3. Neural Networks and Backpropagation

3.1 Neural Network

A neural network is a network or circuit of biological neurons, or, in a modern sense, an artificial neural network, composed of artificial neurons or nodes. Thus, a neural network is either a biological neural network, made up of biological neurons, or an artificial neural network, used for solving artificial intelligence (AI) problems. **The connections of the biological neuron are modelled in artificial neural networks as weights between nodes. A positive weight reflects an excitatory connection, while negative values mean inhibitory connections.** (Figure 3.1) All inputs are modified by a weight and summed. This activity is referred to as a linear combination. Finally, an activation function controls the amplitude of the output. For example, an acceptable range of output is usually between 0 and 1, or it could be 1 and 1.

These artificial networks may be used for predictive modelling, adaptive control and applications where they can be trained via a dataset. Self-learning resulting from experience can occur within networks, which can derive conclusions from a complex and seemingly unrelated set of information.

阅读后结合上课内容需对神经网络的基本结构有充分的理解。

3.2 History

The preliminary theoretical base for contemporary neural networks was independently proposed by Alexander Bain (1873) and William James (1890). In their work, both thoughts and body activity resulted from interactions among neurons within the brain.

For Bain, every activity led to the firing of a certain set of neurons. When activities were

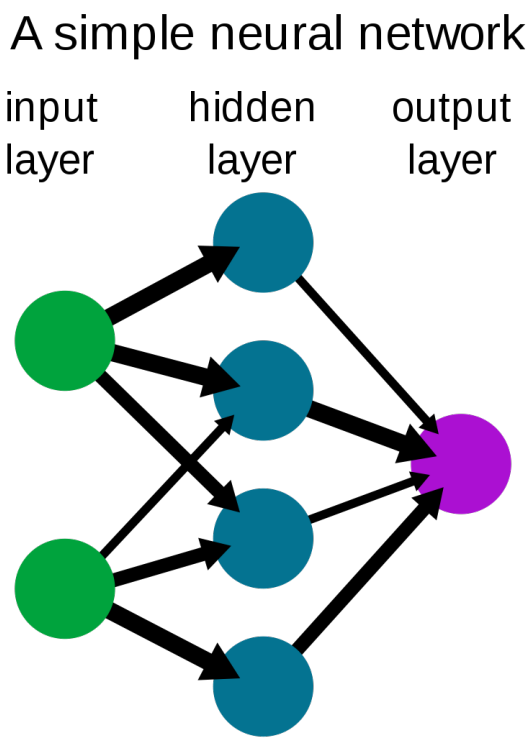


Figure 3.1: Simplified view of a feedforward artificial neural network.

repeated, the connections between those neurons strengthened. According to his theory, this repetition was what led to the formation of memory. The general scientific community at the time was skeptical of Bain's theory because it required what appeared to be an inordinate number of neural connections within the brain. It is now apparent that the brain is exceedingly complex and that the same brain "wiring" can handle multiple problems and inputs.

James's theory was similar to Bain's, however, he suggested that memories and actions resulted from electrical currents flowing among the neurons in the brain. His model, by focusing on the flow of electrical currents, did not require individual neural connections for each memory or action.

C. S. Sherrington (1898) conducted experiments to test James's theory. He ran electrical currents down the spinal cords of rats. However, instead of demonstrating an increase in electrical current as projected by James, Sherrington found that the electrical current strength decreased as the testing continued over time. Importantly, this work led to the discovery of the concept of habituation.

McCulloch and Pitts (1943) created a computational model for neural networks based on mathematics and algorithms. They called this model threshold logic. The model paved the way for

neural network research to split into two distinct approaches. One approach focused on biological processes in the brain and the other focused on the application of neural networks to artificial intelligence.

In the late 1940s psychologist Donald Hebb created a hypothesis of learning based on the mechanism of neural plasticity that is now known as Hebbian learning. Hebbian learning is considered to be a 'typical' unsupervised learning rule and its later variants were early models for long term potentiation. These ideas started being applied to computational models in 1948 with Turing's B-type machines.

Farley and Clark (1954) first used computational machines, then called calculators, to simulate a Hebbian network at MIT. Other neural network computational machines were created by Rochester, Holland, Habit, and Duda (1956).

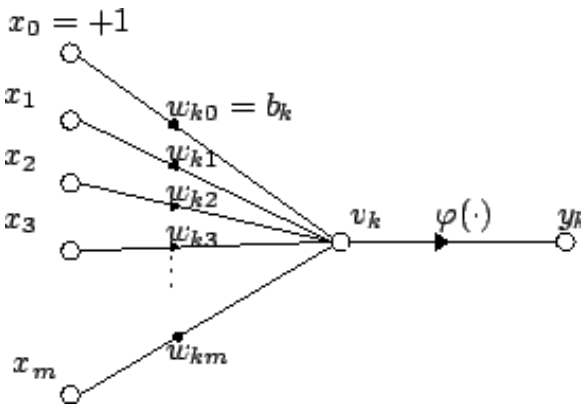
Rosenblatt (1958) created the perceptron, an algorithm for pattern recognition based on a two-layer learning computer network using simple addition and subtraction. With mathematical notation, Rosenblatt also described circuitry not in the basic perceptron, such as the exclusive-or circuit, a circuit whose mathematical computation could not be processed until after the backpropagation algorithm was created by Werbos (1975).

Neural network research stagnated after the publication of machine learning research by Marvin Minsky and Seymour Papert (1969). They discovered two key issues with the computational machines that processed neural networks. The first issue was that single-layer neural networks were incapable of processing the exclusive-or circuit. The second significant issue was that computers were not sophisticated enough to effectively handle the long run time required by large neural networks. Neural network research slowed until computers achieved greater processing power. Also key in later advances was the backpropagation algorithm which effectively solved the exclusive-or problem (Werbos 1975).

The parallel distributed processing of the mid-1980s became popular under the name connectionism. The text by Rumelhart and McClelland (1986) provided a full exposition on the use of connectionism in computers to simulate neural processes.

Neural networks, as used in artificial intelligence, have traditionally been viewed as simplified models of neural processing in the brain, even though the relation between this model and brain biological architecture is debated, as it is not clear to what degree artificial neural networks mirror brain function.

本节神经网络技术发展历程为拓展知识，便于同学们对神经网络这一范畴于人工智能领域有宏观认知。



3.3 Artificial neuron

An artificial neuron is a mathematical function conceived as a model of biological neurons, a neural network. Artificial neurons are elementary units in an artificial neural network. The artificial neuron receives one or more inputs (representing excitatory postsynaptic potentials and inhibitory postsynaptic potentials at neural dendrites) and sums them to produce an output (or activation, representing a neuron's action potential which is transmitted along its axon). Usually each input is separately weighted, and the sum is passed through a non-linear function known as an activation function or transfer function. The transfer functions usually have a sigmoid shape, but they may also take the form of other non-linear functions, piecewise linear functions, or step functions. They are also often monotonically increasing, continuous, differentiable and bounded. Non-monotonic, unbounded and oscillating activation functions with multiple zeros that outperform sigmoidal and ReLU like activation functions on many tasks have also been recently explored. The thresholding function has inspired building logic gates referred to as threshold logic; applicable to building logic circuits resembling brain processing. For example, new devices such as memristors have been extensively used to develop such logic in recent times.

The artificial neuron transfer function should not be confused with a linear system's transfer function.

For a given artificial neuron k, let there be $m+1$ inputs with signals x_0 through x_m and weights w_{k0} through w_{km}. Usually, the x_0 input is assigned the value $+1$, which makes it a bias input with $w_{k0} = b_k$. This leaves only m actual inputs to the neuron: from x_1 to x_m.

The output of the kth neuron is:

$$y_k = \varphi\left(\sum_{j=0}^{m} w_{kj}x_j\right)$$

Where φ is the transfer function (commonly a threshold function).

The output is analogous to the axon of a biological neuron, and its value propagates to the input of the next layer, through a synapse. It may also exit the system, possibly as part of an output vector. It has no learning process as such. Its transfer function weights are calculated and threshold value are predetermined.

阅读后结合上课内容需对神经网络的基础结构有充分的理解与掌握。结合生物神经学中对神经元的结构研究，加深对**线性组合 + 非线性映射**这种模式的理解。

3.4 Transfer Functions

The transfer function (activation function) of a neuron is chosen to have a number of properties which either enhance or simplify the network containing the neuron. Crucially, for instance, any multilayer perceptron using a linear transfer function has an equivalent single-layer network; a non-linear function is therefore necessary to gain the advantages of a multi-layer network.

Below, u refers in all cases to the weighted sum of all the inputs to the neuron,

$$u = \sum_{i=1}^{n} w_i x_i$$

where w is a vector of synaptic weights and x is a vector of inputs.

Step function

The output y of this transfer function is binary, depending on whether the input meets a specified threshold, θ. The "signal" is sent, *i.e.*, the output is set to one, if the activation meets the threshold.

$$y = \begin{cases} 1 & \text{if } u \geq \theta \\ 0 & \text{if } u < \theta \end{cases}$$

This function is used in perceptrons and often shows up in many other models. It performs a division of the space of inputs by a hyperplane. It is specially useful in the last layer of a network intended to perform binary classification of the inputs. It can be approximated from other sigmoidal functions by assigning large values to the weights.

Linear combination

In this case, the output unit is simply the weighted sum of its inputs plus a bias term. A number of such linear neurons perform a linear transformation of the input vector. This is usually more useful in the first layers of a network. A number of analysis tools exist based on linear models, such as harmonic analysis, and they can all be used in neural networks with this linear neuron. The bias term allows us to make affine transformations to the data.

Sigmoid

A fairly simple non-linear function, the sigmoid function such as the logistic function also has an easily calculated derivative, which can be important when calculating the weight updates in the network. It thus makes the network more easily manipulable mathematically, and was attractive to early computer scientists who needed to minimise the computational load of their simulations. It was previously commonly seen in multilayer perceptrons. However, recent work has shown sigmoid neurons to be less effective than rectified linear neurons. The reason is that the gradients computed by the backpropagation algorithm tend to diminish towards zero as activations propagate through layers of sigmoidal neurons, making it difficult to optimise neural networks using multiple layers of sigmoidal neurons.

Rectifier

In the context of artificial neural networks, the rectifier is an activation function defined as the positive part of its argument:

$$f(x) = x^+ = \max(0, x)$$

where x is the input to a neuron. This is also known as a ramp function and is analogous to half-wave rectification in electrical engineering. This activation function was first introduced to a dynamical network by Hahnloser et al. in a 2000 paper in Nature with strong biological motivations and mathematical justifications. It has been demonstrated for the first time in 2011 to enable better training of deeper networks, compared to the widely used activation functions prior to 2011, *i.e.*, the logistic sigmoid (which is inspired by probability theory) and its more practical counterpart, the hyperbolic tangent.

阅读后结合上课内容需对常见的转移函数有直观认识与理解。

3.5 Gradient Descent

Backpropagation computes the gradient in parameter space|weight space of a feedforward neural network, with respect to a loss function. Denote:

x: input (vector of features)

y: target output. For classification, output will be a vector of class probabilities (e.g., $(0.1, 0.7, 0.2)$, and target output is a specific class, encoded by the one-hot/Dummy variable (e.g., $(0, 1, 0)$).

C: loss function or "cost function". For classification, this is usually cross entropy, while for regression it is usually squared error loss.

L: the number of layers. $W^l = (w^l_{jk})$: the weights between layer $l-1$ and l, where w^l_{jk} is the weight between the k-th node in layer $l-1$ and the j-th node in layer lThis means (left) multiplication

by the matrix W^l corresponds to converting output values of layer $l-1$ to input values of layer l: columns correspond to input coordinates, rows correspond to output coordinates.
f^l: activation functions at layer l. For classification the last layer is usually the logistic function for binary classification, and softmax function (softargmax) for multi-class classification, while for the hidden layers this was traditionally a sigmoid function (logistic function or others) on each node (coordinate), but today is more varied, with Rectifier (ReLU) being common.

In the derivation of backpropagation, other intermediate quantities are used; they are introduced as needed below. Bias terms are not treated specially, as they correspond to a weight with a fixed input of 1. For the purpose of backpropagation, the specific loss function and activation functions do not matter, as long as they and their derivatives can be evaluated efficiently. Traditional activation functions include but are not limited to sigmoid, tanh, and ReLU.

The overall network is a combination of function composition and matrix multiplication:

$$g(x) := f^L(W^L f^{L-1}(W^{L-1} \cdots f^1(W^1 x) \cdots))$$

For a training set there will be a set of input–output pairs, $\{(x_i, y_i)\}$. For each input–output pair (x_i, y_i) in the training set, the loss of the model on that pair is the cost of the difference between the predicted output $g(x_i)$ and the target output y_i: $C(y_i, g(x_i))$

Note the distinction: during model evaluation, the weights are fixed, while the inputs vary (and the target output may be unknown), and the network ends with the output layer (it does not include the loss function). During model training, the input–output pair is fixed, while the weights vary, and the network ends with the loss function.

Backpropagation computes the gradient for a "fixed" input–output pair (x_i, y_i), where the weights w^l_{jk} can vary. Each individual component of the gradient, $\partial C/\partial w^l_{jk}$, can be computed by the chain rule; however, doing this separately for each weight is inefficient. Backpropagation efficiently computes the gradient by avoiding duplicate calculations and not computing unnecessary intermediate values, by computing the gradient of each layer –specifically, the gradient of the weighted "input" of each layer, denoted by δ^l –from back to front.

Informally, the key point is that since the only way a weight in W^l affects the loss is through its effect on the "next" layer, and it does so "linearly", δ^l are the only data you need to compute the gradients of the weights at layer l, and then you can compute the previous layer δ^{l-1} and repeat recursively. This avoids inefficiency in two ways. Firstly, it avoids duplication because when computing the gradient at layer l, you do not need to recompute all the derivatives on later layers $l+1, l+2, \ldots$ each time. Secondly, it avoids unnecessary intermediate calculations because at each stage it directly computes the gradient of the weights with respect to the ultimate output (the loss), rather than unnecessarily computing the derivatives of the values of hidden layers with respect to changes in weights $\partial a^{l'}_{j'}/\partial w^l_{jk}$.

Backpropagation can be expressed for simple feedforward networks in terms of matrix multiplication, or more generally in terms of the adjoint graph.

Matrix multiplication

For the basic case of a feedforward network, where nodes in each layer are connected only to nodes in the immediate next layer (without skipping any layers), and there is a loss function that computes a scalar loss for the final output, backpropagation can be understood simply by matrix multiplication. Essentially, backpropagation evaluates the expression for the derivative of the cost function as a product of derivatives between each layer from left to right –"backwards" –with the gradient of the weights between each layer being a simple modification of the partial products (the "backwards propagated error").

Given an input–output pair (x, y), the loss is:

$$C(y, f^L(W^L f^{L-1}(W^{L-1} \cdots f^2(W^2 f^1(W^1 x)) \cdots)))$$

To compute this, one starts with the input x and works forward; denote the weighted input of each layer as z^l and the output of layer l as the activation a^l. For backpropagation, the activation a^l as well as the derivatives $(f^l)'$ (evaluated at z^l) must be cached for use during the backwards pass.

The derivative of the loss in terms of the inputs is given by the chain rule; note that each term is a total derivative, evaluated at the value of the network (at each node) on the input x:

$$\frac{dC}{da^L} \circ \frac{da^L}{dz^L} \cdot \frac{dz^L}{da^{L-1}} \circ \frac{da^{L-1}}{dz^{L-1}} \cdot \frac{dz^{L-1}}{da^{L-2}} \cdots \frac{da^1}{dz^1} \cdot \frac{\partial z^1}{\partial x},$$

where $\circ$ is a Hadamard product, that is an element-wise product, and the products are taken from left to right.

These terms are: the derivative of the loss function; the derivatives of the activation functions; and the matrices of weights:

$$\frac{dC}{da^L} \circ (f^L)' \cdot W^L \circ (f^{L-1})' \cdot W^{L-1} \cdots (f^1)' \cdot W^1.$$

The gradient ∇ is the transpose of the derivative of the output in terms of the input, so the matrices are transposed and the order of multiplication is reversed, but the entries are the same:

$$\nabla_x C = (W^1)^T \cdot (f^1)' \ldots \circ (W^{L-1})^T \cdot (f^{L-1})' \circ (W^L)^T \cdot (f^L)' \circ \nabla_{a^L} C.$$

Backpropagation then consists essentially of evaluating this expression from right to left (equivalently, multiplying the previous expression for the derivative from left to right), computing the gradient at each layer on the way; there is an added step, because the gradient of the weights isn't just a subexpression: there's an extra multiplication.

Introducing the auxiliary quantity δ^l for the partial products (multiplying from right to left), interpreted as the "error at level l" and defined as the gradient of the input values at level l:

$$\delta^l := (f^l)' \circ (W^{l+1})^T \cdots \circ (W^{L-1})^T \cdot (f^{L-1})' \circ (W^L)^T \cdot (f^L)' \circ \nabla_{a^L} C.$$

Note that δ^l is a vector, of length equal to the number of nodes in level l; each component is interpreted as the "cost attributable to (the value of) that node".

The gradient of the weights in layer l is then:

$$\nabla_{W^l} C = \delta^l (a^{l-1})^T . \nabla_{W^l} C = \delta^l (a^{l-1})^T .$$

The factor of a^{l-1} is because the weights W^l between level $l-1$ and l affect level l proportionally to the inputs (activations): the inputs are fixed, the weights vary.

The δ^l can easily be computed recursively, going from right to left, as:

$$\delta^{l-1} := (f^{l-1})' \circ (W^l)^T \cdot \delta^l.$$

The gradients of the weights can thus be computed using a few matrix multiplications for each level; this is backpropagation.

Compared with naively computing forwards (using the δ^l for illustration):

$$\begin{aligned}
\delta^1 &= (f^1)' \circ (W^2)^T \cdot (f^2)' \cdots \circ (W^{L-1})^T \cdot (f^{L-1})' \circ (W^L)^T \cdot (f^L)' \circ \nabla_{a^L} C \\
\delta^2 &= (f^2)' \cdots \circ (W^{L-1})^T \cdot (f^{L-1})' \circ (W^L)^T \cdot (f^L)' \circ \nabla_{a^L} C \\
&\vdots \\
\delta^{L-1} &= (f^{L-1})' \circ (W^L)^T \cdot (f^L)' \circ \nabla_{a^L} C \\
\delta^L &= (f^L)' \circ \nabla_{a^L} C,
\end{aligned}$$

there are two key differences with backpropagation:

Computing δ^{l-1} in terms of δ^l avoids the obvious duplicate multiplication of layers l and beyond. Multiplying starting from $\nabla_{a^L} C$ –propagating the error backwards –means that each step simply multiplies a vector (δ^l by the matrices of weights $(W^l)^T$ and derivatives of activations $(f^{l-1})'$. By contrast, multiplying forwards, starting from the changes at an earlier layer, means that each multiplication multiplies a matrix by a matrix. This is much more expensive, and corresponds to tracking every possible path of a change in one layer l forward to changes in the layer $l+2$ (for multiplying W^{l+1} by W^{l+2}, with additional multiplications for the derivatives of the activations), which unnecessarily computes the intermediate quantities of how weight changes affect the values of hidden nodes.

阅读后结合上课内容达到对多层神经网络的反向传播进行推导的能力，以矩阵/向量的视角看待变量在前馈与反向传播中的计算方式。推荐此部分的外部参考资源于 B 站 https://www.bilibili.com/video/BV15E41117FQ?share_source=copy_web

3.6 Autoencoder

An autoencoder is a type of artificial neural network used to learn efficient codings of unlabeled data (unsupervised learning). The encoding is validated and refined by attempting to regenerate the input from the encoding. The autoencoder learns a representation (encoding) for a set of data, typically for dimensionality reduction, by training the network to ignore insignificant data (“noise”).

Variants exist, aiming to force the learned representations to assume useful properties. Examples are regularised autoencoders (Sparse, Denoising and Contractive), which are effective in learning representations for subsequent classification tasks, and Variational autoencoders, with applications as generative models. Autoencoders are applied to many problems, from facial recognition, feature detection, anomaly detection to acquiring the meaning of words. Autoencoders are also generative models: they can randomly generate new data that is similar to the input data (training data).

Basic Structure

An autoencoder has two main parts: an encoder that maps the input into the code, and a decoder that maps the code to a reconstruction of the input.

The simplest way to perform the copying task perfectly would be to duplicate the signal. Instead, autoencoders are typically forced to reconstruct the input approximately, preserving only the most relevant aspects of the data in the copy.

The idea of autoencoders has been popular for decades. The first applications date to the 1980s. Their most traditional application was dimensionality reduction or feature learning, but the concept became widely used for learning generative models of data. Some of the most powerful AIs in the 2010s involved autoencoders stacked inside deep neural networks.

The simplest form of an autoencoder is a feedforward, non-recurrent neural network similar to single layer perceptrons that participate in multilayer perceptrons (MLP) –employing an input layer and an output layer connected by one or more hidden layers. The output layer has the same number of nodes (neurons) as the input layer. Its purpose is to reconstruct its inputs (minimising the difference between the input and the output) instead of predicting a target value Y given inputs X. Therefore, autoencoders learn unsupervised.

An autoencoder consists of two parts, the encoder and the decoder, which can be defined as transitions ϕ and ψ, such that:

$$\phi : \mathscr{X} \to \mathscr{F}$$

$$\psi : \mathscr{F} \to \mathscr{X}$$

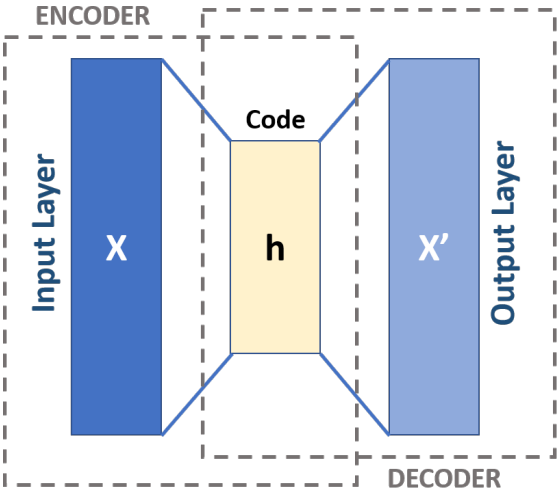


Figure 3.2: Schema of a basic Autoencoder.

$$\phi, \psi = \arg\min_{\phi,\psi} \| \mathscr{X} - (\psi \circ \phi) \mathscr{X} \|^2$$

In the simplest case, given one hidden layer, the encoder stage of an autoencoder takes the input $\mathbf{x} \in \mathbb{R}^d = \mathscr{X}$ and maps it to $\mathbf{h} \in \mathbb{R}^p = \mathscr{F}$:

$$\mathbf{h} = \sigma(\mathbf{W}\mathbf{x} + \mathbf{b})$$

This image $\mathbf{h}$ is usually referred to as code, latent variables, or a latent representation. σ is an element-wise activation function such as a sigmoid function or a rectified linear unit. $\mathbf{W}$ is a weight matrix and $\mathbf{b}$ is a bias vector. Weights and biases are usually initialised randomly, and then updated iteratively during training through backpropagation. After that, the decoder stage of the autoencoder maps $\mathbf{h}$ to the reconstruction $\mathbf{x}'$ of the same shape as $\mathbf{x}$:

$$\mathbf{x}' = \sigma'(\mathbf{W}'\mathbf{h} + \mathbf{b}')$$

where $\sigma', \mathbf{W}'$, and $\mathbf{b}'$ for the decoder may be unrelated to the corresponding $\sigma, \mathbf{W}$, and $\mathbf{b}$ for the encoder.

Autoencoders are trained to minimise reconstruction errors (such as squared errors), often referred to as the "loss":

$$\mathscr{L}(\mathbf{x}, \mathbf{x}') = \|\mathbf{x} - \mathbf{x}'\|^2 = \|\mathbf{x} - \sigma'(\mathbf{W}'(\sigma(\mathbf{W}\mathbf{x} + \mathbf{b})) + \mathbf{b}')\|^2$$

where $\mathbf{x}$ is usually averaged over the training set.

As mentioned before, autoencoder training is performed through backpropagation of the error, just like other feedforward neural networks.

Should the feature space $\mathscr{F}$ have lower dimensionality than the input space $\mathscr{X}$, the feature vector $\phi(x)$ can be regarded as a compressed representation of the input x. This is the case of undercomplete autoencoders. If the hidden layers are larger than (overcomplete), or equal to, the input layer, or the hidden units are given enough capacity, an autoencoder can potentially learn the identity function and become useless. However, experimental results found that overcomplete autoencoders might still learn useful features. In the ideal setting, the code dimension and the model capacity could be set on the basis of the complexity of the data distribution to be modeled. One way to do so is to exploit the model variants known as Regularised Autoencoders.

Denoising autoencoder (DAE)

Denoising autoencoders (DAE) try to achieve a good representation by changing the reconstruction criterion.

Indeed, DAEs take a partially corrupted input and are trained to recover the original undistorted input. In practice, the objective of denoising autoencoders is that of cleaning the corrupted input, or denoising. Two assumptions are inherent to this approach: 1. Higher level representations are relatively stable and robust to the corruption of the input; 2. To perform denoising well, the model needs to extract features that capture useful structure in the input distribution. In other words, denoising is advocated as a training criterion for learning to extract useful features that will constitute better higher level representations of the input.

The training process of a DAE works as follows:

The initial input x is corrupted into $\tilde{x}$ through stochastic mapping $\tilde{x} \sim q_D(\tilde{x}|x)$. The corrupted input $\tilde{x}$ is then mapped to a hidden representation with the same process of the standard autoencoder, $h = f_\theta(\tilde{x}) = s(W\tilde{x}+b)$. From the hidden representation the model reconstructs $z = g_{\theta'}(h)$. The model's parameters θ and θ' are trained to minimise the average reconstruction error over the training data, specifically, minimising the difference between z and the original uncorrupted input x. Note that each time a random example x is presented to the model, a new corrupted version is generated stochastically on the basis of $q_D(\tilde{x}|x)$.

The above-mentioned training process could be applied with any kind of corruption process. Some examples might be additive isotropic Gaussian noise, masking noise (a fraction of the input chosen at random for each example is forced to 0) or salt-and-pepper noise (a fraction of the input chosen at random for each example is set to its minimum or maximum value with uniform probability).

The corruption of the input is performed only during training. After training, no corruption is added.

Sparse autoencoder (SAE)

Learning representations in a way that encourages sparsity improves performance on classification tasks. Sparse autoencoders may include more (rather than fewer) hidden units than inputs, but only a small number of the hidden units are allowed to be active at the same time (thus, sparse). This constraint forces the model to respond to the unique statistical features of the training data.

Specifically, a sparse autoencoder is an autoencoder whose training criterion involves a sparsity penalty $\Omega(h)$ on the code layer h.

$$\mathscr{L}(\mathbf{x},\mathbf{x}') + \Omega(h)$$

Recalling that $h = f(Wx+b)$, the penalty encourages the model to activate (i.e. output value close to 1) specific areas of the network on the basis of the input data, while inactivating all other neurons (i.e. to have an output value close to 0).

This sparsity can be achieved by formulating the penalty terms in different ways.

One way is to exploit the Kullback-Leibler (KL) divergence. Let

$$\hat{\rho}_j = \frac{1}{m}\sum_{i=1}^{m}[h_j(x_i)]$$

be the average activation of the hidden unit j (averaged over the m training examples). The notation $h_j(x_i)$ identifies the input value that triggered the activation. To encourage most of the neurons to be inactive, $\hat{\rho}_j$ needs to be close to 0. Therefore, this method enforces the constraint $\hat{\rho}_j = \rho$ where ρ is the sparsity parameter, a value close to zero. The penalty term $\Omega(h)$ takes a form that penalises $\hat{\rho}_j$ for deviating significantly from ρ, exploiting the KL divergence:

$$\sum_{j=1}^{s} KL(\rho||\hat{\rho}_j) = \sum_{j=1}^{s}\left[\rho\log\frac{\rho}{\hat{\rho}_j} + (1-\rho)\log\frac{1-\rho}{1-\hat{\rho}_j}\right]$$

where j is summing over the s hidden nodes in the hidden layer, and $KL(\rho||\hat{\rho}_j)$ is the KL-divergence between a Bernoulli random variable with mean ρ and a Bernoulli random variable with mean $\hat{\rho}_j$.

Another way to achieve sparsity is by applying L1 or L2 regularisation terms on the activation, scaled by a certain parameter λ. For instance, in the case of L1 the loss function becomes

$$\mathscr{L}(\mathbf{x},\mathbf{x}') + \lambda\sum_{j}|h_i|$$

A further proposed strategy to force sparsity is to manually zero all but the strongest hidden unit activations (k-sparse autoencoder). The k-sparse autoencoder is based on a linear autoencoder (i.e. with linear activation function) and tied weights. The identification of the strongest activations can be achieved by sorting the activities and keeping only the first k values, or by using ReLU hidden units with thresholds that are adaptively adjusted until the k largest activities are identified.

This selection acts like the previously mentioned regularisation terms in that it prevents the model from reconstructing the input using too many neurons.

Contractive autoencoder (CAE)

A contractive autoencoder adds an explicit regulariser in its objective function that forces the model to learn an encoding robust to slight variations of input values. This regulariser corresponds to the Frobenius norm of the Jacobian matrix of the encoder activations with respect to the input. Since the penalty is applied to training examples only, this term forces the model to learn useful information about the training distribution. The final objective function has the following form:

$$\mathscr{L}(\mathbf{x},\mathbf{x}') + \lambda \sum_i ||\nabla_x h_i||^2$$

The autoencoder is termed contractive because it is encouraged to map a neighborhood of input points to a smaller neighborhood of output points.

DAE is connected to CAE: in the limit of small Gaussian input noise, DAEs make the reconstruction function resist small but finite-sized input perturbations, while CAEs make the extracted features resist infinitesimal input perturbations.

Variational autoencoder (VAE)

Variational autoencoders (VAEs) belong to the families of variational Bayesian methods. Despite the architectural similarities with basic autoencoders, VAEs are architecture with different goals and with a completely different mathematical formulation. The latent space is in this case composed by a mixture of distributions instead of a fixed vector.

Given an input dataset $\mathbf{x}$ characterised by an unknown probability function $P(\mathbf{x})$ and a multivariate latent encoding vector $\mathbf{z}$, the objective is to model the data as a distribution $p_\theta(\mathbf{x})$, with θ defined as the set of the network parameters so that $p_\theta(\mathbf{x}) = \int_{\mathbf{z}} p_\theta(\mathbf{x},\mathbf{z}) d\mathbf{z}$.

阅读后结合上课内容掌握自编码器基本结构、学习方式、和主要性质，其中对后续若干变体（降噪自编码器、稀疏自编码器、对比自编码器、变分自编码器）有基本了解，蓝色部分为拓展内容。