AUTOENCODER

ZTS
plote5024@gmail.com

Abstract

An autoencoder is an artificial neural network used for data encoding in unsupervised learning. Its purpose is to learn a compressed representation of data. A typical autoencoder consists of two parts: an encoder and a decoder.

I will learn what they are, what their limitations are, typical use cases, and look at some examples. I will start with a general introduction to autoencoders and discuss the role of activation functions and loss functions in the output layer. I will then discuss what reconstruction error is. Finally, I will look at typical applications such as dimensionality reduction, classification, denoising, and anomaly detection.

I.Introduction

There are three main components of an autoencoder: encoder, latent feature representation, and decoder. Encoders and decoders are just simple functions, while the name latent feature representation usually refers to a tensor of real numbers. In general, we want an autoencoder to reconstruct the input well.

For example, the latent features of the handwritten digit 4 might be the number of lines required to write each digit or the angle of each line and how they are connected. Learning how to write a digit certainly does not require learning the grayscale value of each pixel in the input image. We humans certainly do not learn to write by filling pixels with grayscale values. In the process of learning, we extract basic information that can help us solve a problem (e.g., write a digit). This latent representation (how to write each digit) is very useful for various tasks (e.g., feature extraction that can be used for classification or clustering) or simply understanding the basic characteristics of a dataset.

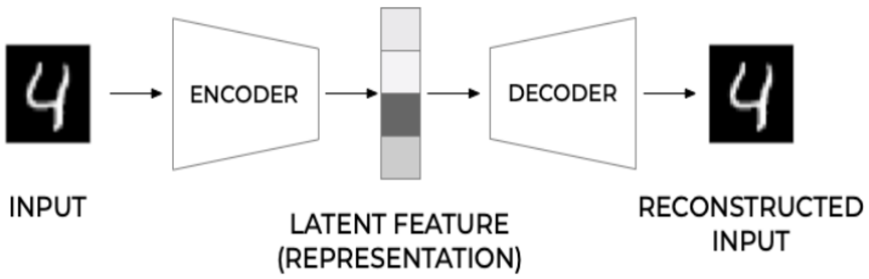

Figure 1: General structure of an *autoencoder*.

II.Structure and Function

The autoencoder has two parts, including the encoder and the decoder. The latent feature is the intermediate representation between the encoder and the decoder. Strictly speaking, it is part of the encoder output, not an independent structural part. In layman's terms, the encoder is responsible for "reducing the burden", reducing the data dimension, and recording all the data with fewer parameters. The decoder is responsible for restoring the data using the provided low-dimensional data. We call the input data c and the restored data b. How do we judge the result of our restoration? We only need to use the loss function to compare the input result with the output result.

For the encoder g:

$$\boldsymbol{h_i} = g(X_i)$$

Single-layer encoder:

$$\boldsymbol{h} = g(X) = f(W_e X + b_e)$$

The input data is $X_i \in \mathbb{R}^n$, $\boldsymbol{h_i} \in \mathbb{R}^q$ is the potential feature representation (low-dimensional representation), $(q < n)$and the g function is the encoder function, **which cleverly represents the dynamic representation of mapping the input data to the latent space (high-dimensional to low-dimensional)**

For the decoder $\boldsymbol{f}$:

$$\widetilde{X_i} = f(h_i) = f(g(X_i))$$

Single layer decoder:

$$\widetilde{X} = f(h) = g(W_d h + b_d)$$

Same as encoder, $h_i \in \mathbb{R}^q$ refers to the latent feature representation. $\widetilde{x_i}$ is the reconstructed data. f is the decoder function that maps the latent features back to the original data space. $\mathbb{R}^n$ represents the high-dimensional space where the input data x_i and the reconstructed data $\widetilde{x_i}$ are located. And $\mathbb{R}^q$ represents the low-dimensional space where the latent feature representation h_i is located, so we can also get $q < n$, which is exactly the opposite of the encoder process.

Loss Function

$$\begin{aligned} L\left(\boldsymbol{X}, \widetilde{\boldsymbol{X}}\right) &= \left\|\boldsymbol{X} - \widetilde{\boldsymbol{X}}\right\|^2 \\ &= \left(\boldsymbol{X_i} - \widetilde{\boldsymbol{X_i}}\right)^T \left(\boldsymbol{X_i} - \widetilde{\boldsymbol{X_i}}\right) \end{aligned}$$

Optimization goals during autoencoder training

$$\begin{aligned} \arg\min_{f,g} \langle \Delta\left(\boldsymbol{X_i}, \widetilde{\boldsymbol{X_i}}\right)\rangle &= \arg\min_{f,g} \langle \Delta(\boldsymbol{X_i}, f(g(\boldsymbol{X_i})))\rangle \\ &= \arg\min_{f,g} \langle \left\|\boldsymbol{X_i} - \widetilde{\boldsymbol{X_i}}\right\|^2 \rangle \\ &= \arg\min_{f,g} \frac{1}{N} \sum_{i=1}^{N} L\left(\boldsymbol{X_i}, \widetilde{\boldsymbol{X_i}}\right) \\ &= \arg\min_{f,g} \frac{1}{N} \sum_{i=1}^{N} \|\boldsymbol{X_i} - f(g(\boldsymbol{X_i}))\|^2 \end{aligned}$$

where Δ indicates a measure of how the input and the output of the autoencoder differ (basically our loss function will penalize the difference between input and output) and $< \cdot >$ indicates the average over all observations. Depending on how one designs the autoencoder, it may be possibleto find f and g so that the autoencoder learns to reconstruct the output perfectly, thus learning the identity function. This is not very useful, as we discussed at the beginning of the article, and to avoid this possibility, two main strategies can be used: creating a bottleneck and add regularization in some form.Adding a "bottleneck," is achieved by making the latent feature's dimensionality lower (often much lower) than the input's. That is the case that we will look in detail in this article. But before looking at this case, let's briefly discuss regularization.

III.Regularization in autoencoders

The purpose of regularization is to prevent the model from overfitting on the training data by limiting the size of model parameters, thereby improving the model's generalization ability on new data.Intuitively it means enforcing sparsity in the latent feature output. The simplest way of achieving this is to add a l_1 or l_2 regularization term to the loss function. That will look like this for the l_2 regularization term:

$$\arg\min_{f,g}\left(\mathbb{E}[\Delta(\boldsymbol{x}_i), g(f(\boldsymbol{x}_i))] + \lambda\sum_i \theta_i^2\right)$$

θ_i is the set of all weight and bias parameters in the autoencoder. These parameters include:

- ♬ Every element in the encoder's weight matrix and bias vector.
- ♬ Every element in the decoder's weight matrix and bias vector.

For the single-layer encoder and single-layer decoder above, we assume the following parameters and can get the parameters of θ_i

Assume the following parameters
- encoder weight matrix: $\boldsymbol{W}_e$
- encoder bias vector: $\boldsymbol{b}_e$
- decoder weight matrix: $\boldsymbol{W}_d$
- decoder bias vector: $\boldsymbol{b}_d$

θ_i includes
- → Each element in the encoder weight matrix $w_{e_{ij}}$
- → Each element in the encoder bias vector b_{e_i}
- → Each element in the decoder weight matrix $w_{b_{ij}}$
- → Each element in the decoder bias vector b_{d_i}

The specific form of the regularization term is as follows:

$$\lambda\sum_i \theta_i^2 = \lambda\left(\sum_{i,j} w_{e_{ij}}^2 + \sum_i b_{e_i}^2 + \sum_{i,j} w_{d_{ij}}^2 + \sum_i b_{d_i}^2\right)$$

IV.Feed Forward Autoencoders

The difference between ffa and the original autoencoder:

- ▷ Symmetry: The structures of encoder and decoder are usually symmetrical. This means that the encoder gradually reduces the number of neurons from the input layer to the latent feature layer, while the decoder gradually increases the number of neurons from the latent feature layer to the output layer, making the entire network structure symmetrical at the middle layer (bottleneck layer).
- ▷ Decrease the number of neurons as you move towards the center of the network: In the encoder part, the number of neurons in each layer gradually decreases until it reaches the middle layer (bottleneck layer). This is designed to compress the data so that it can be represented in a low-dimensional space.
- ▷ Bottleneck layer: The middle layer (bottleneck layer) has the least number of neurons. The output of this layer is a low-dimensional representation or feature representation of the input data. The bottleneck layer is designed to force the network to learn the most important features of the data.
- ▷ Bottleneck effect: By reducing the number of neurons in the middle layer, the network is forced to compress information, remove redundant data, and learn the core features of the data. This structure helps improve the generalization ability of the model and reduce overfitting.

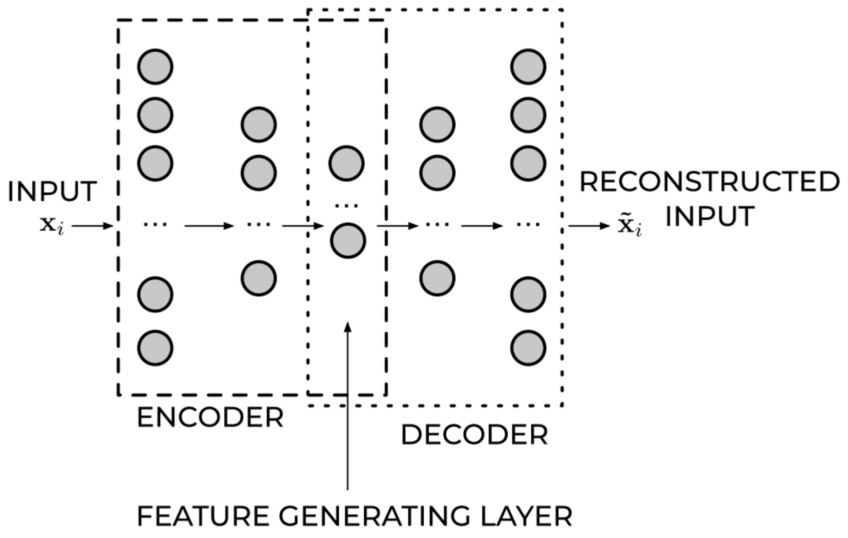

Figure 2: Internal structure of FFA.

Activation Function of the Output Layer

In neural network based autoencoders, the activation function of the output layer plays a particularly important role. The most commonly used functions are ReLU and sigmoid. Let's look at both of these functions and see some tips on when to use which function and when to use which.

ReLU

The **ReLU** activation function can assume all values in the range $[0, \infty]$. As a remainder, its formula is

$$\text{ReLU}(x) = \max(0, x)$$

This choice is good when the input observations x_i assume a wide range of positive values. If the input x_i can assume negative values, the ReLU is, of course, a terrible choice, and the identity function is a much better choice.

$$f(x) = \begin{cases} 0, x \leq 0 \\ x, x > 0 \end{cases}$$

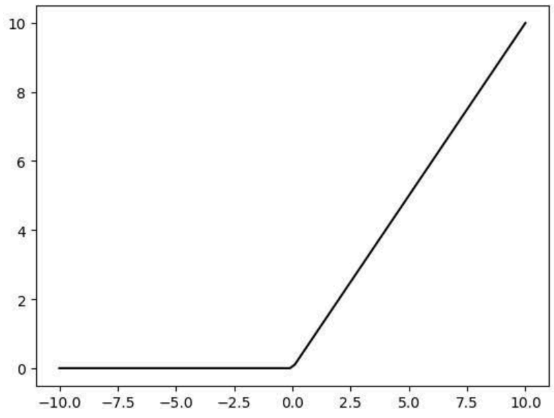

Figure 3: relu image.

- ✓ ReLU is non-linear, which means **it can capture complex patterns in the input data.**
- ✓ **Avoid gradient vanishing**: The derivative of ReLU in the positive interval is a constant 1, and the derivative in the negative interval is 0, **so it can effectively alleviate the gradient vanishing problem when training deep neural networks.** This makes training faster and the model deeper
- ✓ Sparse activation: ReLU outputs zero for negative values, **which means it produces sparse activations (many neurons have zero outputs)**, which helps reduce computation and improves the generalization of the model

ReLU is often used as the activation function for hidden layers because it can effectively train deep neural networks and capture complex nonlinear relationships.

sigmoid

The **sigmoid** function σ can assume all values in the range $[0, 1]$. As a remained its formula is:

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

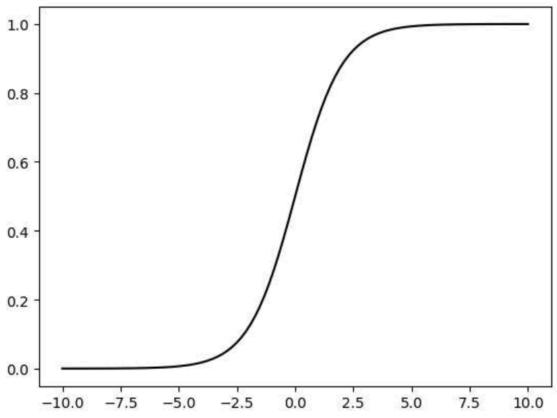

Figure 4: sigmoid image.

This activation function can only be used if the input observations xi are all in the range $[0, 1]$ or if you have normalized them to be in that range.Consider as an example the MNIST dataset. Each value of the input observation x_i (one image) is the gray values of the pixels that can assume any value from 0 to 255. Normalizing the data by dividing the pixel values by 255 would make each observation (each image) have only pixel values between 0 and 1. In this case, the sigmoid would be a good choice for the output layer's activation function.

- ✓ Probability output: Sigmoid maps the input value to the (0, 1) interval, which is suitable as a probability output, especially for **binary classification problems**.
- ✓ Smooth nonlinearity: Sigmoid is a smooth nonlinear function that can map any real number to a finite range.
- ✓ Application: Sigmoid is often used as the activation function of the output layer, especially in binary classification problems, because it can provide a probability value indicating the possibility that a sample belongs to a certain category.

Loss Function

The loss function compares the difference between a and b and tries to reduce the value. The previous loss function is:

$$\mathbb{E}\left[\Delta\left(\boldsymbol{X}_i \widetilde{\boldsymbol{X}}_i\right)\right]$$

For FFA, g; and f will be the functions obtained through the dense layers, as described in the previous sections. Remember that the autoencoder is trying to learn an approximation of the identity function; therefore, you need to find the weights in the network that give the minimum difference according to some metric ($\Delta(\cdot)$) between x_i and $\widetilde{x_i}$. Two loss functions are widely used for autoencoders: mean squared error (MSE) and binary cross entropy (BCE).

Mean Square Error

Since an autoencoder is trying to solve a regression problem, the most common choice as a loss function is the Mean Square Error (MSE):

$$L_{\mathrm{MSE}} = \mathrm{MSE} = \frac{1}{M}\sum_{i=1}^{M}\left\|\boldsymbol{X_i} - \widetilde{\boldsymbol{X_i}}\right\|^2$$

L2 norm:

$$\left\|\boldsymbol{X_i} - \widetilde{\boldsymbol{X_i}}\right\| = \sqrt{\sum_{j=1}^{N}(x_{ij} - \tilde{x}_{ij})}$$

Then square it, expressed as:

$$\left\|\boldsymbol{X_i} - \widetilde{\boldsymbol{X_i}}\right\|^2 = \sum_{j=1}^{N}(x_{ij} - \tilde{x}_{ij})$$

The entire formula represents the mean square error of all data samples:

$$L_{\mathrm{MSE}} = \mathrm{MSE} = \frac{1}{M}\sum_{i=1}^{M}\sum_{j=1}^{N}\left(x_{ij} - \tilde{x}_{ij}\right)^2$$

The symbol | | indicates the norm of a vector,equivalent to the original equation.and M is the number of the observation in the training dataset.

$$\frac{\partial L_{\mathrm{MSE}}}{\partial \tilde{x}_j} = \frac{\partial}{\partial \tilde{x}_j}\left(\frac{1}{M}\sum_{i=1}^{M}(x_i - \tilde{x}_i)^2\right)$$

Since MSE is the average of all sample errors, for each specific x, only the error term corresponding to it is considered:

$$\frac{\partial L_{\mathrm{MSE}}}{\partial \widetilde{x_j}} = -\frac{2}{M}(x_j - \widetilde{x_j}) = 0$$

Solving this equation yields:

$$x_j = \widetilde{x_j}$$

To confirm that this point is a minimum, calculate the second derivative:

$$\frac{\partial^2 L_{\mathrm{MSE}}}{\partial \widetilde{x_j}^2} = \frac{\partial}{\partial \widetilde{x_j}}\left(-\frac{2}{M}(x_j - \widetilde{x_j})\right) = \frac{2}{M}$$

Since the second-order derivative is positive, this means that $x_j = \widetilde{x_j}$,there is a local minimum.

Binary Cross-Entropy

If the activation function of the output layer of the FFA is a sigmoid function, thus limiting neuron outputs to be between 0 and 1, and the input features are normalized to be between 0 and 1 we can use as loss function the binary crossentropy, indicated here with LCE. Note that this loss function is typically used in classification problems, but it works beautifully for autoencoders. The formula for it is:

$$L_{\text{CE}} = -\frac{1}{M}\sum_{i=1}^{M}\sum_{j=1}^{n}\left[x_{j,i}\log\tilde{x}_{j,i} + \left(1 - x_{j,i}\right)\log\left(1 - \tilde{x}_{j,i}\right)\right]$$

- ✓ Model output predicted probability:The model outputs a probability value $\tilde{x}_{j,i}$, which indicates the probability that sample i belongs to category j. This value is usually between 0 and 1.
- ✓ True label:The true label $x_{j,i}$ is the known, actual category label. It is only 0 and 1

For each sample i and each class j, calculate the cross entropy loss term:

$$x_{j,i}\log\tilde{x}_{j,i} + \left(1 - x_{j,i}\right)\log\left(1 - \tilde{x}_{j,i}\right)$$

When the true label $x_{j,i} = 1$, the loss term is $\log\tilde{x}_{j,i}$ Same reason,When the true label $x_{j,i} = 0$, the loss term is $\log\left(1 - \tilde{x}_{j,i}\right)$

for example: $x_{1,i} = 1$ and $\tilde{x}_{1,i} = 0.8$,The loss term is $\log 0.8$,If $x_{1,i} = 0$ and $\tilde{x}_{1,j} = 0.8$ then the loss term is $\log(1 - 0.8) = \log 0.2$.

Sum the loss terms for all samples and classes:

$$\sum_{i=1}^{M}\sum_{j=1}^{n}\left[x_{j,i}\log\tilde{x}_{j,i} + \left(1 - x_{j,i}\right)\log\left(1 - \tilde{x}_{j,i}\right)\right]$$

Since there is a minus sign in front of the cross entropy loss formula, it means that we want to minimize this value. The minus sign is used to turn the loss into a positive value, because the result of the logarithmic function log is usually negative.

Divide the total loss by the number of samples M