30

NEURAL NETWORK AND DEEP LEARNING

30.1 Neural network foundations

30.1.1 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.

30.1.2 Neurons and neural networks

30.1.2.1 *Artificial neurons*

Artificial neuron [Figure 30.1.1] 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 $x = (x_1, ..., x_d) \in \mathbb{R}^d$. 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 $g : \mathbb{R} \to \mathbb{R}$ 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.

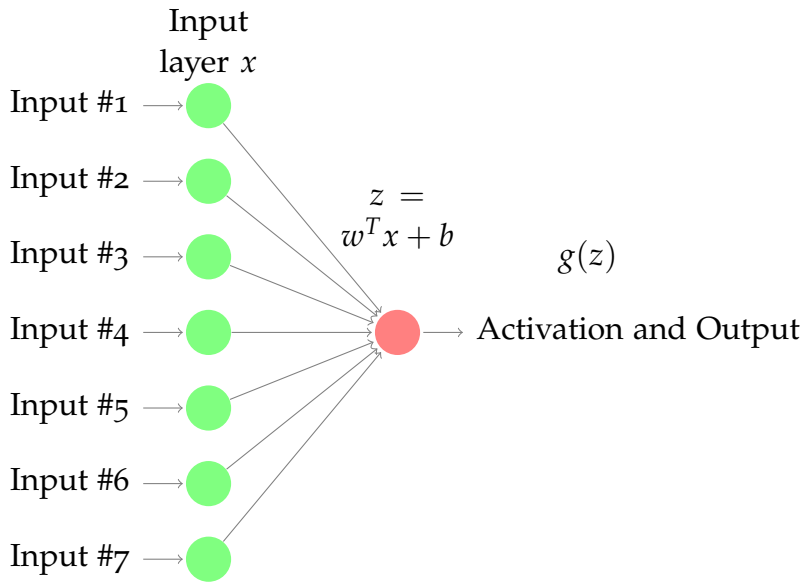


Figure 30.1.1: Scheme of an artificial neuron.

Common activation functions include:

Definition 30.1.1 (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)*

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

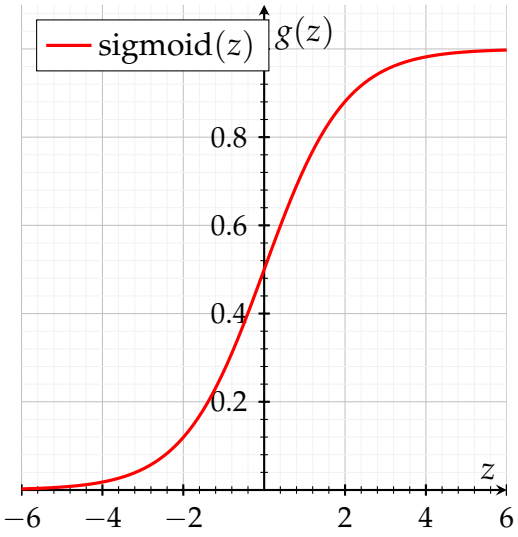
- *(ReLU activation function)*

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

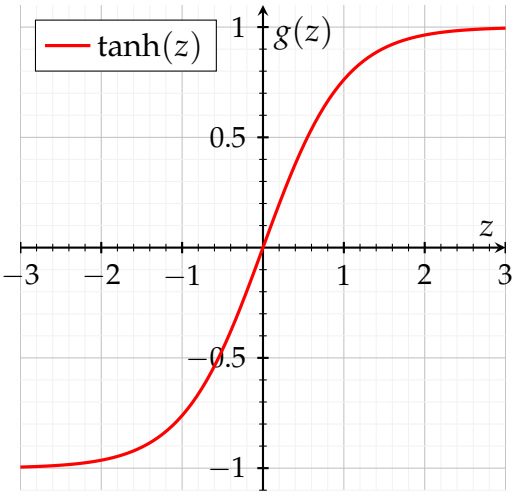
- *(leaky ReLU activation function)*

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

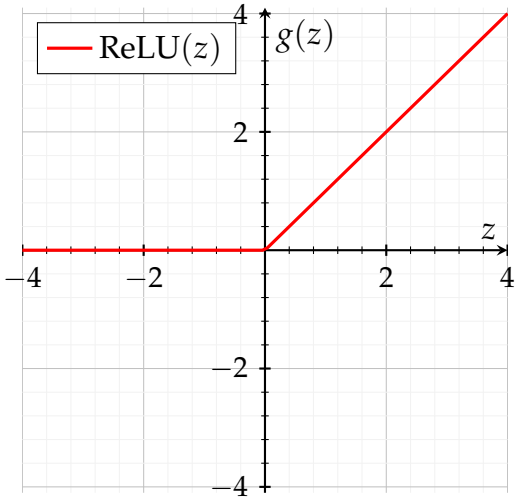
where γ is the leaky parameter usually taking small values like 0.01.



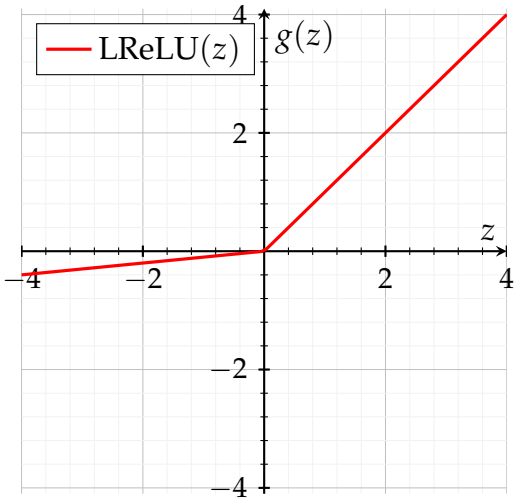
(a) Sigmoid activation function.



(b) tanh activation function.



(c) ReLU activation function.



(d) Leaky ReLU activation function.

Figure 30.1.2: Common activation functions in artificial neural networks.

30.1.2.2 *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 30.1.3.

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 30.1.4), and more output units in the output layer [Figure 30.1.5].

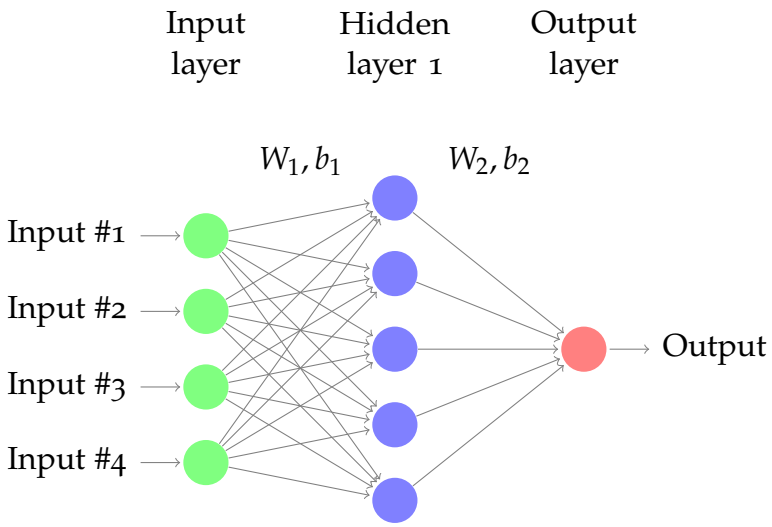


Figure 30.1.3: Scheme for an artificial neural network.

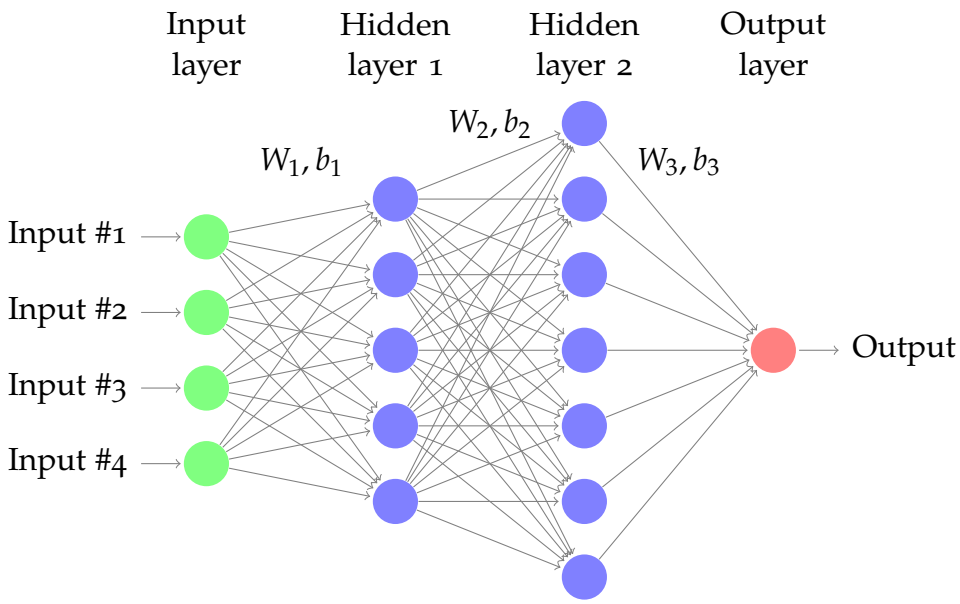


Figure 30.1.4: A four-layer feed-forward neural network.

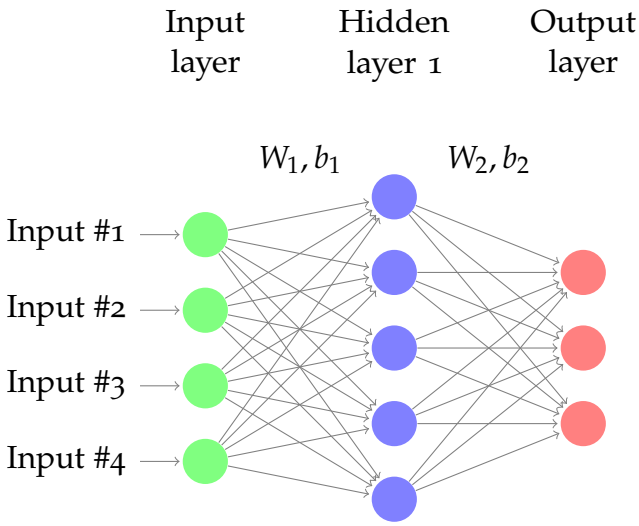


Figure 30.1.5: A three-layer feed-forward neural network with three output units.

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 30.1.2 (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 $\mathbb{R}^H \to [0,1]^C$. The parameter associated with $g(x)$ is a weighting matrix $W = [w_1, w_2, ..., w_C] \in \mathbb{R}^{H \times C}$ such that*

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

30.1.3 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 30.1.1]. 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 30.1.1 (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 $i = 1, 2, ..., N$, 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 $C(I_m)$.

30.1.4 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 30.1.3 (forward process). *Consider a four-layer feed-forward neural network(Figure 30.1.6). 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 $y = g(x_3)$.

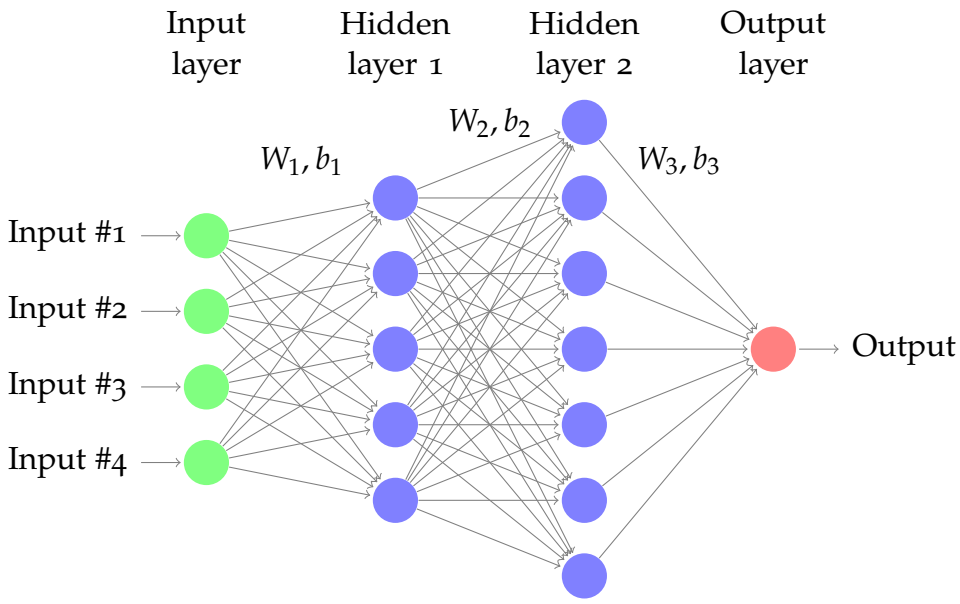


Figure 30.1.6: A four-layer feed-forward neural network.

Lemma 30.1.1 (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 $\delta_3 = g_4'(x_3 - t) \odot g_3'(W_3 x_2 + b_3)$.
- $$\frac{\partial E}{\partial W_2} = \delta_2 x_1^T, \frac{\partial E}{\partial b_3} = \delta_2,$$

  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 30.1.1 (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

$$\frac{\partial E}{\partial W_2} = 0, \frac{\partial E}{\partial b_3} = 0.$$

- $\delta_1 = 0$ and thus

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

Theorem 30.1.2 (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 $W_1, b_1, W_2, b_2, ..., W_L, b_L$, and activation functions $g_1, g_2, ..., g_L$.

Denote the forward feeding result as $x_1, x_2, ..., x_L$.

Consider the loss function on the output x_3 given by

$$E = f(x_3).$$

Then it follows that

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

where

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

[a] also see link

Remark 30.1.2 (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. 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 30.1.2] 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 subsubsection 30.5.2.4.

30.2 Optimization algorithms

30.2.1 Motivation

Backpropogation 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 30.2.1] 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.

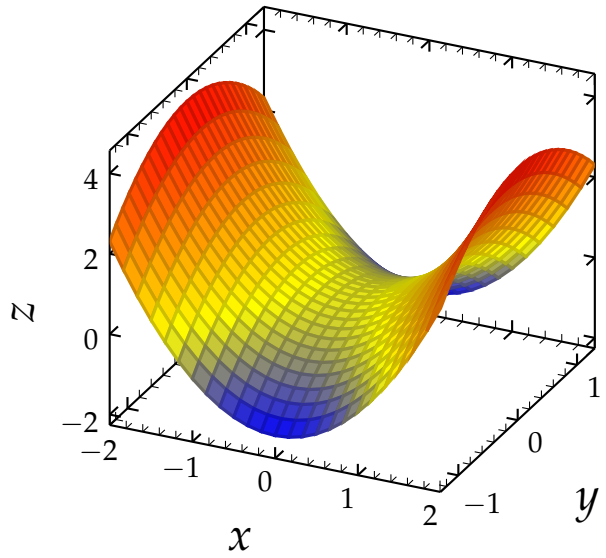


Figure 30.2.1: An example saddle point at $(0, 0, 0)$, which locally minimizes the x direction but maximizes the y direction..

30.2.2 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

Set $k = 1$

repeat

  Compute gradient estimate over N samples via

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

  Apply update $\theta_k = \theta_k - \alpha_k \hat{g}_k$.

  Set $k = k + 1$.

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 $\alpha_k = \alpha_0 \exp(-\beta k), \beta > 0$.

30.2.3 Minibatch stochastic gradient descent

Batch gradient descent is usually not ideal for neural network application that involves large-size training data, such as image recognization 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, iniital model parameter θ.

Set $k = 1$

repeat

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

  Compute a gradient estimate over this minibatch samples via

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

  Apply update $\theta_k = \theta_k - \epsilon_k \hat{g}_k$.

  Set $k = k + 1$.

until *stopping criteria is met*;

Output: θ_k

Remark 30.2.1 (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 [5].

30.2.4 Adaptive gradient method

30.2.4.1 *Adaptive gradient (AdaGrad)*

For simple stochastic gradient methods, we need to set the learning rate hyperparameter or even dynamically schedule learning rate, which are usually difficult tasks and 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[6] 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

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

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.

30.2.4.2 *RMSProp & AdaDelta*

As we mentioned before, AdaGrad tend 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[7, 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

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

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

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

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**[8].

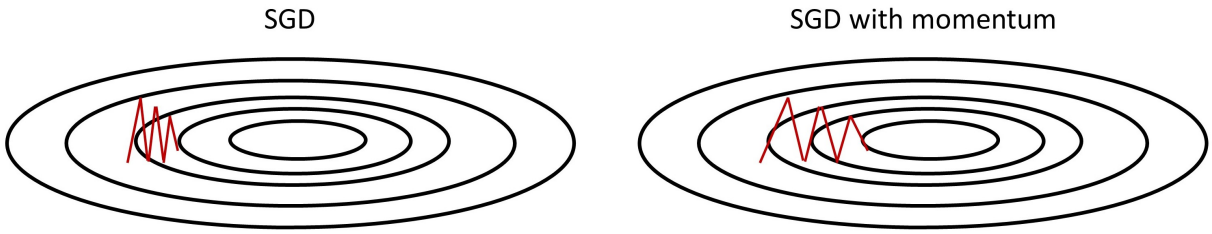


Figure 30.2.2: SGD without momentum and with momentum. SGD with momentum can accumulate gradient/velocity in horizontal direction and move faster towards the minimum located at the center.

30.2.5 Momentum method

Simple SGD with small learning rate can lead to extremely slow learning for functional surfaces with long, narrow valleys [Figure 30.2.2, [9]]. 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 $\theta_k = \theta_k + v_k$.

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

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

Another empircially 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 $\theta_k = \theta_k + v_k$.

30.2.6 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[10]. 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 30.2.4.2 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), iniital model parameter θ, decay parameter $\delta = 1e-8$ and decay rates ρ_1(set to 0.9), ρ_2(set to 0.999).

Set $k = 1$.

Set $M_k = 0, G_k = 0$.

repeat

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

compute gradient estimate over minibatch N samples via

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

Accumulate $M_k = \rho_1 M_{k-1} + (1-\rho_1)\hat{g}_k$. Accumulate $G_k = \rho_2 G_{k-1} + (1-\rho_2)\hat{g}_k \odot \hat{g}_k$.

Correct biases

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

Apply update

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

.

Set $k = k + 1$.

until *stopping criteria is met*;

Output: θ_k

30.3 Training and regularization techniques

30.3.1 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 30.1.2].

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 [section 30.10].

30.3.2 Weight initialization

30.3.2.1 *Motivation*

The goal of weight initialization is to ensure the output of each layer before activation to be neither too small nor too large[11] 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 30.3.1 (zero initialization). When we set all weights and biases to 0, all derivatives except for the last layer's bias will be zero [Remark 30.1.1]. And this will stay the same as we continue the iterations.

30.3.2.2 *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

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

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

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

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

Usually in practice, we require

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

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

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

This is known as **Xvaier** initialization[11].

30.3.2.3 *He initialization*

Xvaier 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

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

By the similar variance argument, we have

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

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[12]

30.3.3 Data normalization

30.3.3.1 *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 sub-subsection 22.5.1.1. Through standardization, input data have similar ranges and scales across different features.

30.3.3.2 *Batch normalization*

Batch normalization is mainly motivated by the need to remedy the negative impact from **internal covariance shift**[13]. 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 sweat spot. A popular solution to internal covariant shift is to normalize the input to each layer, in additional to the initial data standardization.

Practically, we add a 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.

30.3.4 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.

30.3.4.1 *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

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

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.

30.3.4.2 *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

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

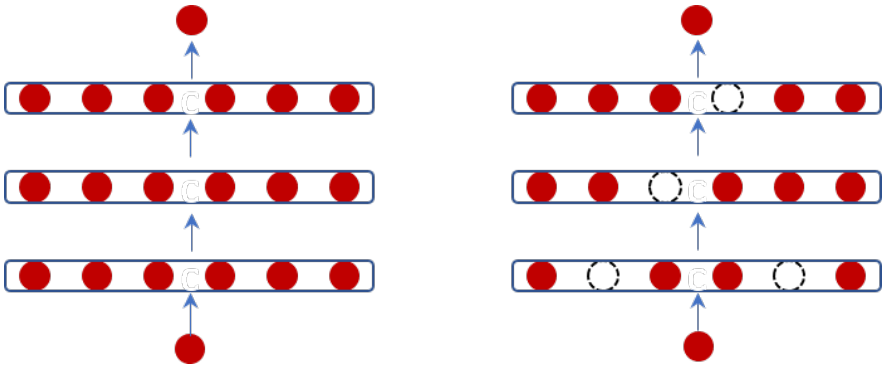


Figure 30.3.1: Dropout technique for a simple feedforward neural networks. The original network (left) and the neural network after some neurons being dropped out (right).

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[14], weight decay and L_2 regularization are not equivalent.

30.3.4.3 *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.

30.3.4.4 *Dropout*

Dropout [15] 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 30.3.1]. 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: 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 [16].

30.3.4.5 *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.

30.3.4.6 *Label smoothing*

Label smoothing[17] is a regularization technique for classification problems. Consider a ground truth one-hot encoded outcome given by

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

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

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

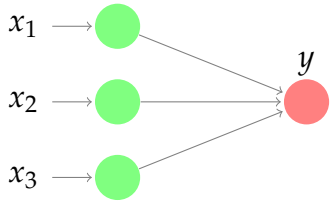
During training, instead of training the network with respect to one-hot predictions, we can train the network with respect to smooth labels to reduce overfitting. The rationale for label smoothing is: One-hot encoding may result in over-fitting. If the model learns to assign full probability to the 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.

30.4 Feed-forward neural network examples

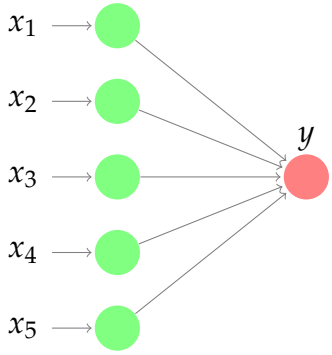
30.4.1 Linear regression and classification

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 30.4.1]. 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 logit or softmax of the output.

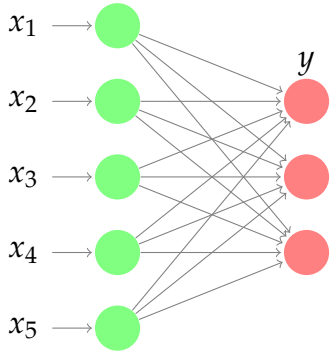
(a) Linear regression or binary classification with one feature.



(b) Linear regression or binary classification with three features.



(c) Linear regression or binary classification with five features.



(d) Multi-output linear regression or multi-class linear classification with five features.

Figure 30.4.1: Neural network architecture for polynomial regression.

Example 30.4.1 (polynomial regression). Polynomial regression can be viewed as linear regression of higher degree features $x, x^2, .., x^d$. Figure 30.4.2 shows polynomial

regression of different degrees using architecture in Figure 30.4.1. 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 $d > 3$.

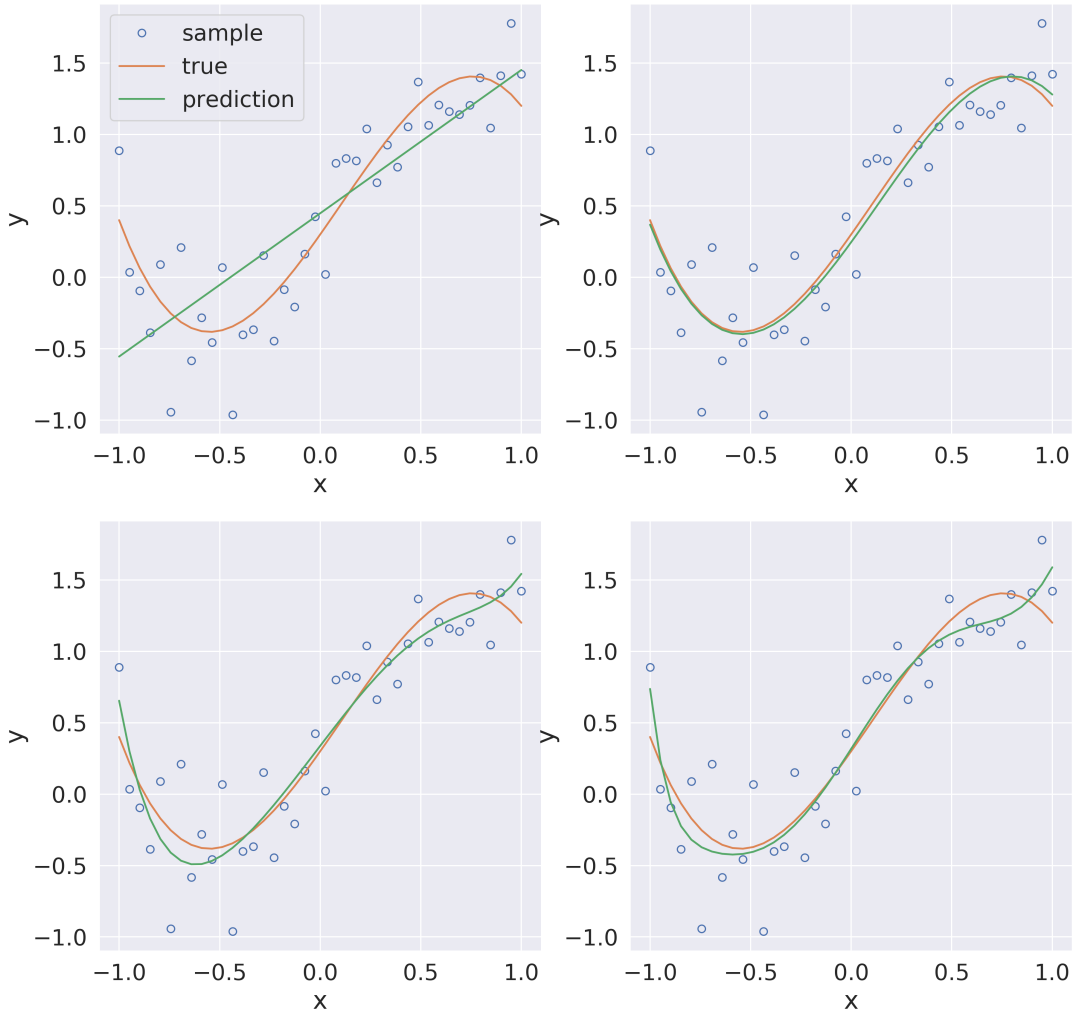


Figure 30.4.2: Polynomial regression with degree d = 1, 3, 6, 10.

Example 30.4.2 (Linear classification regression). Figure 30.4.3 shows the linear classification model realized via the architecture in Figure 30.4.1. Specifically, there are 784 input neurons and 10 output neurons. After training, visualized the learned weighted matrix of size $(784, 10)$ clearly indicates the neural network capture the essential features in distinguish different numbers.

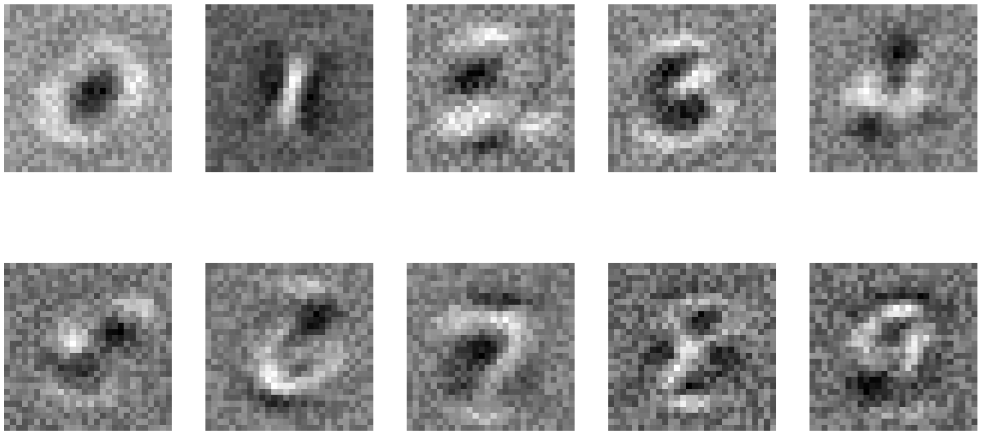


Figure 30.4.3: Visualization of first layer weight for a one-layer linear multi-class classification neural network

30.4.2 Image classification

The Fashion MNIST data set contains tiny 28 by 28 images of different types of clothing [Figure 30.4.4], 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.

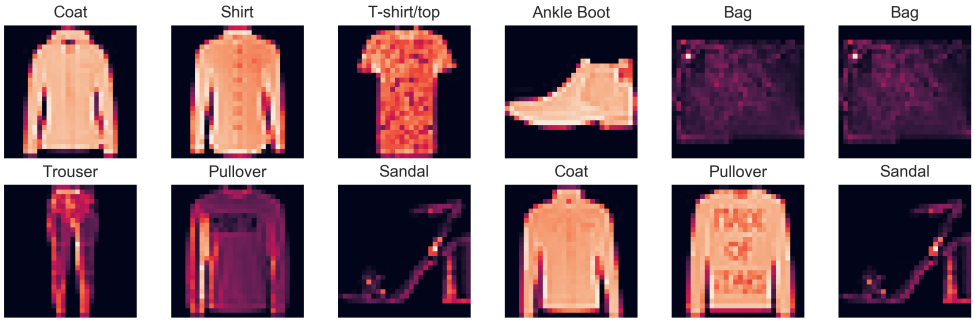


Figure 30.4.4: Example images from the Fashion MNIST dataset.

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

After several epochs of training, we can arrive an accuracy of around 86%. Classification results are summarized in the confusion matrix [Figure 30.4.5] and exemplified by a set of randomly selected samples [Figure 30.4.6]. 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.

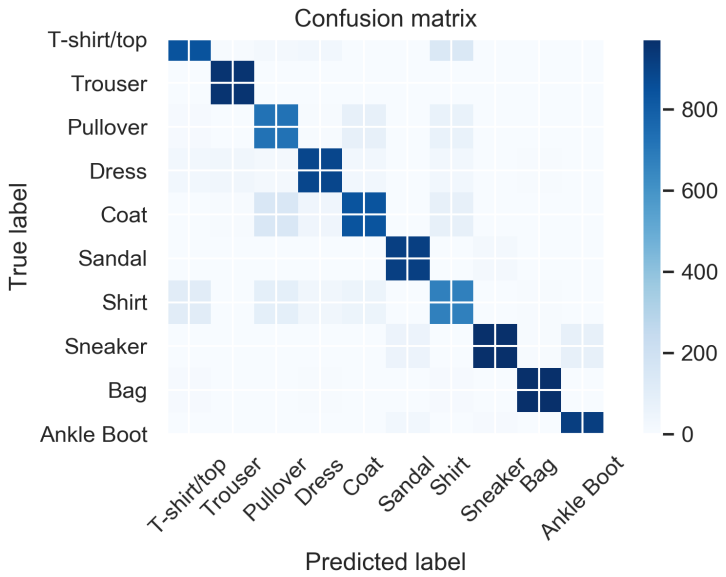


Figure 30.4.5: The confusion matrix from fashion MNIST classification results.

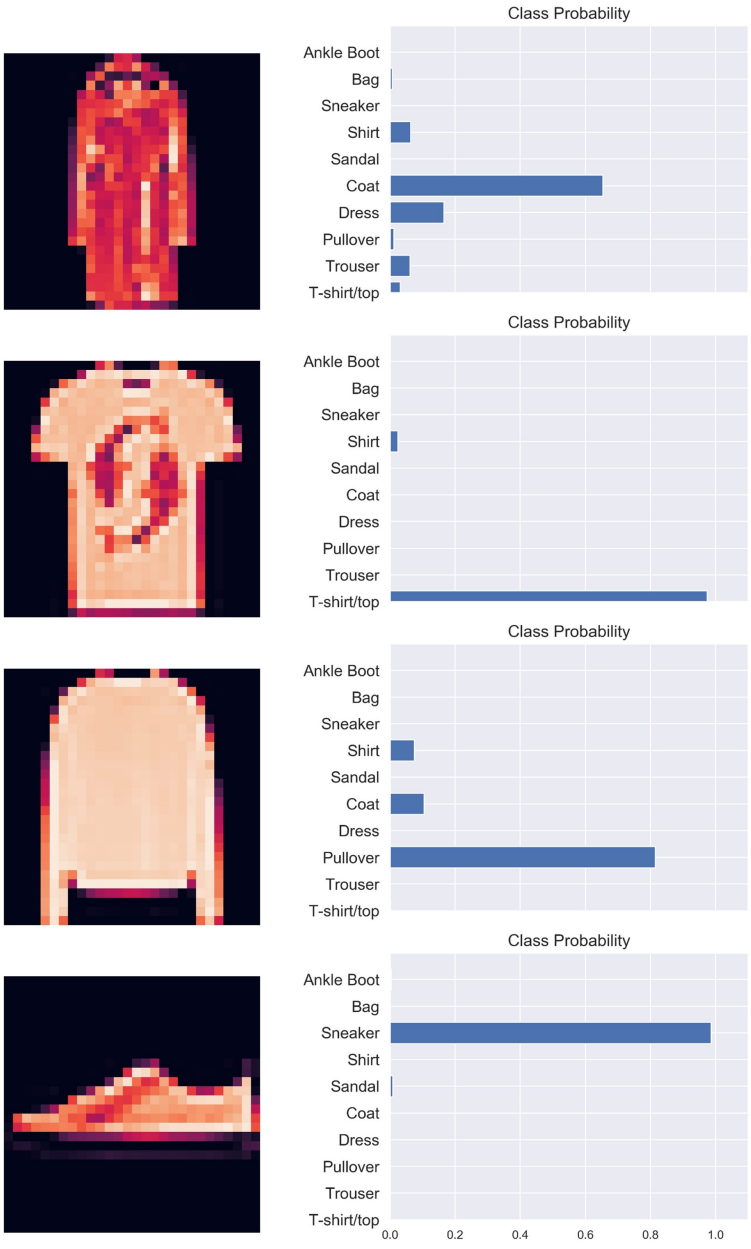


Figure 30.4.6: Classification results for a set of randomly selected samples.

30.4.3 Word embedding

In many text data related tasks, such as sentiment classification, we need to represent text data by numeric values. One naive way to represent the feature of a word is the one-hot word vector, whose length of the typical size of a dictionary. Such sparse representation

does not capture the relations among words (i.e., meanings, lexical semantic) and are thus not considered as good features for advanced natural language processing tasks, such as language modeling. A much better alternative is to embed each word vector into a smaller dense vector (typical dimensionality ranges from 25 to 1,000) that encodes semantic meaning [Figure 30.4.7].

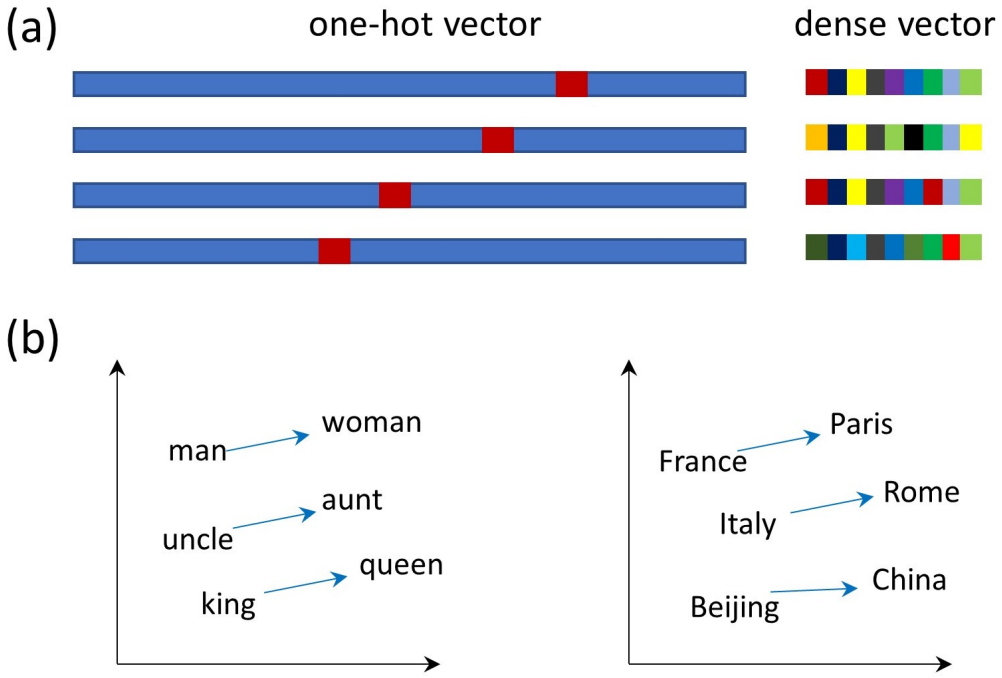


Figure 30.4.7: (a) Embedding layer maps large, sparse one-hot vectors to short, dense vectors. (b) Example of low dimensional embeddings that capture semantic meanings.

In subsection 29.2.3, we introduce a SVD based matrix decomposition method to map one-hot word vector to semantic meaning preserving dense word vector. This section, we introduce a neural networ based method. The two classical methods, called continuous bags of words (**CBOW**)[18] and **Skip-gram**[19]. Both methods employ a three-layer neural networks [Figure 30.4.8], taking a one-hot vector as input and predict the probability of its nearby words. In CBOW, the inputs are surrounding words within a context window of size c and the goal is to predict the central word (same as multi-class classification problems); in Skip-gram, the input is the single central word and the goal is to predict its surrounding words within a context window.

Denote a sequence of words $w_1, w_2, ..., w_T$ in a text, the objective of a Skip-gram model is to maximize the likelihood of observing the occurrence of its surrounding words within a context window of size c, given by

$$\frac{1}{T}\sum_{t=1}^{T}\sum_{-c\le j\le c, j\neq 0}\log p\left(w_{t+j}|w_t\right)$$

where we have assumed conditional independence given word w_t.

In the neural networks of Skip-gram and CBOW, we use Softmax in the output layer for classification probability, given by

$$p(w_k|w_j) = \frac{\exp(v'_k \cdot v_j)}{\sum_{i=1}^{V}\exp(v'_i \cdot v_j)},$$

where V is the size of the vocabulary, v_i is the column i of the input matrix W, and v'_i is the row i in the output matrix W'.

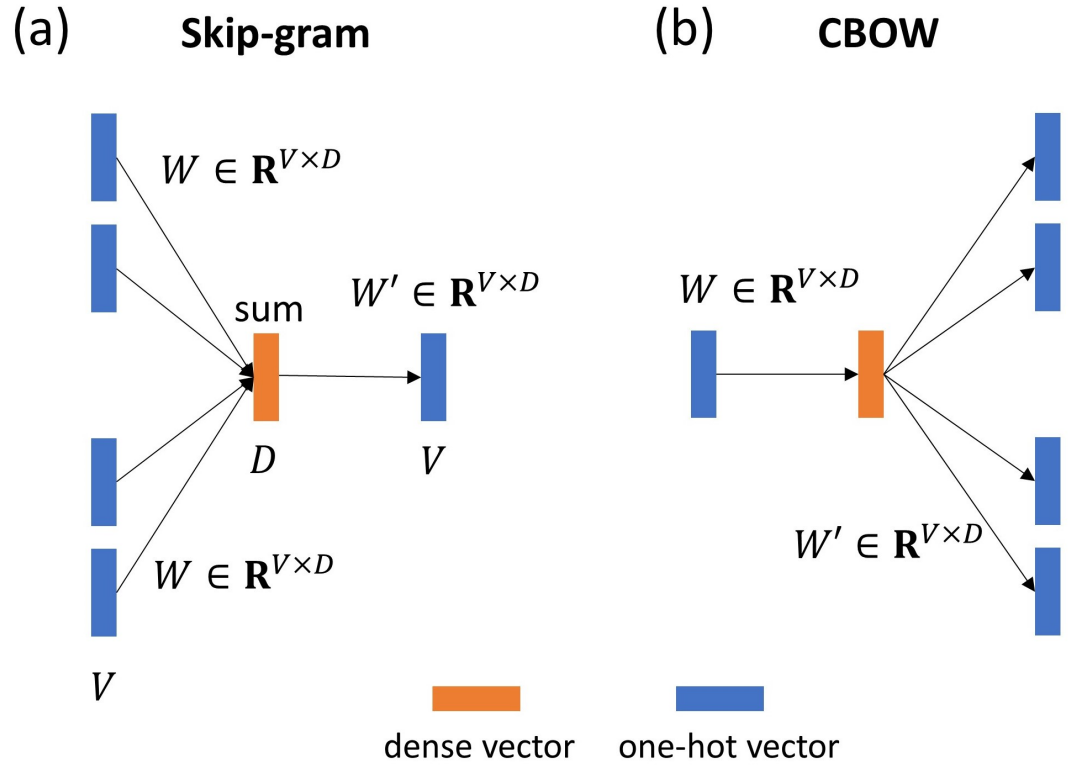


Figure 30.4.8: (a) The Skip-gram architecture that predicts surrounding words given the central word. (b) The CBOW architecture that predicts the central word given its surrounding context words. The one-hot vector has size V; the dense vector has length $D \ll V$. Also note that no nonlinearity activation is applied between input and hidden layer.

The neural network weights of the Skip-gram model are optimized to maximize the observation of a text consisting of words $w_1, w_2, ..., w_T$, which can then be written by

$$\begin{aligned}
&\arg\max_{v,v'} \sum_{t=1}^{T} \sum_{-c\le j\le c, j\neq 0} \log p\left(w_{t+j}|w_t\right) \\
=&\arg\max_{v,v'} \sum_{t=1}^{T} \sum_{-c\le j\le c, j\neq 0} \log \frac{\exp\left(v'_{t+j}\cdot v_{t)}\right)}{\sum_{w\in V}\exp\left(v'_w\cdot v_t\right)} \\
=&\arg\max_{v,v'} \sum_{t=1}^{T} \sum_{-c\le j\le c, j\neq 0} \left[v'_{t+j}\cdot v_t - \log\sum_{w\in V}\exp\left(v'_w\cdot v_t\right)\right]
\end{aligned}$$

In the CBOW model, the optimization problem becomes

$$\begin{aligned}
&\arg\max_{v,v'} \sum_{t=1}^{T} \log p\left(w_t|w_{t-c},...,w_{t+c}\right) \\
&\quad = \arg\max_{v,v'} \sum_{t=1}^{T} \log \frac{\exp(v'_t\cdot\sum_{-c\le j\le c, j\neq 0} v_j)}{\sum_{i=1}^{V}\exp(v'_i\cdot\sum_{-c\le j\le c, j\neq 0} v_j)}
\end{aligned}$$

In both the Skip-gram and the CBOW model, each word will have two embeddings, v_i and v'_i in the input matrix W and the output matrix W', respectively. Notable, the embedding v'_i corresponds to the dense word vector that produces one-hot probability vector in the output. For these two embeddings, we can use one of them, a mixed version of them, and a concatenated one.

With the trained embeddings for each word, we can assemble then into a matrix of size $D \times V$, which is also called an Embedding layer. In applications, the one-hot word vector is fed into the embedding layer and produce the corresponding dense word vectors. From the computational perspective, we do not need to perform matrix multiplication; instead, we can view the Embedding layer as a dictionary that maps integer indices of the word to dense vectors.

In Skip-gram, the weight associated with each word receives adjustment signal (via gradient descent) from its surrounding context words. In CBOW, a central word provides signal to optimize the weights of its multiple surrounding words. Skip-gram is more computational expensive than CBOW as the Skip-gram model has to make predictions of size $O(cV)$ while CBOW makes prediction on the scale of $O(V)$. Further, because of the averaging effect from input layer to hidden layer in CBOW, CBOW is less competent in calculating effective word embedding for rare words than Skip-gram.

30.4.4 Sentiment analysis

We now consider a sentiment analysis task where we are given a movie review text and we need determine if the review's sentiment is positive or negative.

Some negative example reviews are:

- this was an impulse pick up for me from the local video store . don t make the same mistake i did . this movie is **tedious unconvincingly acted** and generally **boring**. the dialogue between the young ...
- yes there are great performances here . unfortunately they happen in the context of a movie that doesn t seem to have a clue what it s doing . during the first minutes of this all the music ta...
- i saw this movie with some indian friends on christmas day . the quick summary of this **movie is must avoid** . jp dutta wrote directed produced and edited this movie and did none of these jobs well.
- this movie is **extremely boring** it tells a story of a female gas station owner and her life.**nothing exciting** ever happens . the director has really kept it real and it feels just like a camera fol...

And some positive example reviews are

- his scary and rather gory adaptation of stephen king s **great** novel features **outstandin**g central performances by dale midkiff fred gwynne who sadly died few years ago and denise crosby and some ...
- okay i ll confess . this is the movie that made me love what michael keaton could do . he does a beautiful parody of someone doing a parody of james cagney with **charm** to spare.
- directors had the same task tell stories of love set in paris . naturally some of them turned out better than others but the whole mosaic is **pretty charming** besides wouldn t it be boring if

Clearly, the existence of words like **tedious, boring** will likely be found in a negative review; similarly, the existence of words like **outstanding, charm** will likely be found in a positive review. Our goal is to define a neural network that can automatically establish such relation.

An initial analysis is to identify these words that are hallmarks of positive and negative reviews. Our method is to first count the word frequency for positive reviews and negative reviews, respectively. The most frequent words in both positive and negative review are actually neural stop words like *the, I,* etc..

Let w_i^P and w_i^N be the frequency of word i shown in positive and negative reviews respectively. We instead of show the words i with highest ratio of w_i^P / w_i^N and lowest ratio

of w_i^P / w_i^N with at least 100 counts. They are expected to be most distinguishing words that determine the sentiment of review.

The result in the following table agrees with our expectation that words like *superb* and *worst* are the key words to determine the sentiment.

Positive	**Negative**
superb	worst
wonderful	awful
fantastic	horrible
excellent	crap
amazing	worse
powerful	terrible
favorite	mess
perfect	stupid
brilliant	dull

Clearly, only words with ratio away from 1 are useful features to distinguish the sentiment of a review. The first step of feature engineering is to keep only words that satisfy ratio requirement. These words are then fed to embedding layer and sum up to dense vector, as a bagged feature embedding vector [Figure 30.4.9]. The bagged embedding is then fed into a feed-forward module to predict labels.

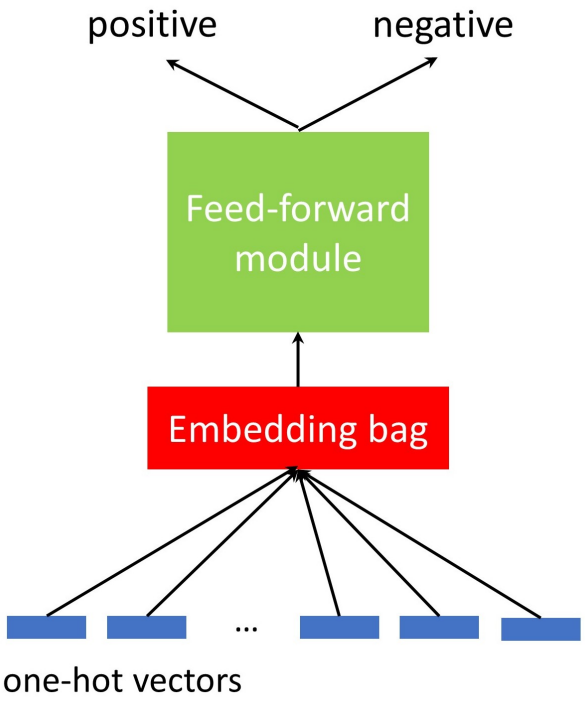


Figure 30.4.9: A feed-forward neural network architecture for sentiment analysis.

30.4.5 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[20–22].

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[23].

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

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

where X_t is a m-dimensional stochastic process, W_t is a p-dimensional Brownian motion, $\sigma(t, X_t, Y_t) \in \mathbb{R}^{m \times p}$, $\mu\left(t, X_t, Y_t, Z_t\right), \varphi\left(t, X_t, Y_t, Z_t\right)$ 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 [23]

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

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

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

Now we give some examples with such formulation.

Example 30.4.3 (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.

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

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

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

where we have discretized the time space $[0, T]$ into N periods $t^0, ..., t^{N-1}$, 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

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

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

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

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

We can summarize the iterative procedure in algorithm 65.

Algorithm 65: Solve forward backward stochastic differential equation via deep learning.

Input: Iniital model parameter θ for u_θ.

Set $n = 1$.

repeat

Generate M trajectories based on

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

Perform gradient descent to update u_θ using loss function defined by

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

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

Set $n = n + 1$.

until *stopping criteria is met*;

Output: u_θ

30.5 Convolutional neural networks (CNN)

30.5.1 Foundations

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 30.5.1], 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.

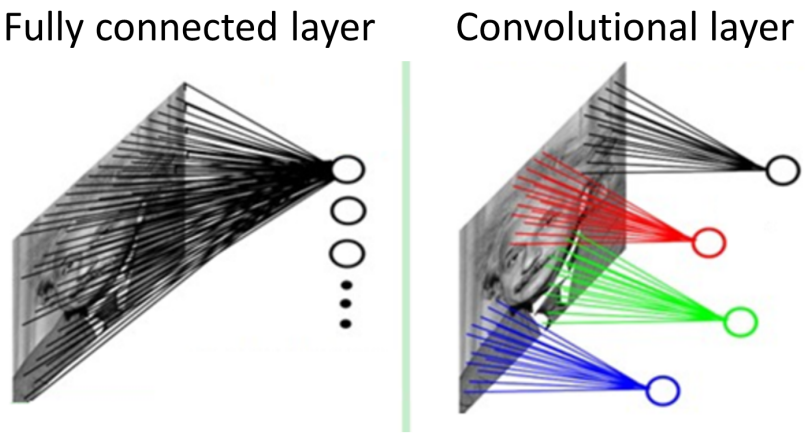


Figure 30.5.1: Comparison of receptive fields in fully-connected layer and local-connected layer in CNN. Credit

.

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 30.5.2.

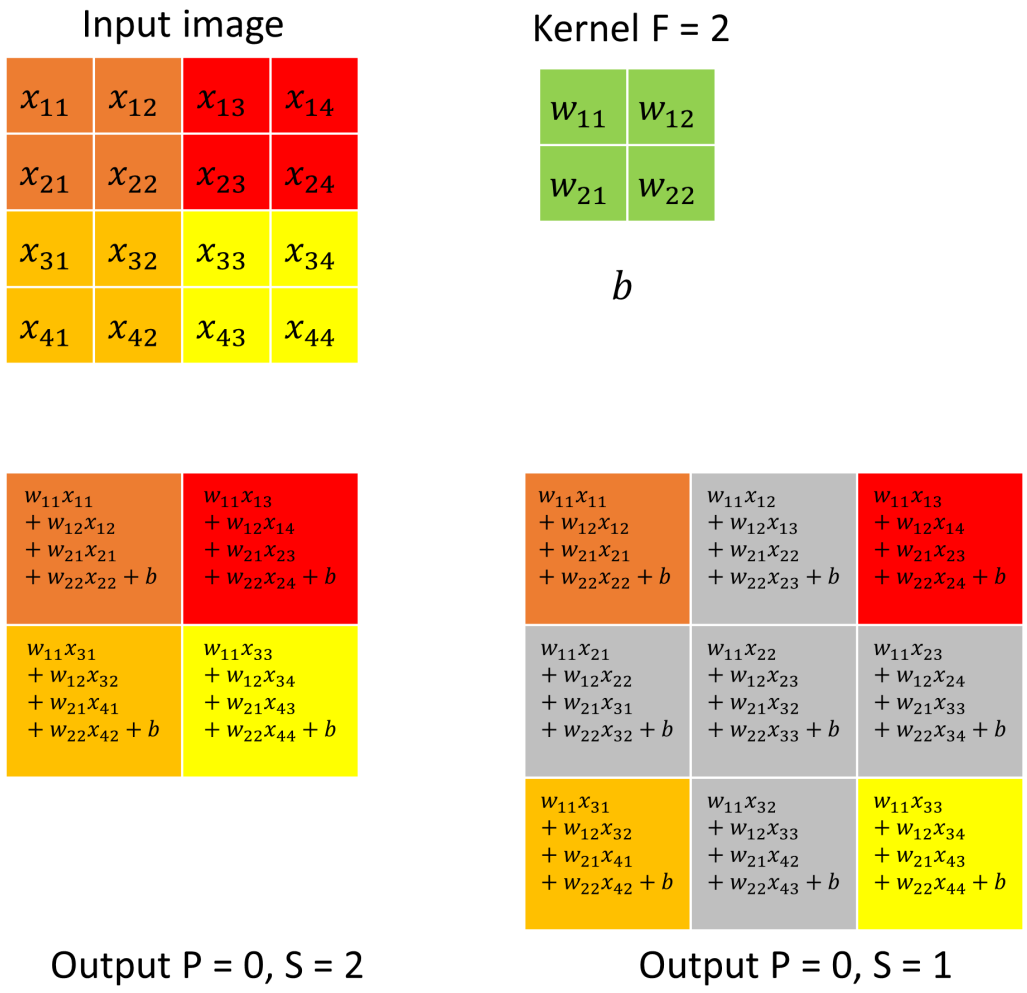


Figure 30.5.2: Demo for one kernel 'convoluting' with an input image.

More generally, a convolutional layer has parameters of strike S, kernel size F, padding P, input channel number C_I and output channel number C_O. Given input with size (C_I, H_I, W_I) the output with sizes

$$H_O = \frac{H_I + 2P - F}{S} + 1$$
$$W_O = \frac{W_I + 2P - F}{S} + 1$$

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 $X^1, ..., X^{C_I}$ be the C_I slices of the input data and let $Z^1, ..., Z^{C_O}$ be the C_O slices of the linear aggregation results. The linear aggregation process can be represented by

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

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

The pooling feature in CNN 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 30.5.3]. 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

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

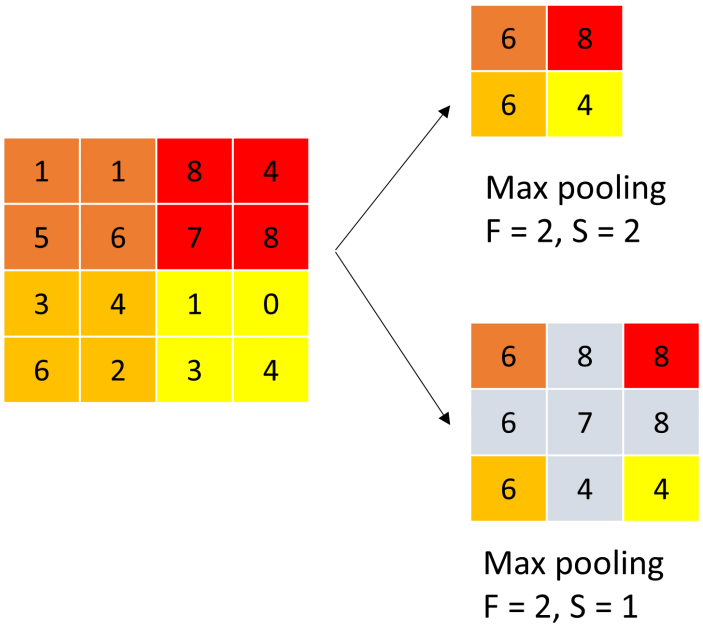


Figure 30.5.3: Pooling layer demo.

Finally, a typical CNN architecture for image classification will be something like Figure 30.5.4.

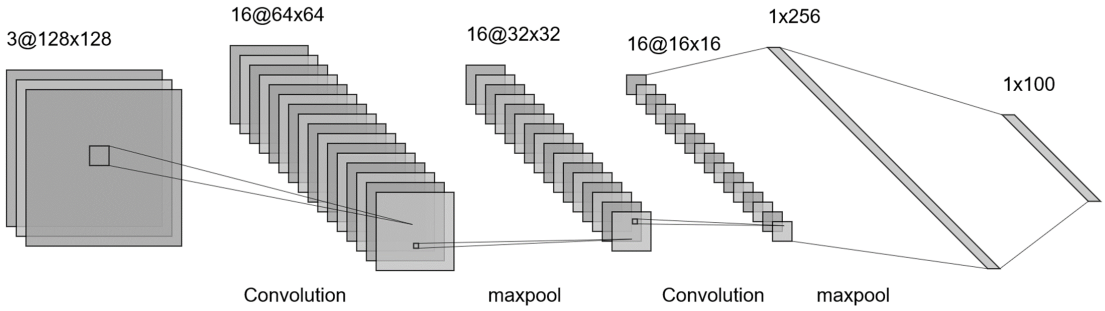


Figure 30.5.4: A typical CNN architecture for image classification tasks.

30.5.2 CNN classical architectures

30.5.2.1 *LeNet*

LeNet is the earliest CNN that applied to image recognition in real-world tasks, mainly recognizing hand-written digits[24]. LeNet introduce the concept of convolution and pooling, laying the foundation of all modern CNNs.

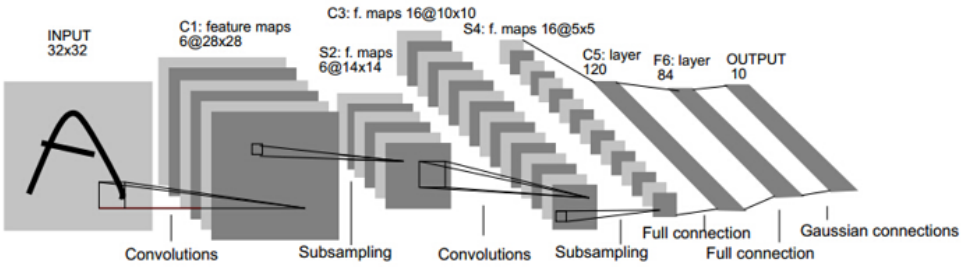


Figure 30.5.5: Scheme for LeNet [24]

30.5.2.2 *AlexNet*

In 2012 Alex Krizhevsky, supervised by Groffrey Hinton, invented AlexNet [25], which won the ImageNet contest by a large margin. Compared to LeNet, key features of AlexNet are

- using much deeper architecture [Figure 30.5.6] than LeNet

- using ReLU instead of sigmoid to avoid saturation effect.
- data augmentation in training via image flipping, zooming in/out, shifting, etc.

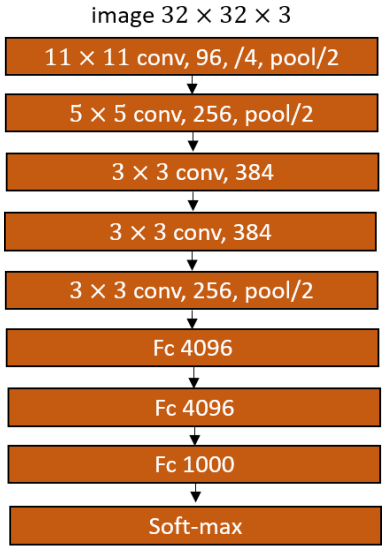


Figure 30.5.6: Architecture of AlexNet.

30.5.2.3 *VGG*

VGG is a family of convolutional neural network architecture invented by Visual Geometry Group (VGG) at University of Oxford in 2014 [26]. It was the first runner-up of the 2014 ImageNet contest in the classification task after the GoogLeNet.

The major innovative components of VGG over AlexNet [Figure 30.5.7, Figure 30.5.8] are the use of small 3 by 3 kernels and larger depth to progressively extract useful features.

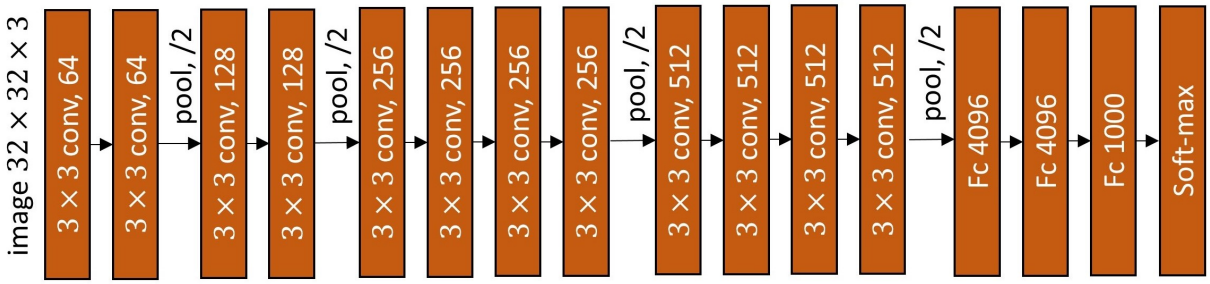


Figure 30.5.7: A typical VGG architecture: VGG-19 scheme.

ConvNet Configuration					
A	A-LRN	B	C	D	E
11 weight layers	11 weight layers	13 weight layers	16 weight layers	16 weight layers	19 weight layers
input (224 × 224 RGB image)					
conv3-64	conv3-64 **LRN**	conv3-64 **conv3-64**	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
maxpool					
conv3-128	conv3-128	conv3-128 **conv3-128**	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128
maxpool					
conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256 **conv1-256**	conv3-256 conv3-256 **conv3-256**	conv3-256 conv3-256 conv3-256 **conv3-256**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
FC-4096					
FC-4096					
FC-1000					
soft-max					

Figure 30.5.8: The depth of the configurations increases from the left (A) to the right (E), as more layers are added (the added layers are shown in bold). The convolutional layer parameters are denoted as "conv-receptive field size-number of channels" [26]

30.5.2.4 *ResNet*

Before the advent of ResNet, CNN variants usually have 20 to 30 layers. However, it was found[27] 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 varnishing gradients [Figure 30.5.9].

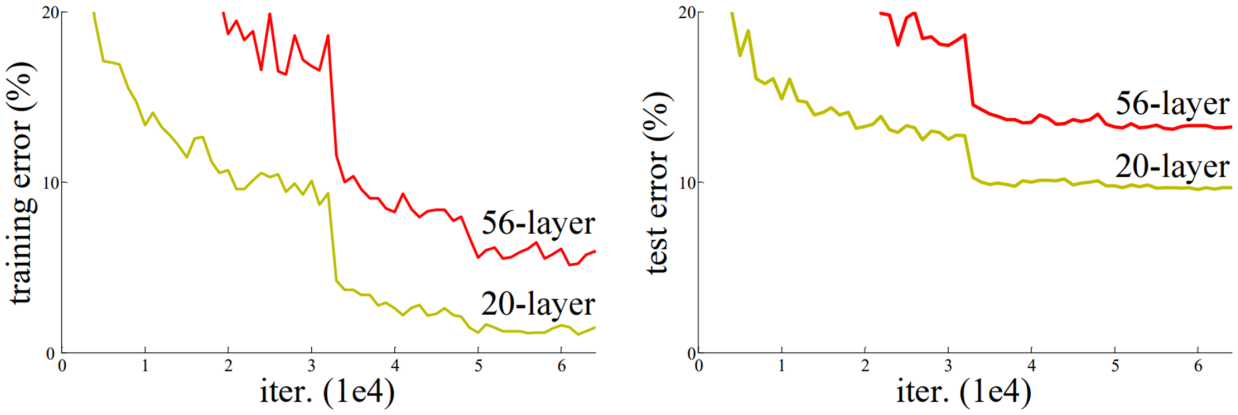


Figure 30.5.9: Training error (left) and testing error (right) on CIFAR-10 with 20-layer and 56-layer vanilla CNN networks. The deeper network has both higher training error and testing error.[27]

The varnishing gradient issue is addressed by introducing short-cuts crossing different layers [Figure 30.5.10]. Particularly, these short-cuts are identity maps that will not reduce gradient magnitude during back-prorogation. From this perspective, ResNet can be viewed as a special case of Highway Network[28].

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 $H(x) - x$.

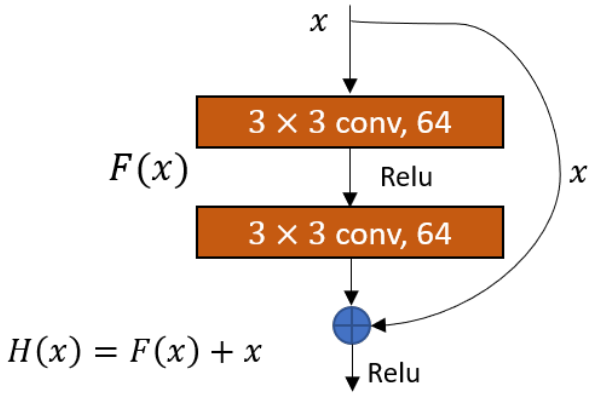


Figure 30.5.10: Scheme for a residual block.

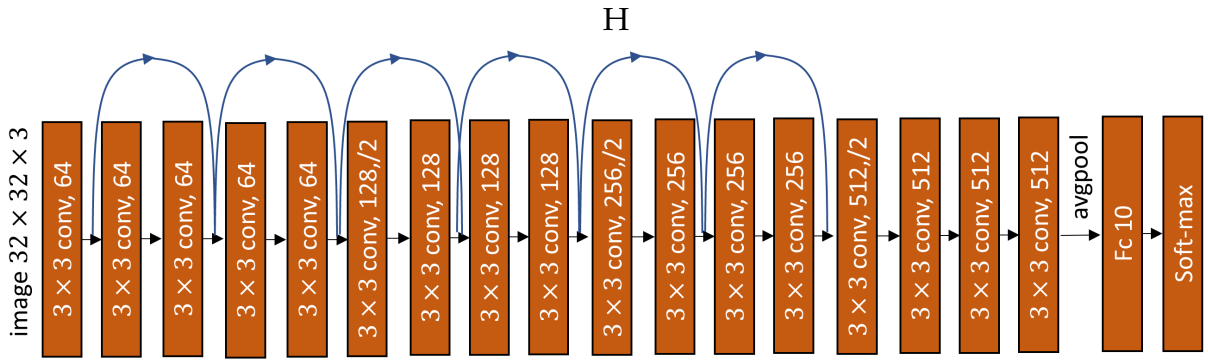


Figure 30.5.11: Scheme of a 18 layer ResNet

A typical ResNet that stacks multiple Residual block is showed in Figure 30.5.11. ResNets can have more than 1000 layers but still have superior performance.

30.6 CNN application examples

30.6.1 Image classification

One of most important application of CNN is image classification, from the hand-written digits to more challenging object recognization. In fact, deep learning regain its popularity because of the superior performance of CNN on the 2012 ImageNet context[25].

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 30.6.1], 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%.

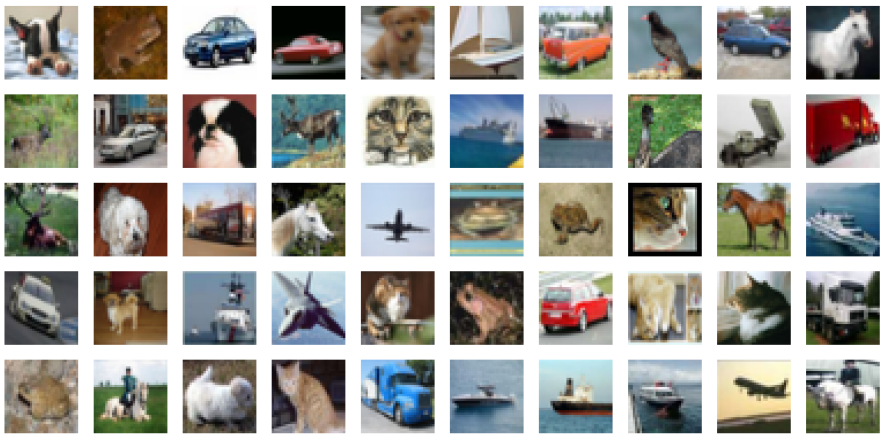


Figure 30.6.1: Example images from CIFAR10 image dataset.

30.6.2 Visualizing CNN

30.6.2.1 *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 [29].

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 [30]

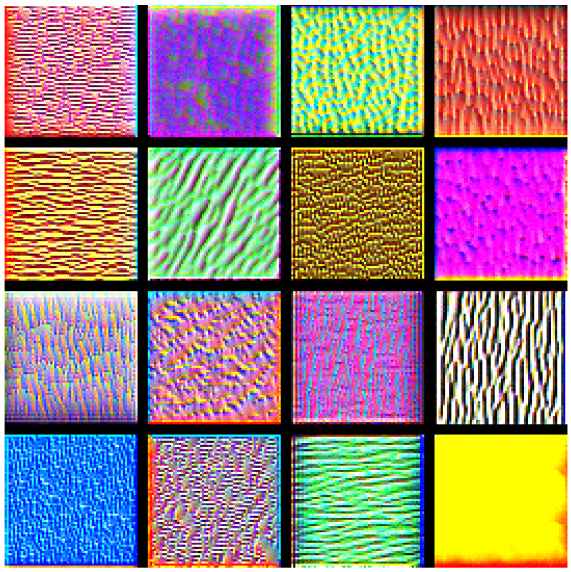
$$R_i^l = \arg\max_x \left\| F_i^l \right\|_F^2$$

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

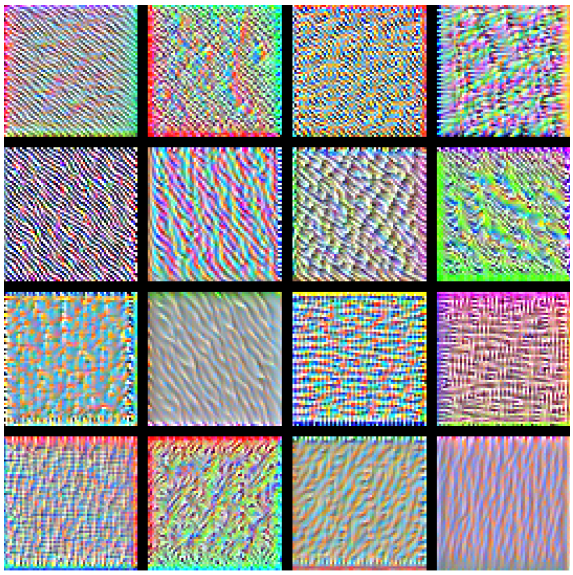
The optimization can be carried out by simple gradient ascent starting from a random noise image. In the following, we visualize filters of different layers in a pretrained VGG-16 on a test image.

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.

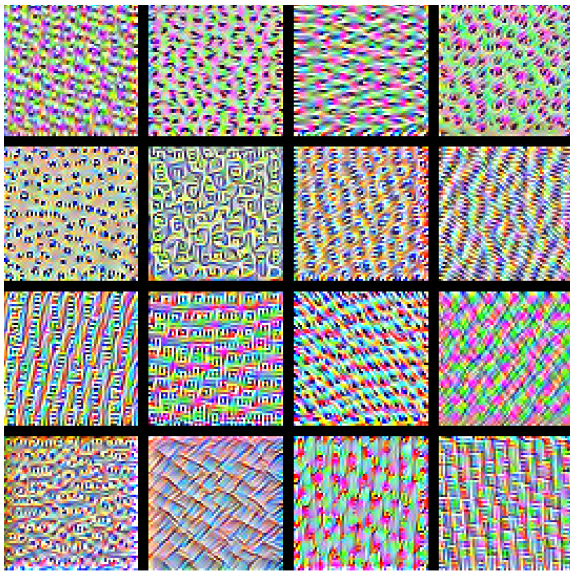
Other algorithms and implementation on visualizing CNN can be found in [31].



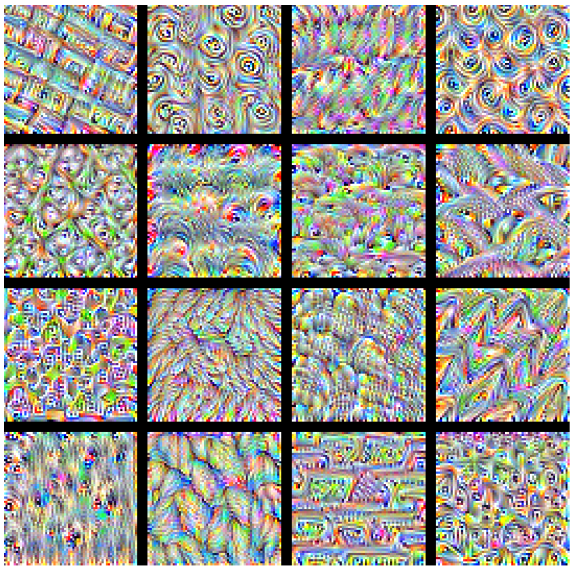
(a) Visualization of kernel in the second convolutional layer.



(b) Visualization of kernel in the seventh convolutional layer.



(c) Visualization of kernel in the tenth convolutional layer.



(d) Visualization of kernel in the seventeenth convolutional layer..

Figure 30.6.2: Visualization of convolution layer.

30.6.2.2 *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)[32].

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 30.6.3.

Figure 30.6.3: Grad-CAM method applied to understand image classification. Middle and right are class-discriminative localization map superimposed onto the original image.

30.6.3 Autoencoders and denoising

30.6.3.1 *Autoencoders*

In subsection 29.1.2, we introduce principal component analysis (PCA) 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[33] can be viewed as a nonlinear generalization of PCA using neural networks. A typical autoencoder neural network consists of two parts [Figure 30.6.4]: 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

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

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 [34]. We also compared the PCA dimensionality reduction performance by reconstructing the same data using the top 50 principal components.

Results in Figure 30.6.5 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.

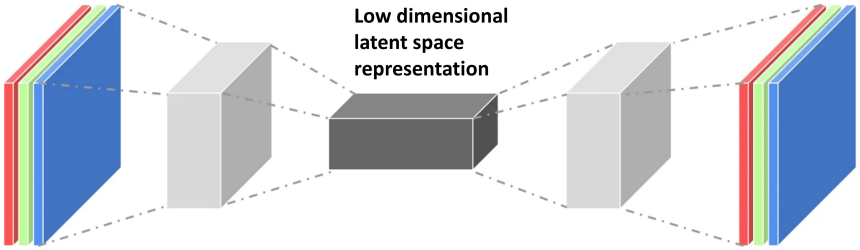


Figure 30.6.4: A CNN based autoencoder. An autoencoder consists of an encoder that transforms high-dimensional image into a low-dimensional code and a decoder that unfolds the code to reconstruct the image.

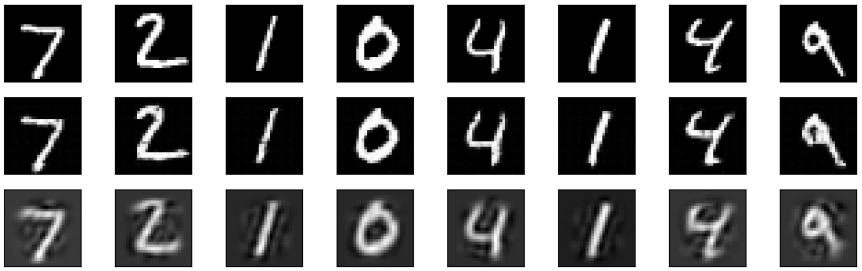


Figure 30.6.5: Comparison of reconstruction performance on random samples from MNIST data set. Top row is the original data. Middle row is autoencoder result based on a 49 dimensional code. Bottom row is PCA result based on a 50 dimensional code.

30.6.3.2 *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 [35]. 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 $\left\|x - x'\right\|^2$.

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

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

Figure 30.6.6 shows the application of a denoising autoencoder to the MNIST dataset.

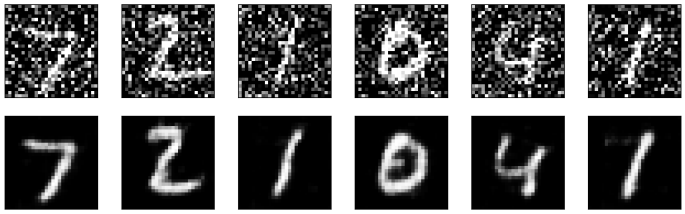


Figure 30.6.6: Denoising autoencoders applied to remove the noise in the MNIST dataset.

30.6.4 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 30.6.7.

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^l_{ij} 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

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

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

The authors in [36] 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

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

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

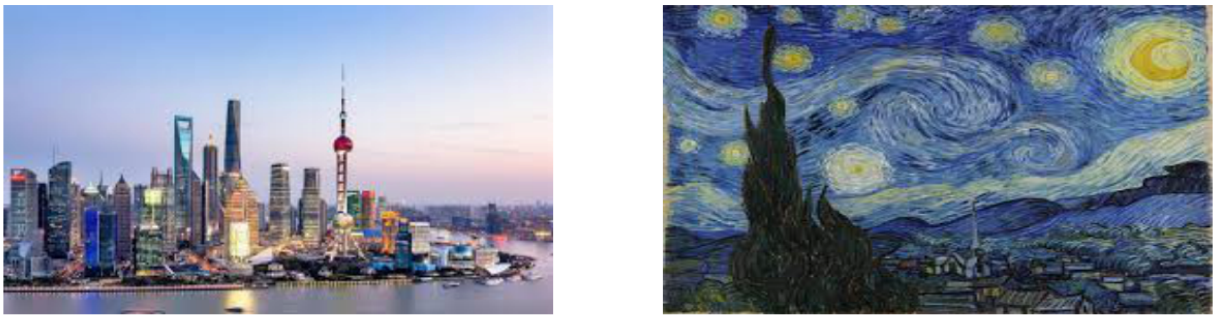
$$L_{\text{style}}(x, s) = \sum_{l \in S_L} \frac{1}{4N_l^2 M_l^2} w_{c,l} \|G - A\|_F^2 = \sum_{l \in S_L} \frac{1}{4N_l^2 M_l^2} \sum_{i,j} w_{s,l} \left(F^l_{ij} - P^l_{ij}\right)^2$$

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

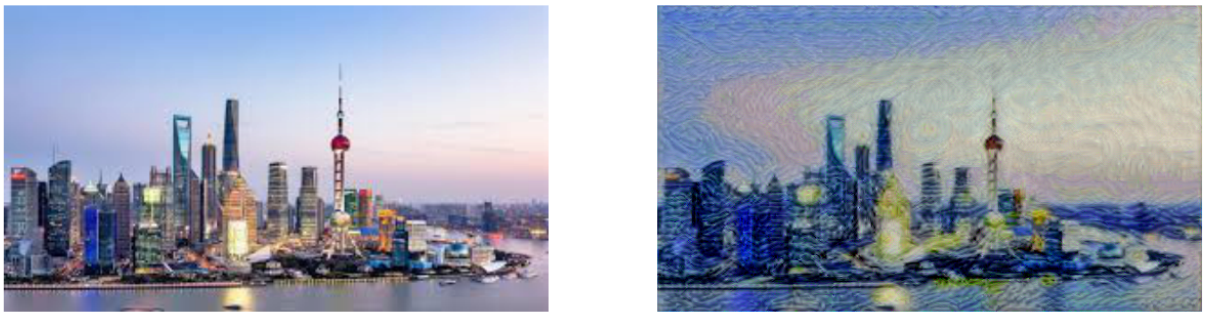
The total loss L is a linear combination of content loss and style loss, given by

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

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 30.6.7]. In the second example, we transfer the Picasso painting style to the Shanghai view picture [Figure 30.6.8].

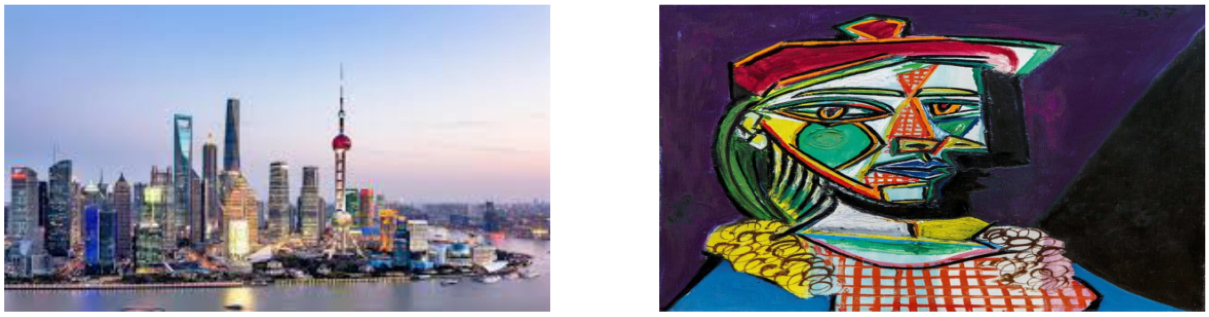


(a) Content image and style image.

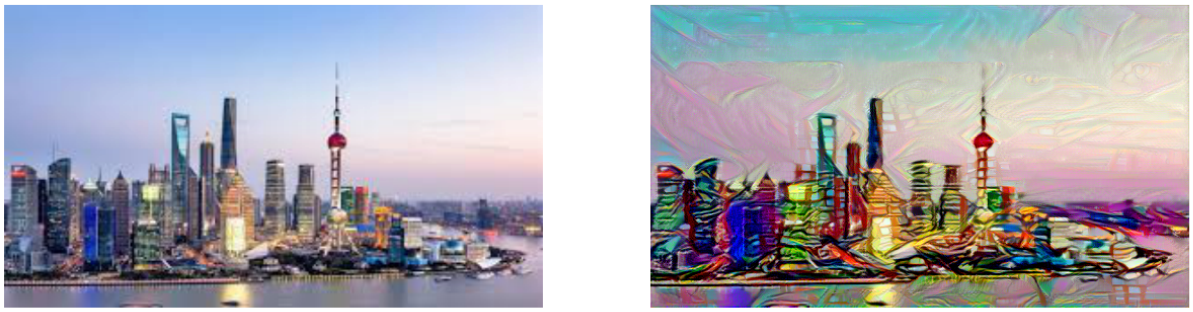


(b) Original content image and the transformed content image.

Figure 30.6.7: 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 30.6.8: Demonstration of neural style transfer with Picasso painting style.

30.6.5 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[37, 38], robotics[39], drug design[40], etc. By equipping reinforcement learning method with convolutional neutral 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[41, 42]. 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 30.6.9.

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 30.6.9(a)] to approximate the optimal action-value function (known as the Q function) given by

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

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 30.6.9(b)],

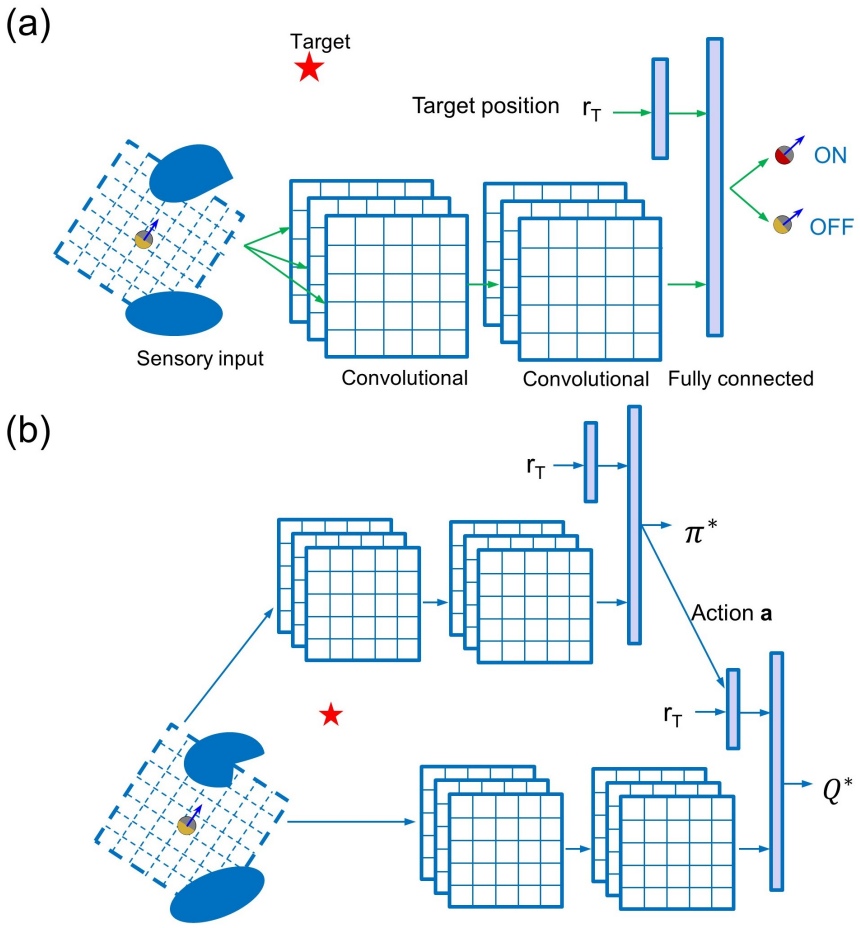


Figure 30.6.9: Application of CNN in deep reinforcement learning in Q learning approach and Actor-Critic approach. Two streams of sensory inputs are fed to the neural network, including a pixel image of the robot's neighborhood fed into a convolutional layer and the target's position fed into a fully connected layer.

30.6.6 Sentence classification

The primary playgrounds of CNNs are image analysis and understanding. The local connectivity in CNN suits spatial locality in images and the multi-layer of filters enable progressive extraction of high-levels features from low-level raw pixel data [sub-subsection 30.6.2.1]. The ability of CNN to extract high-feature from local low-level features also sparks novel application in language related application, such as sentence classification[43].

We first look at how CNN filters can act as feature extractor for sentences. Consider a sentence consisting of N words, with each word represented by a k-dimensional vector $x_i \in \mathbb{R}^k$. In general x_i are low-dimensional dense word embeddings [subsection 30.4.3]. If we view a sentence as a stack of row vectors of its constituent words, then the representation of a sentence is similar to an image with size $N \times k$. If we apply a convolutional filter with kernel size $h \times k$, then the output will be a one-dimensional feature vector of length $N - h + 1$ [Figure 30.6.10].

Mathematically, the CNN filter extracts a new feature vector from a word sequence of window h given by

$$c_i = f\left(\sum_{j=1}^{h} W x_{i+j-1} + b\right),$$

where W is the kernel matrix, b is the bias, and f is a non-linear activation function. In models proposed in [43], CNN filters have different window sizes $h = 3, 4, 5$, with 100 for each of them. Intuitively, these CNN filters are extracting features from 3-grams, 4-grams, and 5-grams. All these CNN extracted features are then pooled and combined to feed into feed-forward layers for sentence classification [Figure 30.6.11].

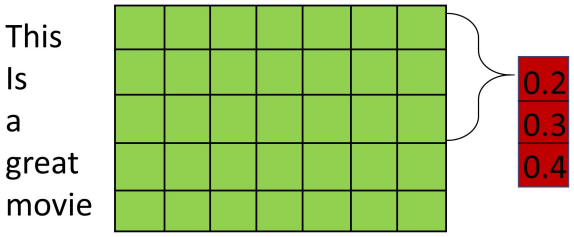


Figure 30.6.10: Demonstration of a CNN filter applying to a sentence to produce a one-dimensional feature vector.

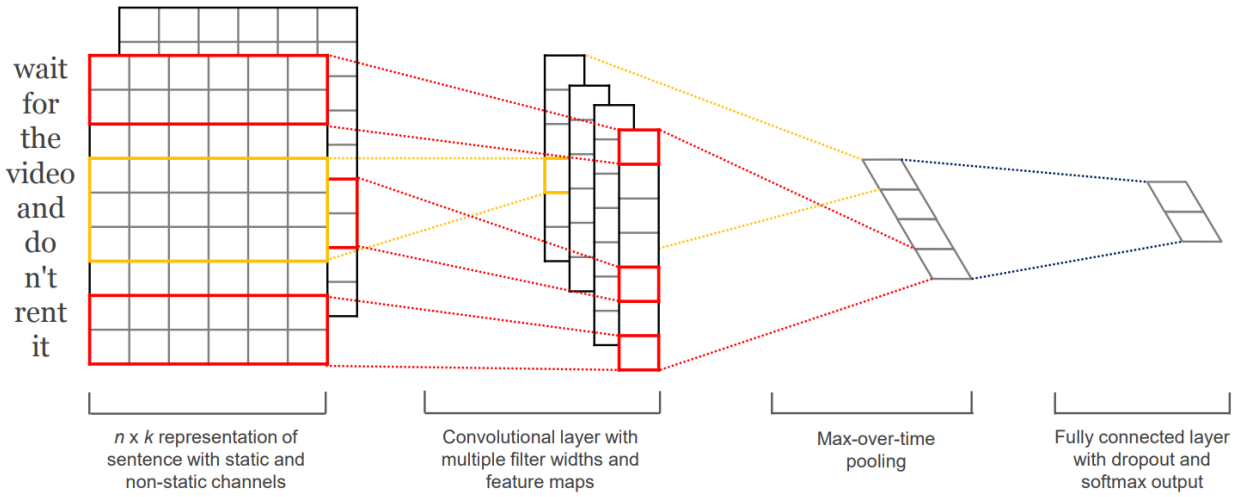


Figure 30.6.11: CNN for sentence classification proposed in [43].

30.7 Recurrent neural networks (RNN)

30.7.1 Recurrent units

30.7.1.1 *Simple recurrent unit (SRU)*

In applications ranging from time-series prediction and classification to speech recognization 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., $(w_1, ..., w_t)$) 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 30.7.1]. 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 $x = (x_1, x_2, ..., x_t, ..., x_L), x_t \in \mathbb{R}^D$. 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 30.7.1]

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

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 $h_1, ..., h_T$ can be fed into different neural network layer structures to enable diverse applications.

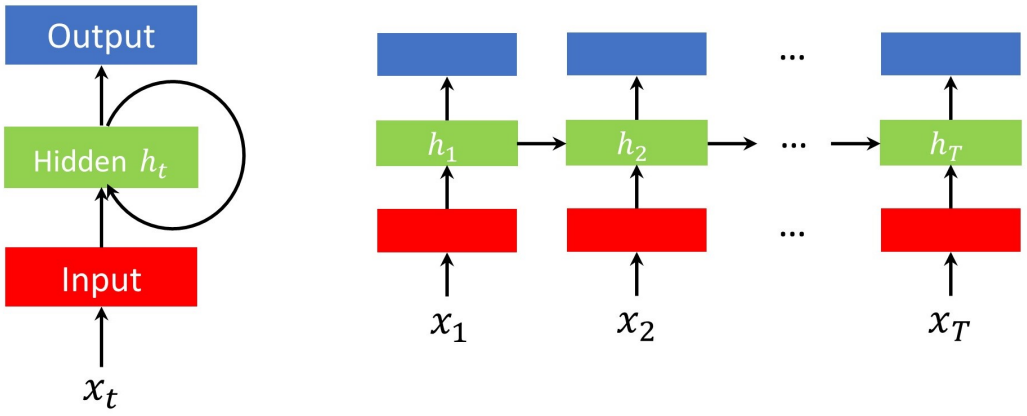


Figure 30.7.1: Scheme of recurrent units in a neural network (left). Recurrent neural network can be unrolled (right)

30.7.1.2 *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 30.7.1 (universal approximation to nonlinear dynamic systems). *[44, p. 798][45, p. 139] Any nonlinear dynamic system given by*

$$h_t = g(h_{t-1}, x_t)$$
$$y_t = o(s_t)$$

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.

30.7.1.3 *Backpropogation 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 backpropogation procedure [subsection 30.1.4]. Using similar derivation, we can obtain a gradient calculation procedure, known as **backpropogation through time (BPTT)**.

To derive BPTT, we consider a simple RNN [Figure 30.7.2] with following formulation:

$$h_t = \tanh\left(W_{hh}h_{t-1} + W_{xh}x_t + b_h\right)$$
$$y_t = \text{softmax}\left(W_{hy}h_t + b_y\right)$$

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

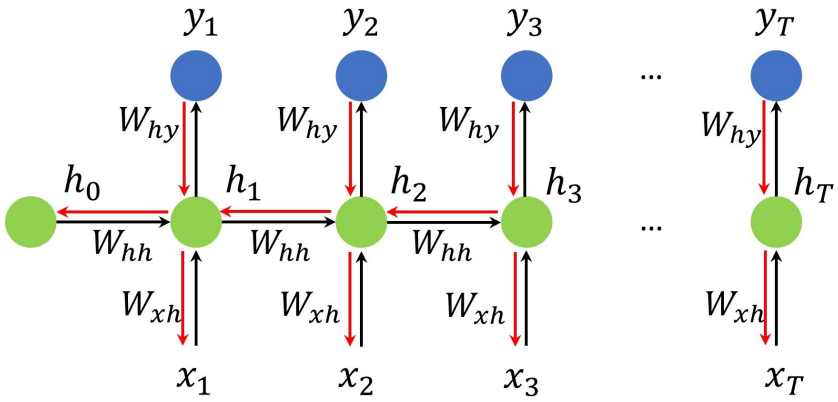


Figure 30.7.2: Scheme for backpropogation through time in a simple RNN. Red arrows are backpropogation directions.

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

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

W_{hh} contributes to the loss at step t via updating $h_1, h_2, ..., h_t$ from $h_0, h_1, ..., h_{t-1}$, 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

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

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 $h_1, h_2, ..., h_t$ from $x_1, x_{t-1}, ..., x_t$, 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 then 1, the gradient will vanish. 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.

30.7.2 Recurrent unit variants

30.7.2.1 *Long short term memory (LSTM)*

One critical variant of RNN is the Long short-term memory (LSTM) unit [Figure 30.7.3. [46]] 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

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

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 date 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

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

The overall information flow scheme can be summarized in Figure 30.7.3.

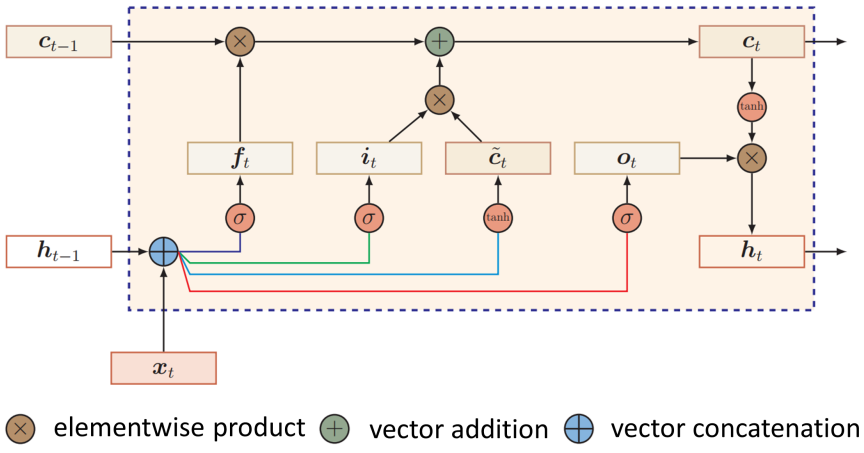


Figure 30.7.3: Scheme of an LSTM cell. Modified from [45, p. 149].

And the mathematical relation can be summarized by

Definition 30.7.1 (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:*

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

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

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

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 $h_t = o_t \odot \tanh\left(c_t\right)$ where $o_t = \sigma\left(W_o x_t + U_o h_{t-1} + b_o\right)$. 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 30.7.1 (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 subsubsection 30.7.1.3, 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

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

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.

30.7.2.2 *Gated Recurrent Unit (GRU)*

Gated recurrent unit [47] 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 control the output h_t as the mixture of h_{t-1} and $\hat{h}_t$. Notably, when $z = 0, r = 1$, GRU becomes simple RNN; when z is large, most historical information is preserved.

The overall information flow scheme scheme can be summarized in Figure 30.7.4.

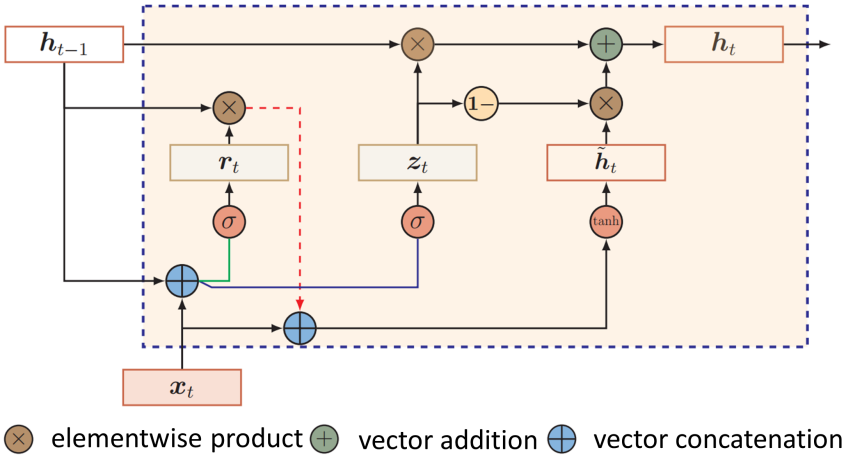


Figure 30.7.4: Scheme of a GRU cell.Modified from [45, p. 152].

And the mathematical relation can be summarized by

Definition 30.7.2 (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 $b_z, b_r, b_h \in \mathbb{R}$.

30.7.3 Common RNN architectures

We have covered several most commonly used recurrent units Figure 30.7.5. 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.

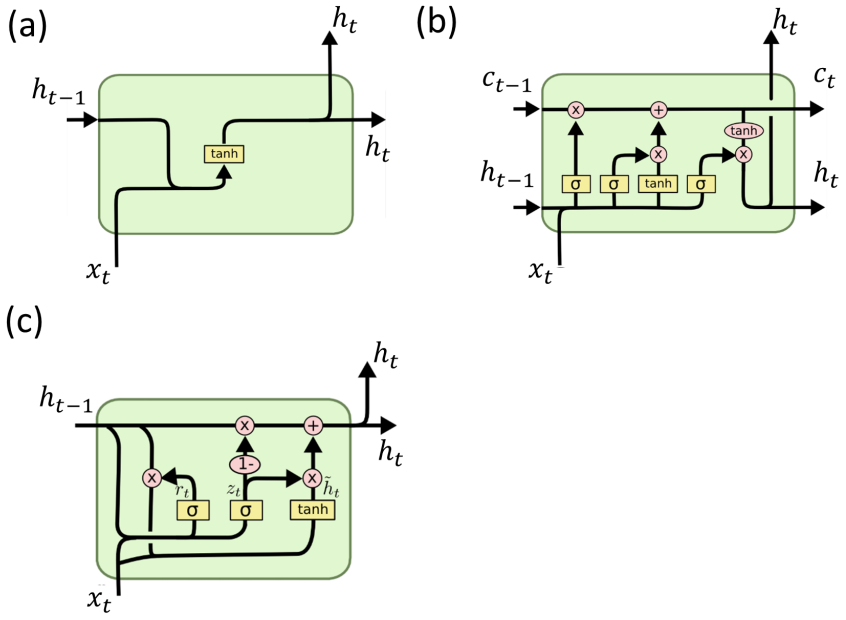


Figure 30.7.5: Different types of units in RNNs: (a) Vanilla RNN cell. (b) LSTM cell. (c) GRU cell. Credit

In this section, we give an overview on classical architecture designs that suit different applications.

First, we consider different types of connections from the last recurrent layers to the output layer, as showed in Figure 30.7.6. The architecture in which recurrent output of every time step t connects to the output layer [Figure 30.7.6(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 $x_1, ..., x_t$. 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 30.7.6(b)(c).

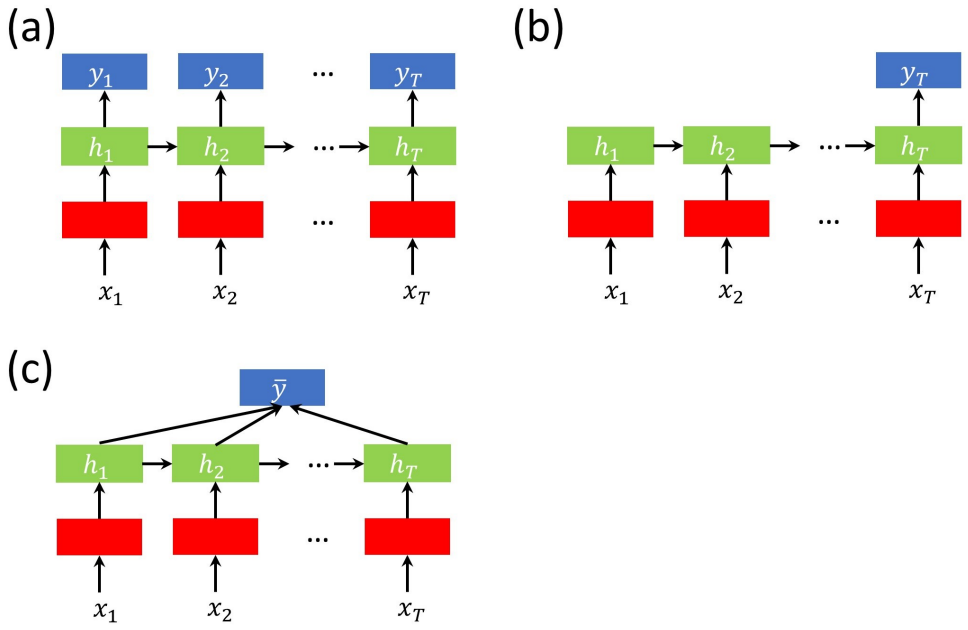


Figure 30.7.6: Typical RNN connection to the output layer.

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 also stack multiple recurrent layers together and form stacked RNN [Figure 30.7.7].

Finally, for applications that require modeling dynamic relations like $y_t = f(x_{1:T}), T > t$, we can use bidirectional RNN. Bidirectional RNN has one layer reverse recurrent layer flowing subsequent input information to preceding hidden states (e.g., from x_T to h_1). 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 30.7.8].

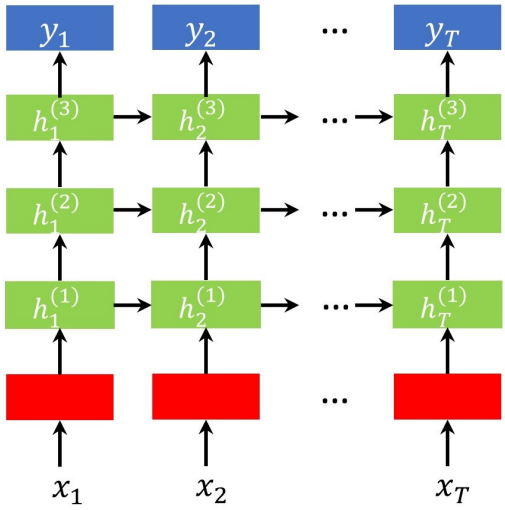


Figure 30.7.7: Scheme for stacked RNN.

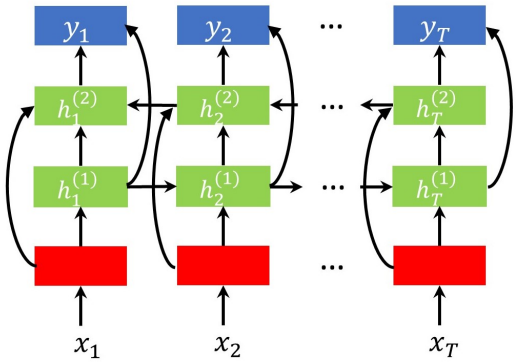


Figure 30.7.8: Scheme for bidirectional RNN.

30.8 RNN application examples

30.8.1 Time series prediction

30.8.1.1 *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 21. A basic architecture for nonlinear time series with possible long memory dependence is given by Figure 30.8.1.

Time series prediction problems are generally formulated as a regression problem. Let $x_1, x_2, ..., x_N, x_i \in \mathbb{R}^D$ 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

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

The training examples are then fed into RNN and yield predicted values [Figure 30.8.1(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 30.8.1(b)].

As a demonstrate, we train a RNN to predict time series

$$y(t) = f(x(t), x(t-1), ...), where x(t) = \cos(t), y(t) = \sin(t).$$

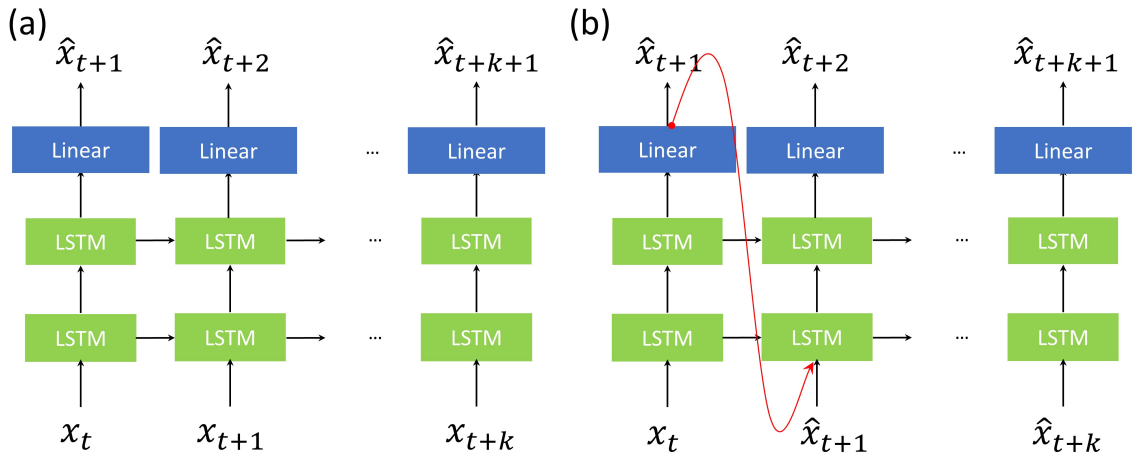


Figure 30.8.1: RNN architecture for time series prediction . (a) In the training phase, RNN are updated by minimizing the next step prediction error. (b) In the prediction phase, trained RNN is used to sequentially predict next step state value based on preceding predicted state value.

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

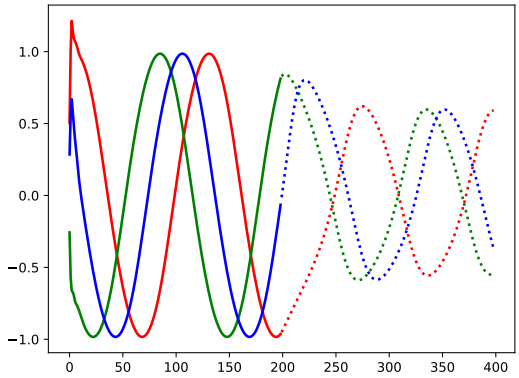
$$
\begin{aligned}
&(\sin(\frac{0}{T}+\phi^0), \sin(\frac{1}{T}+\phi^0, ..., \sin(\frac{N}{T}+\phi^0) \\
&(\sin(\frac{0}{T}+\phi^1), \sin(\frac{1}{T}+\phi^1, ..., \sin(\frac{N}{T}+\phi^1) \\
&\cdots \\
&(\sin(\frac{0}{T}+\phi^{1000}), \sin(\frac{1}{T}+\phi^{1000}, ..., \sin(\frac{N}{T}+\phi^{1000})
\end{aligned}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

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

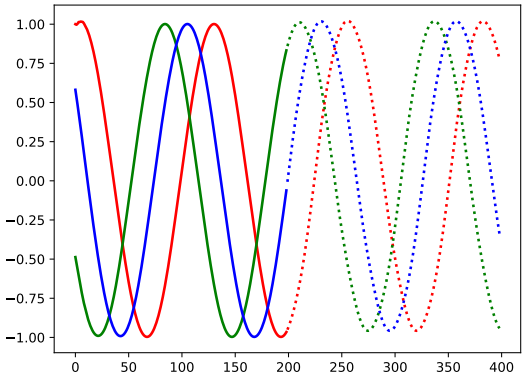
We use a two-layer stack RNN and train for 50 episodes [Figure 30.8.2]. After training, we use 3 out of sample time series to examine the one-step forward and multiple forward prediction performance. We have following observations:

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

In the previous architecture [Figure 30.8.1], 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 30.8.3]. Covariate time series are sequential observations of variables that 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.

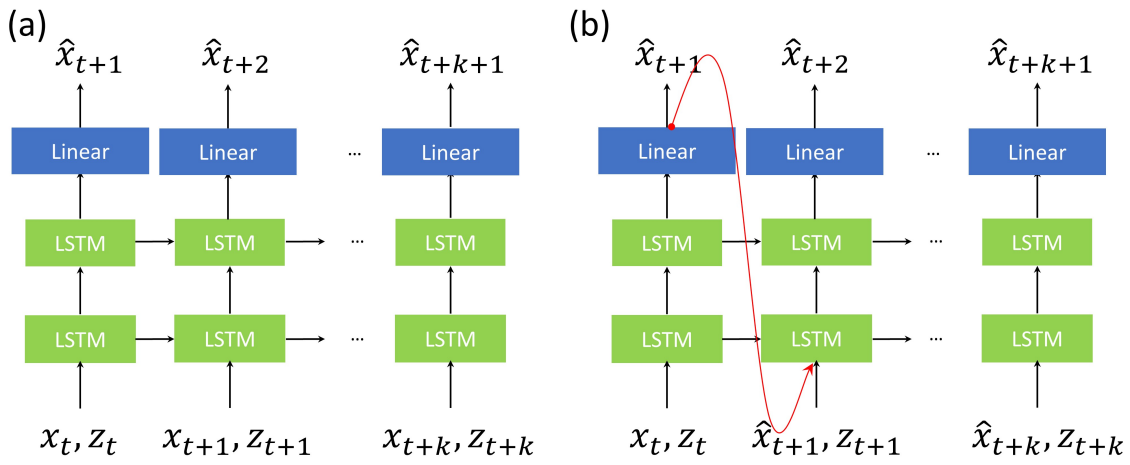


Figure 30.8.3: RNN architecture for time series prediction with covariates. (a) In the training phase, RNN are updated by minimizing the next step prediction error. (b) In the prediction phase, trained RNN is used to sequentially predict next step state value based on preceding predicted state value. Note that covariate time series are assumed available for all time steps.

30.8.1.2 *Deep autoregressive (DeepAR) model*

Previous simple RNN architectures [Figure 30.8.1, Figure 30.8.3] can be used to predict deterministic time series or to predict mean levels for stochastic time series. Deep autoregressive (DeepAR) model [48] 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 $[x_t, x_{t+1}, \ldots, x_T] := z_{t:T}$, the goal is to model the conditional distribution $P\left(x_{T+1}|z_{t:T+1}, x_{1:T}\right)$. In DeepAR model architecture [Figure 30.8.4], 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

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

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.

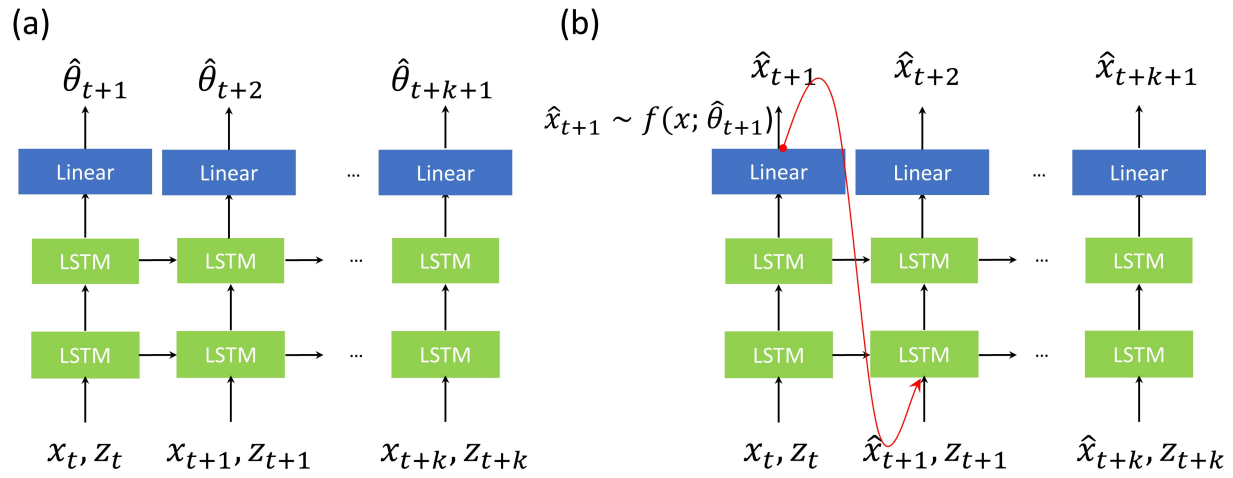


Figure 30.8.4: DeepAR model architecture for time series prediction. Outputs are the parameters characterizing the condition distribution of x_{t+1} conditioned on histories of x_t and z_t. (a) In the training phase, RNN are updated by minimizing the negative log likelihood function. (b) In the prediction phase, a predicted $\hat{x}_{t+1}$ are sampled from predicted conditional distribution and then used to predict next step conditional distribution.

30.8.1.3 *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 $x_{i,t}, i = 1, ..., N$ are the multivariate time series to model, $g_k, k = 1, .., K, K \ll N$ 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** [49].

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

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

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 [49]

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

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.

30.8.2 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 30.8.5. There are 128 hidden units in the LSTM unit. After moderate training, the classification accuracy can achieve 97%, comparable to a simple CNN.

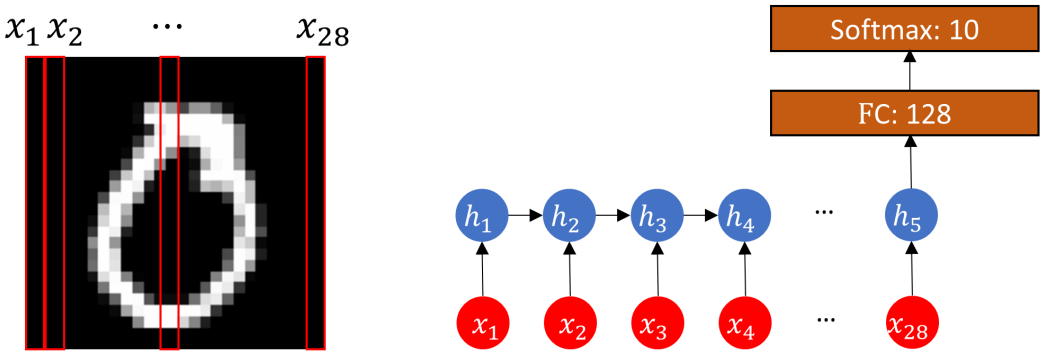


Figure 30.8.5: A RNN architecture for MNIST image recognition.

30.8.3 Sentiment classification

Sentiment analysis via bagging solutions [subsection 30.4.4] does not take into the semantic meaning conveyed by the order of words. For example, sentences like 'do not miss this good movie' and 'this is not a good movie' have opposite sentiments even they have similar word counts.

Using RNN for sentiment analysis, on the other hand, can capture complex semantic meaning in the sequence of words in the classifier. The overall architecture [Figure 30.8.6] contains the following components:

- An embedding layer that convert one-hot vectors of words to efficient low dimensional representations. The embedding layer can have initialized weight from some pretrained models.

- LSTM layers that process sequences of embedded word vectors and capture dependence between words.
- The last output from the LSTM will be fed into a Sigmoid output layer. Sigmoid output is used as the probability of positive or negative labels.

Before feeding the data to the network, we also need some basic data cleaning. Unlike traditional machine learning methods, we here only need minimal feature engineering.

- We need to get rid of punctuation such as comma, periods, etc.
- To facilitate batch processing, we also like to make all input sequence have the same size. For longer sequences, we will truncate the size; for shorter sequences, we will left pad dummy one-hot encoding.

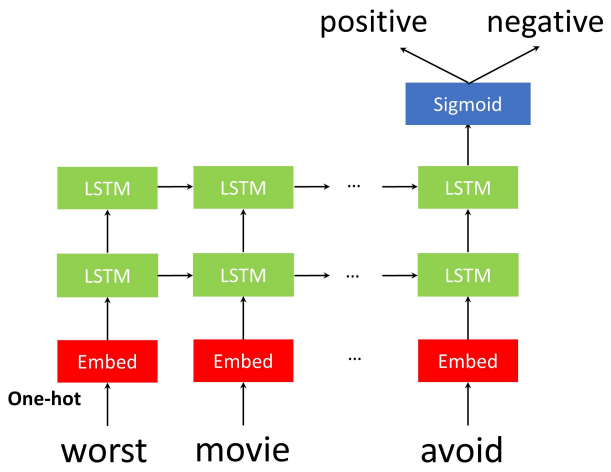


Figure 30.8.6: RNN architecture for sentiment analysis.

30.8.4 Character-level language modeling

30.8.4.1 *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 30.8.7. 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.

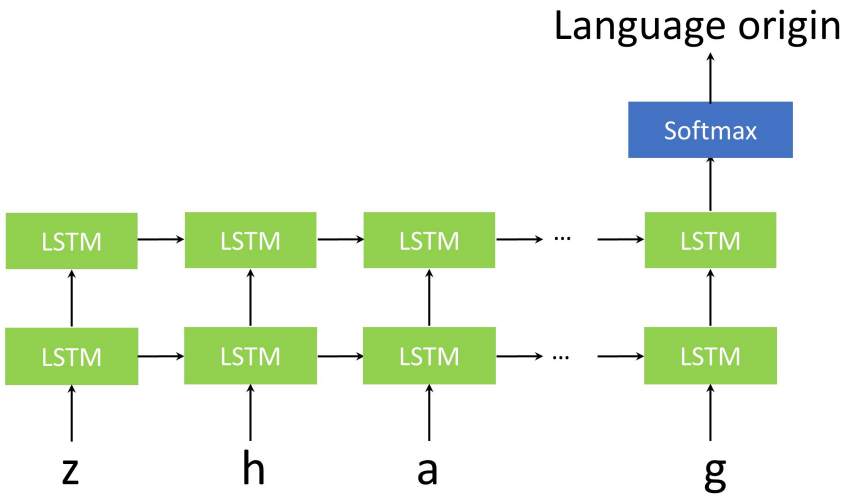


Figure 30.8.7: A RNN for character-level word classification.

30.8.4.2 *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 30.8.8]. 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.

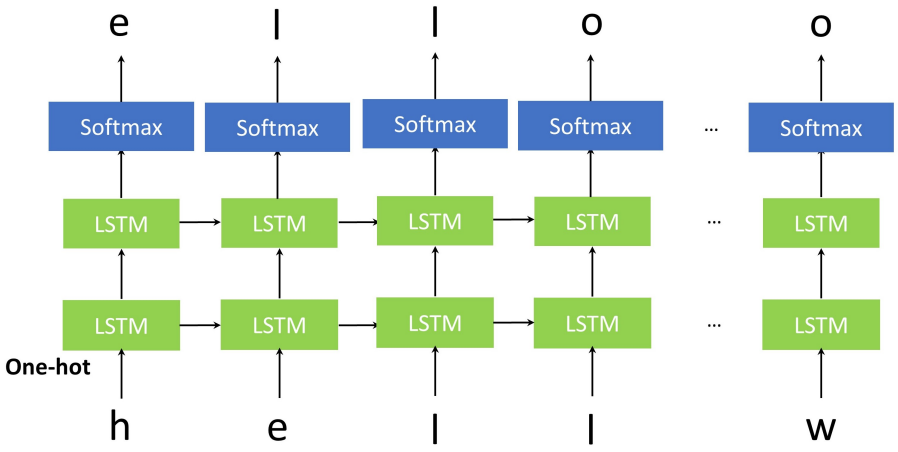


Figure 30.8.8: A RNN for character-level language model. During the training session, one-hot coded characters are directly fed into RNN and predict the next character in the word.

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 30.8.9. 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

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

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

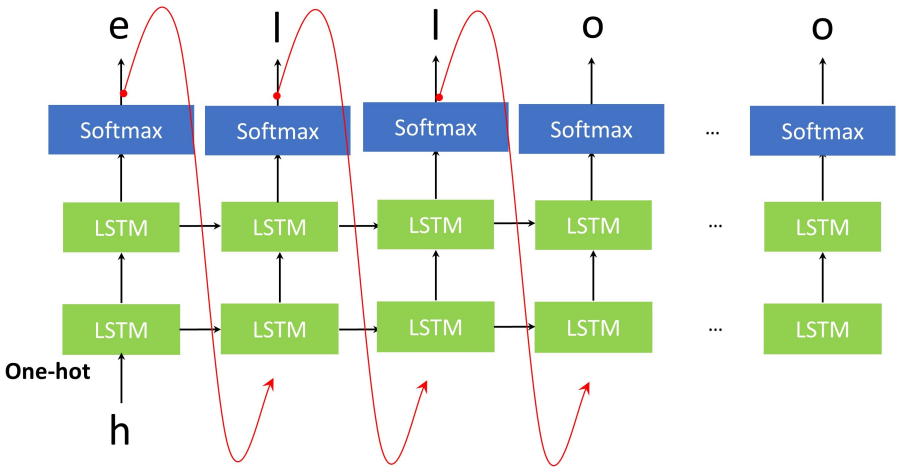


Figure 30.8.9: A RNN for character-level language model. During the word generation session, the network starts with a user input character and continues the generation process with predicted character from the previous step.

30.9 Sequence-to-sequence modeling

30.9.1 Encoder decoder model

In Sequence-to-sequence (seq2seq) modeling, we are seeking a mapping that takes a sequence $X_1, X_2, ..., X_N$ as the input and outputs another sequence $Y_1, Y_2, ..., Y_M$ that meet certain requirements. In general, X_i and Y_j can be a character, a words, or a real-valued vector, an image, etc, and generally $N \neq M$.

Many fascinating AI real-world applications fall into the seq2seq category. Examples include

- machine translation, where a word sequence in one language is mapped to a word sequence in another language;
- speech recognization, where a time series of audio signal is mapped to a word sequence;
- video captioning, where a video, or a sequence of images, is mapped to a word sequence;
- text summarization, where a long sequence of words is mapped to a short sequence of words (i.e., a summary).

One challenge in a seq2seq modeling problem is that usually it cannot be decomposed to a smaller-scale mapping problem between one sequence unit to another sequence unit. For example, in translation task, one word in one language could be translated to one word, or multiple words, or even not translated depending on the context. Further, usually no synchrony exist between sequence X and sequence Y. Take machine translation for another example. Suppose in a word sequence X_i precedes word X_j, and in the translated word sequence, they also have clear correspondence to Y_k and Y_l, respectively. By no synchrony, we mean Y_k might come after Y_l due to language grammar reasons. Therefore, in the seq2seq modeling, sequences must be processed as a whole unit. Note that such architecture can include multiple hidden layers as well. A example of synchrony issue in machine translation is showed in Figure 30.9.1.

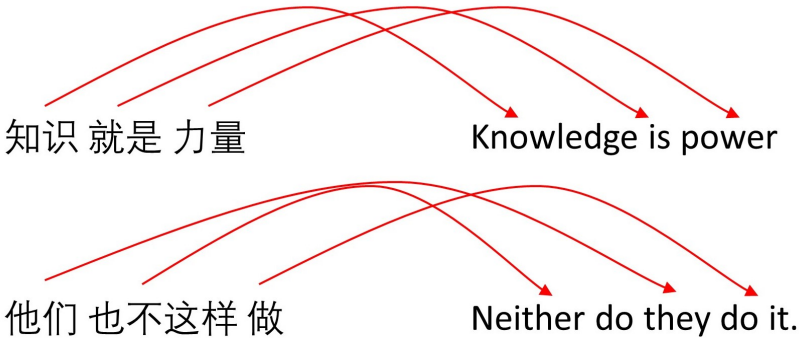


Figure 30.9.1: Seq2seq modeling for language transformation. The input and output sequences might have different lengths, and not in synchrony.

The most basic implementation of a sequence-to-sequence model is via a encoder-decoder network consisting of two RNNs, called encoder and decoder, respectively [Figure 30.9.2]. The encoder-decoder framework stems from efforts in natural language model[50]. The encoder reads a sequence of inputs and outputs the hidden vector at the end of the sequence, known as **the context vector**; the decoder takes the context vector as the initial hidden state input and then sequentially generates a sequence. After training, the context vector is expected to store all relevant, critical information of the input sequence such that the decoder can produce sequences meet certain requirements.

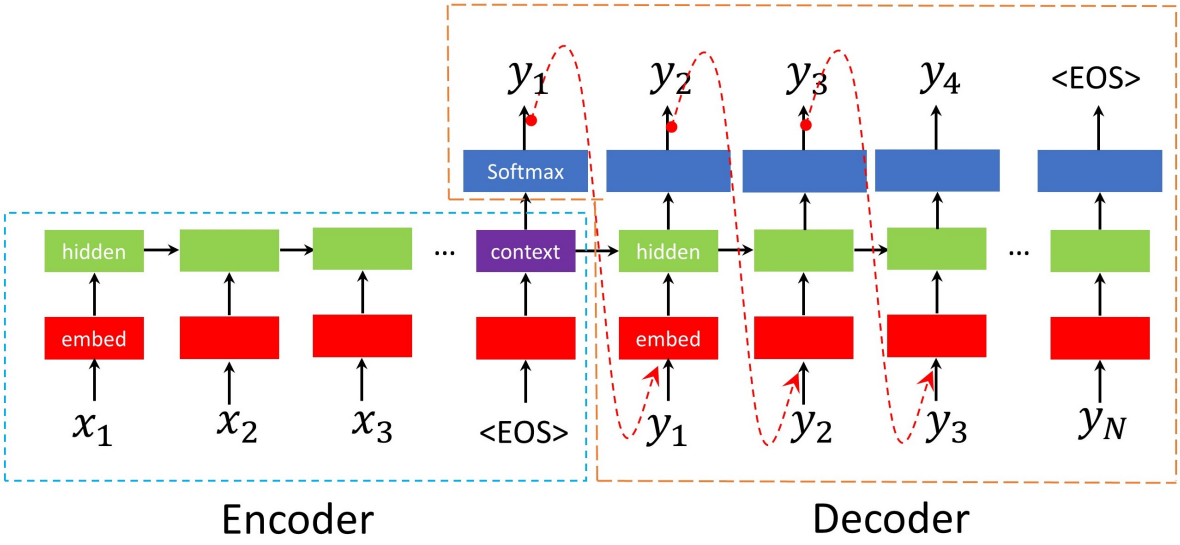


Figure 30.9.2: The Encoder-decoder architecture for seq2seq language modeling. The input sequence is fed into the encoder RNN and terminated by an explicit <EOS> symbol. Then the decoder RNN starts with the context vector and the final prediction of the encoder to generate an output sequence until an explicit <EOS> symbol is produced.

In a video captioning application, this encoder-decoder model will take image sequence as the input, with the usage of convolutional layers replacing embedding layers to extract image feature, and will produce a sequence of words to generate words as captions.

30.9.2 Attention mechanism

The performance of a basic encoder-decoder model [Figure 30.9.2] relies on the **context vector** to store all relevant, critical information of the input sequence. It is not hard to notice that this encoder-decoder network suffers when the input sequences are very long: difficulties arise in both in the encoding part - difficult to encode long sequences in a single vector without losing information, and in the decoding part - difficult to decode a single information vector to a long sequence.

Attention mechanism is inspired how human beings focus on relevant information when performing visual and language tasks[51, 52]. Consider a human translates a sentence. When he translates each word in the sentence, he tends to focus on its related context rather than focusing on the entire sentence. Similarly, when a human is watching closely some part of a object, he tends to focus on the part of interest.

In the context of encoder-decoder, all hidden states carry information and some information would get diluted in the subsequent context vector, especially when the sequence is long. A encoder-decoder architecture with attention mechanism have following features to resolve the difficulties of encoding and decoding [Figure 30.9.3]:

- During the encoding phase, all hidden states, rather than the final one, are saved to construct different context vectors via linear combination for the decoding stage.
- During the decoding phase, relevant context vectors are constructed and fed into the each hidden states in the decoder, alleviating the difficulty to decode all information from one starting context vector.

In summary, the encoder encodes a library of hidden vectors as the building blocks of different context vectors for each output in the decoding stage. The attention mechanism resembles a information lookup system that constructs and feeds the most important and relevant context vector to help the decoder part to do their job. From information flow point of view, the primary purpose attention is to calculate a weight vector to amplify signals that are most relevant in the input sequence and to de-emphasize the part that is irrelevant. The higher weights will enable more relevant hidden state information fed into the the decoder hidden layer.

Now we consider a more concrete implementation of attention mechanism from [51]. Let $(x_1, x_2, ..., x_m)$ and $(y_1, y_2, ..., y_n)$ denote the input and output sequences. The encoder uses bidirectional RNN to encode input sequences to a library of hidden states. Particularly, let $\overrightarrow{h}_i$ denote the hidden state in the forward pass and $\overleftarrow{h}_i$ the hidden state in the backward pass. A hidden state is the concatenation $h_i = cat(\overrightarrow{h}_i, \overleftarrow{h}_i)$. In the decoder part, let s_t be the hidden state information that just precedes the output y_t. s_t is given by

$$s_t = f(s_{t-1}, y_{t-1}, c_i),$$

where c_i is the context vector constructed as a weighted combination of hidden information $\sum_{i=1}^{M} w_i(t) h_i$. Note that the weights is index-dependent, derived from

$$\begin{aligned} e_i(t) &= g(h_i, s_{t-1}) \\ w_i(t) &= \frac{\exp(e_i(t))}{\sum_j \exp(e_j(t))}. \end{aligned}$$

The primary purpose of g is to capture the relevance or correlation between the decoder hidden state and the encoder hidden state. Typical choices for g functions include

$$\begin{aligned} g\left(h_i, s_{t-1}\right) &= h_i^T s_{t-1} \\ g\left(h_i, s_{t-1}\right) &= h_i^T W_g s_{t-1} \\ g\left(h_i, s_{t-1}\right) &= v_g^T \tanh\left(W_g [h_i; s_{t-1}]\right) \end{aligned}$$

where W_g and v_g are learnable parameters.

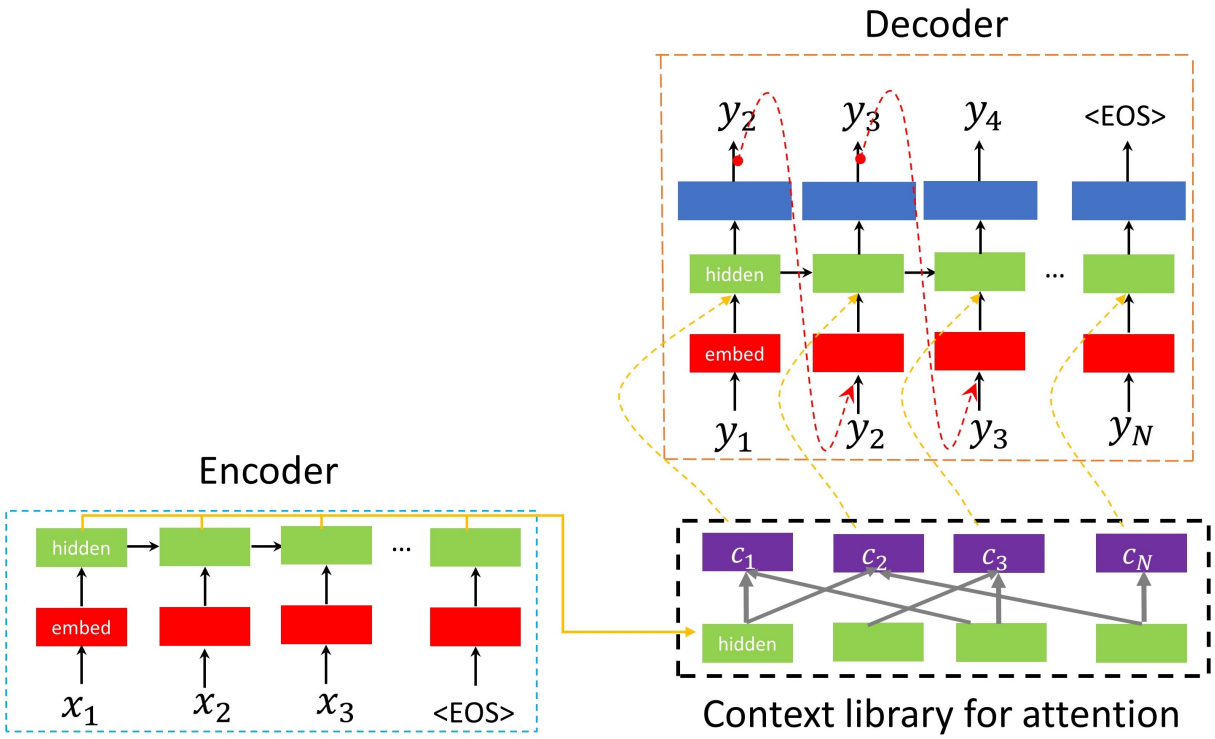


Figure 30.9.3: The Encoder-decoder architecture with attention mechanism for seq2seq modeling. During the encoding phase, all hidden states, rather than the final one, are saved to construct different context vectors via linear combination for the decoding stage. During the decoding phase, relevant context vectors are constructed and fed into the each hidden states in the decoder.

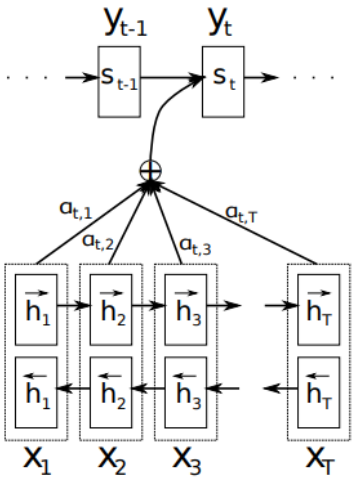


Figure 30.9.4: A bidirectional RNN encoder system with attention mechanism. [51]

30.10 Generative adversarial network (GAN)

30.10.1 Canonical GAN

30.10.1.1 *Basics*

Generative adversarial networks (GAN) are a family of generative deep learning algorithms [53] that aims to generate new data samples with the same statistical distribution as the training data. The training data is of high dimensionality (e.g., images) such that direct derivation of empirical multivariate distribution is intractable. A GAN consists of two different modules: a generator network and a discriminator network. Typically, the generator network learns to map from a latent space to a data sample following a statistical distribution of interest, while the discriminator network distinguishes candidates produced by the generator from the true data distribution. The generative network's training objective is to increase the error rate of the discriminative network (i.e., to "fool" the discriminator network by producing realistic candidates; on the other hand, discriminator, like a regular classifier, is trained to improve its classification accuracy. We can summarize the objectives in GAN as follows.

Definition 30.10.1 (GAN optimization objectives). *Let $x \in \mathbb{R}^M$ be a training example drawn from data distribution p_{data}. Let $z \in \mathbb{R}^k$ be a low dimensional latent random vector sample drawn from distribution p_z. Let $D(x) : \mathbb{R}^M \to \{0, 1\}$ be the discriminator as a binary classifier. Let $G(z) : \mathbb{R}^k \to \mathbb{R}^M$ be the generator producing realistic data samples in $\mathbb{R}^M$.*

- *The objective of GAN is given by a minmax formulation*

$$\min_G \max_D \mathbb{E}_{x \sim p_{data}(x)}[\log D(x)] + \mathbb{E}_{z \sim p_z(z)}[\log(1 - D(G(z))] \ .$$

- *For a fixed G, the optimization objective for D is*

$$\max_D \mathbb{E}_{x \sim p_{data}(x)}[\log D(x)] + \mathbb{E}_{z \sim p_z(z)}[\log(1 - D(G(z))].$$

- *For a fixed D, the optimization objective for G is*

$$\min_G + \mathbb{E}_{z \sim p_z(z)}[\log(1 - D(G(z))],$$

where we omit $\mathbb{E}_{x \sim p_{data}(x)}[\log D(x)]$ since it does not depend on G.

Schematically, training a GAN can be represented by Figure 30.10.1. The latent sample is a random vector z that draws from multivariate Gaussian distribution or uniform

distribution. The generator is a decoder-like neural network that takes the latent sample then generate fake realistic data (i.e., images) and the discriminator is a binary classifier. In the training process, the discriminator network improves its classification performance by minimizing cross-entropy loss; the generator network improves its generation performance by maximizing probability of true image based on discriminator's evaluation.

In the CNN section [subsection 30.6.3], we illustrate the decoder part in an autoencoder neural network can also be used to generate images from a low-dimensional latent input. Could we directly train a decoder network that can map a low-dimensional latent vector to a high-dimensional image? Theoretically we can also do that as long as we have a appropriate loss function to penalize the difference in the generated image distribution and the true image distribution. Clearly, such loss function is not easy to design since we do not even know the true distribution. Indeed, we can view the process of training the discriminator as the process of learning a loss function from data to guide the generator to generate the realistic data sample.

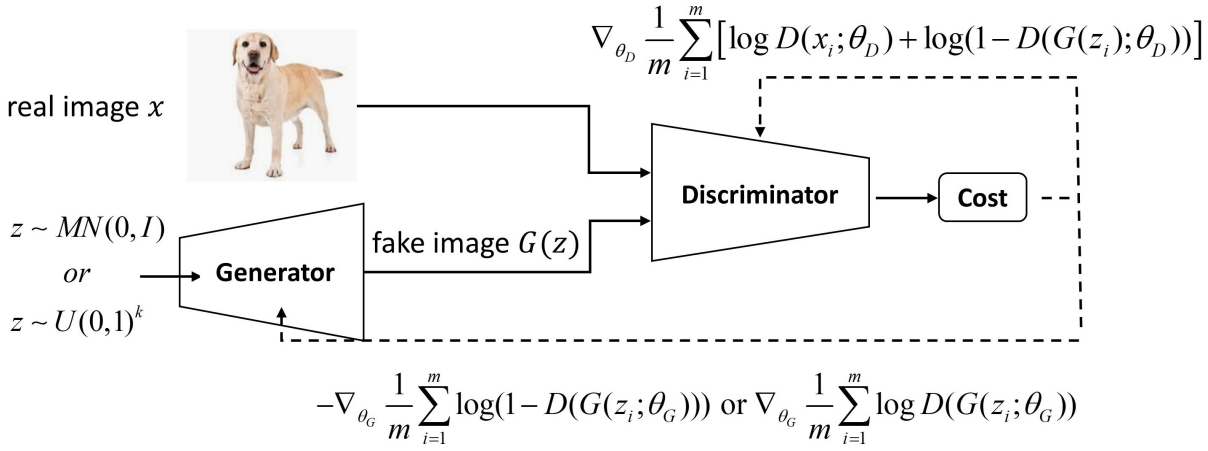


Figure 30.10.1: Scheme for a canonical GAN. The discriminator is trained to distinguish between real and fake image, while the generator is trained to generate realistic image to 'fool' the discriminator. The generator usually uses a decoder-like neural network structure that generate a high-dimensional data from a sample point in the low-dimensional latent space.

Algorithm 66: Minibatch stochastic gradient descent training of GAN.

Set iteration number k for discriminator
for *number of training iterations* **do**
  for *k steps* **do**
    Sample minibatch of m noise samples $\left\{z^{(1)}, \ldots, z^{(m)}\right\}$ from noise prior $p_g(z)$.
    Sample minibatch of m examples $\left\{x^{(1)}, \ldots, x^{(m)}\right\}$ from data geneating distribution $p_{data}(\boldsymbol{x})$.
    Update the discriminatory by ascending its stochastic gradient:

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} \left[\log D\left(x^{(i)}\right) + \log\left(1 - D\left(G\left(z^{(i)}\right)\right)\right) \right]$$

  end
  Sample minibatch of m noise samples $\left\{z^{(1)}, \ldots, z^{(m)}\right\}$ from noise prior $p_g(z)$.
  Update the generator by descending along its stochatic gradient

$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} \log\left(1 - D\left(G\left(z^{(i)}\right)\right)\right)$$

end
Output: Generator network and discriminator network.

Remark 30.10.1 (alternative optimization objective for G). Note that function $\log(1-x)$ approaches 0 when x approaches 0. Such flatness is not suitable for updating G. An alternative optimization objective is $-\log(x)$. Or explicitly,

$$V = \frac{1}{m} \sum_{i=1}^{m} \log(D(G(\boldsymbol{z^i}))).$$

There two objectives are equivalent since $\log(1-x)$ and $-\log(x)$ are both monotonic decreasing function on x.

30.10.1.2 *An example*

In Figure 30.10.2, we demonstrated the generated image from a GAN trained from MNIST. The discriminator and generator neural networks are all feed-forward neural networks.

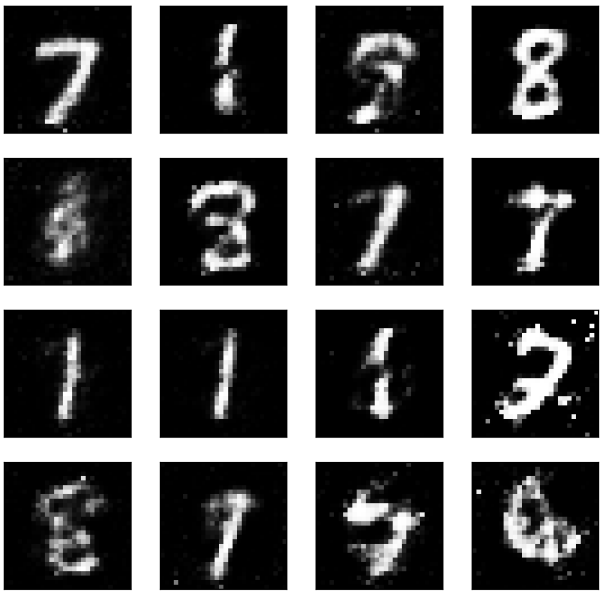


Figure 30.10.2: Generated image samples from a GAN consisting of feed-forward networks.

The neuron units of each layer for the generator networks are 100, 32, 64, 128 and 784. The neuron units of each layer for the discriminator network are 784, 128, 64, 32 and 1. Dropout applies after each layer output and nonlinearity activations between layers are leaky ReLU.

After 100 epoch of training, we can see the generated images do resemble the real MNIST image despite some defects.

30.10.1.3 *Understand training difficulties in GAN*

Training a GAN is generally much more challenging than training a regular neural network. In this section, we will develop some theoretical understanding on the difficulties of training, which further pave the way for improved GAN training techniques[54, 55].

In GAN, the gradient flows to generator networks rely on the classification capability of the discriminator. Surprisingly, an optimal classifier usually does not provides useful gradient flow to improve generator network. Thus an effective training strategy usually involves well-controlled co-adaption of the generator and the discriminator. First, for any generator G, there exists an optimal discriminator D_G^*.

Lemma 30.10.1 (optimal discriminator D for any given generator G). *For a fixed G, the optimal discriminator D is*

$$D_G^*(x) = \frac{p_{data}(x)}{p_{data}(x) + p_g(x)},$$

where p_{data}, p_g are the distribution of the real data and the generated data.

Proof. The training criterion for the discriminator D, given any generator G, is to maximize the quantity

$$\begin{aligned} V(G,D) &= \int_x p_{data}(x)\log(D(x))dx + \int_z p_z(z)\log(1-D(g(z)))dz \\ &= \int_x p_{data}(x)\log(D(x)) + p_g(x)\log(1-D(x))dx \end{aligned}$$

For any $a, b \in \mathbb{R}, a+b \neq 0$, the function $f(y) = a\log(y) + b\log(1-y)$ achieves its maximum in $[0,1]$ at $a/(a+b)$. This can be shown via

$$\frac{\partial f(y)}{\partial y} = \frac{a}{y} - \frac{b}{1-y} = 0 \implies y^* = \frac{a}{a+b}.$$

□

Under the optimal discriminator D_G^*, the loss function used to train the generator G is given by

$$L_G = \mathbb{E}_{x\sim p_{data}} \log \frac{p_{data}(x)}{\frac{1}{2}\left[p_{data}(x) + p_g(x)\right]} + \mathbb{E}_{x\sim p_g} \log \frac{p_g(x)}{\frac{1}{2}\left[p_{data}(x) + P_g(x)\right]} - 2\log 2$$

which can also be written by

$$2JS(p_{data}|p_g) - 2\log(2),$$

where we have used Jensen-Shannon divergence given by

$$JS\left(P_1\|P_2\right) = \frac{1}{2}KL\left(P_1\|\frac{P_1+P_2}{2}\right) + \frac{1}{2}KL\left(P_2\|\frac{P_1+P_2}{2}\right).$$

Here the KL divergence is given by

$$KL\left(P_1\|P_2\right) = \mathbb{E}_{x\sim P_1}\log\frac{P_1}{P_2}.$$

If we have an approximately optimal discriminator such that the loss for the generator is given by the JS divergence, can JS divergence provides enough gradient to drive the

generator to produce desired data distribution? Unfortunately, JS divergence only provides enough gradient when p_g are close to p_{data}. If p_g is nearly disjoint from p_{data}, JS divergence provides vanishing gradients.

In many real-word applications (e.g, image), the distribution of high-dimensional real data p_r generally lies on a low dimensional manifold. The generated data distribution p_g also lies on a lower dimensional manifold because of the architecture setup. In a high-dimensional space, low-dimensional objects are quite likely to be disjoint. When p_r and p_g have disjoint supports, training an optimal discriminator that perfectly separates real and fake samples can be relatively easy, as real and fake samples are likely to have vast differences.

Once the discriminator is trained to close to an optimal one, the loss of the generator reduces to the JS divergence, which has been showed to provide little gradient information to train generators.

Taken together, to successfully train a GAN, the training process for the discriminator and the generator has to be well-balanced. If we train discriminator more than the generator such that the discriminator rapidly proceeds to the optimal discriminator while the p_r and p_g are still far away from each other or disjoint, then the generator can barely receive information from discriminator to improve. As a result, in a well-designed training strategy, we tend to train generator more than the discriminator.

A further severe issue associated with the GAN is that the generated data distribution may collapse to a single or a limited number of modes such that generated data lose diversity in the real data. This is indeed very common for a failed GAN. The root cause is that the loss function from discriminator does not encourage diversity in the generated data.

30.10.1.4 *Deep Convolutional GAN (DCGAN)*

As GAN made it ways into different domains, deep convolutional generative adversarial networks (DCGANs) was one of the marked milestones of generative modeling of images[56].

The generator in DCGAN is an architecture [Figure 30.10.3] that consists of a series of fractionally-strided convolutional layer that maps and expands a small dimensional latent vector z into a high-dimensional image space.

Besides generating realistic images, the authors also presented a way to peek into the latent space. By interpolating a random point z in the latent space, the generated images are smoothly changing from one image to another image [Figure 30.10.4].

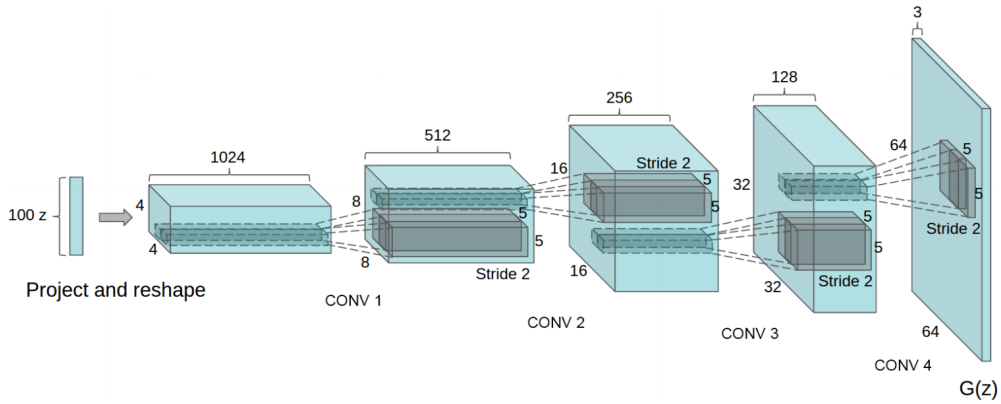


Figure 30.10.3: DCGAN generator network architecture. A 100 dimensional uniform noise is passing through a series of fractionally-strided convolutions then is converted into a final image. Notably, no fully connected or pooling layers are used [56]

Figure 30.10.4: Understanding DCGAN. Each row represents the image generated as we interpolate a random point z in the latent space. Images show smooth transitions from one bedroom and another bedroom.[56]

Because of usual difficulties in training GANs, the authorss also proposed several architecture design guidelines for DCGANs[56]:

- Replace any pooling layers with strided convolutions (discriminator) and fractional-strided convolutions (generator).

- Use batch normalization in both the generator and the discriminator.
- Remove fully connected hidden layers for deeper architectures.
- Use ReLU activation in generator for all layers except for the output, which uses Tanh.
- Use Leaky ReLU activation in the discriminator for all layers.

30.10.2 Conditional GAN

Generative adversarial nets can be extended to a conditional generative model, known as conditional GAN. In a conditional GAN, generators and discriminators are both conditioned on some extra information y. y could be a discrete class labels or a continuous vector values.

In a Conditional GAN, the generator learns to generate a fake sample with the given extra information (e.g., a label) rather than a generic sample from unknown data distribution as GAN. This conditional GAN framework has the following min-max objective function given by

$$\min_G \max_D V(D,G) = \mathbb{E}_{x\sim p_{data}(x)}[\log D(x|y)] + \mathbb{E}_{z\sim p_z(z)}[\log(1 - D(G(z|y)))].$$

Schematically, it can be represented by

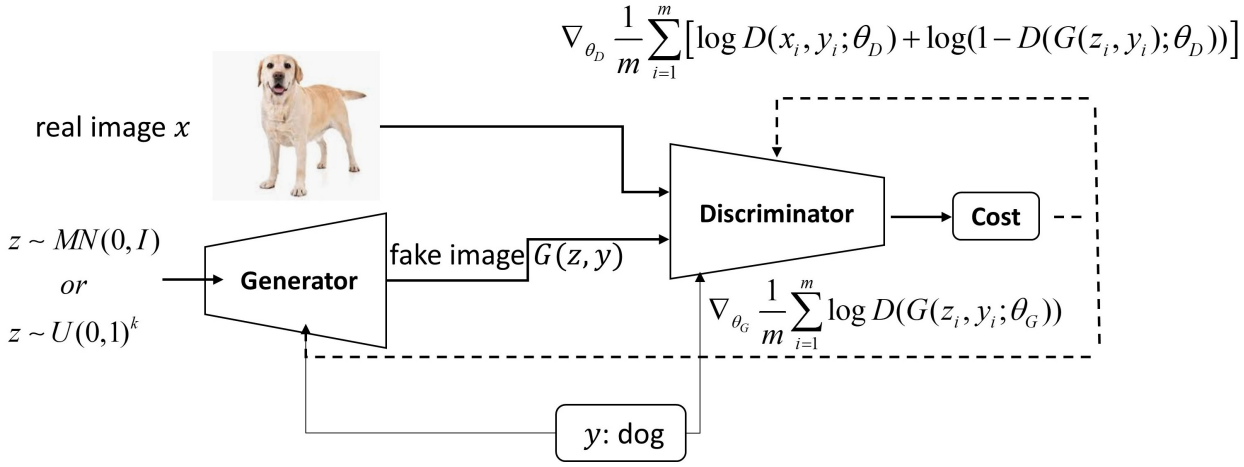


Figure 30.10.5: Scheme for a conditional GAN. The discriminator is trained to distinguish between real and fake image given external label information, while the generator is trained to generate realistic image to 'fool' the discriminator given external label information. The generator usually uses a decoder-like neural network structure that generate a high-dimensional data from a sample point in the low-dimensional latent space.

Algorithm 67: Minibatch stochastic gradient descent training of conditional GAN.

Set iteration number k for discriminator
for *number of training iterations* **do**
  for *k steps* **do**
    Sample minibatch of m noise samples $\left\{z^{(1)}, \ldots, z^{(m)}\right\}$ from noise prior $p_g(z)$.
    Sample minibatch of m samples of conditional inputs $\left\{y^{(1)}, \ldots, y^{(m)}\right\}$
    Sample minibatch of m examples $\left\{x^{(1)}, \ldots, x^{(m)}\right\}$ from conditional data geneating distribution $p_{data}(x|y)$.
    Update the discriminatory by ascending its stochastic gradient:

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} \left[\log D\left(x|y^{(i)}\right) + \log\left(1 - D\left(G\left(z|y^{(i)}\right)\right)\right)\right]$$

  end
  Sample minibatch of m samples of conditional inputs $\left\{y^{(1)}, \ldots, y^{(m)}\right\}$
  Sample minibatch of m noise samples $\left\{z^{(1)}, \ldots, z^{(m)}\right\}$ from noise prior $p_g(z)$.
  Update the generator by descending along its stochatic gradient

$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} \log\left(1 - D\left(G\left(z^{(i)}|y^{(i)}\right)\right)\right)$$

end
Output: Generator network and discriminator network.

30.10.3 Wasserstein GAN (WGAN)

Wasserstain GAN (WGAN) is one of most important improvement on GAN [55], which uses **Wasserstein distance** to quantify the distance between generated data distribution and real data distribution.

Definition 30.10.2 (Wasserstein distance). *The **Wasserstein distance** between two probability density function P_r and P_g is defined by*

$$W\left(p_{data}, p_g\right) = \inf_{\gamma \sim \Pi\left(p_{data}, p_g\right)} \mathbb{E}_{(x,y)\sim\gamma}[\|x-y\|]$$

where $\Pi(p_{data}, p_g)$ is the set of all possible joint distributions for (x, y) that have marginal distribution of $p_{data}(x)$ and $p_g(y)$. For any joint distribution γ, we can draw samples x and y, and thus the expected distance $\|x-y\|$.

Instead of seeking the γ minimizer, which is impractical, WGAN maximizes an alternative objective, based on the Kantorovich Rubinstein duality:

$$W\left(p_{data}, p_g\right) = \frac{1}{K} \sup_{\|f\|_L \leq K} \mathbb{E}_{x\sim p_{data}}[f(x)] - \mathbb{E}_{x\sim p_g}[f(x)]$$

The function f can be approximated by the discriminator neural network. In the canonical GAN, the discriminator is designed to perform binary classification; therefore the last layer is Sigmoid layer. In the WGAN, the discriminator network is to approximate a Lipschitz-smooth function or the Wassertein distance; therefore the last layer is linear layer.

To ensure that the function represented by a neural network satisfies the Lipschitz condition, after every gradient update on the function f, the weights of the network are clamped to a small fixed range. Specifically, the loss function associated with the discriminator is given by

$$L_w = -\left[\frac{1}{m}\sum_{i=1}^{m} f_w\left(x^{(i)}\right) - \frac{1}{m}\sum_{i=1}^{m} f_w\left(G_\theta\left(z^{(i)}\right)\right)\right].$$

And the loss function associated with the generator is given by

$$L_\theta = -\frac{1}{m}\sum_{i=1}^{m} f_w\left(G_\theta\left(z^{(i)}\right)\right).$$

The complete algorithm is in algorithm 68. Empirically the authors also recommended RMSProp optimizer rather than a momentum based optimizer such as Adam, to stabilize the model training.

WGAN has successfully addressed the following issues in GAN [subsubsection 30.10.1.3].

- Training instability. To successfully train a GAN, the training for discriminator and the generator has to been well-balanced.

- Data diversity. The objective function in WGAN encourage the match between real data distribution and generated data distribution, therefore improving the diversity of generated samples.
- Monitoring training process. In canonical GAN, the binary classification loss does not shine light on the quality of generated image. WGAN offers a much better loss function that characterizes the deviation of real data distribution and generated data distribution.

Algorithm 68: Minibatch stochastic gradient descent training of Wasserstein GAN.

Input: weight clipping parameter c, batch size m, iteration number k for discriminator, initial weights θ and w for generator and discriminator networks.

repeat
  for *k steps* **do**
    Sample minibatch of m noise samples $\left\{z^{(1)}, \ldots, z^{(m)}\right\}$ from noise prior $p_g(z)$.
    Sample minibatch of m examples $\left\{x^{(1)}, \ldots, x^{(m)}\right\}$ from data geneating distribution $p_{data}(x)$.
    Update the discriminatory by ascending its stochastic gradient:

$$g_w = \nabla_w \left[\frac{1}{m} \sum_{i=1}^{m} f_w \left(x^{(i)} \right) - \frac{1}{m} \sum_{i=1}^{m} f_w \left(G_\theta \left(z^{(i)} \right) \right) \right].$$

    Clip weight between $[-c, c]$.
  end
  Sample minibatch of m noise samples $\left\{z^{(1)}, \ldots, z^{(m)}\right\}$ from noise prior $p_g(z)$.
  Update the generator by descending along its stochatic gradient

$$g_\theta = -\nabla_\theta \frac{1}{m} \sum_{i=1}^{m} f_w \left(G_\theta \left(z^{(i)} \right) \right).$$

until *generator network converges*;

Output: Generator network and discriminator network.

Weight clipping is certainly not the optimal way to enforce a Lipschitz constraint WGAN still suffers from unstable training, slow convergence after weight clipping. An alternative way to enhance Lipschitz smoothness is to replace weight clipping with gradient penalty[57].

The optimization problem then becomes

$$g_w = \nabla_w \frac{1}{m} \sum_{i=1}^{m} L^{(i)}$$
$$L^{(i)} = f_w(g_\theta(z^{(i)})) - f_w(x^{(i)}) + \lambda \left(\left\| \nabla_x f_w(x^{(i)}) \right\|_2 - 1 \right)^2.$$

30.11 Notes on Bibliography

General books on deep learning include [58].

For a comprehensive review on deep learning methods for text classification, see [59].

Natural language processing, see [60][61].

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. Goyal, P. *et al.* Accurate, large minibatch sgd: Training imagenet in 1 hour. *arXiv preprint arXiv:1706.02677* (2017).
6. Duchi, J., Hazan, E. & Singer, Y. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research* **12,** 2121–2159 (2011).
7. Hinton, G. *Neural Networks for Machine Learning* (University of Toronto, 2012).
8. Zeiler, M. D. ADADELTA: an adaptive learning rate method. *arXiv preprint arXiv:1212.5701* (2012).
9. 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.
10. Kingma, D. P. & Ba, J. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).
11. 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.
12. 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.
13. Ioffe, S. & Szegedy, C. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167* (2015).
14. Loshchilov, I. & Hutter, F. Fixing weight decay regularization in adam. *arXiv preprint arXiv:1711.05101* (2017).

15. 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).

16. Gal, Y. & Ghahramani, Z. *A theoretically grounded application of dropout in recurrent neural networks* in *Advances in neural information processing systems* (2016), 1019–1027.

17. 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.

18. Mikolov, T., Chen, K., Corrado, G. & Dean, J. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781* (2013).

19. Mikolov, T., Sutskever, I., Chen, K., Corrado, G. S. & Dean, J. *Distributed representations of words and phrases and their compositionality* in *Advances in neural information processing systems* (2013), 3111–3119.

20. 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).

21. Raissi, M. & Karniadakis, G. E. Hidden physics models: Machine learning of nonlinear partial differential equations. *Journal of Computational Physics* **357,** 125–141 (2018).

22. 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).

23. Chassagneux, J.-F., Chotai, H. & Muûls, M. *Introduction to Forward-Backward Stochastic Differential Equations* 11–42 (Springer, 2017).

24. 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).

25. Krizhevsky, A., Sutskever, I. & Hinton, G. E. *Imagenet classification with deep convolutional neural networks* in *Advances in neural information processing systems* (2012), 1097–1105.

26. Simonyan, K. & Zisserman, A. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556* (2014).

27. 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.

28. Srivastava, R. K., Greff, K. & Schmidhuber, J. Highway networks. *arXiv preprint arXiv:1505.00387* (2015).

29. Zeiler, M. D. & Fergus, R. *Visualizing and understanding convolutional networks* in *European conference on computer vision* (2014), 818–833.

30. Erhan, D., Bengio, Y., Courville, A. & Vincent, P. Visualizing higher-layer features of a deep network. *University of Montreal* **1341,** 1 (2009).

31. Ozbulak, U. *PyTorch CNN Visualizations* https://github.com/utkuozbulak/pytorch-cnn-visualizations. 2019.

32. 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.

33. Hinton, G. E. & Salakhutdinov, R. R. Reducing the dimensionality of data with neural networks. *science* **313,** 504–507 (2006).

34. Odena, A., Dumoulin, V. & Olah, C. Deconvolution and checkerboard artifacts. *Distill* **1,** e3 (2016).

35. 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).

36. 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.

37. Silver, D. *et al.* Mastering the game of Go with deep neural networks and tree search. *nature* **529,** 484 (2016).

38. Mnih, V. *et al.* Human-level control through deep reinforcement learning. *Nature* **518,** 529 (2015).

39. 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).

40. Popova, M., Isayev, O. & Tropsha, A. Deep reinforcement learning for de novo drug design. *Science advances* **4,** eaap7885 (2018).

41. Li, J., Ávila, B. E.-f. D., Gao, W., Zhang, L. & Wang, J. Micro / nanorobots for biomedicine : Delivery , surgery , sensing , and detoxification (2017).

42. Wang, J. & Gao, W. Nano/microscale motors: Biomedical opportunities and challenges. *ACS Nano* **6,** 5745–5751. ISSN: 19360851 (2012).

43. Kim, Y. Convolutional neural networks for sentence classification. *arXiv preprint arXiv:1408.5882* (2014).

44. Haykin, S. *Neural Networks and Learning Machines, 3/E* (Pearson Education India, 2010).

45. Qiu, X. *Neural Network and Deep Learning* (Fudan University, 2019).

46. Hochreiter, S. & Schmidhuber, J. Long short-term memory. *Neural computation* **9,** 1735–1780 (1997).
47. Chung, J., Gulcehre, C., Cho, K. & Bengio, Y. Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv preprint arXiv:1412.3555* (2014).
48. Salinas, D., Flunkert, V., Gasthaus, J. & Januschowski, T. DeepAR: Probabilistic forecasting with autoregressive recurrent networks. *International Journal of Forecasting* (2019).
49. Wang, Y. *et al.* Deep factors for forecasting. *arXiv preprint arXiv:1905.12417* (2019).
50. Sutskever, I., Vinyals, O. & Le, Q. V. *Sequence to sequence learning with neural networks* in *Advances in neural information processing systems* (2014), 3104–3112.
51. Bahdanau, D., Cho, K. & Bengio, Y. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473* (2014).
52. Luong, M.-T., Pham, H. & Manning, C. D. Effective approaches to attention-based neural machine translation. *arXiv preprint arXiv:1508.04025* (2015).
53. Goodfellow, I. *et al. Generative adversarial nets* in *Advances in neural information processing systems* (2014), 2672–2680.
54. Salimans, T. *et al. Improved techniques for training gans* in *Advances in neural information processing systems* (2016), 2234–2242.
55. Arjovsky, M., Chintala, S. & Bottou, L. Wasserstein gan. *arXiv preprint arXiv:1701.07875* (2017).
56. Radford, A., Metz, L. & Chintala, S. Unsupervised representation learning with deep convolutional generative adversarial networks. *arXiv preprint arXiv:1511.06434* (2015).
57. Gulrajani, I., Ahmed, F., Arjovsky, M., Dumoulin, V. & Courville, A. C. *Improved training of wasserstein gans* in *Advances in Neural Information Processing Systems* (2017), 5767–5777.
58. Goodfellow, I., Bengio, Y., Courville, A. & Bengio, Y. *Deep learning* (MIT press Cambridge, 2016).
59. Minaee, S. *et al.* Deep Learning Based Text Classification: A Comprehensive Review. *arXiv preprint arXiv:2004.03705* (2020).
60. Manning, C. D. & Schütze, H. *Foundations of statistical natural language processing* (MIT press, 1999).
61. Goldberg, Y. Neural network methods for natural language processing. *Synthesis Lectures on Human Language Technologies* **10,** 1–309 (2017).

Part VI

OPTIMAL CONTROL AND REINFORCEMENT LEARNING METHODS