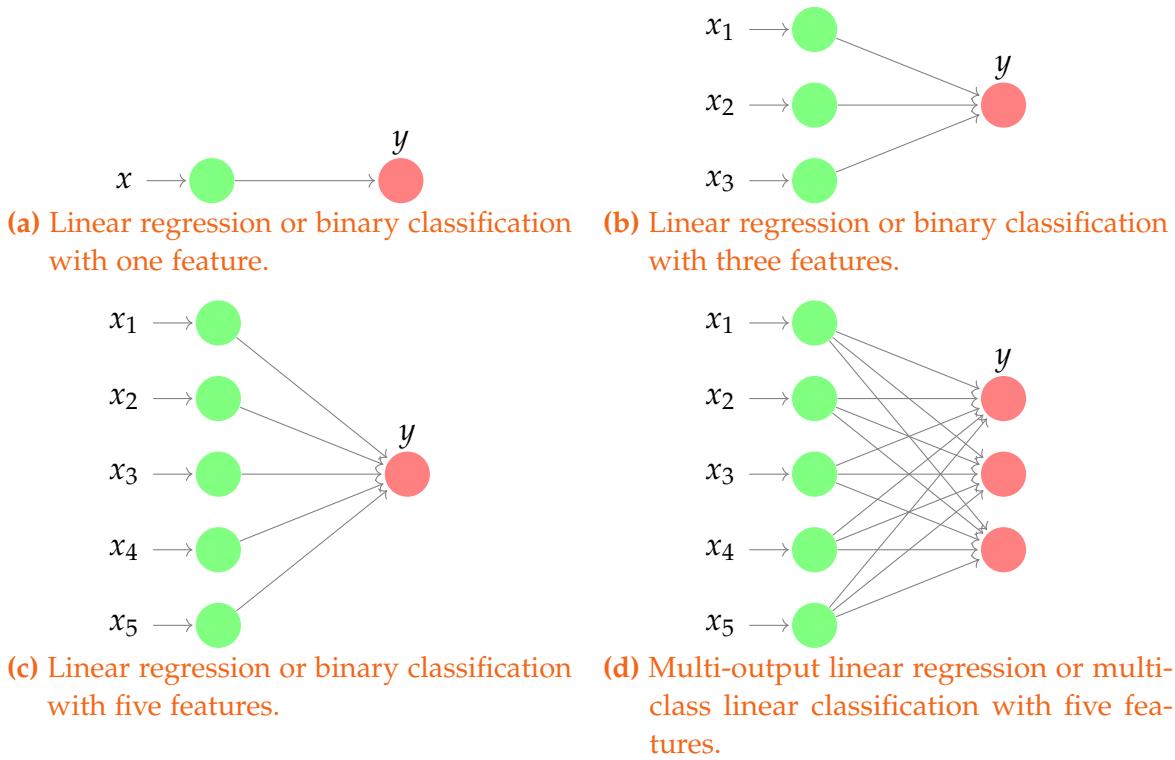


Figure 32.2.1: Simple neural network architecture for linear regression and classification application.

Specially, the mathematical model underlying [Figure 32.2.1\(a\)\(b\)\(c\)](#) for single output is simply

$$\hat{y} = \begin{cases} Wx + b, & \text{linear regression} \\ \sigma(Wx + b), & \text{linear classification} \end{cases},$$

where $W, x \in \mathbb{R}^D, b \in \mathbb{R}$ and σ is the Sigmoid activation function. \hat{y} in the context of linear classification is interpreted as predicted probability.

Given n training examples $\{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \dots, (x^{(n)}, y^{(N)})\}, x^{(i)}, y^{(i)} \in \mathbb{R}$, the loss function associated with a regression problem is typically mean square error (MSE) given by

$$L_{MSE} = \frac{1}{N} \sum_{i=1}^N (y^{(i)} - \hat{y}^{(i)})^2.$$

The loss function associated with binary classification ($y^{(i)} \in \{0, 1\}$) is typical binary cross entropy (BCE) given by

$$L_{BCE} = -\frac{1}{N} \sum_{i=1}^N y^{(i)} \ln \hat{y}^{(i)} - (1 - y^{(i)}) \ln (1 - \hat{y}^{(i)}).$$

One can refer to logistic regression for more discussion on the formulation [[section 25.1](#)].

The **multiple output regression** model underlying [Figure 32.2.1\(d\)](#) is

$$\hat{y} = Wx + b,$$

where $\hat{y}, b \in \mathbb{R}^p, x \in \mathbb{R}^D, W \in \mathbb{R}^{D \times p}$. The loss is similarly formulated using L2 norm given by

$$L_{MSE} = \frac{1}{N} \sum_{i=1}^N \|y^{(i)} - \hat{y}^{(i)}\|_2^2.$$

The **multi-class classification** (C classes) model underlying [Figure 32.2.1\(d\)](#) is

$$\hat{y} = \text{Softmax}(Wx + b),$$

where $\hat{y} \in \mathbb{R}^C, \sum_{i=1}^C \hat{y}_i = 1$. Specifically,

$$\hat{y}_i = \frac{\exp(z_i)}{\sum_{i=1}^C \exp(z_i)},$$

where $z = Wx + b, z \in \mathbb{R}^C$ is the logits for C classes.

Assume training example label $y^{(i)}$ is taking integer value among $\{1, \dots, C\}$, the multi-class cross entropy loss (CE) is given by

$$\begin{aligned} L_{CE} &= -\frac{1}{N} \sum_{i=1}^N \sum_{k=1}^C I(y^{(i)} = k) \ln \hat{y}_k^{(i)} \\ &= -\frac{1}{N} \sum_{i=1}^N \ln \frac{\exp(z_{y^{(i)}}^{(i)})}{\sum_{j=k}^C \exp(z_j^{(i)})} \\ &= -\frac{1}{N} \sum_{i=1}^N \left(z_{y^{(i)}}^{(i)} - \ln \sum_{j=1}^C \exp(z_j^{(i)}) \right). \end{aligned}$$

To understand the impact of each logit on the loss function, we consider the loss for one example whose ground truth is i class. The loss is given by

$$l_i = -(z_i - \ln \sum_{j=1}^C \exp(z_j)).$$

The derivative of l_i with respect to z_j , $j = i$ and $j \neq i$, are given by

$$\begin{aligned} \frac{\partial l_i}{\partial z_i} &= -1 + \frac{\exp(z_i)}{\sum_{j=1}^C \exp(z_j)} \\ \frac{\partial l_i}{\partial z_j} &= \frac{\exp(z_i)}{\sum_{j=1}^C \exp(z_j)}, i \neq j. \end{aligned}$$

Clearly, if the true label is i , we can reduce the loss if we increase z_i and increase the loss by increase other logits.

Example 32.2.1 (Linear classification regression). [Figure 32.2.2](#) shows the linear classification model for MNIST multi-class classification realized via the architecture in [Figure 32.2.1](#). Specifically, there are 784 input neurons (each neuron takes each pixel value in an image) and 10 output neurons. After training, visualizing the learned weighted matrix of size (784, 10) clearly indicates that the neural network is acting as feature extractor to capture the essential features in distinguish different digits.



Figure 32.2.2: Visualization of first layer weight for a one-layer linear multi-class classification neural network

32.2.1.2 Nonlinear extension

The linear regression via one-layer network can be extended to polynomial regression, which can be viewed as linear regression of higher degree features x, x^2, \dots, x^d . Figure 32.2.3 shows polynomial regression of different degrees using architecture in Figure 32.2.1. The degree of freedoms is simply controlled by the number of input neurons. Since the sample data is generated from a three degree polynomial plus noise, we see good fit at $d = 3$, underfitting at $d = 1$ and overfitting at $d > 3$.

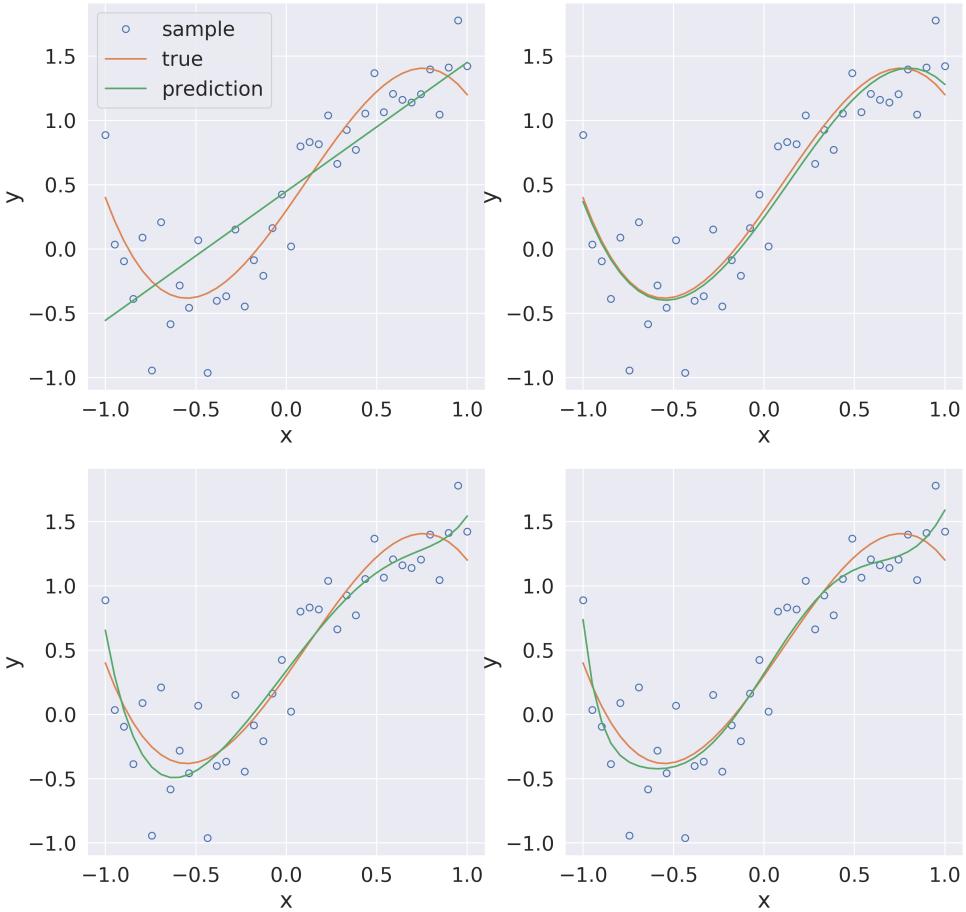


Figure 32.2.3: Polynomial regression with degree $d = 1, 3, 6, 10$.

A more general approach to nonlinear regression is to use multi-layer feed-forward neural network. Feed-forward neural network can carry out fairly complicated nonlinear regression with even one hidden layer. As showed in , we perform nonlinear regression on

$$y = 0.1x \cos(x) + 0.1\epsilon, \epsilon \sim N(0, 1).$$

The network used has one hidden layer with 20 neurons.

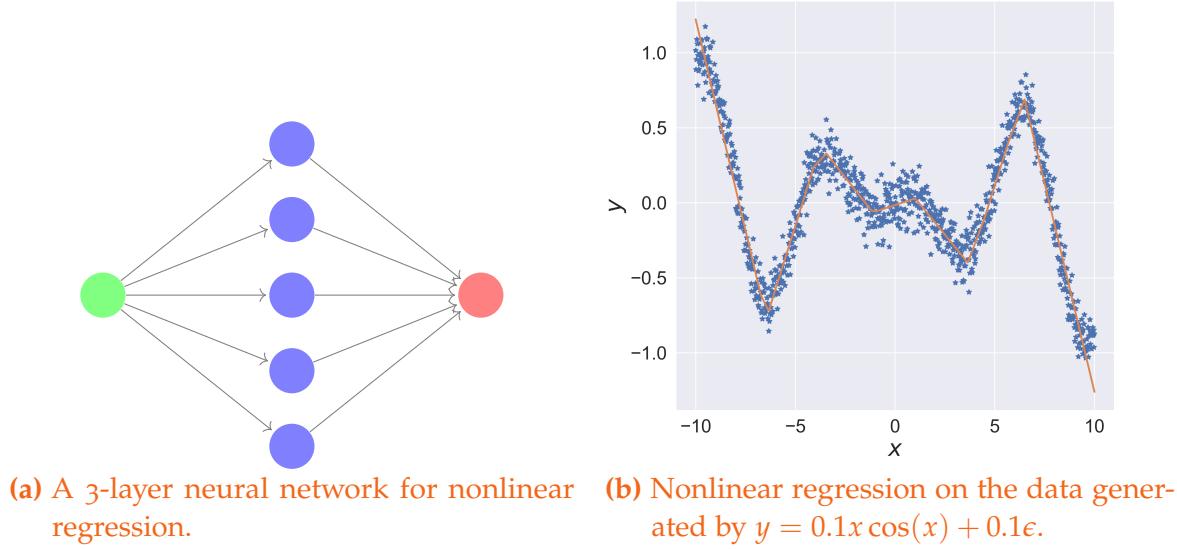


Figure 32.2.4: Multi-layer feed-forward neural network for nonlinear regression.

32.2.1.3 Radial basis function network

Radial basis function network (RBFN) is a type of feed-forward neural network widely used as functional approximators[5]. A typical RBFN has three layers: an input layer , a hidden layer with a radial basis activation function (RBF) and a linear output layer. The radial basis activation function taking input $x \in \mathbb{R}^D$ is given by

$$g(x) = \exp\left(-\frac{\|x - c\|^2}{\sigma^2}\right),$$

where $c \in \mathbb{R}^D$ and $\sigma \in \mathbb{R}$ are the function parameters. The output $\hat{y} = (\hat{y}_1, \dots, \hat{y}_M)$, $\hat{y}_i \in \mathbb{R}$ is given by

$$\hat{y}_i = \sum_{k=1}^H W_{ik} g_k(x) + b_i, i = 1, \dots, M.$$

where W_{ik} are weight parameters connecting from hidden layer to the output layer, $g_k(x)$ is the activated value from a RBF, and $b_i \in \mathbb{R}$ is the bias term. More compactly, we can write

$$y = WG(x),$$

where $g(x) = (g_1(x), \dots, g_H(x))^T$ is the vector of activated values.

Note that in RBFN, between the input layer and the hidden layer, there is no weight parameters, instead, the parameters are those in the radial basis function.

Given training samples $D = \{(x^{(1)}, y^{(1)}), \dots, (x^{(N)}, y^{(N)})\}$, the parameter learning in RBFN is via minimization of the following loss function

$$L = \frac{1}{N} \sum_{i=1}^N \|y^{(i)} - \hat{y}_i^{(i)}\|^2 = \frac{1}{N} \|y^{(i)} - Wg(x_i) + b\|^2.$$

This is a non-convex optimization problem. Although it is possible to directly perform back-propagation gradient descent, such a brute force approach does not take advantage of the locality property of radial basis functions.

Another approach is a two-stage learning process. In the first stage, we select RBF centers c and variance σ by clustering; in the second stage, we perform back-propagation to update weights between the hidden layer and the output layer and to fine tune the centers c .

The clustering process proceeds with grouping x into different cluster based on appropriate distance metrics. In [section 30.4](#), we have discussed a number of clustering algorithms. The number of clusters is either pre-specified or learned from data. After determining centers c from clustering algorithms, the variance parameter σ can be selected via several methods[5]. For example, it can be chosen as the average distance among inputs x to their nearest centers.

The second stage involving learning W and fine tuning c via gradient descent. The gradients with respect to W and c are given by

$$\frac{\partial L}{\partial W_{ij}} = -\frac{1}{N} \sum_{n=1}^N (y_i^{(n)} - \hat{y}_i^{(n)}) g_j(x^{(n)}; c_j),$$

and

$$\frac{\partial E}{\partial c_j} = \frac{1}{N} W_{ij} \sum_{n=1}^N \sum_{i=1}^M (y_i^{(n)} - \hat{y}_i^{(n)}) g'_j(x^{(n)}; c_j) \frac{x^{(n)} - c_j}{\sigma^2}$$

where $i = 1, \dots, M, j = 1, \dots, H$ and $g'(\cdot) = \exp(\cdot)$.

There is another simplified and analytical method to fit weight parameters. Suppose we have already determine the RBF parameters, we directly solve optimal W via theories in basis function regression [[section 24.3](#)] and multivariate linear regression [[subsection 15.5.1](#)]. The resulting expression is

$$Y = WG,$$

and optimal W is

$$W^* = YG^T(GG^T)^{-1},$$

where $Y \in \mathbb{R}^{M \times N}$ is the outcome matrix, and $G \in \mathbb{R}^{H \times N}$ is given by $G = [g(x^{(1)}), \dots, g(x^{(N)})]$.

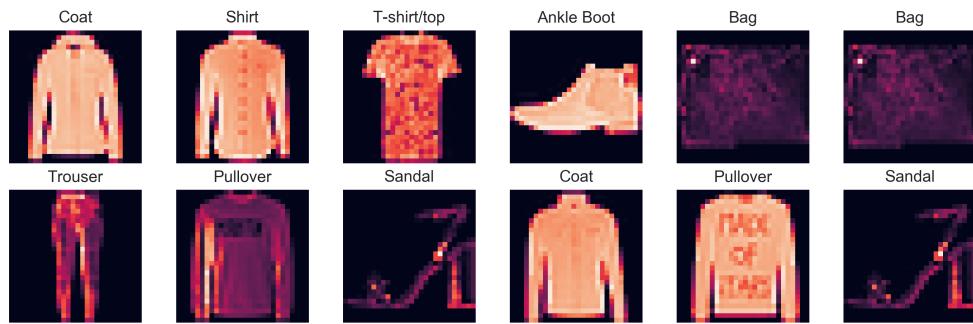


Figure 32.2.5: Example images from the Fashion MNIST dataset.

32.2.2 Image classification

The Fashion MNIST data set contains tiny 28 by 28 images of different types of clothing [Figure 32.2.5], including T-shirt/top, trouser, pullover, dress, coat, sandal, shirt, sneaker, bag, and ankle boot.

In the following MNIST Fashion example, we use a five-layer feed forward neural network consisting of 784, 256, 128, 64, and 10 neural units. Softmax is applied to the last layer to get the classification probability for 10 classes and ReLu nonlinearity is applied to the output of all previous layer. Each image is flattened into a long vector of 784 components before being fed into the network.

Let one output from the final softmax output layer be $p^{(i)} \in \mathbb{R}^K$, where K is the number of classes and $\sum_{j=1}^K p_j^{(i)} = 1$. Then the loss function for N examples is the cross-entropy loss given by

$$L = -\frac{1}{N} \sum_{i=1}^N \mathbf{1}(y^{(i)}=j) \log p_j^{(i)},$$

where $y^{(i)} \in \{1, 2, \dots, K\}$ is the label of example i .

After several epochs of training, we can arrive at an accuracy of around 86%. Classification results are summarized in the confusion matrix [Figure 32.2.6] and exemplified by a set of randomly selected samples [Figure 32.2.7]. The confusion matrix shows that the neural network has encountered difficulty in distinguishing the following items:

- Ankle boot and sneaker.
- Shirt, T-shirt, pullover, and coat.

It agrees with our expectation since these items look similar.

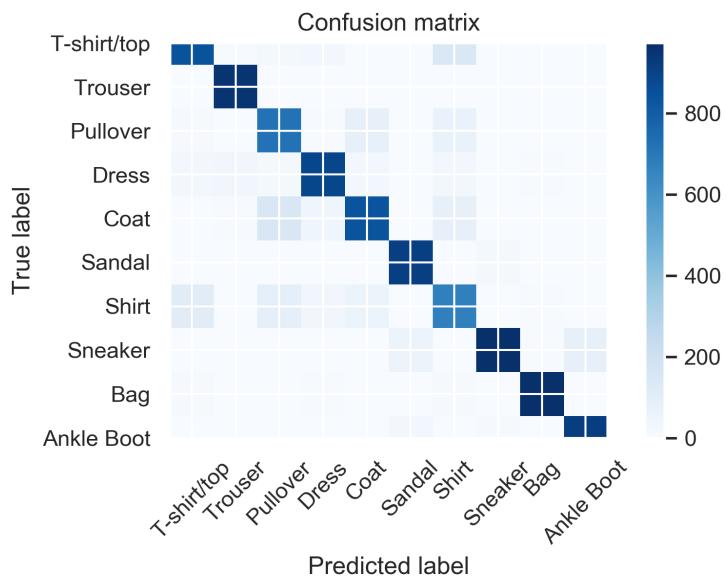


Figure 32.2.6: The confusion matrix from fashion MNIST classification results.

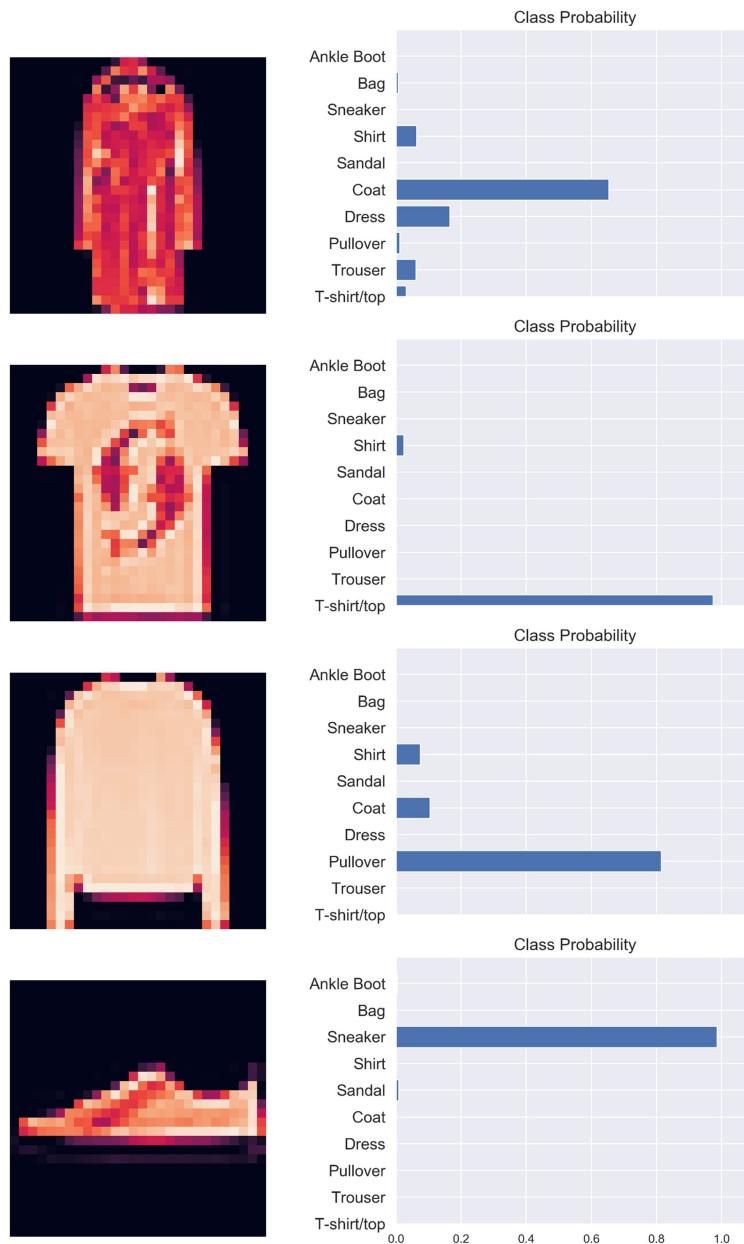


Figure 32.2.7: Classification results for a set of randomly selected samples.

32.2.3 Recommender systems

In product (books, movies, goods, etc) recommendation applications, we are given a **rating matrix** [??], which contains customer's rating on each item. This rating matrix has many missing entries, representing the customer has not yet rate and bought the item.

Our goal is to predict these missing entries and recommend items with highest predicted ratings to customers.

In subsection 30.2.2, we discuss a matrix factorization approach to the rating matrix. This matrix decomposition can also be implemented via feed-forward neural network with a bottleneck layer [Figure 32.2.8].

is for neural networks. The input is a one-hot encoded user id, which represents the user feature matrix for the hidden layer weight vector of the data. For the hidden layer, it can represent the user feature vector, and for the output weight vector of the hidden layer, it represents the item feature matrix, Item feature matrix. Finally, the output of the neural network is the user's preference for the item.

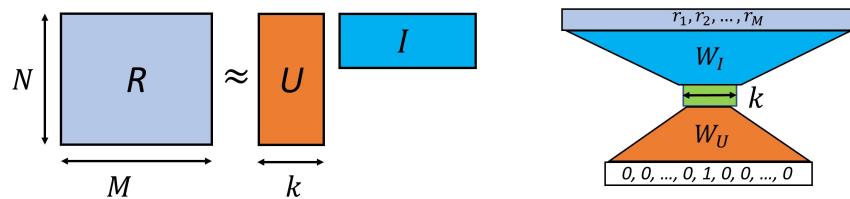


Figure 32.2.8: Feed-forward neural network for rating matrix decomposition in a recommender system.

Also see [6] for a comprehensive review on deep learning methods for recommender systems,

32.2.4 Approximating numerical partial differential equations

Partial differential equations (PDE) have been widely used to describe dynamics and equilibrium properties of physical and chemical systems. Numerical methods using grids to solve PDE are mainly limited to a few dimensions. Recently, deep learning methods have entered the territory of numerical PDE, with the remarkable achievement in solving high-dimensional PDE[7–9].

In classification and regression tasks, the loss function of a neural network are basically errors with respect to target values observed in the data. In PDE, the loss function is derived from the deviations from the underlying physical law defined by the PDE.

In the following, we give an example of solving forward-backward stochastic differential equations, which have found important applications in financial modeling and stochastic optimal control[10].

We consider coupled forward-backward stochastic differential equations of the general form

$$\begin{aligned} dX_t &= \mu(t, X_t, Y_t, Z_t) dt + \sigma(t, X_t, Y_t) dW_t, \quad t \in [0, T] \\ X_0 &= \xi \\ dY_t &= \varphi(t, X_t, Y_t, Z_t) dt + Z_t^T \sigma(t, X_t, Y_t) dW_t, \quad t \in [0, T] \\ Y_T &= g(X_T) \end{aligned}$$

where X_t is a m -dimensional stochastic process, W_t is a p -dimensional Brownian motion, $\sigma(t, X_t, Y_t) \in \mathbb{R}^{m \times p}$, $\mu(t, X_t, Y_t, Z_t)$, $\varphi(t, X_t, Y_t, Z_t)$ are the drifts. Y_t is a one-dimensional stochastic process with terminal condition given by $g(X_T)$. Z_t is another m -dimensional stochastic process. To solve this forward-backward stochastic differential equation, we need to find a solution consists of the stochastic processes X_t , Y_t , and Z_t . It has been shown that this forward-backward stochastic differential equation is related to PDE of the form [10]

$$u_t = f(t, x, u, Du, D^2u)$$

with terminal condition $u(T, x) = g(x)$, where $u(t, x)$ is the unknown solution and

$$\begin{aligned} f(t, x, y, z, \gamma) &= \varphi(t, x, y, z) - \mu(t, x, y, z)^T z - \frac{1}{2} \text{Tr} [\sigma(t, x, y) \sigma(t, x, y)' \gamma] \\ Y_t &= u(t, X_t) \\ Z_t &= Du(t, X_t) \end{aligned}$$

Now we give some examples with such formulation.

Example 32.2.2 (Black-Scholes equation). In mathematical finance, the dynamics of multiple stock prices and the value of an option with these stocks as underlyings can be described by the following simplified forward-backward stochastic differential equations.

$$\begin{aligned} dX_t &= \sigma \text{diag}(X_t) dW_t, \quad t \in [0, T] \\ X_0 &= \xi \\ dY_t &= r \left(Y_t - Z_t^T X_t \right) dt + \sigma Z_t' \text{diag}(X_t) dW_t, \quad t \in [0, T] \\ Y_T &= g(X_T) \end{aligned}$$

where X_t are the stocks price dynamics govern by geometric Brownian motion, and Y_t is the option value price with governing equation derived from Black-Scholes framework. The payoff function of the option at expiry T is given by $g(X_T)$.

The classic numerical PDE method to solve X_t, Y_t, Z_t is to use finite difference on a grid in the state space, which is computational prohibitive when dimensionality of the problem is large. Deep learning method will use a multi-layer feed forward neural network to directly approximate $u(t, X_t)$, which takes (X_t, t) as the input. Further leveraging the automatic differentiation of neural network software (such as PyTorch and TensorFlow), we can also obtain $Z_t = Du(t, X_t)$.

The weight of the neural network is updated iteratively. First consider the discretized forms of the stochastic differential equation, which are given by

$$\begin{aligned} X^{n+1} &\approx X^n + \mu(t^n, X^n, Y^n, Z^n) \Delta t + \sigma(t^n, X^n, Y^n) \Delta W^n \\ Y^{n+1} &\approx Y^n + \varphi(t^n, X^n, Y^n, Z^n) \Delta t + (Z^n)^T \sigma(t^n, X^n, Y^n) \Delta W^n \end{aligned}$$

where we have discretized the time space $[0, T]$ into N periods t^0, \dots, t^{N-1} , with Δt being the time step size. $\Delta W^n \sim \mathcal{N}(0, \Delta t)$. The loss function penalize the deviation from above differential equations, and it is given by

$$\begin{aligned} L &= \sum_{m=1}^M \sum_{n=0}^{N-1} \left| Y_m^{n+1} - Y_m^n - \varphi(t^n, X_m^n, Y_m^n, Z_m^n) \Delta t^n - (Z_m^n)^T \sigma(t^n, X_m^n, Y_m^n) \Delta W_m^n \right|^2 \\ &\quad + \sum_{m=1}^M \left| Y_m^N - g(X_m^N) \right|^2 \end{aligned}$$

where the subscript $m = 1, \dots, M$ denote different realizations. Note that the neural network comes into play via $Y_m^n = u(t^n, X_m^n)$, $Z_m^n = Du(t^n, X_m^n)$, and the simulation trajectory

$$X_m^{n+1} = X_m^n + \mu(t^n, X_m^n, Y_m^n, Z_m^n) \Delta t^n + \sigma(t_m^n, X_m^n, Y_m^n) \Delta W_m^n$$

and $X_m^0 = \xi$ for every m .

We can summarize the iterative procedure in [algorithm 61](#).

Algorithm 61: Solve forward backward stochastic differential equation via deep learning.

Input: Initial model parameter θ for u_θ .

1 Set $n = 1$.

2 **repeat**

3 Generate M trajectories based on

$$X^{n+1} \approx X^n + \mu(t^n, X^n, Y^n, Z^n) \Delta t + \sigma(t^n, X^n, Y^n) \Delta W^n$$

$$Y^{n+1} \approx Y^n + \varphi(t^n, X^n, Y^n, Z^n) \Delta t + (Z^n)^T \sigma(t^n, X^n, Y^n) \Delta W^n$$

4 Perform gradient descent to update u_θ using loss function defined by

$$L = \sum_{m=1}^M \sum_{n=0}^{N-1} \left| Y_m^{n+1} - Y_m^n - \varphi(t^n, X_m^n, Y_m^n, Z_m^n) \Delta t^n - (Z_m^n)^T \sigma(t^n, X_m^n, Y_m^n) \Delta W_m^n \right|^2 \\ + \sum_{m=1}^M \left| Y_m^N - g(X_m^N) \right|^2$$

5 where $Y_m^n = u_\theta(t^n, X_m^n)$, $Z_m^n = Du_\theta(t^n, X_m^n)$.

6 Set $n = n + 1$.

7 **until** stopping criteria is met;

Output: u_θ

BIBLIOGRAPHY

1. LeCun, Y., Bengio, Y. & Hinton, G. Deep learning. *nature* **521**, 436–444 (2015).
2. Cybenko, G. Approximation by superpositions of a sigmoidal function. *Mathematics of control, signals and systems* **2**, 303–314 (1989).
3. Hornik, K., Stinchcombe, M. & White, H. Multilayer feedforward networks are universal approximators. *Neural networks* **2**, 359–366 (1989).
4. Rumelhart, D. E., Hinton, G. E. & Williams, R. J. Learning representations by back-propagating errors. *nature* **323**, 533–536 (1986).
5. Wu, Y., Wang, H., Zhang, B. & Du, K.-L. Using radial basis function networks for function approximation and classification. *ISRN Applied Mathematics* **2012** (2012).
6. Zhang, S., Yao, L., Sun, A. & Tay, Y. Deep learning based recommender system: A survey and new perspectives. *ACM Computing Surveys (CSUR)* **52**, 1–38 (2019).
7. Raissi, M., Perdikaris, P. & Karniadakis, G. E. Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations. *Journal of Computational Physics* **378**, 686–707 (2019).
8. Raissi, M. & Karniadakis, G. E. Hidden physics models: Machine learning of nonlinear partial differential equations. *Journal of Computational Physics* **357**, 125–141 (2018).
9. Han, J., Jentzen, A. & Weinan, E. Solving high-dimensional partial differential equations using deep learning. *Proceedings of the National Academy of Sciences* **115**, 8505–8510 (2018).
10. Chassagneux, J.-F., Chotai, H. & Muûls, M. *Introduction to Forward-Backward Stochastic Differential Equations* 11–42 (Springer, 2017).

33

NETWORK TRAINING AND OPTIMIZATION

33	NETWORK TRAINING AND OPTIMIZATION	1429
33.1	Optimization algorithms	1430
33.1.1	Motivation	1430
33.1.2	Full Batch gradient descent	1431
33.1.3	Minibatch stochastic gradient descent	1432
33.1.4	Adaptive gradient method	1433
33.1.4.1	Adaptive gradient (AdaGrad)	1433
33.1.4.2	RMSProp & AdaDelta	1433
33.1.5	Momentum method	1435
33.1.6	Combined together: adaptive momentum (Adam)	1436
33.2	Training and regularization techniques	1438
33.2.1	Choices of activation functions	1438
33.2.2	Weight initialization	1438
33.2.2.1	Motivation	1438
33.2.2.2	Xavier initialization	1439
33.2.2.3	He initialization	1440
33.2.3	Data normalization	1440
33.2.3.1	Initial data standardization	1440
33.2.3.2	Batch normalization	1441
33.2.3.3	Layer normalization	1443
33.2.4	Regularization	1444
33.2.4.1	L_p regularization	1444

33.2.4.2	Weight decay	1444
33.2.4.3	Early stopping	1445
33.2.4.4	Dropout	1445
33.2.4.5	Data augmentation	1447
33.2.4.6	Label smoothing	1447
33.3	Notes on Bibliography	1448

33.1 Optimization algorithms

33.1.1 Motivation

Backpropagation provides a systematic way to calculate gradients of weight parameters, which can be used to construct gradient-descent based algorithms. In real-world applications, neural network optimization is of substantial high dimensionality since neural networks tend to be over-parameterized, with number of parameters going over 1 million. The nonlinearity of activation functions also makes the optimization to be of non-convex nature. Studies have suggested that the following two challenges deserve special attention while we are designing optimization algorithms.

The first challenge is to escape **saddle point** in the objective function surface more effectively. A saddle point [Figure 33.1.1] is a point that minimizes some coordinates but maximize some other coordinates. Saddle points have zero-gradient and thus can easily trap algorithms solely update minimizers based on gradients of current step. In low-dimensional non-convex optimization, efforts are directed towards escaping local minimums of low quality. By contrast, high-dimensional non-convex optimization is more focused on escaping saddle points. In a high-dimensional optimization problem, the occurrence of saddle points are much more frequent than local minimums. Consider a M dimensional optimization, a local minimum has to be the minimizer in all M dimensions. If a candidate point has probability p to be a minimizer in one dimension, then the probability for it to be a local minimum is p^M , a extremely small number when M is large.

The second challenge is regarding **moving fast at relatively flat functional surfaces**, particularly in the vicinity of local minimums. An over-parameterized neural networks plus additional training strategies (e.g., weight initialization, normalization, etc.), for stability concerns, further restrict each parameter to only contribute in nearly negligible amount to the final output. As a result, regions in the vicinity of local minimums are often quite flat, causing slow progress for gradient descent methods with fixed small step size.

This section we will cover different types of first-order gradient based optimization methods that make attempt to address aforementioned challenges. Although in some applications, second order method through Hessians are also used, first-order methods are the main stream optimization methods due to their simplicity and scalability.

Besides optimization algorithms covered in this section, additional training strategies are often required to promote convergence speed and stabilize training process. We will cover these training strategies in the next section.

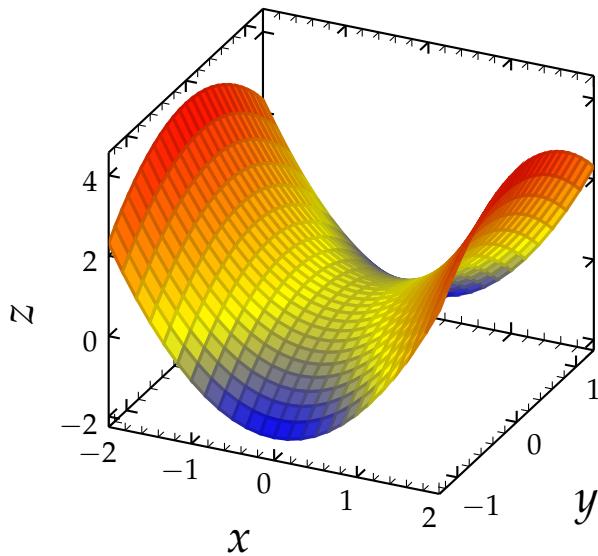


Figure 33.1.1: An example saddle point at $(0, 0, 0)$, which locally minimizes the x direction but maximizes the y direction..

33.1.2 Full Batch gradient descent

Full batch gradient descent is usually used in relatively simple machine learning problems using small-scale neural networks. A typical algorithm is showed in [algorithm 62](#). In every iteration, the gradient is computed from all data samples. All weight parameters will take a step of size α_k along the negative gradient direction.

Algorithm 62: Full batch gradient descent algorithm

Input: Scheduled learning rate α_k , initial model parameter θ

1 Set $k = 1$

2 **repeat**

3 Compute gradient estimate over N samples via

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

4 Apply update $\theta_k = \theta_k - \alpha_k \hat{g}_k$.
Set $k = k + 1$.

5 **until** stopping criteria is met;

Output: θ_k

Empirically, large learning rate is preferred at the beginning of the training, where the iterate is usually far from the minimum; smaller learning rate is preferred as the

training proceeds to avoid overshooting or even divergence near the local minimum. Learning rate is usually set to decay according to some prescribed schedules. Notably, sufficient condition to guarantee convergence is given by

$$\sum_{k=1}^{\infty} \alpha_k = \infty, \sum_{k=1}^{\infty} \alpha_k^2 < \infty.$$

Common learning rate schedules include

- inverse time decay $\alpha_k = \alpha_0 \frac{1}{1+\beta k}$.
- exponential decay $\alpha_k = \alpha_0 \beta^k, \beta < 1$ or $\alpha_k = \alpha_0 \exp(-\beta k), \beta > 0$.

33.1.3 Minibatch stochastic gradient descent

Batch gradient descent is usually not ideal for neural network application that involves large-size training data, such as image recognition and natural language processing. Evaluating the gradient over the whole set of training data is computational prohibitive; moreover, many samples are similar, making the gradient of the whole data sample is simply the multiplier of the gradient of a much smaller, representative sample data set.

Minibatch stochastic gradient descent uses a random sample of the training data set to estimate the gradient on each step. A typical algorithm is showed in [algorithm 63](#)

Algorithm 63: Minibatch stochastic gradient descent algorithm

Input: learning rate ϵ_k , initial model parameter θ .

- 1 Set $k = 1$
 - 2 **repeat**
 - 3 Sample a minibatch of training samples of size m $(x^{(i)}, y^{(i)}), i = 1, 2, \dots, m$.
 - 4 Compute a gradient estimate over this minibatch samples via
 - 5
$$\hat{g}_k = \frac{1}{m} \nabla_{\theta} \sum_{i=1}^m L(f(x^{(i)}; \theta), y^{(i)}).$$
 - 6 Apply update $\theta_k = \theta_k - \epsilon_k \hat{g}_k$.
 - 7 Set $k = k + 1$.
 - 8 **until** stopping criteria is met;
 - 9 **Output:** θ_k
-

Remark 33.1.1 (choice of minibatch size).

- The estimation quality of gradient via minibatch gradient descent is strongly affected by the minibatch size. In general, the gradient estimate is unbiased, irrespective of the choice of minibatch size, but its variance will decrease as the minibatch increases.
- For larger minibatch size, we can increase learning rate since the estimated gradient is more certain. There is an empirical **Linear Scaling Rule**: When the minibatch size is multiplied by k , multiply the learning rate by k [1].

33.1.4 Adaptive gradient method

33.1.4.1 Adaptive gradient (AdaGrad)

For simple stochastic gradient methods, we need to set the learning rate hyperparameter or even dynamically schedule learning rate, which is usually a difficult task or problem specific. Further, a uniform learning rate is usually not an effective way for high-dimensional gradient descent methods, since one learning rate could be too large for one-dimension but, on the contrary, too small for another dimension. The AdaGrad algorithm[2] addresses the issue by choosing different learning rates for each dimension. The algorithm adaptively scales the learning rate for each dimension based on the accumulated gradient magnitude on that dimension so far.

Let G_t be the **accumulated gradient** up to iteration t , given by

$$G_t = G_{t-1} + \hat{g}_t \odot \hat{g}_t, G_0 = 0.$$

The parameter update is given by

$$\theta_t = \theta_{t-1} - \frac{\alpha_0}{\delta + \sqrt{G_t}} \hat{g}_t$$

where α_0 is the initial learning speed, usually set at a small number (say, $1e-9$ to $1e-7$) and δ is a small positive constant to avoid division by zero.

As we can see, the learning rate in AdaGrad is monotonically decreasing, which may dramatically slow down the convergence as the learning rate becomes too small. In general, AdaGrad algorithm performs best for convex optimization.

33.1.4.2 RMSProp & AdaDelta

As we mentioned before, AdaGrad tends to shrink learning rate too aggressively. This is an advantage when applying AdaGrad to convex function optimization as it enables the algorithm to converge fast and stably. However, non-convex function optimization usually require large, adaptive learning rate to escape bad local minimum and converge stably to better local minimums.

The first remedy is to prevent the learning rate from shrinking too fast. Let G_t be the accumulated gradient up to iteration t , given by

$$G_t = \rho G_{t-1} + (1 - \rho)g_t \odot g_t, G_0 = 0.$$

Then we compute update

$$\theta_t = \theta_{t-1} - \frac{\alpha_0}{\delta + \sqrt{G_t}} \hat{g}_t.$$

which is the core part of the RMSProp algorithm[3, lec. 6].

How this modification can make the G_t smaller than that in the AdaGrad, as can be seen from following expansion.

(expansion of G_t) In RMSProp, we have

$$\begin{aligned} G_t &= \rho G_{t-1} + (1 - \rho)g_t \odot g_t \\ &= \rho G_{t-2} + \rho(1 - \rho)g_t \odot g_t + \rho(1 - \rho)g_{t-1} \odot g_{t-1} \\ &\approx g \cdot g(1 - \rho)(1 + \rho + \dots + \rho^{t-1}) \\ &= (1 - \rho^t)g \cdot g \end{aligned}$$

where we assume $g_t \cdot g_t \approx g \cdot g, \forall t$. Clearly, we have roughly,

$$G_t^{\text{RMSProp}} \approx \frac{(1 - \rho^t)}{t} G_t^{\text{AdaGrad}}, G_t^{\text{AdaGrad}} \approx t g \odot g.$$

The second remedy is to correct step size. In Newton based method with Hessian, the gradient step is given by $H^{-1}g$, which has scaled step size given by

$$\Delta\theta \propto H^{-1}g \propto \frac{\frac{\partial f}{\partial \theta}}{\frac{\partial^2 f}{\partial \theta^2}} \propto \theta.$$

On the other hand, step size in SGD has an unit of $\frac{1}{\theta}$ and step size in RMSProp has an unit of 1. We introduce

$$S_t = \rho S_{t-1} + (1 - \rho)\Delta\theta \odot \Delta\theta$$

and use $\sqrt{S_t + \epsilon}$ to approximate the unit of θ .

Finally, the updating step size becomes

$$\theta_t = \theta_{t-1} - \frac{\sqrt{\delta + S_t}}{\sqrt{\delta + G_t}} \hat{g}_k$$

where α_0 is the initial learning speed, usually set at a small number (say, 1e-9 to 1e-7).

The resulting algorithm is **AdaDelta**[4].

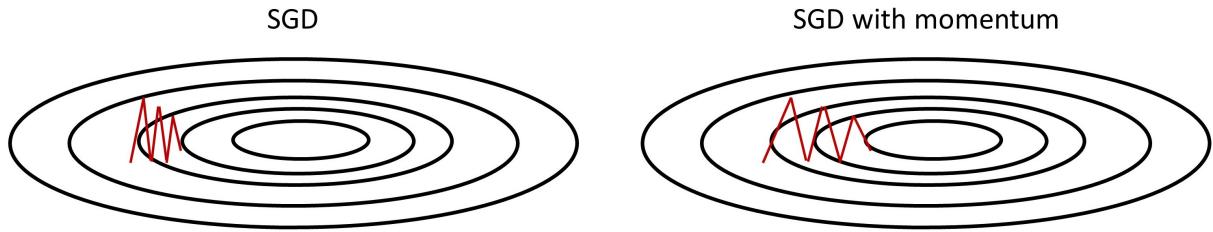


Figure 33.1.2: SGD without momentum and with momentum. SGD with momentum can accumulate gradient/velocity in horizontal direction and move faster towards the minimum located at the center.

33.1.5 Momentum method

Simple SGD with small learning rate can lead to extremely slow learning for functional surfaces with long, narrow valleys [Figure 33.1.2, [5]]. One intuition inspired by the physics of a heavy ball falling down is to add momentum to the gradient descent steps. Mathematically, adding momentum is equivalent to adding historical weighted averaged gradient to the current gradient. The total gradient will then be hopefully large enough to enable fast movement on relatively flat regions.

Consider the gradient

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

We can compute a speed via

$$v_k = \alpha v_{k-1} - \alpha_k \hat{g}_k,$$

which is used to update parameter θ via $\theta_k = \theta_{k-1} + v_k$.

To see that the update velocity is the weighted average gradient, we now show that the velocity is an exponentially decaying moving average (similar to AR(1) process) of the negative gradients, given by

$$\begin{aligned} v_k &= \alpha v_{k-1} - \hat{g}_k \\ &= \alpha(\alpha v_{k-2} - \hat{g}_{k-1}) - \hat{g}_k \\ &= \alpha^2 v_{k-2} - \alpha \hat{g}_{k-1} - \hat{g}_k \\ &= \dots \\ &= \sum_{i=0}^{\infty} \alpha^i \hat{g}_{k-i} \end{aligned}$$

Another empirically better momentum scheme is the Nesterov momentum. In the aforementioned classic momentum, we first correct velocity by adding historical velocities and then make descent step according to the corrected velocity; in Nesterov momentum, we first make a step into velocity direction and then make a correction to a velocity vector based on new location.

The velocity update is then given by

$$v_k = \alpha v_{k-1} - \alpha_k \nabla f(\theta_k + \alpha v_{k-1}).$$

Then we update $\theta_k = \theta_k + v_k$.

33.1.6 Combined together: adaptive momentum (Adam)

By combining the ideas of momentum and adaptive learning rate, we yield Adam, one of most popular gradient descent algorithm in deep learning community[6]. The name Adam is derived from adaptive moment estimation. As its name suggests, Adam will compute velocity via momentum type of averaging and adjust the learning rate using inverse of accumulated gradients.

Specifically, the velocity is computed via

$$\begin{aligned} M_k &= \rho_1 M_{k-1} + (1 - \rho_1) \hat{g}_k \\ \tilde{M}_k &= \frac{M_k}{1 - \rho_1^k} \end{aligned}$$

and the accumulated gradient is compute via

$$\begin{aligned} G_k &= \rho_2 G_{k-1} + (1 - \rho_2) \hat{g}_k \odot \hat{g}_k \\ \tilde{G}_k &= \frac{G_k}{1 - \rho_2^k} \end{aligned}$$

Note that we correct the M_k and G_k be dividing the factor $1 - \rho_i^k, i = 1, 2$ to get the average estimation (see [subsubsection 33.1.4.2](#) for more details.)

The final algorithm is given by [algorithm 64](#).

Algorithm 64: Adam stochastic gradient descent algorithm.

Input: learning rate α (set to 0.001), initial model parameter θ , decay parameter

$\delta = 1e - 8$ and decay rates ρ_1 (set to 0.9), ρ_2 (set to 0.999).

1 Set $k = 1$.

2 Set $M_k = 0, G_k = 0$.

3 **repeat**

4 Sample a minibatch of training samples of size m $(x^{(i)}, y^{(i)}), i = 1, 2, \dots, m$.

5 compute gradient estimate over minibatch N samples via

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

6 Accumulate $M_k = \rho_1 M_{k-1} + (1 - \rho_1) \hat{g}_k$. Accumulate

$$G_k = \rho_2 G_{k-1} + (1 - \rho_2) \hat{g}_k \odot \hat{g}_k.$$

7 Correct biases

$$\tilde{M}_k = \frac{M_k}{1 - \rho_1^k}, \tilde{G}_k = \frac{G_k}{1 - \rho_2^k}.$$

8 Apply update

$$\theta_k = \theta_{k-1} - \frac{\alpha \cdot \tilde{M}_k}{\delta + \sqrt{\tilde{G}_k}}$$

9 Set $k = k + 1$.

9 **until** stopping criteria is met;

Output: θ_k

33.2 Training and regularization techniques

33.2.1 Choices of activation functions

The underpinning of the universal approximation power of neural networks are the usage of nonlinear activation functions. If using simple linear activation functions, a neural network reduces to an affine transformation. The choice of nonlinear activation function can strongly affect the performance for gradient based optimization methods. A large input to activation functions like Sigmoid and Tanh will produce vanishing gradient value and cause dramatic slowdown in the learning process. Here we briefly discuss the advantages and pitfalls in choosing different activation functions.

Sigmoid and Tanh activation functions are from early generations of the activation family. They have smooth gradients and restricted outputs, which might meet the requirement of some applications. However, neural networks with Sigmoid and Tanh suffer from Vanishing gradient because of their saturation effect [[Remark 32.1.2](#)].

ReLU is another popular activation function and is empirically found to facilitate faster learning because it will not saturate as long as input is positive. However, when input is negative, ReLU provides zero gradient, which is known as the *dying ReLU* problem.

Leaky ReLU remedies the zero gradient problem by introducing a smaller negative gradient when input is negative. Leaky ReLU is widely used in training generative adversarial networks [??].

33.2.2 Weight initialization

33.2.2.1 Motivation

The goal of weight initialization is to ensure the output of each layer before activation to be neither too small nor too large[[7](#)] in order to accelerate the training process. Large output will make activation functions saturated, causing vanishing gradient and slow learning. Small outputs will not exploit the nonlinearity of the activation functions and dramatically reduce the approximation power of neural networks.

Common practice is to set zero bias and randomly initialize weights to have zero mean and specified variance. In the following, we cover two most common weight initialization methods.

Remark 33.2.1 (zero initialization). When we set all weights and biases to 0, all derivatives except for the last layer's bias will be zero [Remark 32.1.1]. And this will stay the same as we continue the iterations.

33.2.2.2 Xvaier initialization

Let $w^{(k)}$ be the weight of k layer, which receives the output $a^{(k-1)}$ from its previous layer. Assume components $w_i^{(k)}$ and $a_i^{(k-1)}$ have zero mean and independent to each other, then their dot product output $a_j^{(k)}$ will have zero mean. $a^{(k)}$ will have variance of

$$\begin{aligned} \text{Var}[a^k] &= \text{Var}\left[\sum_{i=1}^{n^{(k-1)}} w_i^k a_i^{(k-1)}\right] \\ &= \sum_{i=1}^{n^{(k-1)}} \text{Var}[w_i^k] \text{Var}[a_i^{(k-1)}] \\ &= n^{(k-1)} \text{Var}[w_i^k] \text{Var}[a_i^{(k-1)}] \end{aligned}$$

where $n^{(k-1)}$ is the neuron number in the $k-1$ layer.

Since we want $\text{Var}[a_i^{(k-1)}]$ and $\text{Var}[a_j^{(k)}]$ to have similar scale, which gives

$$1 = n^{(k-1)} \text{Var}[w_i^k] \implies \text{Var}[w_i^k] = \frac{1}{n^{(k-1)}}.$$

Usually in practice, we require

$$\text{Var}[w_i^k] = \frac{2}{n^{(k-1)} + n^{(k)}}.$$

Depending on the desired distribution, we have

- If initial weight distribution is Gaussian, then

$$w_i^{(k)} \sim N(0, \frac{2}{n^{(k-1)} + n^{(k)}}).$$

- If initial weight distribution is uniform, then

$$w_i^{(k)} \sim U\left(-\sqrt{\frac{6}{n^{(k-1)} + n^{(k)}}}, \sqrt{\frac{6}{n^{(k-1)} + n^{(k)}}}\right).$$

This is known as **Xvaier** initialization[7].

33.2.2.3 He initialization

Xavier initialization is suitable for neural networks with activation functions of Sigmoid or Tanh. For ReLU activation function, by probability about half outputs are zeros. Following similar arguments above, we have

$$\text{Var}[a^k] = \frac{1}{2} n^{(k-1)} \text{Var}[w_i^k] \text{Var}[a_i^{(k-1)}].$$

By the similar variance argument, we have

$$\text{Var}[w_i^k] = \frac{2}{n^{(k-1)}}.$$

Then we have

- If initial weight distribution is Gaussian, then

$$w_i^{(k)} \sim N(0, \frac{2}{n^{(k-1)}}).$$

- If initial weight distribution is uniform, then

$$w_i^{(k)} \sim U(-\sqrt{\frac{6}{n^{(k-1)}}}, \sqrt{\frac{6}{n^{(k-1)}}}).$$

This is known as **He** initialization[8]

33.2.3 Data normalization

33.2.3.1 Initial data standardization

Similar to weight initialization, initial data standardization plays critical role in neural network training performance. A feature of large scale will cause the neural network to ignore other features with small values. Further, input data with different scales can cause additional difficulty in gradient based optimization method as it creates unnecessary valley in the objective function surface.

The most common data standardization is same as the procedure covered in [subsection 31.3.2.1](#). Through standardization, input data have similar ranges and scales across different features.

33.2.3.2 Batch normalization

Batch normalization is mainly motivated by the need to remedy the negative impact from **internal covariance shift**[9]. Internal covariant shift refers to the change of input data distribution as the training proceeds. In the context of deep learning, input distribution to each layer is affected by parameters in all the input layers before. As a result, a small change to the network can produce distribution shift for all the subsequent layers. During the training process, weights are updating in a way such that multiplication of weights and input plus subsequent nonlinear activation can approximate the desired target function. As the weights in each layer are highly adapted to the input distribution, change of input distribution will strongly impact of the change of weights in the learning process.

It is also well established that training progresses better when the inputs have zero mean and unit variances and thus the resulting gradients are in the sweet spot. A popular solution to internal covariant shift is to normalize the input to each layer, in addition to the initial data standardization.

Practically, we add a batch normalization layer to normalize each input channel across a minibatch samples before the nonlinear activation. The layer performs normalization by subtracting the minibatch mean and scaling by the inverse minibatch standard deviation. The normalized input \hat{x} is then given by

$$\hat{x}_i = \frac{x_i - \hat{\mu}_B}{\sqrt{\hat{\sigma}_B^2 + \epsilon}}$$

where x_i is the input, $\hat{\mu}_B$ is the estimated mean from the minibatch, $\hat{\sigma}_B^2$ is the estimated variance.

Instead of using \hat{x} directly, we usually shift the input \hat{x} by a learnable offset β and scale it by a learnable scaling factor γ . This will give the network some extra flexibility to improve its approximation performance. Mathematically, we have

$$y_i = \gamma \hat{x}_i + \beta.$$

Note that when we use a trained network in prediction test, the layer should use the mean and variance calculated based on the whole training data set, instead of the minibatch mean and variance.

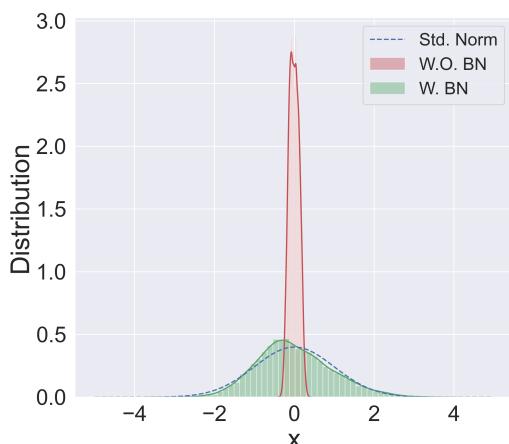
In practice, it has been found that batch normalization has the following advantages:

- Networks can be trained faster and better as weights are updated in a manner with stabilized input distribution.
- Increased tolerance for higher learning rate. Without batch normalization, the internal shift issue requires small learning rate to allow weights to update and co-adapt at a very slow pace.

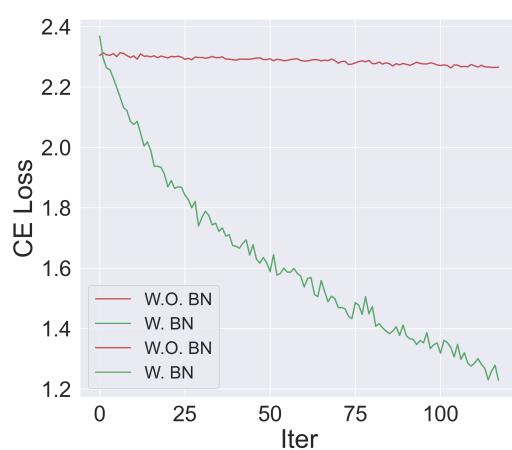
- Increased tolerance for weight initialization. If weights are not properly initialized, the output of a layer can cause nonlinearity to saturate and produce vanishingly small gradients that hinder learning. Batch normalization can help network to have more flexibility in weight initialization.
- Allow more activation functions. Without batch normalization, some useful activations like ReLU can easily enter the zero gradient region.

We consider a simple experiment where we train neural networks (two hidden layers) to MNIST classification. One neural network has batch normalization before ReLU activation and one does not. After several iteration of training, we compare the output of the second layer [Figure 33.2.1\(a\)](#), a neural layer with batch normalization produces output with nearly standard normal distribution. On the other hand, a neural layer without batch normalization can produce data distribution fairly different from standard normal and could be highly dependent on the weight initialization and other training strategies. When using a relatively large learning rate, network with batch normalization can significantly out-perform neural networks without batch normalization.

It is also clear that the use of batch normalization can lead to fast decrease of training loss and speed up the learning process.



(a) Internal data distribution.



(b) Training loss for models with and without batch normalization.

Figure 33.2.1: Impact batch normalization on internal data distribution and learning process. [Code](#)

33.2.3.3 Layer normalization

To have accurate estimation of batch mean and variance, Batch normalization requires that batch size not to be too small. Still, batch normalization cannot be applied in a recurrent neural network because the inputs are dynamically changing due to recurrent update.

Layer Normalization [10] is proposed to address the limitation of batch normalization. Unlike batch normalization, which performs normalization across samples, layer normalization performs normalization across all neurons in an intermediate layer.

Mathematically, let the input to the l th layer neuron be $z^{(l)} = (z_1^{(l)}, z_2^{(l)}, \dots, z_{n_l}^{(l)})$, where n^l is the number of neuron on layer l . We can compute layerwise mean and variance by

$$\mu^{(l)} = \frac{1}{n^l} \sum_{i=1}^{n^l} z_i^{(l)}$$

$$\sigma^{(l)} = \sqrt{\frac{1}{n^l} \sum_{i=1}^{n^l} (z_i^{(l)} - \mu^{(l)})^2}$$

The layerwise normalized input can be directly defined as

$$\hat{z}^{(l)} = \frac{z^{(l)} - \mu^{(l)}}{\sigma^{(l)}}.$$

Note that all the hidden units in a layer share the same normalization mean and variance parameter and different training examples have different normalization parameters. This contrasts batch normalization, where each dimension in a batch has its own normalization parameters. The benefit of layer normalization is that it can be applied to arbitrary mini-batch, including batch size of 1.

In practice, to allow additional flexibility, we often introduce learnable parameters and the layer normalization is often defined as

$$\hat{z}^{(l)} = \frac{z^{(l)} - \mu^{(l)}}{\sigma^{(l)} + \epsilon} \odot \gamma + \beta$$

$$\triangleq \text{LayerNorm}_{\gamma, \beta}(z^{(l)})$$

where β, γ are learnable parameter with dimensionality n^l .

Layer normalization has important application in stabilize recurrent neural network. The hidden state iteration with normalization is given by

$$z_t = Uh_{t-1} + Wx_t$$

$$h_t = f(\text{LayerNorm}(z_t))$$

where x_t is input at step t , h_{t-1} is previous hidden state, U, W are network weight, and f is an activation function.

In a standard recurrent neural network [subsection 35.1.3], gradients tend to grow or shrink at each time step, leading to exploding or vanishing gradients. With layer normalization, the normalization terms make inputs scale-invariant and can help stabilize the forward and backward process.

33.2.4 Regularization

Owing to its over-parameterization nature, deep learning has an inherent tendency for over-fitting, that is, good performance in training stage but poor generalization performance in the testing stage. Regularization, therefore, is one essential procedure to prevent overfitting and improve the generalization performance. Regularization techniques span from classical approaches like L_p penalization and early stopping that are widely used in traditional machine learning to novel techniques such as weight decay and dropout that are specially designed for deep learning.

33.2.4.1 L_p regularization

L_p penalization aims to penalize large magnitude of neural network parameters in order to reduce model complexity and overfitting. The most widely used penalization are L_1 , L_2 , and their mixture known as Elastic Net. Parameter penalization is achieved by optimization over the following modified loss function given by

$$\theta^* = \arg \min_{\theta} \frac{1}{N} \sum_{n=1}^N \mathcal{L} \left(y^{(n)}, f \left(x^{(n)}; \theta \right) \right) + \lambda_1 \|\theta\|_1 + \lambda_2 \|\theta\|_2$$

where λ_1 and λ_2 control the strength of penalization.

However, in general, L_p regularization is not as often used as in traditional machine learning algorithms; other equivalent approaches like weight decay and dropout are used instead.

33.2.4.2 Weight decay

Weight decay performs similar regularization like L_2 norm to penalize the magnitude. The core idea is: after each update, the weights are multiplied by a factor slightly less than 1 to prevents the weights from growing too large. Mathematically, the update formula is given by

$$\theta^{(m)} = (1 - w)\theta^{(m-1)} - \alpha g,$$

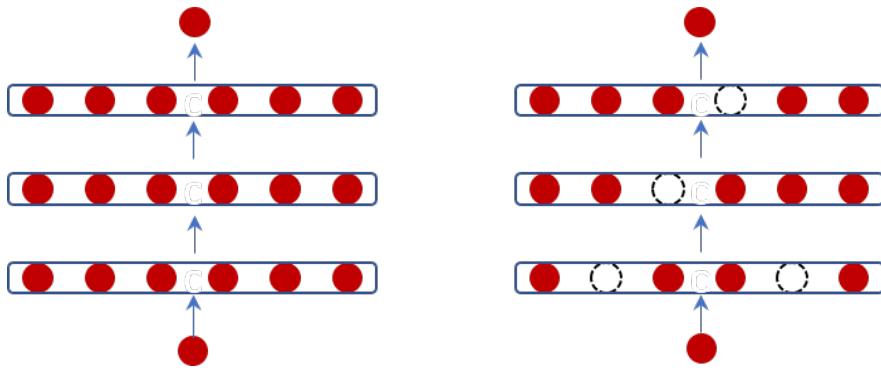


Figure 33.2.2: Dropout technique for a simple feed-forward neural networks. The original network (left) and the neural network after some neurons being dropped out (right).

where the superscript is the iteration number, w is the decay coefficient (small value say 0.005), α is the step size, and g is the gradient. The update formula can be seen as gradient descent on the original loss plus a quadratic regularization term. Note that although in the standard stochastic descent, weight decay is equivalent to L_2 regularization. However, for more advanced optimization methods like Adam[11], weight decay and L_2 regularization are not equivalent.

33.2.4.3 Early stopping

As the training proceeds, the training error can continuously drop whereas the test error will first drop, and then might at some point, starts to increase. The phenomenon indicates the onset of overfitting and can be prevented by early stopping the training.

In practice, we uses a validation data set to detect the start of overfitting and then early stop the training process when the validation test starts to increase.

33.2.4.4 Dropout

Dropout [12] technique works as follows: during the training process, we randomly remove some portion of neurons (including their connected edges). This can be achieved be setting the neuron's output to zero. In practice, we draw a Bernoulli random sample with probability p for each neuron to determine if this neuron should be dropped [Figure 33.2.2]. In the testing stage, all neurons are engaged in calculation and we multiplies a factor p for each layer output.

The regularization effect from dropout can be interpreted under the ensemble learning framework (e.g., bagging in section 29.3): in the training stage, we sample a

simpler, smaller sub-network to train in each iteration; in the testing stage, we average the output from different sub-networks. These sub-networks are equivalent to the base learners in the ensemble learning framework.

Dropout in recurrent neural networks are significantly complicated, which has been addressed in [13].

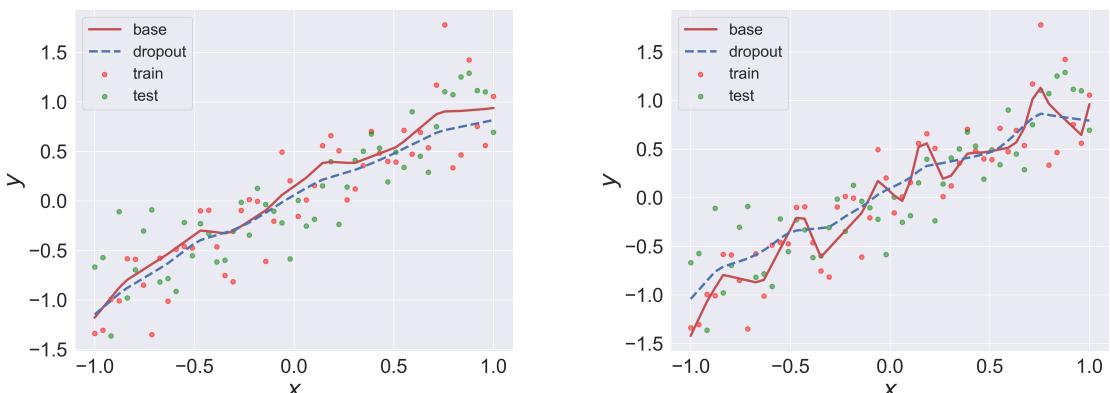
Remark 33.2.2 (bagging vs dropout). Bagging in traditional machine learning has several differences compared to dropout.

- In bagging, each model is trained on a bootstrapped sample set and each model has its own parameters.
- In dropout, subnetworks share parameters.

We consider the a simple regression problem [Figure 33.2.3] to demonstrate the impact of dropout on model generation. The training and testing data are generated by

$$y = x + 0.3\epsilon, \epsilon \sim N(0, 1).$$

We use neural network with two hidden layers (each has 100 hidden units). Because of the noise in the data and the complexity of the neural network model, we expect the model, after extensive training, to learn noisy patterns in the training set and generalize poorly to the testing data. We compare predictions at early iterations vs late stage iteration. Clearly, the presence of dropout can alleviate the tendency of fitting to noise when the model is being extensively trained.



(a) Model prediction comparison after training 200 iterations. (b) Model prediction comparison after training 800 iterations.

Figure 33.2.3: Impact of dropout on model generalization. [Code](#)

33.2.4.5 Data augmentation

Collecting more data samples is one of the most effective ways to overcome overfitting and improve generalization performance. Data augmentation mainly applies to image data, where various transformations are applied to generate new image samples from original images. Common transformations include: rotation, flipping, zooming in/out, shifting, and adding noise.

33.2.4.6 Label smoothing

Label smoothing[14] is a regularization technique for classification problems. Consider a ground truth one-hot encoded outcome given by

$$y = (0, \dots, 0, 1, 0, \dots, 0).$$

Label smoothing with a uniform noise of magnitude ϵ will give a smoothed label as

$$y' = \left(\frac{\epsilon}{K-1}, \dots, \frac{\epsilon}{K-1}, 1 - \epsilon, \frac{\epsilon}{K-1}, \dots, \frac{\epsilon}{K-1} \right).$$

During training, instead of training the network with respect to one-hot predictions, we can train the network with respect to smooth labels to reduce overfitting. The rationale for label smoothing is: one-hot encoding may result in over-fitting. If the model learns to assign full probability to the ground truth label for each training example, it is not guaranteed to generalize well. Intuitively, a model tends to overfit when the model becomes too confident about its predictions. Label smoothing offers a mechanism for encouraging the model to be less confident and thus more robust to unseen samples. In addition, if some outcome are incorrectly labeled, label smoothing technique can make the neural network robust to such mistakes.

33.3 Notes on Bibliography

General books on deep learning include [15].

For a comprehensive review on deep learning methods for text classification, see [16].

Natural language processing, see [17][18].

BIBLIOGRAPHY

1. Goyal, P. *et al.* Accurate, large minibatch sgd: Training imagenet in 1 hour. *arXiv preprint arXiv:1706.02677* (2017).
2. Duchi, J., Hazan, E. & Singer, Y. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research* **12**, 2121–2159 (2011).
3. Hinton, G. *Neural Networks for Machine Learning* (University of Toronto, 2012).
4. Zeiler, M. D. ADADELTA: an adaptive learning rate method. *arXiv preprint arXiv:1212.5701* (2012).
5. Sutskever, I., Martens, J., Dahl, G. & Hinton, G. *On the importance of initialization and momentum in deep learning* in *International conference on machine learning* (2013), 1139–1147.
6. Kingma, D. P. & Ba, J. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).
7. Glorot, X. & Bengio, Y. *Understanding the difficulty of training deep feedforward neural networks* in *Proceedings of the thirteenth international conference on artificial intelligence and statistics* (2010), 249–256.
8. He, K., Zhang, X., Ren, S. & Sun, J. *Delving deep into rectifiers: Surpassing human-level performance on imagenet classification* in *Proceedings of the IEEE international conference on computer vision* (2015), 1026–1034.
9. Ioffe, S. & Szegedy, C. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167* (2015).
10. Ba, J. L., Kiros, J. R. & Hinton, G. E. Layer normalization. *arXiv preprint arXiv:1607.06450* (2016).
11. Loshchilov, I. & Hutter, F. Fixing weight decay regularization in adam. *arXiv preprint arXiv:1711.05101* (2017).
12. Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I. & Salakhutdinov, R. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research* **15**, 1929–1958 (2014).
13. Gal, Y. & Ghahramani, Z. *A theoretically grounded application of dropout in recurrent neural networks* in *Advances in neural information processing systems* (2016), 1019–1027.

14. Szegedy, C., Vanhoucke, V., Ioffe, S., Shlens, J. & Wojna, Z. *Rethinking the inception architecture for computer vision* in *Proceedings of the IEEE conference on computer vision and pattern recognition* (2016), 2818–2826.
15. Goodfellow, I., Bengio, Y., Courville, A. & Bengio, Y. *Deep learning* (MIT press Cambridge, 2016).
16. Minaee, S. *et al.* Deep Learning Based Text Classification: A Comprehensive Review. *arXiv preprint arXiv:2004.03705* (2020).
17. Manning, C. D. & Schütze, H. *Foundations of statistical natural language processing* (MIT press, 1999).
18. Goldberg, Y. Neural network methods for natural language processing. *Synthesis Lectures on Human Language Technologies* **10**, 1–309 (2017).

34

CONVOLUTIONAL NEURAL NETWORKS

34 CONVOLUTIONAL NEURAL NETWORKS 1449

34.1 Foundations 1450

34.1.1 Convolutional connection 1450

34.1.2 Pooling and translational invariance 1453

34.1.3 Special convolution units 1455

34.1.3.1 The 1×1 convolution 1455

34.1.3.2 Dilated convolution 1456

34.2 CNN classical architectures 1458

34.2.1 LeNet 1458

34.2.2 AlexNet 1458

34.2.3 VGG 1460

34.2.4 ResNet 1461

34.3 Image recognition 1466

34.3.1 Image classification 1466

34.4 Visualizing CNN 1467

34.4.1 Visualizing filters 1467

34.4.2 Visualizing classification activation map 1468

34.5 Autoencoders and denoising 1470

34.5.1 Autoencoders 1470

34.5.2 Denoising autoencoder 1471

34.6 Neural style transfer 1473

34.7 Visual based deep reinforcement learning 1476

34.1 Foundations

34.1.1 Convolutional connection

Convolutional Neural Networks (CNN) have achieved remarkable success in image related applications such image segmentation and recognition, and object detection. Since the advent of CNN, there have been many variants, with some of them, including LeNet, Alexnet, GoogLeNet, VGG, etc., achieving milestone performance in real-world applications or ImageNet contests.

Compared to dense neural networks, the most distinct characteristics are: **local receptive field, parameter sharing, and pooling**.

In feed-forward neural networks [Figure 34.1.1], every hidden feature is constructed from *all* input features in the previous layer. In convolutional neural networks, every hidden feature is constructed from *part* of input features that are spatially close. This is realized by locally connected layers, where each neuron in the hidden layer seems to have a local receptive field. The local connectivity exploits the fact that pixel far away from each other usually has negligible correlation and significantly reduce network parameters and overfitting.

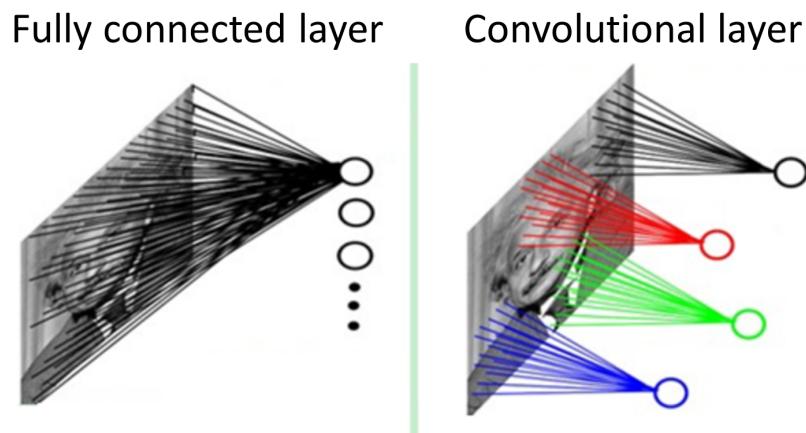


Figure 34.1.1: Comparison of receptive fields in fully-connected layer and local-connected layer in CNN. Credit

Another feature that contributes to the reduction of network parameters is sharing the same weights across different local connectivity. We call these weight kernels. How kernel processes inputs can be illustrated in Figure 34.1.2. Specifically, for an input X

with 4×4 size and 3 channels, when it convolves with a 2×2 kernel and step size of 2, it produces 4 activations, given by

$$\begin{aligned} z_{11} &= \sum_{l=1}^3 x_{11}^{(l)} w_{11}^{(l)} + x_{12}^{(l)} w_{12}^{(l)} + x_{21}^{(l)} w_{21}^{(l)} + x_{22}^{(l)} w_{22}^{(l)} \\ z_{12} &= \sum_{l=1}^3 x_{13}^{(l)} w_{11}^{(l)} + x_{14}^{(l)} w_{12}^{(l)} + x_{23}^{(l)} w_{21}^{(l)} + x_{24}^{(l)} w_{22}^{(l)} \\ \dots &= \dots \\ z_{22} &= \sum_{l=1}^3 x_{33}^{(l)} w_{11}^{(l)} + x_{34}^{(l)} w_{12}^{(l)} + x_{43}^{(l)} w_{21}^{(l)} + x_{44}^{(l)} w_{22}^{(l)} \end{aligned}$$

where $x_{ij}^{(l)}, w_{ij}^{(l)}$ denotes input value and kernel value at spatial location (i, j) from channel l .

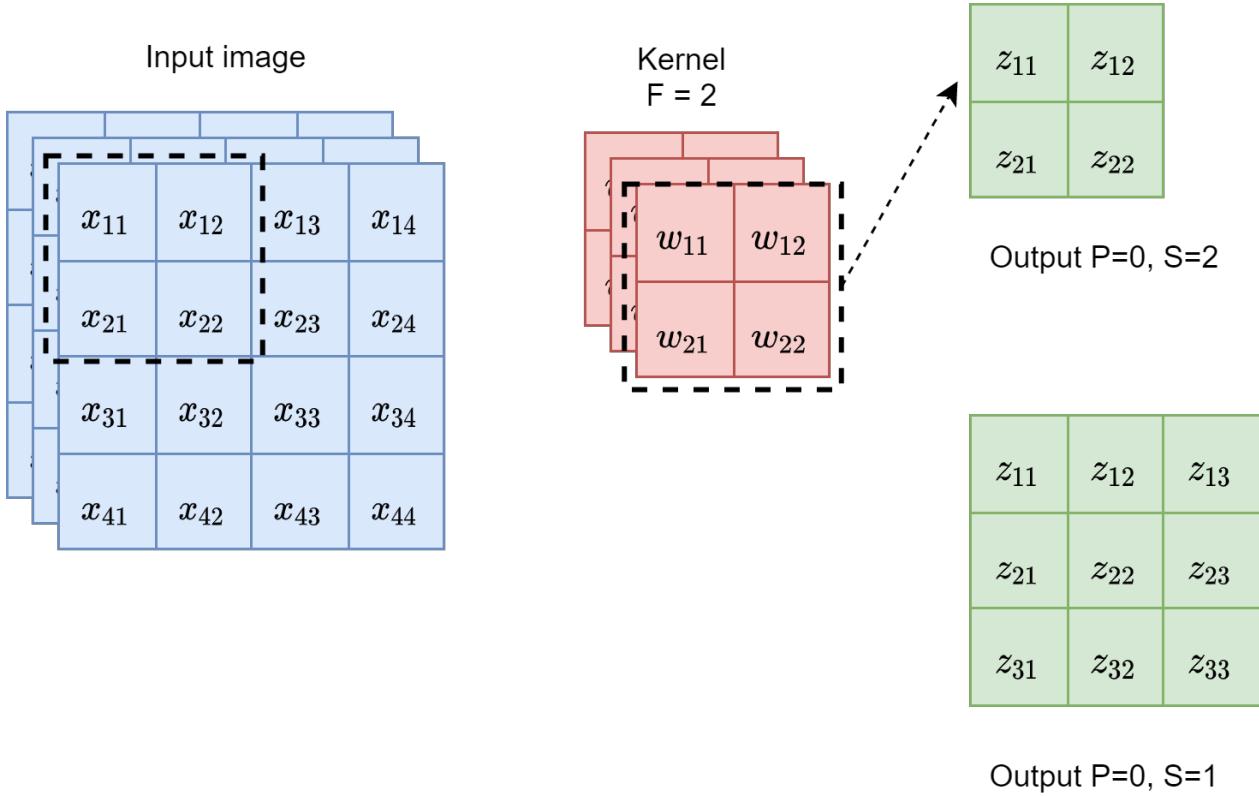


Figure 34.1.2: Demo for one kernel 'convoluting' with an input image. The values (across all channels) in the dashed box will convolve with the kernel to produce a value in the output.

Definition 34.1.1 (convolutional calculation). A convolutional layer (S, F, P, C_I, C_O) will have $C_I \times C_O$ matrix kernels represented by matrices $K^{m,n} \in \mathbb{R}^{F \times F}, 1 \leq m \leq C_I, 1 \leq n \leq C_O$.

Let X^1, \dots, X^{C_I} be the C_I slices of the input data and let Z^1, \dots, Z^{C_O} be the C_O slices of the **linear aggregation** results. The linear aggregation process to produce one slide in the output can be represented by

$$Z^n = \sum_{m=1}^{C_I} K^{m,n} \circledast X^m.$$

where $K^{m,n} \circledast X^m$ is a matrix whose entries are given by

$$[K^{m,n} \circledast X^m]_{ij} = \sum_{u=1}^F \sum_{v=1}^F K_{u,v}^{m,n} X_{i+u-1, j+v+1}^m$$

Note that features in different input channels are 'merged' to generate features on the output channels.

More generally, we allow a 2D convolutional layer to have more flexibility, including

- Stride $S = (S_1, S_2)$, which control the slide window step size in each direction.
- Kernel size $F = (F_1, F_2)$, which allows the convolutional kernel to have different in each direction.
- Padding $P = (P_1, P_2)$, which specifies how many zeros are padded on the border of the input.

Consider a 2D convolutional layer with stride $S = (S_1, S_2)$, kernel size $F = (F_1, F_2)$, padding $P = (P_1, P_2)$, input channel number C_I and output channel number C_O . Given 2D input image with size (C_I, H_I, W_I) the output will have sizes (H_O, W_O) given by

$$H_O = \frac{H_I + 2P_1 - F_1}{S_1} + 1$$

$$W_O = \frac{W_I + 2P_2 - F_2}{S_2} + 1.$$

One interesting question for us see the connection between fully-connected layer and convolutional layer is can we represent the former by the latter? It turns out that if we want to create a fully connected layer for input $H \times W \times C_I$ with C_2 hidden units, we just a convolutional kernel of size $H \times W \times C_2$. This kernel will have $H \times W \times C_I \times C_O$ weights!

34.1.2 Pooling and translational invariance

Another important characteristics of CNN is pooling, which is realized via pooling layers. A pooling layer is usually used between successive convolutional layers, with the goal to select and reduce feature numbers and thus network parameters via down-sampling. Such reduction is necessary to speed up training and more importantly, to mitigate overfitting.

A pooling can down-sample the input via maximization or averaging. The key parameters of a pooling layer are the pooling kernel F and the stride S [Figure 34.1.3]. Commonly we choose $F = 2, S = 2$; If we want to keep more features, one can choose $F = 2, S = 1$ for overlapping pooling.

Given an input with size $W_I \times H_I \times D_I$, the output from a pooling layer will have size $W_O \times H_O \times D_O$, where

$$W_O = \frac{W_I - F}{S} + 1, H_O = \frac{H_I - F}{S} + 1, D_O = D_I.$$

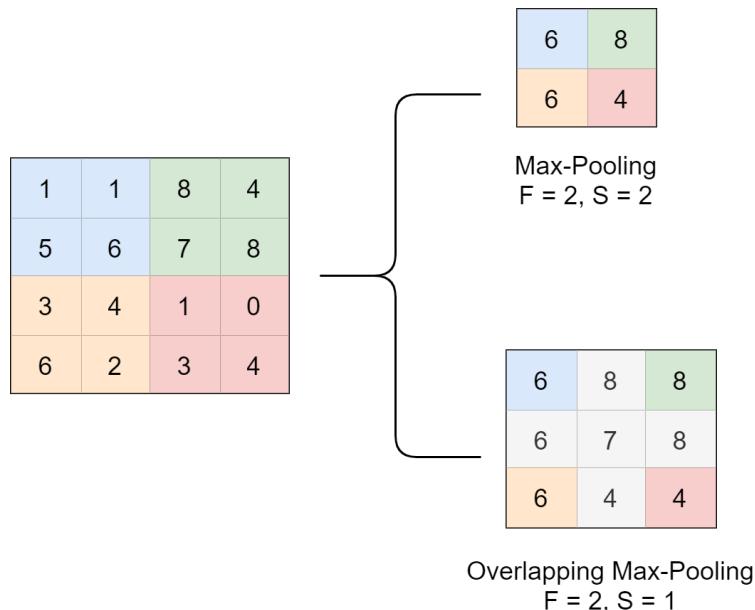


Figure 34.1.3: Max-Pooling layer illustration. Depending on the kernel size and stride, we will have different output shape.

The special architecture of parameter sharing and max pooling can allow CNN to acquire some level of **translation invariance** capability for taking images as input. Suppose an image shift down a bit, the activation in the filters of the first layer will also shift down a bit. Note that max-pooling layer will only take the maximum value among

of a patch pixels. Eventually, the output from max-pooling will remain the same despite the shift in the input.

The level of translation invariance depends on the pooling size; in general, CNN is not invariant for large shifts.

Pooling layer reduces data spatial dimension and belongs to non-differential operation. How can we properly perform back-propagation and distribute gradients across all layers? For mean pooling, the gradient will be averaged and distributed to locations of the preceding layer; For max pooling, the gradient will be only distributed to locations associated with local maximum values in the preceding layer.

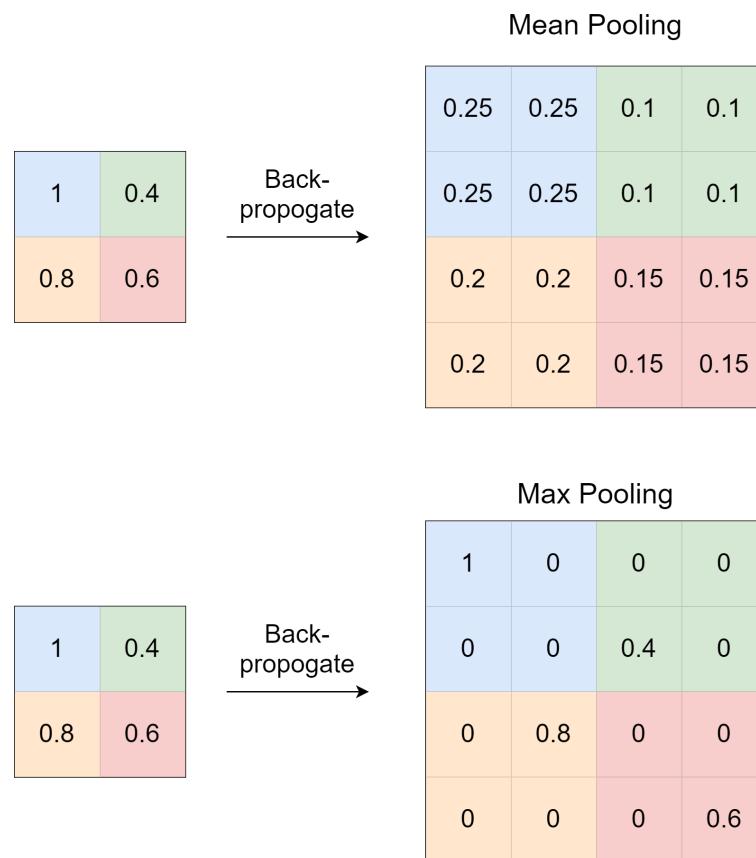


Figure 34.1.4: Gradient distribution during back-propagation for mean pooling and max pooling (both have kernel size 2).

Finally, a typical CNN architecture for either classification or regression tasks will be something like Figure 34.1.5. The first several layers consist of convolutional layers and pooling layers progressively extract high-level representation features from the input. Final outputs from convolutional layers will be further pooled or flattened and fed into

fully connected layers. Depending on the task being classification or regression, the final output layer is either a linear layer with Softmax or just a linear layer.

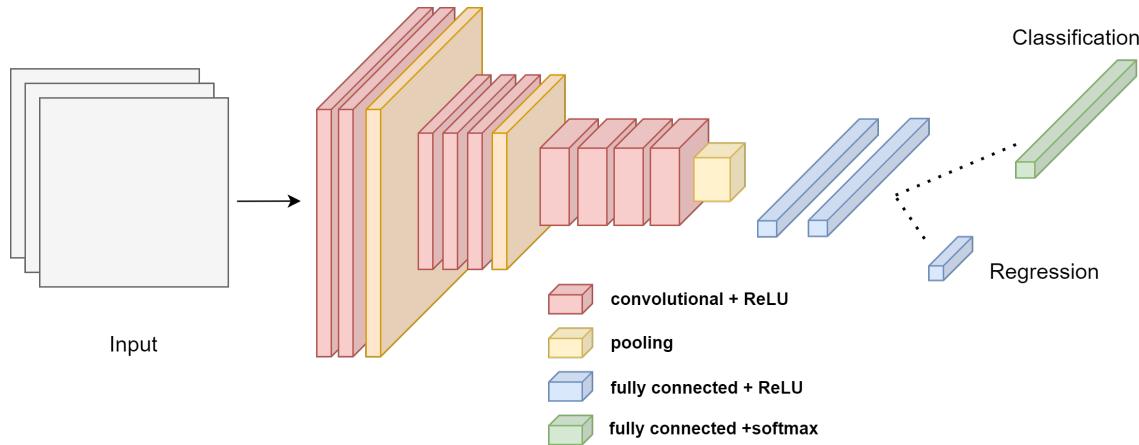


Figure 34.1.5: A typical CNN architecture for classification or regression tasks.

34.1.3 Special convolution units

34.1.3.1 The 1×1 convolution

An important type of convolution filter widely used across varying CNN architectures is 1×1 convolutional filters. A 1×1 convolutional layer with input channel number being C_I and output channel number C_O) has weight parameters represented by a matrix $W \in \mathbb{R}^{C_I \times C_O}$. Let $X^1, \dots, X^{C_I} \in \mathbb{R}^{H \times W}$ be the C_I slices of the input data and let $Z^1, \dots, Z^{C_O} \in \mathbb{R}^{H \times W}$ be the C_O slices of the output. It follows that

$$[Z^n]_{ij} = \sum_{m=1}^{C_I} W^{m,n} X_{ij}^m, n = 1, \dots, C_O.$$

Suppose that we have an input with shape (H_0, W_0, C_0) shaped tensor where F_0 is the number of channels and H, W are the spatial dimensions. Apply the convolutional computation scheme, the output tensor will have shape (F_1, H, W) ; note that the feature map size (H, W) remain unchanged. So 1×1 conv filters can be used to change the dimensionality of features at each pixel. If $F_1 > F$ then we are increasing dimensionality (i.e., fewer channels mapped to more channels); if $F_1 < F$ we are decreasing dimensionality (i.e., more channels mapped to fewer channels) [Figure 34.1.6].

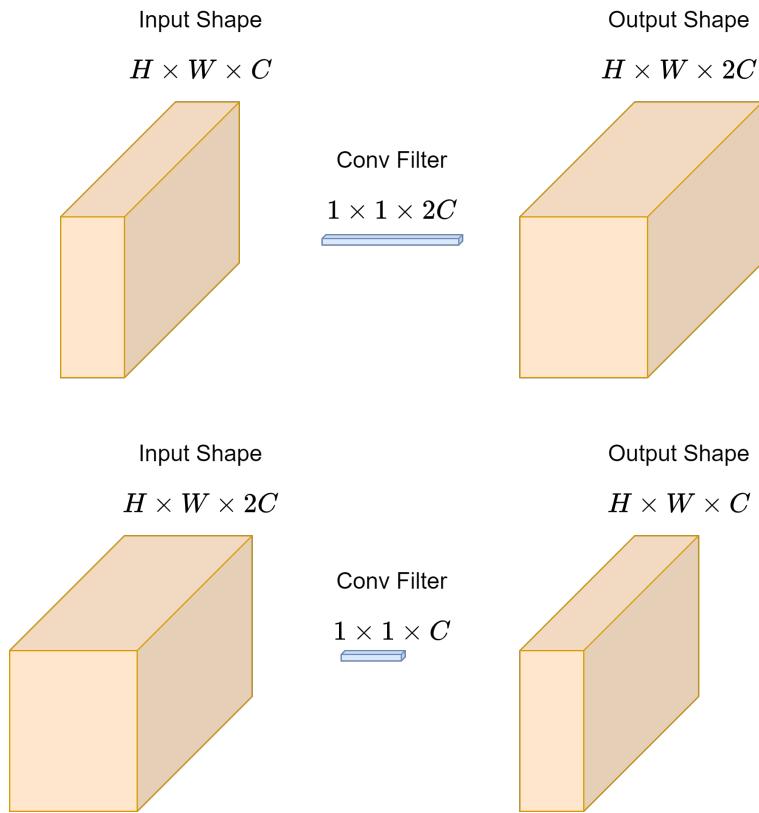


Figure 34.1.6: 1×1 convolution (stride 1 and zero padding) can be used to increase channel size (upper) or decrease channel size (lower) without changing the spatial size.

34.1.3.2 Dilated convolution

In a deep CNN, convolutional kernels act on small patches in the input and uses subsequent pooling layer progressively expand the receptive field and integrate context information. The usage of pooling layer can cause information loss and reduce accuracy. In order to have sufficiently large receptive field (e.g., in sequence modeling problem), we typically need a large number of convolutional layer with small kernels (e.g., 3×3 kernel) or need to increase the kernel size or need to increase the pooling size. But the first two will increase the number of parameters, and the third way will cause information loss.

Dilated convolution [1] is a strategy to expand the receptive field without increasing network depth and kernel size. The strategy is to introduce *holes* during the kernel computation[Figure 34.1.7].

If $D - 1$ holes are inserted between every two elements of the convolution kernel, the effective size of the convolution kernel is $K' = K + (K - 1) \times (D - 1)$. Among them,

K is the size of the original convolution kernel, and D is called the dilation rate (or just dilation). When $D = 1$, the convolution kernel is an ordinary convolution kernel.

For example, the receptive field size of a 3×3 regular convolution kernel is 3; The receptive field size of a 3×3 convolution kernel with dilation 2 is 5; The receptive field size of a 3×3 convolution kernel with dilation 4 is 9;

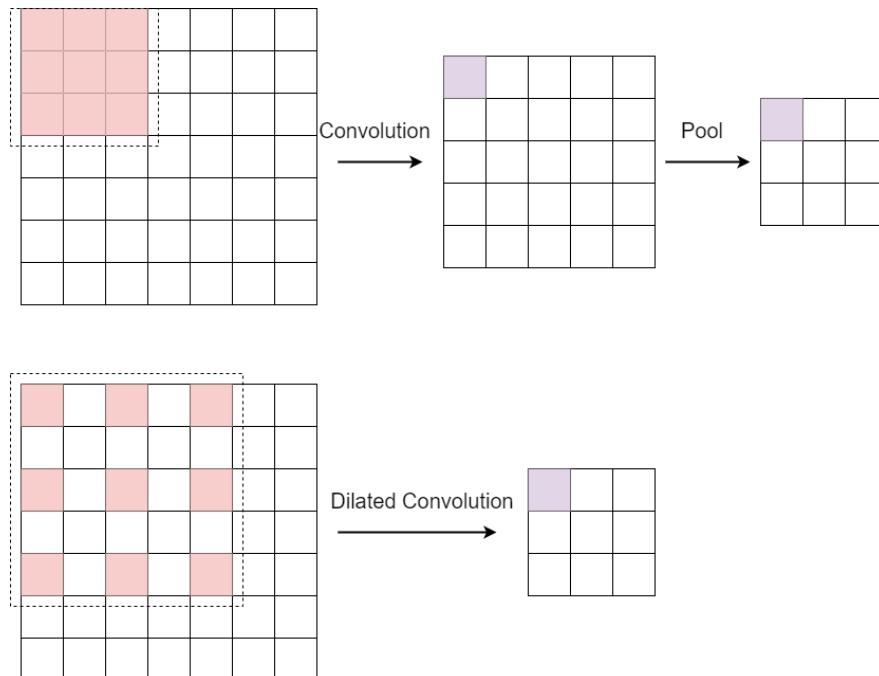


Figure 34.1.7: Illustration of regular 3×3 convolution and dilated convolution. (Upper) Regular convolution act on contiguous patches in the input and use pooling to increase receptive field for layers in the back. (Lower) Dilated convolution insert *holes* in the convolution computation and expand the receptive field without pooling.

34.2 CNN classical architectures

34.2.1 LeNet

LeNet is the earliest CNN that applied to image recognition in real-world tasks, mainly recognizing hand-written digits[2]. LeNet introduced the concept of convolution and pooling, laying the foundation of all modern CNNs.

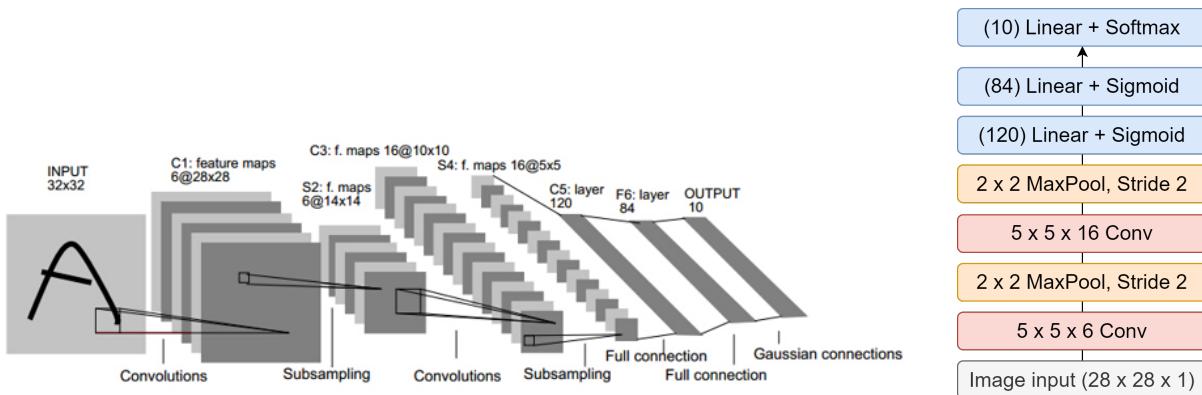


Figure 34.2.1: Scheme and architecture for LeNet[2].

34.2.2 AlexNet

The wide spread of interest in deep learning for computer vision was sparked by the great success of AlexNet [3], which beat the traditional methods in the ImageNet Large-Scale Visual Recognition Challenge (ILSVRC) by a large margin. ImageNet is a large-scale annotated dataset for image recognition tasks. It has over 15 million labeled high-resolution images belonging to roughly 22,000 categories. ILSVRC uses a subset of ImageNet with roughly 1000 images in each of 1000 categories. In all, there are roughly 1.2 million training images, 50,000 validation images, and 150,000 testing images.

Model	Top-1	Top-5
Sparse coding	47.1%	28.2%
SIFT + FVs	45.7%	25.7%
CNN	37.5%	17.0%

The architecture of AlexNet is shown in [Figure 34.2.2](#). The net contains eight layers with weights; the first five are convolutional and the last three are fully-connected layers.

The output of the last fully-connected layer is fed to a 1000-way Softmax to produce prediction probability over the 1000 class labels. Our network is trained by minimizing multi-class entropy loss [subsection 32.2.1].

Compared to LeNet, AlexNet has several key features: 1) AlexNet uses 5 convolutional layers, a much deeper architecture than LeNet (2 convolutional layers); 2) AlexNet uses ReLU nonlinearity unit, which is found to accelerate training by several factor compared to Sigmoid or Tanh nonlinearity. AlexNet also introduces local response normalization layer, which facilitate the learning with ReLU. Local response normalization basically normalize each activation by dividing the sum over activations at neighboring channels.

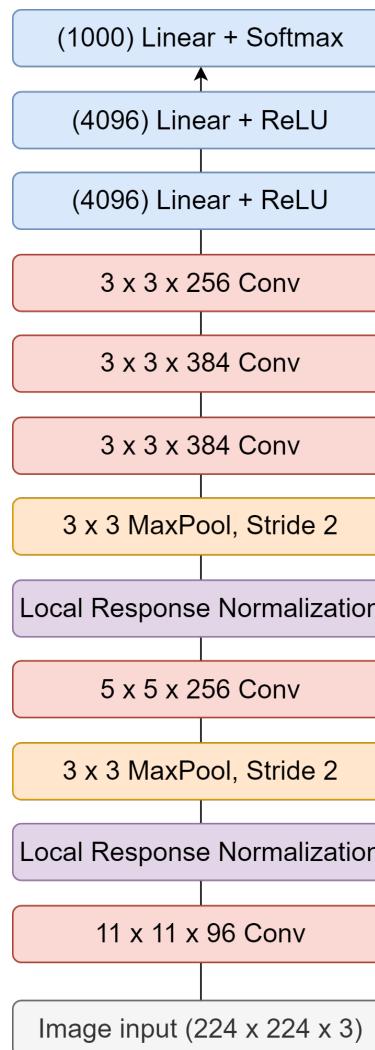


Figure 34.2.2: Architecture of AlexNet.

AlexNet also pioneered several different data augmentation strategies to prevent overfitting. The first strategy involves generating image translations and horizontal reflections. We do this by extracting random 224 by 224 patches (and their horizontal reflections) from the 256 by 256 images and training the network on these extracted patches. This creates an augmented training dataset that has a size of 2048 times of the original training dataset.

To maintain consistency between training and testing, at test time, the network makes a prediction by extracting five 224 by 224 patches (the four corner patches and the center patch) as well as their horizontal reflections (hence ten patches in all). Final prediction is the average of individual predictions on the ten patches.

The second scheme is generating additional training samples by perturbing each pixel location's RGB values (there are a total 224 × 224 pixels). Instead of adding a 3-dim independent random Gaussian noise at each pixel location, the authors added 3-dim correlated Gaussian noise whose covariance matrix is 0.01 multiplied by the covariance matrix of RGB values across all training examples. The author utilized an eigen-decomposition approach to generate such correlated Gaussian noises, which can also be realized via Cholesky decomposition.

Without those data augmentation scheme, the large network is found to suffer from substantial overfitting and poor performance on testing.

34.2.3 VGG

VGG is another milestone convolutional neural network architecture after the AlexNet. VGG was invented by Visual Geometry Group (VGG) [4] and it was the first runner-up of the 2014 ImageNet contest in the classification task after the GoogLeNet.

The major innovative components of VGG [Figure 34.2.3] over AlexNet [Figure 34.2.3] are the use of stacked 3×3 kernels and larger depth (up to 19 layers) to progressively extract useful features. This contrasts Alexnet, which utilize 5×5 and 7×7 filters in first and second convolution layers to enable larger receptive fields.

It is important to know that a stack of two 3×3 filters has an effective receptive field of 5×5 and a stack of three 3×3 filters has an effective field of 7×7 . Beside the equivalence in the receptive field, having multiple 3×3 filters also incorporates more nonlinearities in the transformation, thus increasing the expressive power of the network.

In addition, stacked 3×3 filters are actually more parameter efficient. For example, consider the input and the output of a three-layer 3×3 convolution stack has C channels. The stack has $3(3^2C^2) = 27C^2$ weights; at the same time, a single 7×7 filter would require $7^2C^2 = 49C^2$ parameters.



Figure 34.2.3: A typical VGG architecture: VGG-19 scheme.

34.2.4 ResNet

Residual network (ResNet) is another CNN that has profound impact on deep learning. Before the advent of ResNet, CNN variants usually have 20 to 30 layers. However, it was found[5] that simply stacking more layers will not further boost but rather degrade performance. It is suggested that, although deep networks certainly have better expressive power, they become significantly difficult to train due to vanishing gradients [Figure 34.2.4].

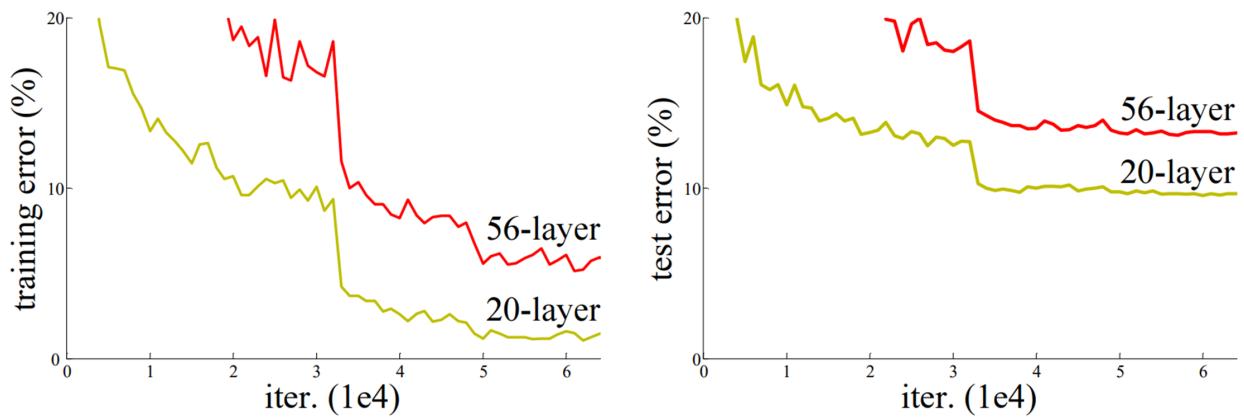


Figure 34.2.4: Training error (left) and testing error (right) on CIFAR-10 with 20-layer and 56-layer vanilla CNN networks. The deeper network has both higher training error and testing error.[5]

The vanishing gradient issue was addressed by introducing short-cuts crossing different layers [Figure 34.2.5]. Particularly, these short-cuts are identity maps that will not reduce gradient magnitude during back-propagation. From this perspective, ResNet can be viewed as a special case of Highway Network[6].

Another aspect to understand the name 'residual' is by the fact that the output of a residual block is $H(x) = F(x) + x$, or $F(x) = H(x) - x$. If $H(x)$ is the target, then $F(x)$ is trained to approximate the residual $H(x) - x$.

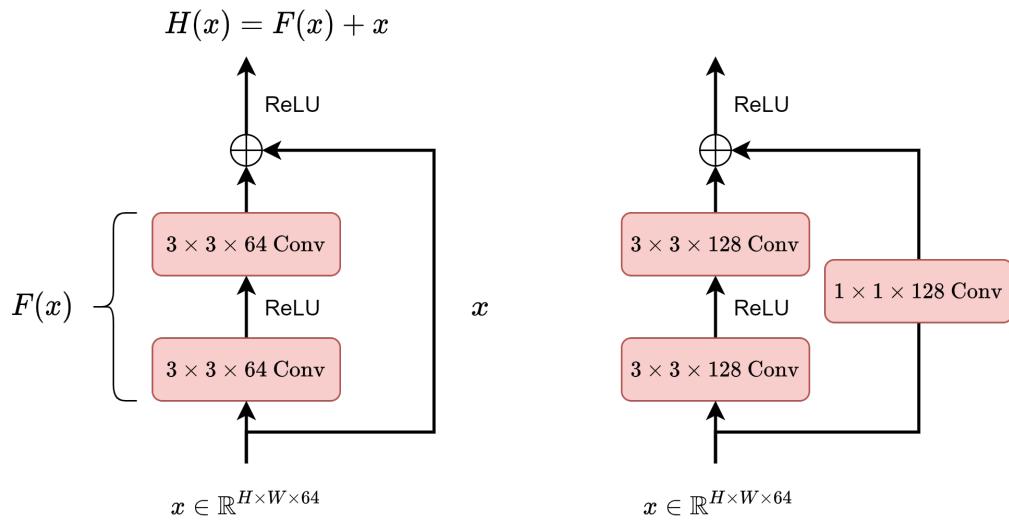


Figure 34.2.5: Scheme for a residual block without (left) and with (right) 1×1 convolution. 1×1 convolution is used to match the channel number before and after the convolutional layers.

A typical ResNet that stacks multiple Residual block is showed in [Figure 34.2.6](#). ResNets can have more than 1000 layers but still have superior performance.

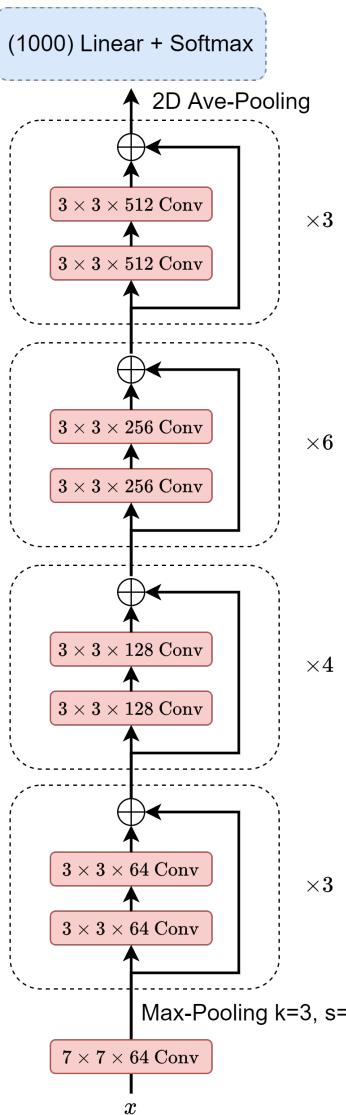


Figure 34.2.6: Scheme of a 34-layer ResNet

Another economic variant of ResNet basic block is the bottleneck block [Figure 34.2.7]. A bottleneck block has three layers of 1×1 , 3×3 and 1×1 convolutions, where the 1×1 convolutions are responsible for reducing and then increasing (restoring) dimensions. The 3×3 layer has a smaller channel number that facilitate the extraction of high-level important features. The use of a bottleneck reduces the number of parameters and matrix multiplications and increases the overall network depth.

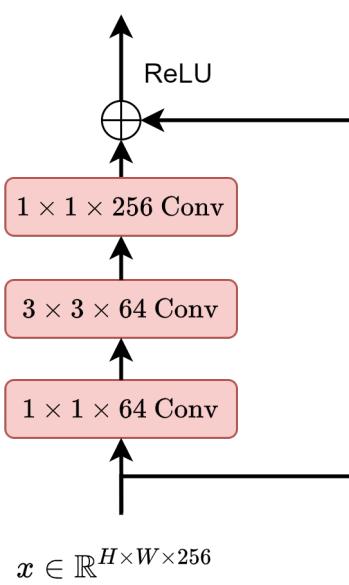


Figure 34.2.7: Scheme for a residual bottleneck block.

34.3 Image recognition

34.3.1 Image classification

One of most important application of CNN is image classification, from the hand-written digits to more challenging object recognition. In fact, deep learning regain its popularity because of the superior performance of CNN on the 2012 ImageNet context[3].

As CNN can extract common image feature from data, a pre-trained network can be used to as a generic tool to extract features. In image classification tasks, instead of training a deep CNN from scratch, we usually take a pre-trained network, replace the last fully connected layer and the final classification layer of the network, and use the new combined network as a starting point to train for a new task. Fine-tuning this combined network is usually much faster and easier than training a network with randomly initialized weights from scratch. This method is also known as **transfer learning**.

In the below example [Figure 34.3.1], we reuse the a pretrained ResNet-18 with the last layers replaced by a new fully-connected layer. This new network is able to classify CIFAR10 image dataset with accuracy up to 90%.

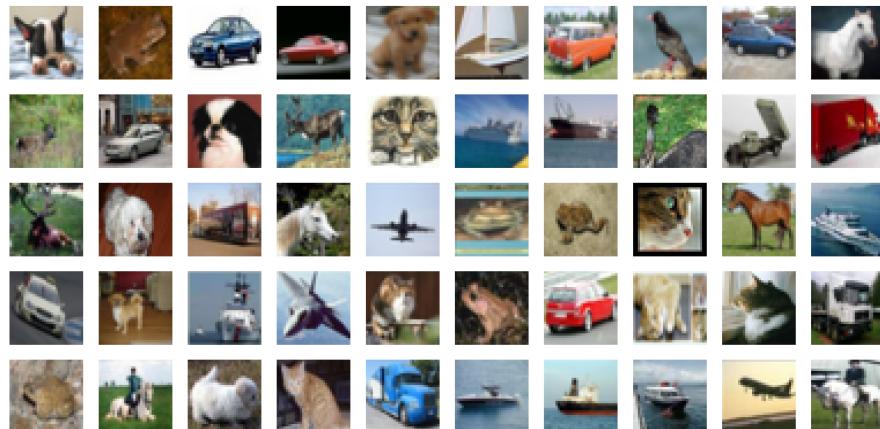


Figure 34.3.1: Example images from CIFAR10 image dataset.

34.4 Visualizing CNN

34.4.1 Visualizing filters

Traditional computation vision use human-crafted feature, which offer interpretability on how computers understand image. A CNN, by contrast, progressively builds features from data, and, to some extent, works like a ‘black box’. Understand what is learned in CNN and how CNN based on classifier make decision has been in active research since the first paper on this topic published in 2014 [7].

When we feed an image to a CNN, filters of different layers will be activated. To understand the working mechanism of a CNN, it is important to understand what information on the image each filter is primarily respond to.

Let $x \in \mathbb{R}^{W \times H}$ be the input image, the response map $R_i^l \in \mathbb{R}^{W \times H}$ for filter i at layer l can be obtained by [8]

$$R_i^l = \arg \max_x \|F_i^l\|_F^2$$

where F_i^l is the activation matrix of filter i at layer l .

The optimization can be carried out by simple gradient ascent starting from a random noise image. In the following, we visualize filters of different layers in a pretrained VGG-16 on a test image.

These filter visualizations illuminate how convolutional layers perceive the input. Each layer learns a collection of filters such that the inputs can be expressed as a combination of the filter responses. This is analogous to how the Fourier transform decomposes signals onto a series of cosine functions. The filters from the bottom layer encode simple information like directional edges. Filters in top layers begin to resemble textures found in natural images: feathers, eyes, leaves, and so on.

Other algorithms and implementation on visualizing CNN can be found in [9].

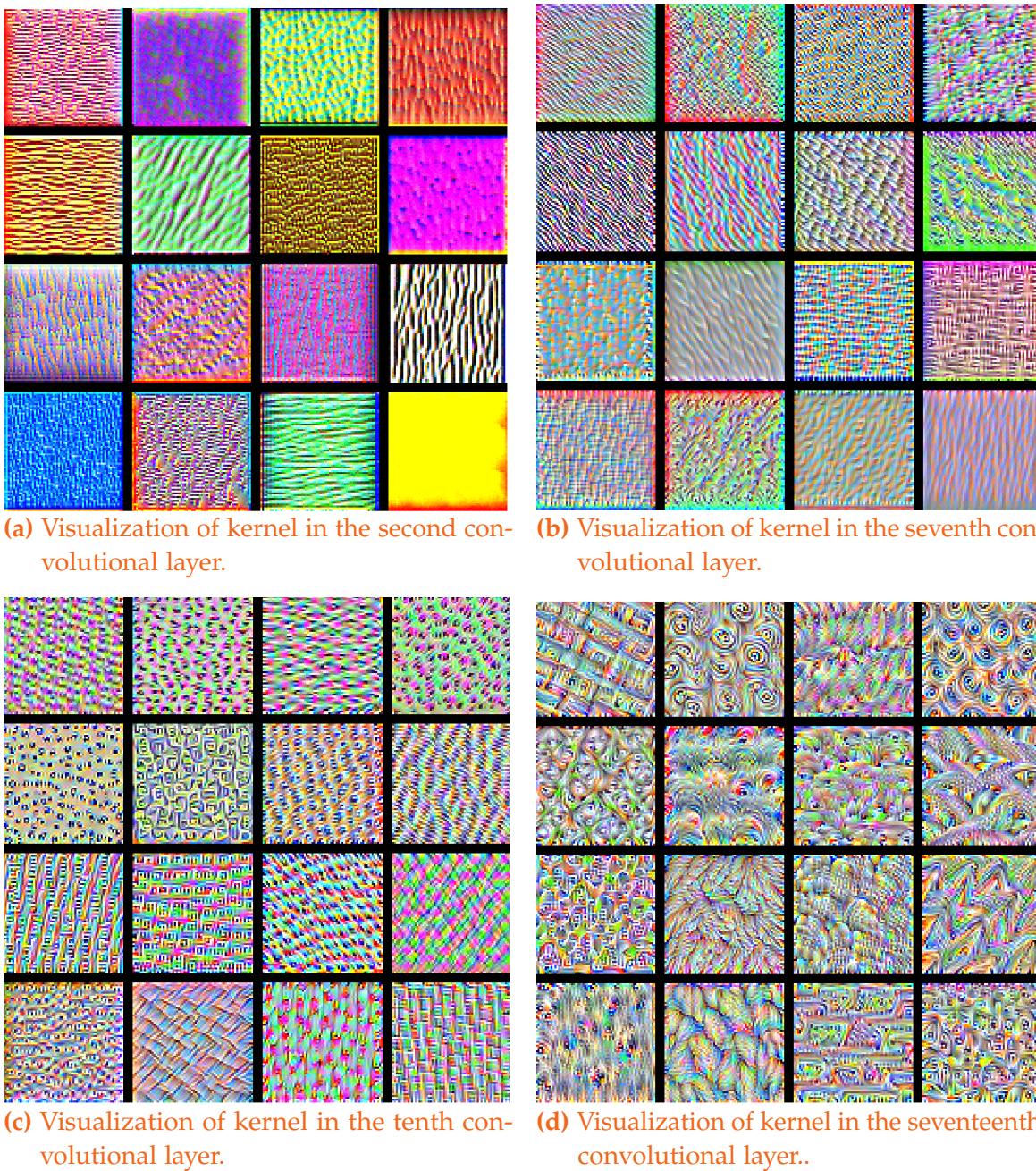


Figure 34.4.1: Visualization of convolution layer.

34.4.2 Visualizing classification activation map

Consider applying the CNN classifier to classify an image containing a dog and a cat. The classifier will output nonzero classification probabilities corresponding to a dog

and a cat. It is curious to know that how the classifier comes up with the classification probability and how the classifier makes decision based on different portion of the image.

One method that offer excellent visual explanation is Grad-CAM (Gradient-weighted Class Activation Mapping)[10].

Given a neural network, the visualization contains the following steps

- Let y_c denote the output classification probability of class c . First we compute the gradient with respect to the feature map activations A^k of the convolutional layer k , that is

$$\frac{\partial y_c}{\partial A^k}.$$

Note that the gradient represents the magnitude of the activation A^k contributes to the classification probability.

- We can compute the averaged gradient

$$\alpha_c^k = \frac{1}{WH} \sum_{i=1}^H \sum_{j=1}^W \frac{\partial y_c}{\partial A^k},$$

where W and H are the width and height of the feature map of layer k .

- The class-discriminative localization map $L^c \in \mathbb{R}^{H \times W}$ is given by

$$L^c = \max(0, \alpha_c^k A^k).$$

Note that in the final step take only takes the positive part of the summation via mapping $\max(0, x)$ since we are only interest in feature maps that have a positive influence on the class classification probability. Besides, because different layers might have different sizes of feature maps, we should properly resize them before the summation.

Back to the dog and cat classification problem. The superposition of the localization map on top of the original image is showed in Figure 34.4.2.

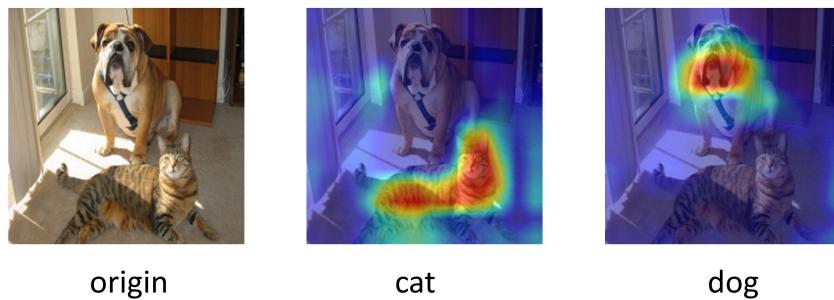


Figure 34.4.2: Grad-CAM method applied to understand image classification. Middle and right are class-discriminative localization map superimposed onto the original image.

34.5 Autoencoders and denoising

34.5.1 Autoencoders

Principal component analysis (PCA) is one of the most widely used method to reduce the dimensionality of data. PCA seeks the low-dimensional latent space such that original high dimensional data can be maximally reconstructed as a linear transformation from a sample point in the low-dimensional latent space. Notably, the low-dimensional latent space is just a subspace in the original high-dimensional space.

Autoencoder[11] can be viewed as a nonlinear generalization of PCA using neural networks. A typical autoencoder neural network consists of two parts [Figure 34.5.1]: one encoder part where high dimensional data is mapped via feed-forward neural layers to a low-dimensional representation, and one decoder part that decodes the low-dimensional representation to the original data.

The training of neural network is achieved by minimizing the decoder's reconstruction error. In the simplest case, consider an autoencoder consisting of a one-layer encoder and a one-layer decoder. Then the code for input x is given by the hidden layer representation h

$$h = \sigma(W_E x + b_E),$$

where σ is the Sigmoid activation function, W_E and b_E are the weight and bias of the encoder.

The decoder will reconstruct x' from h via

$$x' = \sigma(W_D h + b_D),$$

where W_D and b_D are the weight and bias of the decoder.

The reconstruction error is then given by

$$L(x, x') = \|x - x'\|^2.$$

Now we consider applying an autoencoder to compress MNIST dataset. We use an autoencoder network consisting of convolutional layers. The encoder network part is composed of two convolutional neural network and one max pooling layer to encoder the 784-dimensional image into a 49-dimensional code. The decoder consists of two transposed convolutional layers to increase the width and height of the input data, or upsampling the data. Alternative upsampling methods can be found in [12]. We also compared the PCA dimensionality reduction performance by reconstructing the same data using the top 50 principal components.

Results in Figure 34.5.2 show the comparison of reconstruction based on low-dimensional codes from neural networks and PCA. It is clear that the autoencoder method gives much better reconstruction than the PCA method.

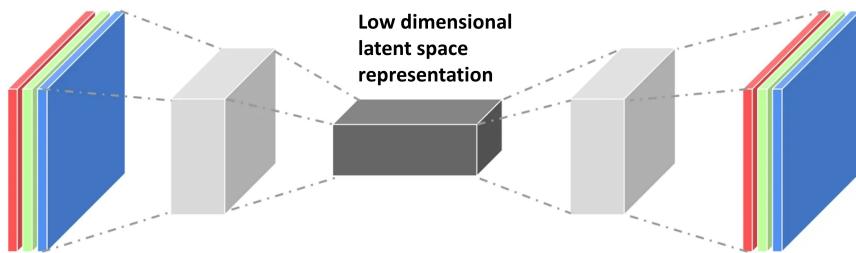


Figure 34.5.1: A CNN based autoencoder. An autoencoder consists of an encoder that transforms high-dimensional image into a low-dimensional code and a decoder that unfolds the code to reconstruct the image.

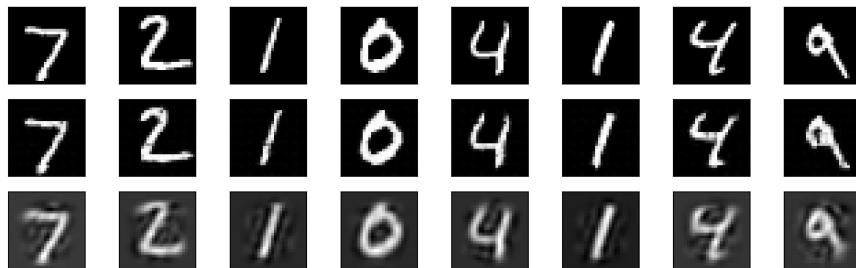


Figure 34.5.2: Comparison of reconstruction performance on random samples from MNIST data set. Top row is the original data. Middle row is autoencoder result based on a 49 dimensional code. Bottom row is PCA result based on a 50 dimensional code.

34.5.2 Denoising autoencoder

Since autoencoders can learn the latent representation, it can be used to remove noises similar to the noise removal functionalities of PCA and probabilistic PCA [13]. The training of autoencoder with the capability of removing noise has the following steps:

- Given an original input x , we add noise on to it to get $\tilde{x} = x + z, z \sim MN(0, I)$ and feed \tilde{x} into the autoencoder to get the reconstruction x' .
- The weights are updated by minimizing $\|x - x'\|^2$.

In the simplest case, consider an autoencoder consisting of one-layer encoder and one-layer decoder. Then the code for input $\tilde{x} = x + z$ is given by the hidden layer representation h

$$h = \sigma(W_E \tilde{x} + b_E),$$

where σ is the Sigmoid activation function, W_E and b_E are the weight and bias of the encoder.

The decoder will reconstruct x' from h via

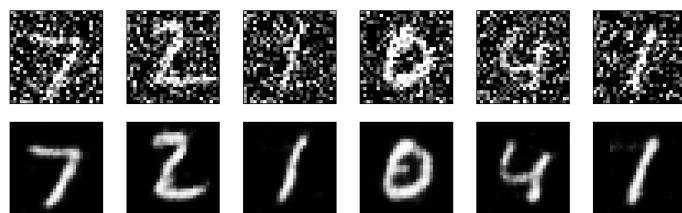
$$x' = \sigma(W_D h + b_D),$$

where W_D and b_D are the weight and bias of the decoder.

The reconstruction error is then given by

$$L(x, x') = \|x - x'\|^2.$$

[Figure 34.5.3](#) shows the application of a denoising autoencoder to the MNIST dataset.



[Figure 34.5.3](#): Denoising autoencoders applied to remove the noise in the MNIST dataset.

34.6 Neural style transfer

Neural style transfer is one of most fascinating applications of CNNs. The main idea is to use a pretrained CNN to extract the style and the content from two different images and then combine them together to synthesize a new image. In other words, transferring the style of one image to another using neural networks. An example of style transfer is given by [Figure 34.6.1](#).

To begin with, we first look into how the style and the content of an image are represented in a CNN. By the content of an image, we mean objects and their spatial information (such as positions and arrangement) in the image. Objects and their spatial information are roughly defined by geometry and edges. By style of an image, we mean the colors, appearance, and texture information of an image.

When we feed an image into a CNN classifier, the last layer can be viewed as a logistic classifier based on the features extracted from preceding layers. Particularly, the first few layer extracts low-level features like edges and textures and the final few layers extract higher-level features such as object parts like ears, noises, and wheels.

Consider a convolutional layer l with has N^l feature maps (each from a filter). Denote the feature map size by M_l , where M_l is the height multiplied by the width of the feature map. We further denote the feature map in layer l by a matrix $F^l \in \mathbb{R}^{N_l \times M_l}$. Here F_{ij}^l is the activation of the component j of the i filter at in layer l .

Let C_L denote the set of layers we use to extract content representation. Let F and P denote the content representation of a generated image x and target content image c . The content loss is then given by

$$L_{\text{content}}(x, c) = \frac{1}{2} \sum_{l \in C_L} w_{c,l} \|F - P\|_F^2 = \frac{1}{2} \sum_{l \in C_L} \sum_{i,j} w_{c,l} (F_{ij}^l - P_{ij}^l)^2$$

where $w_{c,l}$ are weights assigned to each layerwise loss.

The authors in [\[14\]](#) proposed that the style of an image is captured by the Gram matrix of filter activations in any layer of CNN network. The entries of the Gram matrix capture the correlations between the different filter responses via inner production. More specifically, the Gram matrix for layer l is a matrix $G^l \in \mathcal{R}^{N_l \times N_l}$, given by

$$G_{ij}^l = \sum_k F_{ik}^l F_{jk}^l, \text{ or } G^l = F^l [F^l]^T.$$

Let S_L denote the set of layers we use to extract content representation. Let G and A denote the content representation of generated image x and target content image s . The style loss is then given by

$$L_{\text{style}}(x, s) = \sum_{l \in S_L} \frac{1}{4N_l^2 M_l^2} w_{s,l} \|G - A\|_F^2 = \sum_{l \in S_L} \frac{1}{4N_l^2 M_l^2} \sum_{i,j} w_{s,l} (F_{ij}^l - P_{ij}^l)^2$$

where $w_{s,l}$ are weights assigned to each layerwise loss.

The total loss L is a linear combination of content loss and style loss, given by

$$L = \alpha L_{\text{content}} + \beta L_{\text{style}}.$$

In the following, we illustrate neural style transfer in two examples. In the first example, we transfer the Van Gogh painting style to the Shanghai view picture [Figure 34.6.1]. In the second example, we transfer the Picasso painting style to the Shanghai view picture [Figure 34.6.2].



(a) Content image and style image.

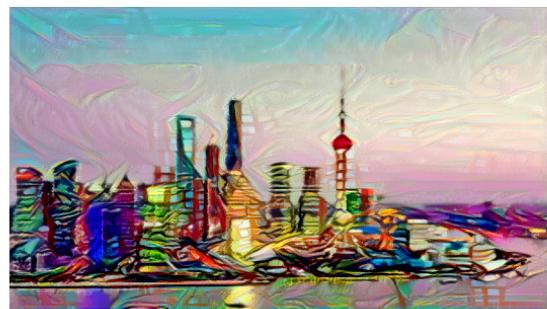


(b) Original content image and the transformed content image.

Figure 34.6.1: Demonstration of neural style transfer with Van Gogh painting style.



(a) Content image and style image.



(b) Original content image and the transformed content image.

Figure 34.6.2: Demonstration of neural style transfer with Picasso painting style.

34.7 Visual based deep reinforcement learning

Deep neural networks have found important applications in solving challenging sequential decision making problems. The method that combines deep learning and reinforcement learning is known as deep reinforcement learning (DRL). DRL has recently achieved human-level performance in diverse domains such as games[15, 16], robotics[17], drug design[18], etc. By equipping reinforcement learning method with convolutional neural networks, DRL expanded the success to challenging visual domains like robotics and self-driving cars.

In the following, we illustrate CNN based DRL in solving the optimal control problems in the field of nanorobot navigation. Nanorobot navigation is the key step to realize emerging biomedical applications like in autonomous targeted nanodrug delivery and precision surgery[19, 20]. In these application, nanorobots are carrying nano-sized drug to hard-to-reach locations (e.g., tissues, cancer cells) and deliver the drug there.

Consider a nanorobot in a 2D plane has equation of motion given by

$$\begin{aligned}\partial_t x &= v \cos(\theta) + \xi_x(t) \\ \partial_t y &= v \sin(\theta) + \xi_y(t) \\ \partial_t \theta &= \xi_\theta(t)\end{aligned}$$

where x , y , and ϕ denote the robot's position and orientation v is the propulsion speed as **control input**. ξ_x, ξ_y, ξ_θ are independent zero mean white noise stochastic processes resulting from Brownian motion and rotation. Compared the traditional macro-size robots, nanorobots are usually under-actuated (i.e., not all degrees of freedom can be controlled) and subject to Brownian motion.

The goal of DRL is find out a rule, known as control policy, to specify v based on the state of the robot (x, y, θ) and its local visual information (i.e. if there are obstacles nearby) such that the robot can arrive at a given target in minimum time.

Let us simplify our discussion in the discrete-time setting. We denote the robot state at time step n by $s_n = (x_n, y_n, \theta_n)$. The observation $\phi(s_n)$ at s_n consists of the image of the robot's neighborhood and the target position r^T , as shown in [Figure 34.7.1](#).

Suppose v only takes binary values of 0 and v_{max} , labeled by OFF and ON actions. In the Q learning approach, the optimal control policy is obtained by training the neural network [[Figure 34.7.1\(a\)](#)] to approximate the optimal action-value function (known as the Q function) given by

$$Q^*(\phi(s), v) = \mathbb{E} \left[R(s_0) + \gamma^1 R(s_1) + \gamma^2 R(s_2) + \dots | \phi(s_0) = \phi(s), v_0 = v, \pi^* \right]$$

which is the expected sum of rewards along the process by following the optimal policy π^* , after observing $\phi(s)$ and activating speed v . The neural network employs CNN to

process visual information of the neighborhood, represented by a binary image and a fully connected layer to take the target's position as the input. The neural network finally outputs the two optimal Q values associated with the ON and OFF actions.

If we allow v to take continuous value between 0 and v_{max} , we can use Actor-Critic approach to compute the optimal control policy. In the Actor-Critic approach, we use one CNN the actor network, to approximate the optimal control policy π^* , and another CNN as the critic network, to approximate the optimal Q function [Figure 34.7.1(b)],

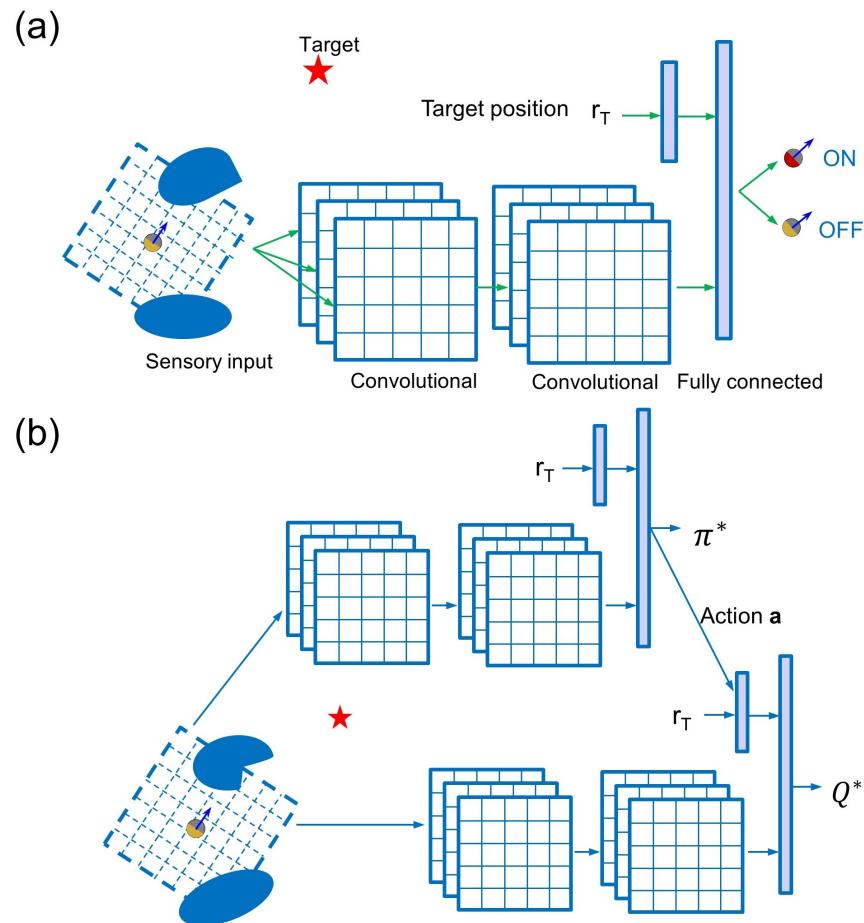


Figure 34.7.1: Application of CNN in deep reinforcement learning in Q learning approach and Actor-Critic approach. Two streams of sensory inputs are fed to the neural network, including a pixel image of the robot's neighborhood fed into a convolutional layer and the target's position fed into a fully connected layer.

BIBLIOGRAPHY

1. Yu, F. & Koltun, V. Multi-scale context aggregation by dilated convolutions. *arXiv preprint arXiv:1511.07122* (2015).
2. LeCun, Y., Bottou, L., Bengio, Y., Haffner, P., et al. Gradient-based learning applied to document recognition. *Proceedings of the IEEE* **86**, 2278–2324 (1998).
3. Krizhevsky, A., Sutskever, I. & Hinton, G. E. *Imagenet classification with deep convolutional neural networks* in *Advances in neural information processing systems* (2012), 1097–1105.
4. Simonyan, K. & Zisserman, A. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556* (2014).
5. He, K., Zhang, X., Ren, S. & Sun, J. Deep residual learning for image recognition in *Proceedings of the IEEE conference on computer vision and pattern recognition* (2016), 770–778.
6. Srivastava, R. K., Greff, K. & Schmidhuber, J. Highway networks. *arXiv preprint arXiv:1505.00387* (2015).
7. Zeiler, M. D. & Fergus, R. Visualizing and understanding convolutional networks in *European conference on computer vision* (2014), 818–833.
8. Erhan, D., Bengio, Y., Courville, A. & Vincent, P. Visualizing higher-layer features of a deep network. *University of Montreal* **1341**, 1 (2009).
9. Ozbulak, U. *PyTorch CNN Visualizations* <https://github.com/utkuozbulak/pytorch-cnn-visualizations>. 2019.
10. Selvaraju, R. R. et al. Grad-cam: Visual explanations from deep networks via gradient-based localization in *Proceedings of the IEEE international conference on computer vision* (2017), 618–626.
11. Hinton, G. E. & Salakhutdinov, R. R. Reducing the dimensionality of data with neural networks. *science* **313**, 504–507 (2006).
12. Odena, A., Dumoulin, V. & Olah, C. Deconvolution and checkerboard artifacts. *Distill* **1**, e3 (2016).
13. Vincent, P., Larochelle, H., Lajoie, I., Bengio, Y. & Manzagol, P.-A. Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion. *Journal of machine learning research* **11**, 3371–3408 (2010).

14. Gatys, L. A., Ecker, A. S. & Bethge, M. *Image style transfer using convolutional neural networks* in *Proceedings of the IEEE conference on computer vision and pattern recognition* (2016), 2414–2423.
15. Silver, D. *et al.* Mastering the game of Go with deep neural networks and tree search. *nature* **529**, 484 (2016).
16. Mnih, V. *et al.* Human-level control through deep reinforcement learning. *Nature* **518**, 529 (2015).
17. Levine, S., Finn, C., Darrell, T. & Abbeel, P. End-to-end training of deep visuomotor policies. *The Journal of Machine Learning Research* **17**, 1334–1373 (2016).
18. Popova, M., Isayev, O. & Tropsha, A. Deep reinforcement learning for de novo drug design. *Science advances* **4**, eaap7885 (2018).
19. Li, J., Ávila, B. E.-f. D., Gao, W., Zhang, L. & Wang, J. Micro / nanorobots for biomedicine : Delivery , surgery , sensing , and detoxification (2017).
20. Wang, J. & Gao, W. Nano/microscale motors: Biomedical opportunities and challenges. *ACS Nano* **6**, 5745–5751. ISSN: 19360851 (2012).

35

RECURRENT NEURAL NETWORKS

35	RECURRENT NEURAL NETWORKS	1478
35.1	Recurrent neural network	1479
35.1.1	Simple recurrent unit (SRU)	1479
35.1.2	Simple RNN and its approximation capability	1480
35.1.3	Backpropogation through time (BPTT)	1480
35.2	Recurrent unit variants	1483
35.2.1	Long short term memory (LSTM)	1483
35.2.2	Gated Recurrent Unit (GRU)	1486
35.3	Common RNN architectures	1488
35.3.1	Stacked RNN and bidirectional RNN	1488
35.3.2	Task dependent output layers	1490
35.3.3	Dropout	1491
35.3.4	LSTM with recurrent projection layer	1492
35.4	RNN application examples	1494
35.4.1	Time series prediction	1494
35.4.1.1	Simple RNN prediction	1494
35.4.1.2	Deep autoregressive (DeepAR) model	1497
35.4.1.3	Deep factor model	1498
35.4.2	MNIST classification with sequential observation	1499
35.4.3	Character-level language modeling	1500
35.4.3.1	Word classification	1500
35.4.3.2	Text generation	1501

35.1 Recurrent neural network

35.1.1 Simple recurrent unit (SRU)

In applications ranging from time-series prediction and classification to speech recognition and machine translation, we need to process sequence data of variable length. The Forward feeding neural networks such as MLP and CNN introduced in the previous section use fixed architectures, thus require the input data to be have fixed dimensionality and size. For example, image data should be cropped or enlarged to fixed data before feeding to a CNN. However, the fixed input size requirement is not flexible for machine learning problem involving sequence data as input. For example, in machine translation, each sample input is a word sequence (e.g., (w_1, \dots, w_t)) consisting different number of words. Another detailed feature in a neural network is the ability to capture dependence within a sequence. For applications involving natural language, these dependencies can also be of long range.

Recurrent neural networks (RNN) extend feed-forward neural networks's capability in addressing sequence modeling tasks. The main idea of RNN is to introduce recurrent units among the hidden layers [Figure 35.1.1]. In this way, the number of recurrent iteration can be varied according to the input sequence length. By feeding past hidden layer output into next hidden layers, dependence among a sequence is modeled.

More formally, consider a sequence input $x = (x_1, x_2, \dots, x_t, \dots, x_L), x_t \in \mathbb{R}^D$. Let x_t be the input at time index t in the sequence, the evolution of the hidden state h_t is given by [Figure 35.1.1]

$$\begin{aligned} z_t &= W_h h_{t-1} + W_x x_t + b_h \\ h_t &= g_h(z_t) \end{aligned}$$

where g_h is the activation function (usually g_h is Sigmoid or Tanh), $W_h \in \mathbb{R}^{H \times H}$, and $W_x \in \mathbb{R}^{D \times H}, b \in \mathbb{R}^H$. h_0 is usually taken to be zero.

The recurrent process can be seen more clearly if we unroll the a RNN structure explicitly through time (up to the length of sequence). The unrolled RNN is equivalent to MLP with restricted connections and shared weights. The output h_1, \dots, h_T can be fed into different neural network layer structures (e.g., fully-connected layer) to enable diverse applications.

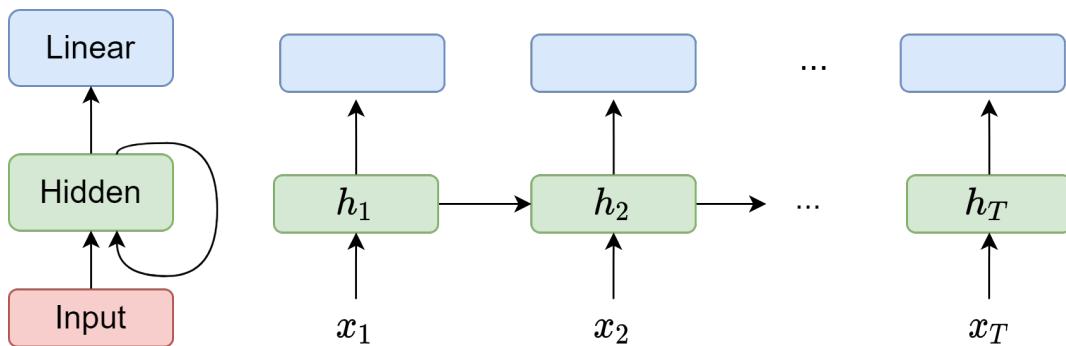


Figure 35.1.1: Scheme of recurrent units in a neural network (left). Recurrent neural network can be unrolled (right)

35.1.2 Simple RNN and its approximation capability

Many sequence related models and applications can be abstracted as the modeling of a nonlinear dynamic systems. One extraordinary property of a RNN is its capability to approximate the dynamics of a broad range of nonlinear dynamic systems.

Theorem 35.1.1 (universal approximation to nonlinear dynamic systems). [1, p. 798][2, p. 139] Any nonlinear dynamic system given by

$$\begin{aligned} h_t &= g(h_{t-1}, x_t) \\ y_t &= o(s_t) \end{aligned}$$

where h_t is the hidden state, x_t is external input, can be approximated by a recurrent neural network to any desired degree of accuracy if the network is equipped with an adequate number of hidden neurons.

35.1.3 Backpropagation through time (BPTT)

The most common way to train a RNN is through stochastic gradient descent. In feed-forward neural networks, gradient calculation simplifies to the backpropagation procedure [subsection 32.1.4]. Using similar derivation, we can obtain a gradient calculation procedure, known as **backpropagation through time (BPTT)**.

To derive BPTT, we consider a simple RNN [Figure 35.1.2] with following formulation:

$$h_t = \tanh(W_{hh}h_{t-1} + W_{xh}x_t + b_h)$$

$$y_t = \text{softmax}(W_{hy}h_t + b_y)$$

And we assume the total loss L is a function of all the output y_1, \dots, y_T . To simplify, we further assume that the total loss can be decomposed to the summation of loss at each step t , i.e., $L = \sum_{i=1}^T L_t$.

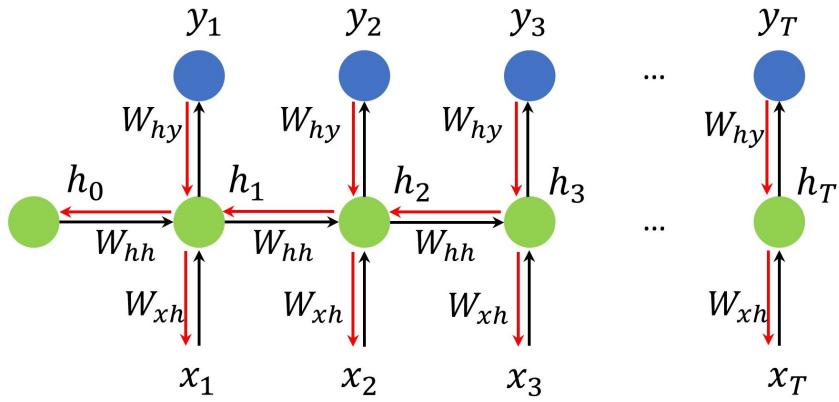


Figure 35.1.2: Scheme for backpropagation through time in a simple RNN. Red arrows are backpropagation directions.

Because W_{hy}, b_y contribute to the loss via y_1, \dots, y_T , the gradient with respect to W_{hy}, b_y is a summation of T items, given by

$$\begin{aligned}\frac{\partial L}{\partial W_{hy}} &= \sum_{t=1}^T \frac{\partial L}{\partial y_t} \frac{\partial y_t}{\partial W_{hy}} \\ \frac{\partial L}{\partial b_y} &= \sum_{t=1}^T \frac{\partial L}{\partial z_t} \frac{\partial z_t}{\partial b_y}.\end{aligned}$$

W_{hh} contributes to the loss at step t via updating h_1, h_2, \dots, h_t from h_0, h_1, \dots, h_{t-1} , respectively. Note that we use L_t to denote the loss associated with y_t , we have the gradient of L_t with respect to W_{hh} given by

$$\frac{\partial L_t}{\partial W_{hh}} = \sum_{k=0}^{t-1} \frac{\partial L_t}{\partial z_t} \frac{\partial z_t}{\partial h_t} \frac{\partial h_t}{\partial h_k} \frac{\partial h_k}{\partial W_{hh}}$$

Aggregating together, we have

$$\frac{\partial L}{\partial W_{hh}} = \sum_{t=1}^T \frac{\partial L_t}{\partial W_{hh}} = \frac{\partial L_t}{\partial W_{hh}} = \sum_{k=0}^{t-1} \frac{\partial L_t}{\partial z_t} \frac{\partial z_t}{\partial h_t} \frac{\partial h_t}{\partial h_k} \frac{\partial h_t}{\partial W_{hh}}$$

Similarly, W_{xh} contributes to the loss at step t via updating h_1, h_2, \dots, h_t from x_1, x_{t-1}, \dots, x_t , respectively. Therefore, we have

$$\frac{\partial L_t}{\partial W_{xh}} = \sum_{k=1}^t \frac{\partial L_t}{\partial z_t} \frac{\partial z_t}{\partial h_t} \frac{\partial h_t}{\partial h_k} \frac{\partial h_t}{\partial W_{xh}},$$

Aggregating together, we have

$$\frac{\partial L}{\partial W_{xh}} = \sum_{t=1}^T \frac{\partial L_t}{\partial W_{xh}} = \sum_{t=1}^T \sum_{k=1}^t \frac{\partial L_t}{\partial z_t} \frac{\partial z_t}{\partial h_t} \frac{\partial h_t}{\partial h_k} \frac{\partial h_t}{\partial W_{xh}}$$

Note that $\frac{\partial h_t}{\partial h_k}$ involves matrix multiplication over the sequence. Specifically,

$$\frac{\partial h_t}{\partial h_k} = \frac{\partial h_t}{\partial h_{t-1}} \frac{\partial h_{t-1}}{\partial h_{t-2}} \cdots \frac{\partial h_{k+1}}{\partial h_k} = \prod_{i=k}^{t-1} \frac{\partial h_{i+1}}{\partial h_i}.$$

Each single term in the chain is given by

$$\frac{\partial h_t}{\partial h_{t-1}} = \text{diag}[\tanh'(W_{hh}h_{t-1} + W_{xh}x_t + b_h)]W_{hh}.$$

if the largest eigenvalue (in terms of absolute value) of the matrix W_{hh} is greater than 1, the gradient will explode for long sequences; on the other hand, if the largest eigenvalue is less than 1, the gradient will vanish [3]. For gradient vanishing issues, the states and inputs that are far away from the current time step contribute negligibly to the gradient computing of current step.

35.2 Recurrent unit variants

35.2.1 Long short term memory (LSTM)

One critical variant of RNN is the Long short-term memory (LSTM) unit [Figure 35.2.1. [4]] LSTM aims to overcome the issue of gradient vanishing or explosion in RNN and thus models long-term dependence in a more robust and reliable way. The key success of LSTM is to offer more parameters, in a structural way, to control the BPTT process. The specific improvements are the introduction of internal cell state c_t and three gates (input, forget, and output gates) to control the information flow.

In LSTM, the recurrent relation connecting c_t and h_t to their previous step now becomes

$$\begin{aligned} c_t &= f_t \odot c_{t-1} + i_t \odot \tilde{c}_t \\ h_t &= o_t \odot \tanh(c_t) \end{aligned}$$

where \tilde{c}_t is candidate cell state given by

$$\tilde{c}_t = \tanh(W_c x_t + U_c h_{t-1} + b_c),$$

f_t and i_t are the forget gate and input gate vector, respectively, with elements between 0 and 1 to determine the composition of the final cell state c_t .

The three gates output value (achieved by using sigmoid function, denoted by σ) between 0 and 1 to control information flow. Forget gate f_t controls how much to discard in the previous cell state c_{t-1} ; input gate i_t controls how much to keep in current candidate cell state \tilde{c}_t ; output gate o_t controls how much in c_t will output to h_t . We have

$$\begin{aligned} i_t &= \sigma(W_i x_t + U_i h_{t-1} + b_i) \\ f_t &= \sigma(W_f x_t + U_f h_{t-1} + b_f) \\ o_t &= \sigma(W_o x_t + U_o h_{t-1} + b_o) \end{aligned}$$

The overall information flow scheme can be summarized in Figure 35.2.1.

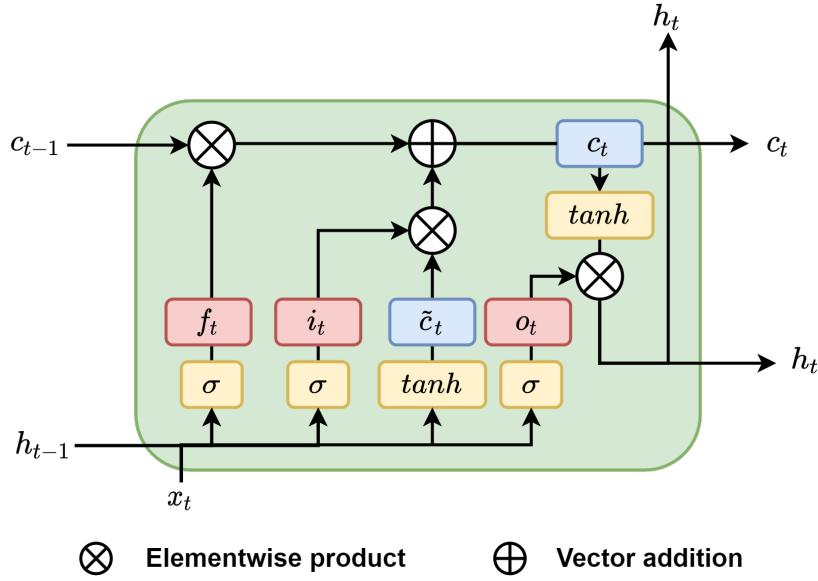


Figure 35.2.1: Scheme of an LSTM cell.

And the mathematical relation can be summarized by

Definition 35.2.1 (LSTM cell mathematical relations). A LSTM cell takes input x_t , previous output h_{t-1} , and previous cell state c_{t-1} and outputs h_t and c_t via the following relations:

$$\begin{aligned}
 \tilde{c}_t &= \tanh(W_c x_t + U_c h_{t-1} + b_c) \\
 i_t &= \sigma(W_i x_t + U_i h_{t-1} + b_i) \\
 f_t &= \sigma(W_f x_t + U_f h_{t-1} + b_f) \\
 o_t &= \sigma(W_o x_t + U_o h_{t-1} + b_o) \\
 c_t &= f_t \odot c_{t-1} + i_t \odot \tilde{c}_t \\
 h_t &= o_t \odot \tanh(c_t)
 \end{aligned}$$

where parameter and inputs dimensions include $h_t \in \mathbb{R}^H$, $x_t \in \mathbb{R}^D$, $c_t, i_t, o_t, f_t \in \mathbb{R}^H$, $W_i, W_f, W_o \in \mathbb{R}^{H \times D}$, $U_i, U_f, U_o \in \mathbb{R}^{H \times H}$, and $b_c, b_i, b_f, b_o \in \mathbb{R}$.

Now we discuss the intuition underlying LSTM and its mechanism in modeling long term dependence among sequences. addresses gradient vanishing and explosion.

LSTM has two hidden units c_t and h_t , which can be intuitively interpreted as **long-term memory** and **short-term memory**, respectively. During every iteration, long-

term memory c_t is updated to include information from x_t, h_{t-1} while maintaining old information in c_{t-1} , which is given by

$$\begin{aligned} c_t &= f_t \odot c_{t-1} + i_t \odot \tilde{c}_t \\ \tilde{c}_t &= \tanh(W_c x_t + U_c h_{t-1} + b_c) \end{aligned}$$

where the ratio of storing new information and maintaining old information is controlled by the input gate value i_t and the forget gate value f_t . The use of forget gate enables information stored in c_t change slowly and last longer, therefore c_t is called the long term memory.

Information stored in long terms c_t will output to short-term memory h_t via $h_t = o_t \odot \tanh(c_t)$ where $o_t = \sigma(W_o x_t + U_o h_{t-1} + b_o)$. Note that the output gate o_t is affected by input x_t and short-term memory h_{t-1} .

The updated short-term memory h_t , which contains both short-term and long-term information, will be subsequently connected to additional feed-forward layers specific to tasks.

Note 35.2.1 (forget gate initialization to enable gradient flow). In practice, to make most of forget gate to cope with the vanishing gradient issue, we usually initialize b_f close to 1 (so f_t will be close 1) to enable gradients carry over multiple time steps.

In subsection 35.1.3, we have quantitatively discussed that BPTT in simple RNN can cause gradients to vanish, which ultimately prevents learning long-term relationship from sequences. Now we look at how LSTM quantitatively alleviates the gradient vanishing issues.

Because the cell state maintains long term memory, we want to see if gradient flow could suffer exponential decay as in a vanilla RNN. We look at the derivative for $\frac{\partial c_t}{\partial c_{t-1}}$. Note that

$$\begin{aligned} c_t &= f_t \odot c_{t-1} + i_t \odot \tilde{c}_t \\ \tilde{c}_t &= \tanh(W_c x_t + U_c h_{t-1} + b_c) \\ i_t &= \sigma(W_i x_t + U_i h_{t-1} + b_i) \\ f_t &= \sigma(W_f x_t + U_f h_{t-1} + b_f) \end{aligned}$$

so we have

$$\frac{\partial c_t}{\partial c_{t-1}} = f_t + \frac{\partial f_t}{\partial c_{t-1}} \odot c_{t-1} + \frac{\partial i_t}{\partial c_{t-1}} \odot \tilde{c}_t + i_t \odot \frac{\partial \tilde{c}_t}{\partial c_{t-1}}.$$

We are not going to pursue further details; instead, the key idea in LSTM is that there are multiple learnable components controls the damping or growing effects on each step via gates. The network can learn to set the gate values, conditioning on the current input and hidden state, to decide when to damp gradient to eliminate long term influence, and when to grow the gradient to preserve the long term dependence. On the contrary, the vanilla RNN lacks such flexibility (RNN only has $W_h h$ to control the gradient change) and the gradient either explodes or vanishes for long sequences.

35.2.2 Gated Recurrent Unit (GRU)

Gated recurrent unit [5] is a simpler RNN variant than LSTM (e.g., no cell states) but is still endowed with key features of LSTM for long-term dependence modeling. GRU employs an **update gate** $z_t \in [0, 1]$ and a **reset gate** $r_t \in [0, 1]$ to control information flow. The output of the hidden layer is then given by

$$\begin{aligned} h_t &= z_t \odot h_{t-1} + (1 - z_t) \odot \tilde{h}_t \\ z_t &= \sigma(W_z x_t + U_z h_{t-1} + b_z) \\ \tilde{h}_t &= \tanh(W_h x_t + U_h (r_t \odot h_{t-1}) + b_h) \\ r_t &= \sigma(W_r x_t + U_r h_{t-1} + b_r) \end{aligned}$$

where \tilde{h}_t is a candidate state. The reset gate controls the dependence of \tilde{h}_t on the previous h_{t-1} , and the update gate controls the output h_t as the mixture of h_{t-1} and \tilde{h}_t . Notably,

- When $z = 0, r = 1$, GRU becomes simple RNN;
- When z is large, most historical information is preserved.

The overall information flow scheme can be summarized in [Figure 35.2.2](#).

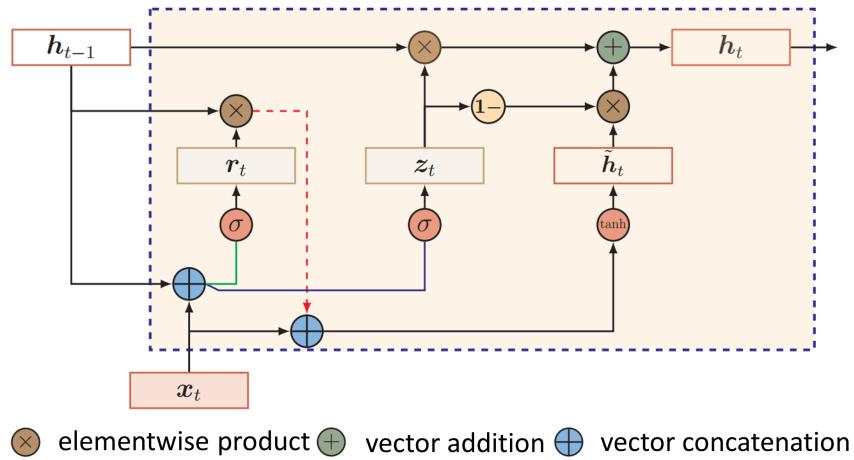


Figure 35.2.2: Scheme of a GRU cell. Modified from [2, p. 152].

And the mathematical relation can be summarized by

Definition 35.2.2 (GRU cell mathematical relations). A GRU cell takes input x_t and previous output h_{t-1} and outputs h_t via the following relations:

$$\begin{aligned}
 z_t &= \sigma(W_z x_t + U_z h_{t-1} + b_z) \\
 \tilde{h}_t &= \tanh(W_h x_t + U_h (r_t \odot h_{t-1}) + b_h) \\
 r_t &= \sigma(W_r x_t + U_r h_{t-1} + b_r) \\
 h_t &= z_t \odot h_{t-1} + (1 - z_t) \odot \tilde{h}_t
 \end{aligned}$$

where parameter and inputs dimensions include $h_t \in \mathbb{R}^H$, $x_t \in \mathbb{R}^D$, $r_t, z_t \in \mathbb{R}^H$, $W_z, W_r \in \mathbb{R}^{H \times D}$, $U_z, U_h, U_r \in \mathbb{R}^{H \times H}$, and $b_z, b_r, b_h \in \mathbb{R}$.

35.3 Common RNN architectures

35.3.1 Stacked RNN and bidirectional RNN

We have covered several most commonly used recurrent units [Figure 35.3.1](#). These recurrent units can be used with other neural network units, which are collectively called recurrent neural networks, to perform general regression and classification tasks.

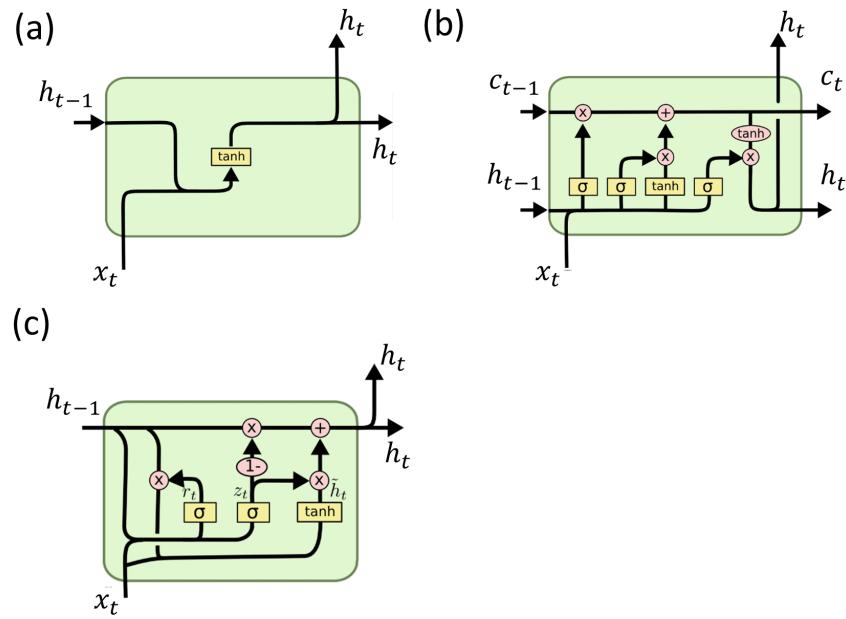


Figure 35.3.1: Different types of units in RNNs: (a) Vanilla RNN cell. (b) LSTM cell. (c) GRU cell. Credit

First, to enable recurrent layer to learn more high-level representative features in the sequence (e.g., in complex time series prediction and language modeling), we can stack multiple recurrent layers together and form stacked RNN [[Figure 35.3.2a](#)].

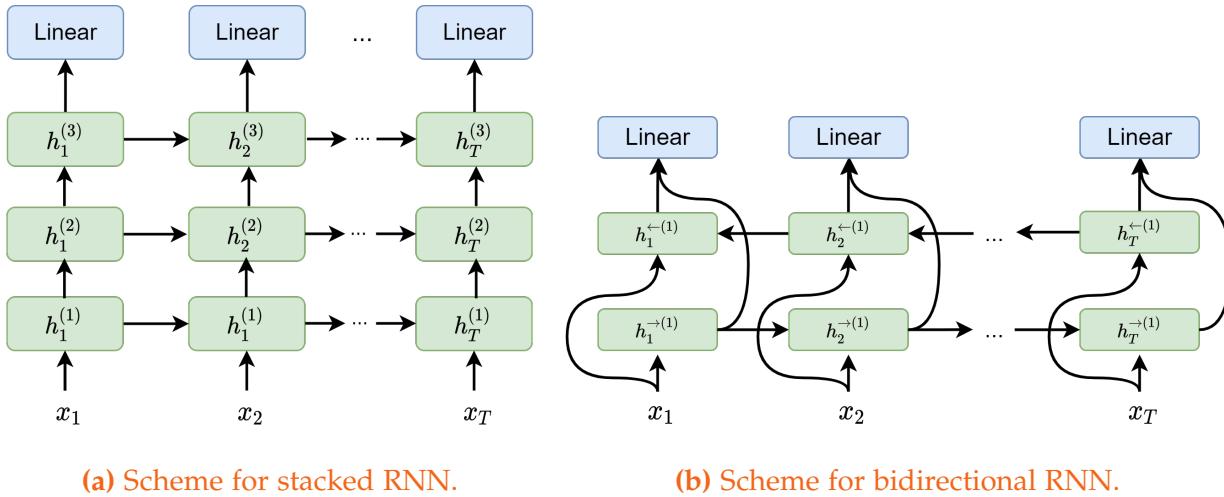


Figure 35.3.2: Stacked RNN and bidirectional RNN.

In the stacked RNN with input dimension D and hidden unit dimension H , the weight matrices associated with first layer RNN cells are $W_{ih}^{(1)} \in \mathbb{R}^{D \times H}$ and $W_{hh}^{(1)} \in \mathbb{R}^{H \times H}$; the weight matrices associated with second layer RNN cells are $W_{ih}^{(1)} \in \mathbb{R}^{H \times H}$ and $W_{hh}^{(1)} \in \mathbb{R}^{H \times H}$ since first layer RNN cell output are the input to the second layer RNN cells.

Another important architecture of RNN is bidirectional RNN [Figure 35.3.2b], which can be used for context modeling with context from both past and future. Bidirectional RNN has one layer reverse recurrent layer flowing subsequent input information to preceding hidden states (e.g., from x_T to x_1).

Finally, we can combine stacking and bidirectionality together and get stacked bidirectional RNN [Figure 35.3.3]. Example application includes tagging a word's entity and machine translation in where the meaning and tag of a word usually depends on its preceding and subsequent surrounding words.

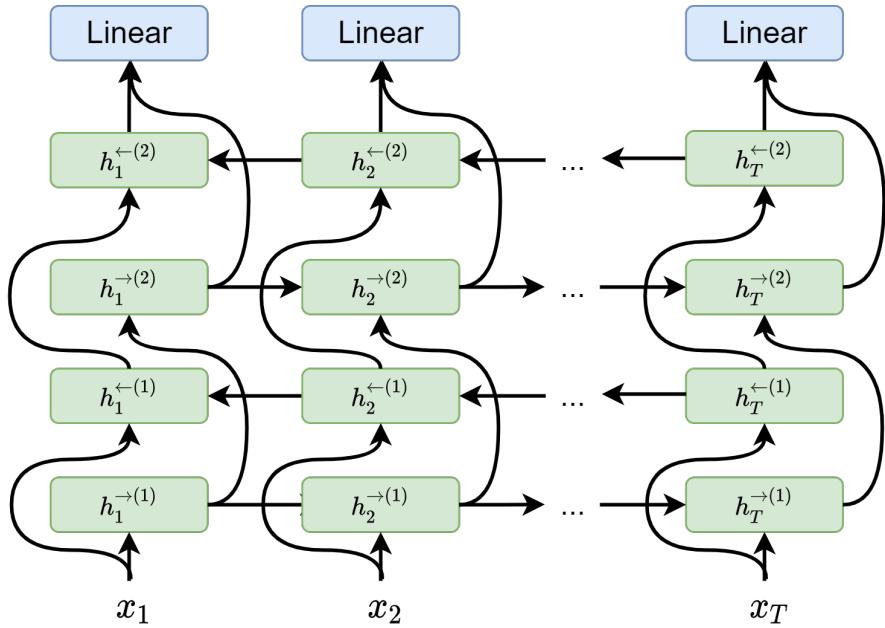


Figure 35.3.3: Stacked bidirectional RNN

35.3.2 Task dependent output layers

Without RNN extracts features from sequential inputs, we use different output layers to solve different tasks [[Figure 35.3.4](#)]. The architecture in which recurrent output of every time step t connects to the output layer [[Figure 35.3.4\(a\)](#)] is suitable for modeling dynamic system represented by $y_t = f(x_{1:t})$ (e.g., time series prediction), where y_t has dependency on x_1, \dots, x_t . On the other hand, for sequence data classification problem, we only need one output vector given the whole sequence data, then we can use architecture like [Figure 35.3.4\(b\)\(c\)](#).

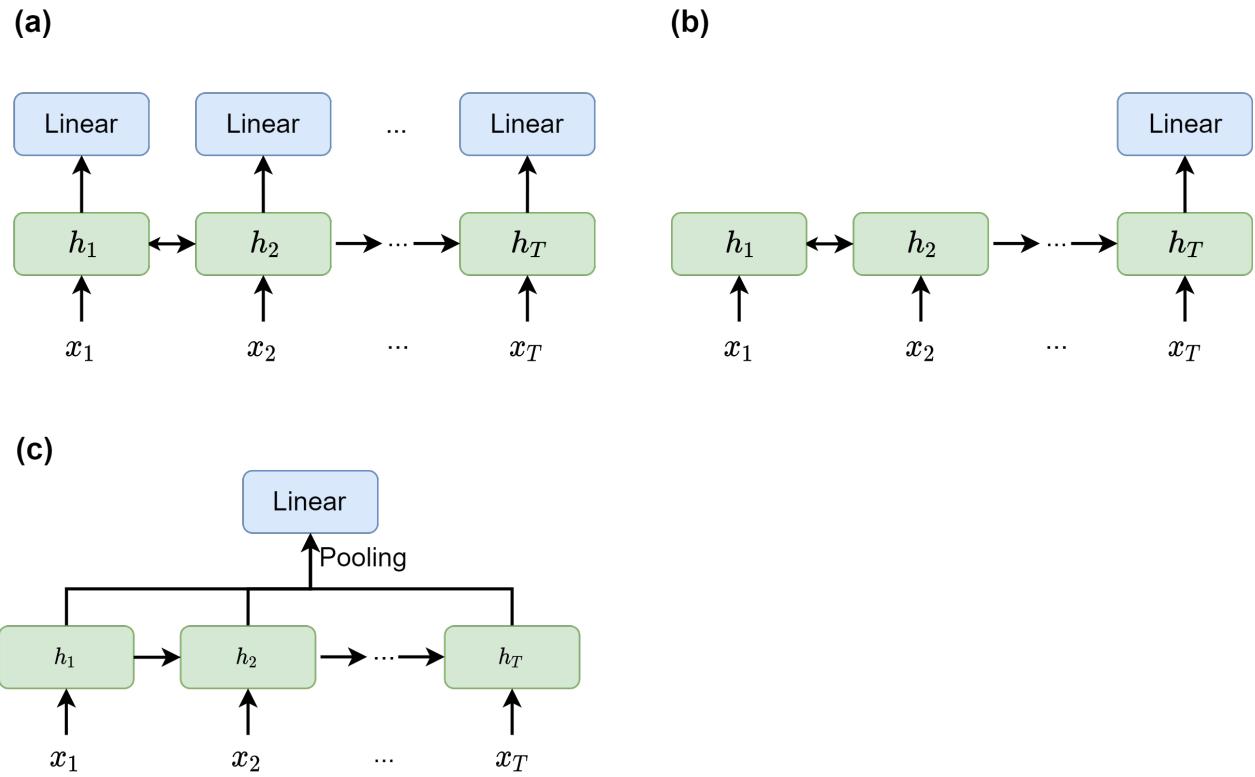


Figure 35.3.4: Typical RNN connection to the output layer.

35.3.3 Dropout

Like feed-forward multi-layer neural network, dropout can be applied to recurrent neural network to improve robustness and generalization[6]. Dropout can be applied at the input, output, and recurrent connections [Figure 35.3.5]. Given a sequence of inputs, it is unclear whether dropout mask applied should be different or same for every time step. [7] points out that the theoretically sound approach is to apply the dropout mask for every time step.

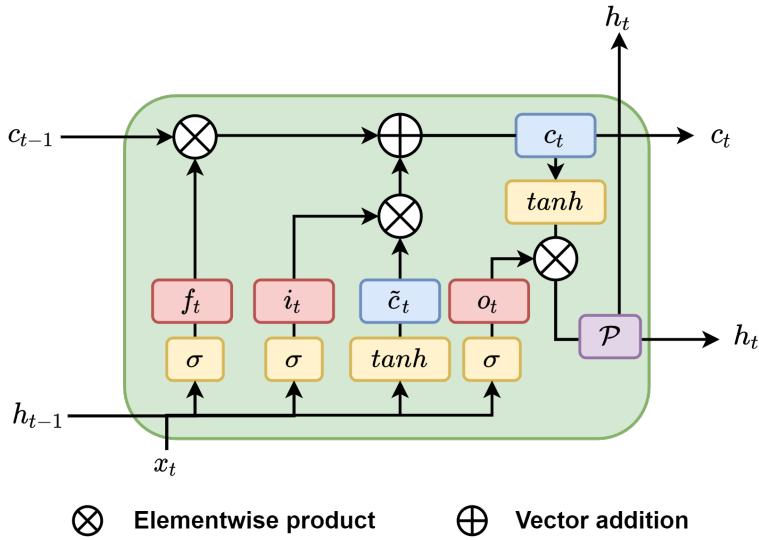


Figure 35.3.6: LSTM cell with a projection layer (purple box) added after the hidden output h_t .

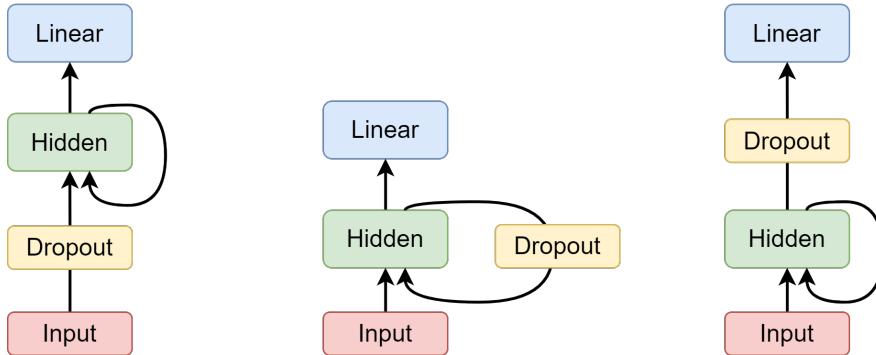


Figure 35.3.5: Three places to add Dropout to a RNN: at the input, at the recurrent connection, and at the output.

35.3.4 LSTM with recurrent projection layer

A recent improvement on LSTM [8] in terms of effective use of model parameters and reducing computational cost is to introduce a projection layer after the hidden output h_t , as shown in [Figure 35.3.6](#).

This additional projection layer changes the LSTM cell in the following way. Let the \tilde{h}_t be the original hidden output given by [Definition 35.2.1](#). The project layer is associated with a weight matrix $W_{ph} \in \mathbb{R}^{P \times H}$ such that the new hidden output $h_t \in \mathbb{R}^P$ is given by

$$h_t = W_{ph}\tilde{h}_t.$$

Here P is the projection size.

Additional changes in the LSTM cell include

- While the hidden h_t has dimension P , cell states c_t still has dimension H .
- Since all hidden h_t has dimension P , all matrices associated with gates will have size $H \times P$.

35.4 RNN application examples

35.4.1 Time series prediction

35.4.1.1 Simple RNN prediction

The most straight forward application of RNN is time series modeling. RNN can model complex, nonlinear time series beyond linear time series we consider in [chapter 22](#). A basic architecture for nonlinear time series with possible long memory dependence is given by [Figure 35.4.1](#).

Time series prediction problems are generally formulated as a regression problem. Let $x_1, x_2, \dots, x_N, x_i \in \mathbb{R}^D$ be an example time series observation. Let the context window by K , we can transform the time series observation into training samples with input and outcome pairs like

$$(x_1, \dots, x_k), x_k; (x_2, \dots, x_{k+1}), x_{k+1}; (x_1, \dots, x_k), x_{k+1}; \dots$$

The training examples are then fed into RNN and yield predicted values [[Figure 35.4.1\(b\)](#)]. The loss function can simply be the mean squared error between the outcome and the predicted value.

In the prediction stage, RNN can predict more than the immediate next step by constantly feeding preceding predicted values to the network [[Figure 35.4.1\(b\)](#)]. As a demonstration, we train a RNN to predict time series

$$y(t) = f(x(t), x(t-1), \dots),$$

where $x(t) = \cos(t), y(t) = \sin(t)$.

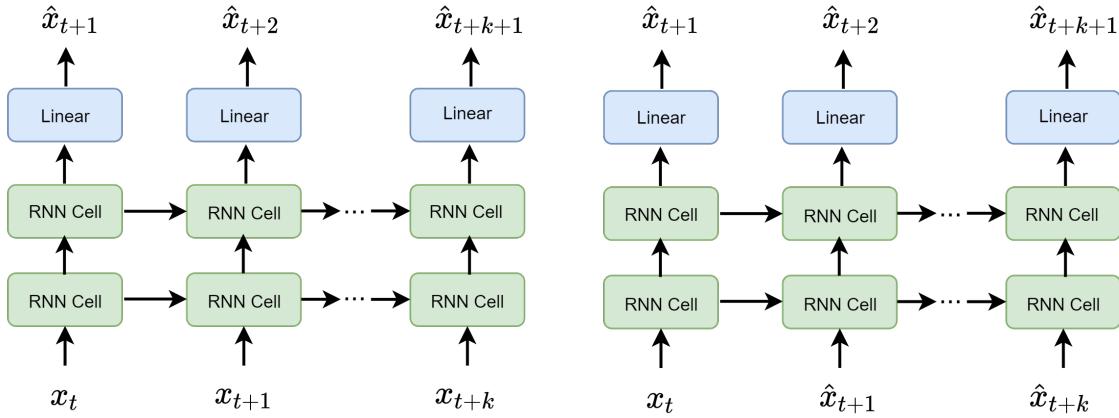


Figure 35.4.1: RNN architecture for time series prediction . (left) In the training phase, RNN are updated by minimizing the next step prediction error. (right) In the prediction phase, trained RNN is used to sequentially predict next step state value based on preceding predicted state value.

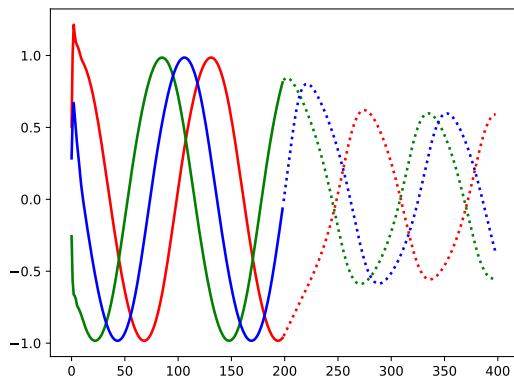
Consider a toy model. The training data are 1000 trajectories of sine wave data with random initial phases ϕ

$$\begin{aligned}
 & (\sin(\frac{0}{T} + \phi^0), \sin(\frac{1}{T} + \phi^0), \dots, \sin(\frac{N}{T} + \phi^0)) \\
 & (\sin(\frac{0}{T} + \phi^1), \sin(\frac{1}{T} + \phi^1), \dots, \sin(\frac{N}{T} + \phi^1)) \\
 & \dots \\
 & (\sin(\frac{0}{T} + \phi^{1000}), \sin(\frac{1}{T} + \phi^{1000}), \dots, \sin(\frac{N}{T} + \phi^{1000}))
 \end{aligned}$$

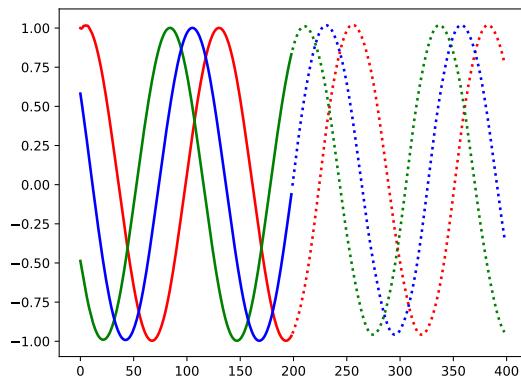
where $N = 10T, T = 20$.

We use a two-layer stack RNN and train for 50 episodes [Figure 35.4.2]. After training, we use 3 out of sample time series to examine the one-step forward and multiple forward prediction performance. We have following observations:

- After 2 episodes of training, we can see that one step forward prediction straggles only when input sequence is short (i.e., less than 10). Multi-step forward prediction has poor performance.
- After 17 episodes of training, we can see good performance at both one-step forward and multi-step forward prediction.



(a) RNN (after 2 episode's training) predicted values with 3 different observed time series of 200 steps as the input. The solid lines are one-step forward predictions; the dashed lines are multi-step forward predictions.



(b) RNN (after 17 episode's training) predicted values with 3 different observed time series of 200 steps as the input. The solid lines are one-step forward predictions; the dashed lines are multi-step forward predictions..

Figure 35.4.2: RNN one-step forward and multiple forward prediction performance for Sine time series.

In the previous architecture [Figure 35.4.1], the input are the historical observations of the time series. Now we consider a more general architecture that allows the input to include additional covariate time series z_t [Figure 35.4.3]. Covariate time series are sequential observations of variables that have some relationship with the time series. We usually assume covariate time series are available during future horizon (they are from either external model prediction or from user input). The use of covariate time series are quite common in practice as a means to enhance prediction performance. For example, let the stock price of APPLE be the time series of our interest, and SP500 index be the covariate time series. Predicting APPLE stock price given SP500 observation will be easier than directly predicting APPLE stock price from its historical prices, as there is strong correlation between SP500 and APPLE stock.

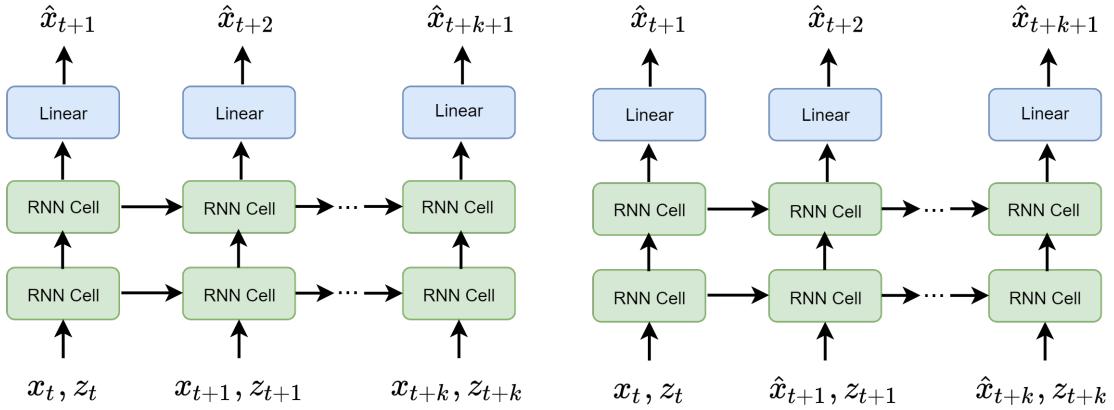


Figure 35.4.3: RNN architecture for time series prediction with covariates. (left) In the training phase, RNN are updated by minimizing the next step prediction error. (right) In the prediction phase, trained RNN is used to sequentially predict next step state value based on preceding predicted state value. Note that covariate time series are assumed available for all time steps.

35.4.1.2 Deep autoregressive (DeepAR) model

Previous simple RNN architectures [Figure 35.4.1, Figure 35.4.3] can be used to predict deterministic time series or to predict mean levels for stochastic time series. Deep autoregressive (DeepAR) model [9] is an extension that combines statistical modeling methods and RNN. DeepAR models allows we make additional distribution assumption on the time series.

Suppose We are given a number of time series $x_t^{(i)}$ where i is the time series index and t is the time index; and a number of covariate time series $z_t^{(i)}$. The goal is to learn the conditional distribution of the future distribution of x_t of the time series, given its past and the covariates.

More formally, denote $[x_t, x_{t+1}, \dots, x_T] := z_{t:T}$, the goal is to model the conditional distribution $P(x_{T+1}|z_{t:T+1}, x_{1:T})$. In DeepAR model architecture [Figure 35.4.4], we assume the conditional distribution follows some parametric forms and use RNN to learn the distribution parameters. For example, if assumed conditional distribution is Gaussian, then the outputs are parameters including the mean μ_t and standard deviation σ_t for prediction \hat{x}_{T+1} .

The network parameters for the feed-forward module and the recurrent modules are trained by minimizing the negative log likelihood function. If we assume z_t follows Gaussian, then the loss function is given by

$$L(x_t | \hat{\mu}_t, \hat{\sigma}_t) = -\sum_{i=1}^N \log(2\pi\sigma_i^2)^{-\frac{1}{2}} + \sum_{i=1}^N \left(\frac{(z_t^{(i)} - \mu_t)^2}{2\sigma_t^2} \right),$$

where $\{x_t\}$ are the outcomes in the training examples, $\{\mu_t\}, \{\sigma_t\}$ are predicted parameters for the conditional distribution.

During prediction, the time series values in the prediction horizon are not available (because that's what we are trying to predict). Therefore, samples from the conditional distribution (whose parameters are predicted during the previous step) are used as the input values for the current time step.

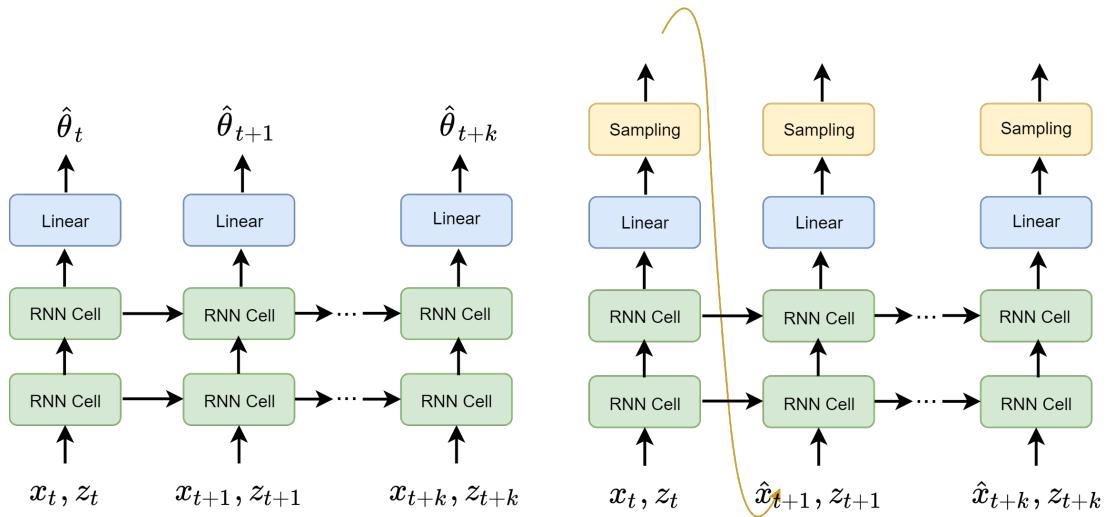


Figure 35.4.4: DeepAR model architecture for time series prediction. Outputs are the parameters characterizing the condition distribution of x_{t+1} conditioned on histories of x_t and z_t . (a) In the training phase, RNN are updated by minimizing the negative log likelihood function. (b) In the prediction phase, a predicted \hat{x}_{t+1} are sampled from predicted conditional distribution and then used to predict next step conditional distribution.

35.4.1.3 Deep factor model

In modeling multivariate stochastic time series, a linear factor model represents individual time series to a few common global factors via

$$x_{i,t} = \sum_{k=1}^K \beta_{ik} g_{k,t} + r_{i,t},$$

where $x_{i,t}, i = 1, \dots, N$ are the multivariate time series to model, $g_k, k = 1, \dots, K, K \ll N$ are the global factor time series, and r_i are the individual random effects time series following common zero mean stochastic process (e.g., Gaussian processes).

Clearly, global factor time series are playing critical roles in the following aspects:

- Capturing the dominant collective dynamics coexist among time series.
- Revealing latent structures among time series and offering model interpretation (i.e., explaining the dynamics via global factors).

Usually, the global factor time series are constructed as a linear combination of $\{x_{i,t}\}, \{z_i\}$ via PCA or reduced rank multivariate regression. However, these linear constructed global factors often fall short in facing complex time series modeling challenges.

One approach to construct more competent global factor time series is to harness RNN's capability of capturing complex pattern and structure among time series. The resulting model is known as **deep factor model** [10].

In the deep Factor model, each time series $x_i(t)$ is governed by a non-random global component and a random component, which can be written by

$$x_{i,t} = \sum_{k=1}^K \beta_{i,k} g_{k,t} + r_{i,t}$$

The global factors are given by

$$g_{k,t} = RNN_k(\{x_{i,t}, z_i\}),$$

where RNN_k denote a RNN taking input sequences of $\{x_i, z_i\}$ and producing one output. In the basic case, we can assume $r_{i,t} \sim N(0, \sigma_{i,t})$, where $\sigma_{i,t}$ is specified by neural networks or other deterministic functions. Beside being simple Gaussian processes, the random effect $r_{i,t}$ can further be modeled by a linear state space model to introduce additional linear structures among the noises, given by [10]

$$\begin{aligned} h_{i,t} &= F_{i,t} h_{i,t-1} + q_{i,t} \epsilon_{i,t}, \quad \epsilon_{i,t} \sim N(0, 1) \\ r_{i,t} &= a_{i,t}^\top h_{i,t} \end{aligned}$$

where h_i are latent states, F is the transition matrix for h_i , and a is emission coefficient matrix, and $q_{i,t}$ are strength for the random shocks.

35.4.2 MNIST classification with sequential observation

One interesting application of RNN is image classification based on sequential input of small image patches. Contrasting a CNN that directly takes the whole image to the

neural network, RNN can only take one stripe of a image at a time as input. For a 28 by 29 image, one image is then decomposed into 28 such stripes, reminiscent of a multi-dimensional time series of length 28. Different from time series modeling, the RNN architecture only employs the last hidden output for classification. The recurrent nature of the RNN ‘stores’ all previously observed image stripes in the last hidden output for classification.

The final neural network architecture is showed in [Figure 35.4.5](#). There are 128 hidden units in the LSTM unit. After moderate training, the classification accuracy can achieve 97%, comparable to a simple CNN.

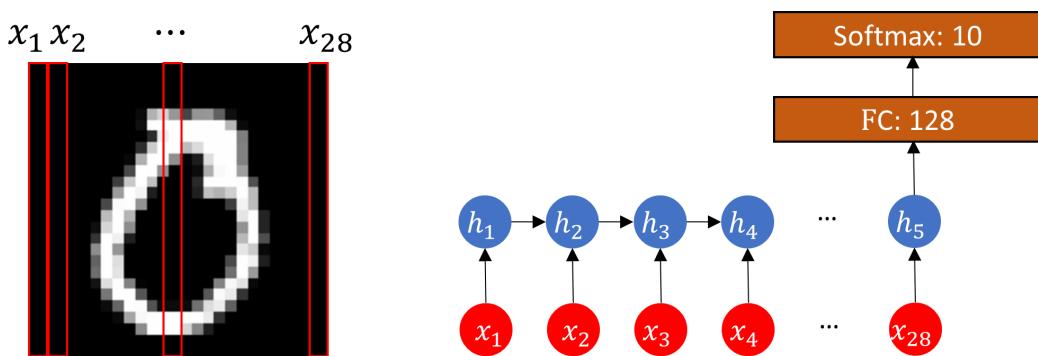


Figure 35.4.5: A RNN architecture for MNIST image recognition.

35.4.3 Character-level language modeling

35.4.3.1 Word classification

In sentiment classification, we take a sequence of words as the input and output a sentiment label. In word classification, we take a sequence of characters and output a label characterizing the word.

An example application is to classify names origin via RNN in [Figure 35.4.6](#). The overall architecture contains the following components:

- Characters are fed into the LSTM layer one by one.
- The last output from the LSTM will go to a Softmax output layer to produce classification probabilities on the language origin (Chinese, American, Japanese, etc.).

Note that we usually do not need embedding layer since the one-hot encoding on character level is already of low dimensionality.

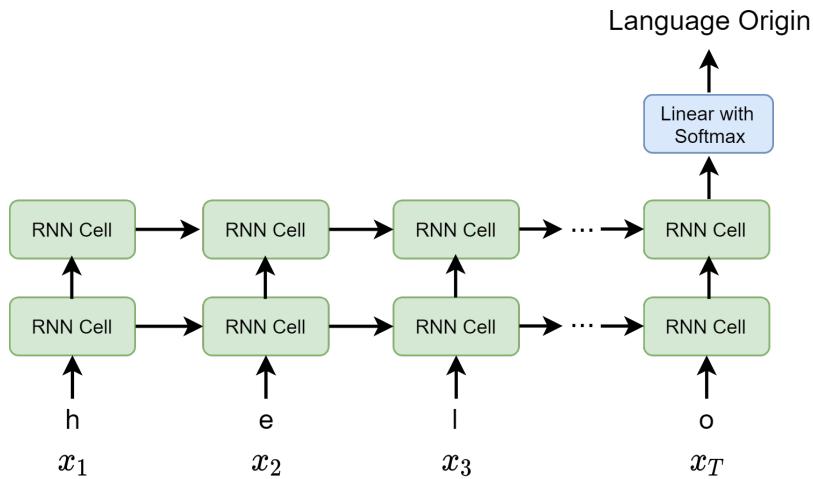


Figure 35.4.6: A RNN for character-level word classification.

35.4.3.2 Text generation

RNN has been used for advanced language modeling tasks, like machine translation, caption generation, and chatting robot. Before diving into these advanced applications, we cover one very basic language modeling task in this section, known as character-level language modeling. The basic goal of character-level language modeling is to predict or generate a sequence of sensible characters based on preceding characters.

One concrete application example of character-level language model is when we write an email in Gmail and type messages on a phone, we observe Gmail and our phones will suggest next character/word. Alternatively, a simple word completion program can also be made by n-gram statistical model, which basically counts n-gram frequencies. However, with RNN designed to efficiently capture the long term relation among characters, we expect RNN language modeling can do far more than a empirical n-gram statistical model.

The first step towards the language modeling task is to train a RNN that is able to predict the next character based on an input character sequence [Figure 35.4.7 (left)]. A naive n-gram model would required K^n memory, where K is the vocabulary size (say $K = 26$ for alphabetic letters). From this perspective, RNN can be viewed as an efficient parametric n-gram model.

The overall RNN architecture contains the following components:

- Characters from a training text corpus (e.g., a English novel) are fed into the LSTM layer one by one.

- Each output from LSTM will go to a Softmax output layer to produce prediction probabilities on the next letter (26 classes).
- Network parameters are updated by taking gradient descent based on a cross-entropy loss function applied to each prediction.

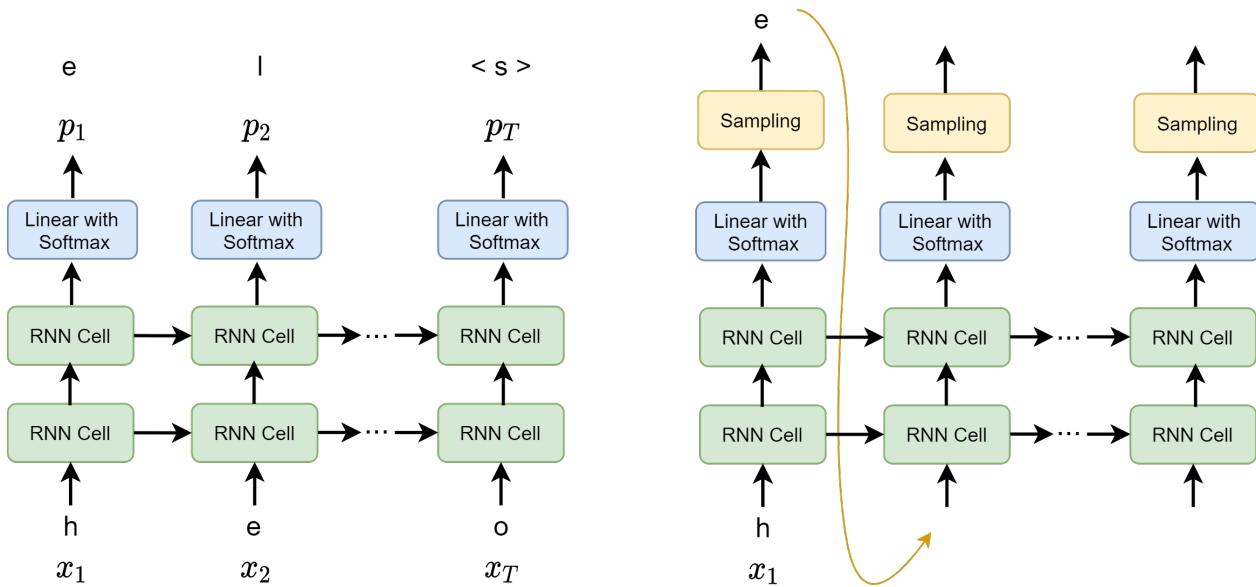


Figure 35.4.7: A RNN for character-level language model. (left) During the training session, one-hot coded characters are directly fed into RNN and predict the next character in the word. (right) A RNN for character-level language model. During the word generation session, the network starts with a user input character and continues the generation process with predicted character from the previous step.

The stage of text generation is to use the trained RNN to sequentially predict new character based on previous predicted character.

The text generation scheme and architecture are showed in Figure 35.4.7 (right). The actual predicted character requires sampling from the prediction probability output by the softmax layer.

There are different sampling strategies from predicted probabilities to promote the total probability of a full generated word. On one extreme, a greedy sampling strategy will take character with the largest probability. But such an approach usually results in repetitive, deterministic patterns that don't look like coherent language. A more general

and flexible strategy is to introduce a temperature controlled sampling strategy, where the sampling probability for character i is given by

$$p_i^T \propto \exp\left(\frac{p_i^S}{T}\right),$$

where p_i^S is the sampling probability from softmax output. As $T \rightarrow \infty$, we get uniform sampling, and as $T \rightarrow 0$, we have greedy sampling. Compared to greedy sampling, probabilistic sampling can generate interesting language patterns that resemble human language.

BIBLIOGRAPHY

1. Haykin, S. *Neural Networks and Learning Machines*, 3/E (Pearson Education India, 2010).
2. Qiu, X. *Neural Network and Deep Learning* (Fudan University, 2019).
3. Pascanu, R., Mikolov, T. & Bengio, Y. *On the difficulty of training recurrent neural networks* in *International conference on machine learning* (2013), 1310–1318.
4. Hochreiter, S. & Schmidhuber, J. Long short-term memory. *Neural computation* **9**, 1735–1780 (1997).
5. Chung, J., Gulcehre, C., Cho, K. & Bengio, Y. Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv preprint arXiv:1412.3555* (2014).
6. Zaremba, W., Sutskever, I. & Vinyals, O. Recurrent neural network regularization. *arXiv preprint arXiv:1409.2329* (2014).
7. Gal, Y. & Ghahramani, Z. A theoretically grounded application of dropout in recurrent neural networks in *Advances in neural information processing systems* (2016), 1019–1027.
8. Sak, H., Senior, A. W. & Beaufays, F. Long short-term memory recurrent neural network architectures for large scale acoustic modeling (2014).
9. Salinas, D., Flunkert, V., Gasthaus, J. & Januschowski, T. DeepAR: Probabilistic forecasting with autoregressive recurrent networks. *International Journal of Forecasting* (2019).
10. Wang, Y. *et al.* Deep factors for forecasting. *arXiv preprint arXiv:1905.12417* (2019).