

Universidade Federal de Minas Gerais  
Escola de Engenharia  
Curso de Graduação em Engenharia de Controle e Automação

**Análise do desempenho de *autoenconders* em um problema  
de detecção de falhas**

Sara Regina Ferreira de Faria

Orientador: Prof. André Paim Lemos, Dr.

Belo Horizonte, junho de 2017



## **Monografia**

**Análise do desempenho de *autoenconders* em um problema de detecção de falhas**

Monografia submetida à banca examinadora designada pelo Colegiado Didático do Curso de Graduação em Engenharia de Controle e Automação da Universidade Federal de Minas Gerais, como parte dos requisitos para aprovação na disciplina Projeto Final de Curso II.

Belo Horizonte, junho de 2017



# Resumo

O objetivo deste trabalho é analisar o desempenho de diferentes tipos de *autoencoders* para problemas de detecção de falhas. *Autoencoder* é uma rede neural artificial de aprendizado não supervisionado que codifica uma entrada para obter uma representação dos dados em uma camada oculta através de um *encoder* e logo em seguida a decodifica para obter dados iguais aos de entrada através de um *decoder*. Com esse processo, a rede aprende características dos dados de entrada. Esse trabalho estuda o desempenho de cinco diferentes *autoencoders*. O primeiro deles é o *autoencoder* simples, conforme descrito. Em seguida, há o *stacked autoencoder*, que é uma rede neural profunda construída através do empilhamento de *autoencoders* para obter representações intermediárias dos dados. Depois, é colocado o *denoising autoencoder*, que aplica um ruído aos dados de entrada, mas que procura reconstruir os dados sem falhas. Então há o *sparse autoencoder* que acrescenta uma restrição nas ativações dos neurônios da camada oculta. E, por fim, há o *variational autoencoder*, que ao invés de aplicar uma função arbitrária, aprende parâmetros de uma distribuição de probabilidade dos dados de entrada. Para o problema de detecção de falhas, o *autoencoder* é treinado com dados normais e então dados com falhas são apresentados ao modelo. Como a rede neural desconhece os dados com falhas, a reconstrução da camada oculta gera dados diferentes dos de entrada, indicando que a entrada não é um dado normal, e sim uma falha. Para avaliar e comparar os desempenhos dos modelos, eles foram desenvolvidos e aplicados na base de dados *Tennessee Eastman Process*, que apresenta 20 diferentes falhas. Como indicadores de performance foram utilizados a área sob a curva ROC e também o tempo necessário para cada teste. Os resultados obtidos indicam que o *sparse autoencoder* é capaz de detectar melhor as falhas que os outros modelos, enquanto o *stacked autoencoder* apresenta menor tempo de treino e teste que os outros modelos.

**Palavras-chave:** Autoencoder, Detecção de Falhas, Tennessee Eastman Process.

# Abstract

The objective of this work is to analyze the performance of different types of autoencoders for fault detection problems. Autoencoder is an artificial neural network with unsupervised learning that encodes an input to obtain a representation of the data in a hidden layer through an encoder and then decodes it to obtain the same input through a decoder. With this process, the network learns characteristics of the input data. This work studies the performance of five different autoencoders. The first of them is the simple autoencoder, as described. Then there is the stacked autoencoder, which is a deep neural network built through the stacking of autoencoders to get intermediate representations of the data. Then the denoising autoencoder is presented, which applies a noise to the input data, but still aims to rebuild the original data. Then there is the sparse autoencoder which adds a constraint on the activations of the neurons on the hidden layer. And finally, there is the variational autoencoder, which instead of applying an arbitrary function, learns parameters of a probability distribution of the input data. For the fault detection problem, the autoencoder is trained with normal data and then faulty data is presented to the model. Because the neural network is unaware of faulty data, the reconstruction of the hidden layer generates data different from the input, indicating that the input is not a normal data, but a failure. To evaluate and compare the performances of the models, they were developed and applied in the Tennessee Eastman Process database, which presents 20 different failures. As performance indicators, the area under the ROC curve was used as well as the time required for each test. The obtained results indicate that the sparse autoencoder is able to detect faults better than the other models, while the stacked autoencoder presents a shorter training and test time than the other models.

**Keywords:** Autoencoder, Failure detection, Tennessee Eastman Processes.

# Sumário

<b>Resumo</b>	<b>i</b>
<b>Abstract</b>	<b>ii</b>
<b>Lista de Figuras</b>	<b>v</b>
<b>Lista de Tabelas</b>	<b>vii</b>
<b>1 Introdução</b>	<b>1</b>
1.1 Motivação e Justificativa . . . . .	1
1.2 Objetivos do Projeto . . . . .	2
1.3 Local de Realização . . . . .	2
1.4 Estrutura da Monografia . . . . .	2
<b>2 Revisão Bibliográfica</b>	<b>5</b>
2.1 Redes Neurais Artificiais . . . . .	5
2.2 Análise de Componentes Principais (PCA) . . . . .	6
2.3 <i>Autoencoder</i> . . . . .	7
2.3.1 <i>Stacked Autoencoder</i> . . . . .	9
2.3.2 <i>Denoising autoencoder</i> . . . . .	10
2.3.3 <i>Sparse autoencoder</i> . . . . .	10
2.3.4 <i>Variational autoencoder</i> . . . . .	11
2.4 Resumo do Capítulo . . . . .	13
<b>3 Metodologia</b>	<b>15</b>
3.1 Aplicação de <i>autoencoder</i> em problemas de detecção de falhas . . . . .	15
3.2 Metodologia de desenvolvimento do projeto . . . . .	16
3.2.1 Indicadores de Performance . . . . .	17
3.3 Resumo do Capítulo . . . . .	18
<b>4 Resultados</b>	<b>19</b>
4.1 <i>Tennessee Eastman Process</i> . . . . .	19
4.2 Testes Realizados . . . . .	20
4.3 Resultados Obtidos . . . . .	23
4.4 Resumo do Capítulo . . . . .	25

<b>5 Conclusões</b>	<b>27</b>
5.1 Considerações Finais . . . . .	27
5.2 Propostas de Continuidade . . . . .	27
<b>Referências Bibliográficas</b>	<b>28</b>

# Listas de Figuras

2.1	Comparação da estrutura do neurônio biológico (a) e do neurônio artificial (b). Fonte: [10]	5
2.2	Exemplo de rede neural com arquitetura <i>feedforward</i> . Fonte: [8]	6
2.3	Exemplo de rede neural com arquitetura recorrente. Fonte: [8]	6
2.4	Estrutura do <i>autoencoder</i> com a camada de entrada, a camada oculta e a camada de saída. Fonte: [4]	8
2.5	Arquitetura de um <i>stacked autoencoder</i> . Fonte: [19]	9
2.6	Arquitetura do <i>denoising autoencoder</i> . Alguns elementos de $x$ são zerados aleatoriamente e a entrada corrompida $\tilde{x}$ é utilizada para mapear as características $y$ e então reconstruir $x$ produzindo $z$ . A função de perda $L(x, z)$ é calculada segundo a saída e a entrada original. Fonte: [15]	11
2.7	Arquitetura do <i>encoder</i> e do <i>decoder</i> probabilísticos. Fonte: [1]	12
2.8	Arquitetura do <i>variational autoencoder</i> . Fonte: [1]	12
3.1	Dois exemplos de curvas ROC e a curva esperada para tentativa aleatória. Quanto maior a área sob a curva ROC, melhor a classificação realizada pelo modelo. Fonte:[17]	18
4.1	Esquema do Tennessee Eastman Process Fonte:[5]	19
4.2	Exemplo dos dados da variável $t(1)$ .	21
4.3	Curvas ROC para os testes com as falhas 1, 2, 5, 7, 8, 9, 10, 11, 12, 14, 15, 19.	24



# **Lista de Tabelas**

4.1	Variáveis do TEP utilizadas nas análises. Fonte: [9] . . . . .	20
4.2	Descrição das situações de falhas do TEP. Fonte: [12] . . . . .	22
4.3	Parâmetros utilizados para treinar os diferentes modelos . . . . .	23
4.4	Resultados obtidos nos testes . . . . .	23



# Capítulo 1

## Introdução

### 1.1 Motivação e Justificativa

Em várias áreas, como na aviação, em uma planta industrial ou nos veículos modernos, sistemas altamente confiáveis têm sido exigidos. Para satisfazer essa necessidade, eles precisam ser testados nas mais diversas situações de formas a se identificar uma possível falha. Caso alguma falha seja detectada, ela deve ser estudada e analisada por especialistas na área para encontrar sua causa e corrigi-la [19].

Na maioria dos casos, essa detecção de falhas é feita por técnicos capacitados, cuja decisão pode ser influenciada por seus diferentes treinamentos e níveis de experiência. Assim, esse trabalho pode não ser tão confiável quanto se exige [4]. Além disso, o aumento da complexidade dos sistemas implica também em uma análise ainda mais custosa [19].

Uma possível solução é utilizar técnicas de aprendizado de máquina (*machine learning*). Para esses casos, um modelo é desenvolvido e aprendido pela máquina a partir de dados normais obtidos do sistema. Então dados de teste são apresentados a este modelo para detectar falhas ou anomalias em relação ao processo normal. A falha encontrada é depois analisada para que seja identificada sua causa. Esse é o problema chamado de detecção e análise de falhas [14].

Uma dessas técnicas é o *autoencoder*, uma rede neural artificial onde a seleção de características e sua extração são não supervisionadas [4]. Para isso, ele é composto por duas partes: um *encoder* (codificador) e um *decoder* (decodificador) [1]. Essas duas partes podem ser compostas de uma única camada oculta ou então é possível que a representação oculta de um *autoencoder* seja utilizada como entrada para outro, formando um *deep autoencoder* [1].

No processo de codificação, o número de neurônios presentes em cada camada é diminuído até que se chegue à sua última camada, a *bottleneck layer* (camada gargalo) [4]. Nesse processo, características dos dados de entrada são extraídas e armazenadas nessa última camada [13].

Já no processo de decodificação, o número de neurônios presentes em cada camada é aumentado até que a entrada seja reconstruída. Sendo assim, os dados de saída desejados são os mesmos dados de entrada [16][13]. Então, ao ser treinado, o modelo baseado em *autoencoder* busca a minimização do erro de reconstrução [4], isto é, a diferença entre a entrada original e sua reconstrução [1].

Em síntese, treinar um *autoencoder* consiste em aprender os pesos e bias ao longo das

camadas de neurônios de forma que o vetor de entrada possa ser reconstruído o mais próximo possível do dado original. Nisso, a *bottleneck layer* obtida é a melhor representação comprimida possível para o vetor de entrada [13].

Uma das vantagens em se utilizar os *autoencoders* é o fato de poder tratar entradas reais, distribuições probabilísticas ou entradas binárias apenas fazendo a modificação na função de ativação da rede neural e também na função de perda [19].

Além dos *autoencoders* simples, como foram colocados até o momento, há algumas variações possíveis em sua programação que podem apresentar outras vantagens em diferentes aplicações.

No caso do *variational autoencoder* (*autoencoder* variacional), por exemplo, a principal diferença é ser um modelo gerativo estocástico capaz de calcular probabilidades calibradas. Enquanto o *autoencoder* simples é um modelo discriminador determinístico, sem qualquer fundação probabilística [1].

Já o *sparse autoencoder* (*autoencoder* escasso) é utilizado principalmente em situações em que há muitos ruídos presentes nos dados de entrada, podendo dispensar inclusive a presença de filtros de sinal [4]. Para tanto, sua diferença do *autoencoder* simples é a adição de escassez na função de custo do modelo [4]. Assim, o número de ativações nos neurônios na camada oculta é limitado, fazendo com que o espaço de características selecionadas seja ainda mais restrito [4]. Além disso, [16, Wang] conclui em seu trabalho que esse tipo de *autoencoder* é mais adequado para reconstrução de dados contínuos.

Outra variação é o *denoising autoencoder* (*autoencoder* diminuidor de ruído), em que um ruído é adicionado no vetor de entrada [14]. O objetivo desse modelo é reconstruir a entrada limpa a partir de sua versão corrompida, o que força o *autoencoder* a manter apenas características mais robustas do sistema [19].

Por ser treinado para manter apenas os padrões significativos do sistema [1], o *denoising autoencoder* requer menos conhecimento sobre os distúrbios e ruídos, que podem possuir causas ou formas desconhecidas. Essa, portanto, é uma das maiores vantagens desse tipo de *autoencoder* [19]. Além disso, segundo [19, Zhang], esse método é capaz de alcançar uma razão de compressão maior se comparado ao *autoencoder* simples.

## 1.2 Objetivos do Projeto

O objetivo deste projeto final de curso é estudar os diferentes tipos de *autoencoders* e comparar seu desempenho na solução de problemas de detecção de falhas.

## 1.3 Local de Realização

O projeto será estudado e implementado nas dependências de Universidade Federal de Minas Gerais.

## 1.4 Estrutura da Monografia

O trabalho está apresentado em cinco capítulos. O capítulo 1 traz a introdução ao problema e os objetivos que se pretende alcançar ao final do desenvolvimento do trabalho. O capítulo

2 contempla conceitos básicos de redes neurais artificiais e *autoencoders*, de formas que se compreenda o que foi utilizado no trabalho. No capítulo 3 há uma visão do que já foi desenvolvido na área por outros pesquisadores e também traz a descrição da metodologia utilizada para o desenvolvimento do projeto. Já o capítulo 4 apresenta os dados utilizados para testes, a explicação dos modelos construídos, em quais testes eles foram aplicados e os resultados obtidos em cada um deles. E no capítulo 5 há a conclusão deste trabalho e propostas de continuidade.



# Capítulo 2

## Revisão Bibliográfica

### 2.1 Redes Neurais Artificiais

As redes neurais artificiais são uma abordagem não-linear para modelar problemas matemáticos de classificação, predição ou regressão com base no funcionamento do cérebro humano [8]. E o neurônio biológico é a unidade básica para este funcionamento: impulsos elétricos chegam a um neurônio, que trata a informação e passa para o próximo de forma a trazer o resultado esperado. Analogicamente, cada neurônio artificial recebe diversas entradas com seus respectivos pesos que passam por uma função de ativação e seguem para a saída. Na figura 2.1, essas duas estruturas podem ser comparadas. [10]

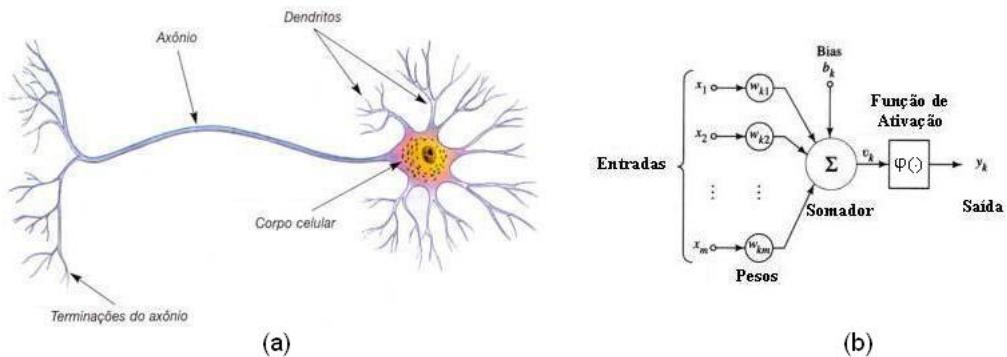


Figura 2.1: Comparação da estrutura do neurônio biológico (a) e do neurônio artificial (b).  
Fonte: [10]

As redes neurais artificiais podem ser classificadas segundo diversos critérios, como método de aprendizado, arquitetura, tipos de saída, tipos de nós ou implementação [8]. Os principais critérios para o desenvolvimento deste trabalho estão melhor explicados nos itens a seguir.

#### Arquitetura - tipos de conexão

1. *Feedforward*: a saída de cada nó da rede neural é a entrada da próxima camada (*layer*).

Na figura 2.2, há um exemplo de rede neural *feedforward*. Nesse caso, pode-se ver que as saídas das camadas 1 e 2 são entradas apenas das suas camadas subsequentes.[8]

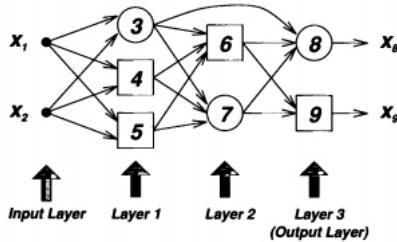


Figura 2.2: Exemplo de rede neural com arquitetura *feedforward*. Fonte: [8]

2. Recorrente: se há algum caminho circular entre os nós da rede, ela é classificada como recorrente. Como pode ser visto na figura 2.3, que a saída de um dos nós é também uma de suas entradas. [8]

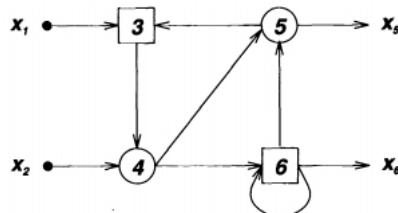


Figura 2.3: Exemplo de rede neural com arquitetura recorrente. Fonte: [8]

## Métodos de Aprendizado

1. Supervisionado: tipo de aprendizado quando há um grupo de dados com entradas e saídas conhecidas. A rede neural será treinada com parte desses dados para replicar seu comportamento e, com o restante dos dados, é testada sua capacidade de seguir os resultados desejados.
2. Não-supervisionado: utilizado quando o conjunto de dados apresenta apenas as entradas, mas não as saídas desejadas. Nesse caso, a rede deverá ser treinada de forma a encontrar padrões nos dados de treinamento e então testar seu funcionamento com os dados de teste.
3. Offline: também conhecido como aprendizado em bateladas, ele acontece a partir do uso de um conjunto de dados para treinamento da rede. Uma vez definida, a rede não será alterada até que um novo conjunto de dados seja apresentado. [8]
4. Online: nesse caso, o treinamento da rede é feito a cada amostra de dados e seus parâmetros são sempre atualizados. Esse tipo de aprendizado é essencial para processos em que há mudanças constantes de características.[8]

## 2.2 Análise de Componentes Principais (PCA)

O PCA (*Principal Component Analysis*) é um método utilizado para reduzir o número de informações utilizadas para representar um dado [2]. Com  $N$  dados de entrada  $x$ , o PCA

gera a matriz de transformação linear  $W = w_1, \dots, w_d^T$  para extrair os  $d \leq N$  componentes principais  $y = Wx$ . Após o PCA projetar  $x$  para um subespaço de menor dimensão, ele reconstrói o dado novamente no espaço de observação, seguindo a equação de predição do modelo: [14]

$$x_r = W^T y, \quad (2.1)$$

com um vetor de erro de reconstrução  $e$  dado por:

$$e = x - x_r, \quad (2.2)$$

Sendo  $e = e_1, \dots, e_N$ , pode-se calcular o escore de anomalias e com isso encontrar possíveis falhas. [14]

Apesar do PCA ser simples e de fácil implementação, é um modelo linear. Assim, não atinge uma performance alta em detecções de falhas se a estrutura dos dados for complexa.[14]

## 2.3 Autoencoder

Do ponto de vista da modelagem de processos, um *autoencoder* se comporta como um PCA se o problema for linear. Caso contrário, apenas o *autoencoder* será capaz de obter características consistentes do processo [19]. A partir disso, pode-se entender o *autoencoder* como um PCA não-linear.

Portanto, um *autoencoder* é um modelo de rede neural artificial construído para aprender uma representação comprimida dos dados de entrada e então reconstruí-los o mais próximo possível a partir dessa representação [13][1][4]. Além disso, seu treinamento se dá por aprendizado não-supervisionado [1]. Ele é composto de duas partes: um *encoder* e um *decoder*.

- **Encoder:** é o mapeamento determinístico que transforma o vetor de dados de entrada  $x$  em uma representação oculta  $y$ . Esse mapeamento geralmente é definido por uma função  $f(\cdot)$  afim seguida de uma não-linearidade:

$$y = f_\theta(x) = s(Wx + b), \quad (2.3)$$

onde  $\theta = W, b$ ;  $s(\cdot)$  é uma função não linear, como a sigmoidal por exemplo;  $W$  é uma matriz de pesos de dimensão  $d \times d$ ; e  $b$  é um vetor de *offset* de dimensão  $d$ . [15]

- **Decoder:** é o mapeamento que transforma a representação oculta  $y$  em um vetor  $z$  no espaço dos dados de entrada. Comumente esse mapeamento é definido por uma função  $g(\cdot)$  afim seguida de uma não-linearidade:

$$z = g_{\theta'}(y) = s(W'y + b'), \quad (2.4)$$

sendo  $\theta' = W', b'$ ;  $W'$  uma matriz de pesos de dimensão  $d \times d$ ; e  $b'$  um vetor de *offset* de dimensão  $d$ . Opcionalmente  $W'$  pode ser definida por  $W' = W^T$ , indicando a situação de "pesos ligados". [15]

O vetor  $z$  não é uma reconstrução exata de  $x$ , mas sim o parâmetro da distribuição condicional  $p(x|X = z)$  que gerará  $x$  com uma maior probabilidade. [19]

As características dos dados de entrada extraídas pelo *encoder* são armazenadas na camada oculta. O número de unidades nessa camada é um parâmetro importante para o *autoencoder*, uma vez que limita o número de características que poderão ser utilizadas na reconstrução dos dados de entrada. Por conta disso, essa camada recebe o nome de *bottleneck layer* (camada gargalo). [13]

A estrutura do *autoencoder* pode ser vista na figura 2.4, com a presença de  $m$  neurônios na camada de entrada e de saída e  $n$  neurônios na camada oculta, sendo  $m > n$ . É também possível ver as matrizes de peso  $W_1$  e  $W_2$  no *encoder* e no *decoder* e os offsets  $B_1$  e  $B_2$ .

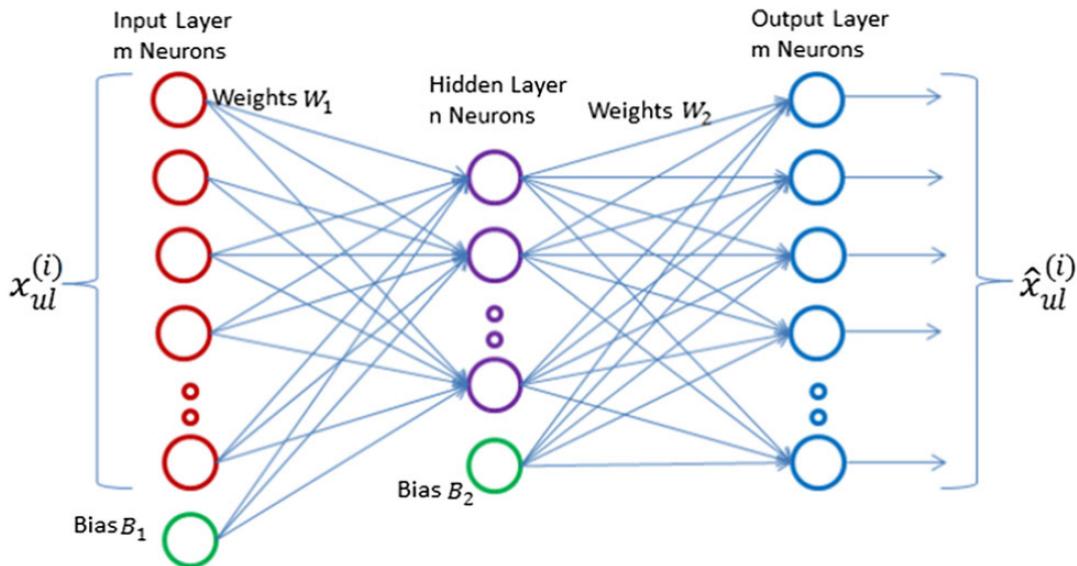


Figura 2.4: Estrutura do *autoencoder* com a camada de entrada, a camada oculta e a camada de saída. Fonte: [4]

A diferença entre a entrada original de dados  $x$  e a reconstrução obtida  $z$  é chamada de erro de reconstrução [1]. A função objetivo do treinamento de um *autoencoder* é encontrar os valores para os parâmetros do *encoder* e do *decoder* que minimizem esse erro, dado por:

$$\arg_{\theta, \theta'} \min E_{q^0(X)}[L(X, Z(\tilde{X}))], \quad (2.5)$$

sendo que o erro  $L(X, Z(\tilde{X}))$  pode ser definido de diversas formas, como por exemplo o erro quadrado: [19]

$$L(x, z) = \|x - z\|^2. \quad (2.6)$$

Em síntese, treinar um *autoencoder* implica em aprender os pesos e *offsets* utilizados nas camadas da rede neural para que as amostras de entrada possam ser reconstruídas o mais próximo possível do original. Para isso, os pesos das matrizes  $W$  e  $W'$  podem ser aperfeiçoados por *backpropagation* [13].

Sendo assim, um algoritmo de treinamento de um *autoencoder* pode ser visto no algoritmo 1.

**Algoritmo 1:** Algoritmo de treinamento do *autoencoder*. Fonte: [1]

---

**Data:** Dataset:  $x^{(1)}, x^{(2)}, \dots, x^{(N)}$   
**Result:** encoder  $f_\theta$ , decoder  $g_\theta$

- 1  $\phi, \theta \leftarrow$  Inicia parâmetros
- 2 **while** convergência dos parâmetros  $\phi, \theta$  **do**
- 3   | Calcula a soma do erro de reconstrução
- 4   |  $\phi, \theta \leftarrow$  atualiza parâmetros utilizando gradiente de  $E$
- 5 **end**

---

### 2.3.1 Stacked Autoencoder

O *stacked autoencoder* é uma rede neural profunda que pode ter sua arquitetura vista na figura 2.5. Nesse *autoencoder*, o número de neurônios se reduz a cada camada, até que se chegue à representação final  $y$ . Depois, o número de neurônios volta a aumentar na ordem inversa até que se chegue à reconstrução  $z$  de  $x$ . [19]

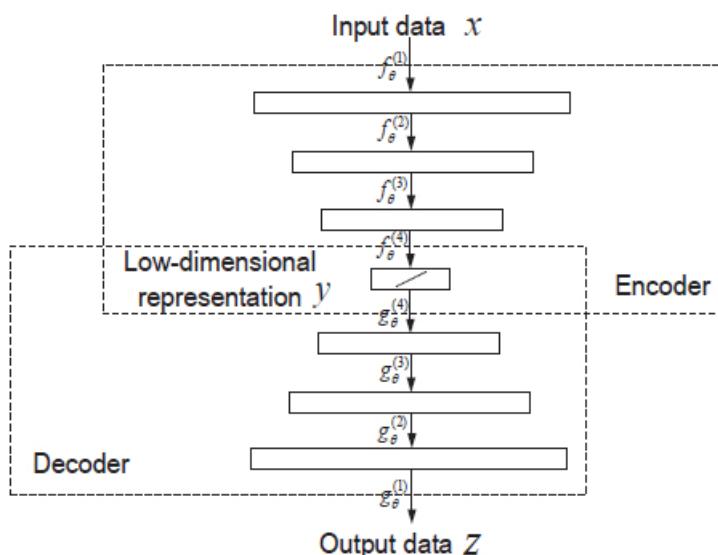


Figura 2.5: Arquitetura de um *stacked autoencoder*. Fonte: [19]

A representação de uma entrada na camada anterior é calculada pelo *encoder*, e então ela é utilizada como entrada para a próxima camada. Ao repetir esse procedimento, a representação final da entrada original é obtida. Para reconstruir essa entrada original a partir da representação final, os *decoders* correspondentes devem ser empilhados em ordem reversa aos *encoders*. [19]

Por fim é feito o ajuste fino da rede profunda, ou seja, ajustar os parâmetros da rede inteira para minimizar o erro de reconstrução. Isso pode ser feito pelo algoritmo de *backpropagation* baseado em gradiente descendente. [19]

Uma vez que o empilhamento de *autoencoders* foi então construído, a representação final (a saída do nível mais alto dos *encoders*) pode ser usada como entrada para um algoritmo de classificação. [15]

### 2.3.2 Denoising autoencoder

O *denoising autoencoder* é uma variação do *autoencoder* que adiciona um ruído ao vetor de entrada original  $x$  e adota essa entrada ruidosa como o novo vetor de entrada [1][14][19]. E há vários métodos para corrupção dos dados, como ruído de mascaramento, ruído "sal e pimenta" e ruído gaussiano aditivo [19]. O mapeamento das características na camada oculta, nesse caso, pode ser expresso pela equação:

$$y = f_{\theta}(\tilde{x}) = s(W\tilde{x} + b) \quad (2.7)$$

onde  $\theta = W, b$ ;  $s(\cdot)$  é uma função não linear;  $\tilde{x}$  é o vetor corrompido de dados de entrada;  $W$  é uma matriz de pesos; e  $b$  é um vetor de *offset* [19].

De posse do vetor  $y$ , o mapeamento de  $z$  como reconstrução do vetor corrompido  $x$  é feito como em um *autoencoder* simples.

O erro de reconstrução continua sendo a diferença entre a saída resultante e a entrada original [1]. E assim como no *autoencoder* simples, devem-se otimizar os parâmetros do *encoder* e do *decoder* para que se minimize o erro de reconstrução [19]. Isso significa ter  $z$  tão próximo quanto possível da entrada não corrompida [15].

É importante colocar que, para cada amostra de treinamento  $x$  apresentada ao modelo, uma versão diferente de  $\tilde{x}$  é gerada. Os parâmetros do *encoder* e do *decoder* são inicializados aleatoriamente e então otimizados por gradiente descendente [15].

A principal diferença, então, entre o *autoencoder* simples e sua variante *denoising autoencoder* é que, no segundo caso,  $y$  é obtido ao aplicar o mapeamento determinístico  $f$  a uma entrada corrompida [15]. E ainda assim a saída  $z$  deve ser uma reconstrução dos dados  $x$  não corrompidos [14].

Em síntese, a extração das características dos dados deve ser robusta às corrupções dos dados de entrada e preservar as informações da entrada original [19]. Isso então força o aprendizado de um mapeamento que extraia somente os padrões significativos dos dados [1].

Na figura 2.6, há um esquema explicativo do treinamento do *denoising autoencoder* utilizando o ruído de mascaramento como método de corrupção dos dados. Nesse caso, alguns elementos de  $x$  são zerados randomicamente segundo uma taxa de corrupção  $v$ . [19]

### 2.3.3 Sparse autoencoder

O *sparse autoencoder* é uma variação do *autoencoder* com uma penalidade adicionada à sua função de custo. E seu objetivo é resolver o seguinte problema de otimização para aprender as características dos dados de entrada: [4]

$$\min ||x - z||^2 + \lambda ||W||^2, \quad (2.8)$$

sendo o termo  $z$  a reconstrução aproximada da entrada  $x$ . O vetor  $W$  é a ativação do vetor de entrada  $x$ . O vetor  $y$  é o de características aprendidas. E  $\lambda$  é o regulador das ativações de  $W$ . Durante o treinamento, os pesos da matriz  $W$  e o vetor de *offset*  $b$  são atualizados para reduzir o erro de reconstrução de  $x$ . [4]

A ativação da camada oculta  $y$  e a ativação da camada de saída  $z$  se dão por um *encoder* e por um *decoder* respeitando as mesmas equações do *autoencoder* original. A diferença está no fato do *sparse autoencoder* adicionar escassez na ativação da camada oculta. [4]

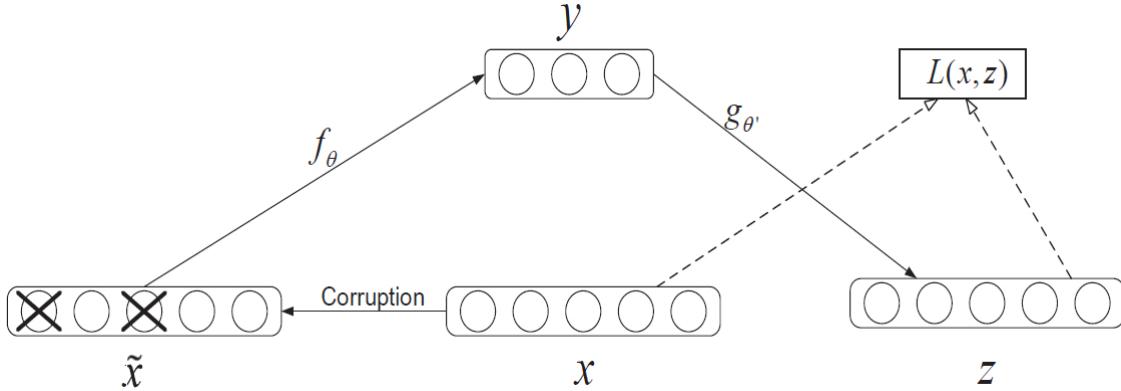


Figura 2.6: Arquitetura do *denoising autoencoder*. Alguns elementos de  $x$  são zerados aleatoriamente e a entrada corrompida  $\tilde{x}$  é utilizada para mapear as características  $y$  e então reconstruir  $x$  produzindo  $z$ . A função de perda  $L(x, z)$  é calculada segundo a saída e a entrada original. Fonte: [15]

A penalidade adicionada limita o número de ativações na camada oculta de neurônios. Isso faz o espaço de características ser mais comprimido e aumenta a separabilidade dos dados. [4]

### 2.3.4 Variational autoencoder

A principal diferença entre o *variational autoencoder* e o *autoencoder* simples é que o *variational autoencoder* é um modelo gerativo estocástico que pode gerar probabilidades calibradas, enquanto o *autoencoder* é um modelo discriminativo determinístico que não possui fundação probabilística. [1]

Na analogia com o *autoencoder* e tendo  $z$  como a variável latente e  $x$  como os dados de entrada, a aproximada posterior  $q_\phi(y|x)$  é o *encoder* e o modelo probabilístico direto gráfico  $p_\theta(x|y)$  é o *decoder*. Assim, os *encoders* e *decoders* do *variational autoencoder* podem ser chamados de *encoders* e *decoders* probabilísticos. [1]

Vale enfatizar que o *variational autoencoder* modela os parâmetros da distribuição ao invés do valor em si. Ou seja,  $f(x, \phi)$  no *encoder* gera os parâmetros da aproximada posterior  $q_\phi(y|x)$  e para obter o valor real da variável latente  $y$ , a amostragem de  $q_\phi(y; f(x, \phi))$  é necessária. Na figura 2.7, as arquiteturas descritas do *encoder* e do *decoder* podem ser vistas.[1]

A função objetivo de um *variational autoencoder* é maximizar a restrição variacional da probabilidade marginal de dados dada pela equação 2.9. Já a probabilidade marginal de cada ponto é dada pela equação 2.10. [1]

$$\log p_\theta(x^{(i)}, \dots, x^{(N)}) = \sum_{i=1}^N \log p_\theta(x^{(i)}) \quad (2.9)$$

$$\log p_\theta(x^{(i)}) = -D_{KL}(q_\phi(y|x)||p_\theta(y)) + E_{q_\phi(y|x^{(i)})}[\log p_\theta(x|y)] \quad (2.10)$$

onde  $p_\theta(x|y)$  é a probabilidade de se obter  $x$  dada a variável latente  $y$ . O primeiro termo da soma da equação 2.10 é a divergência KL entre a aproximada posterior e a prévia da

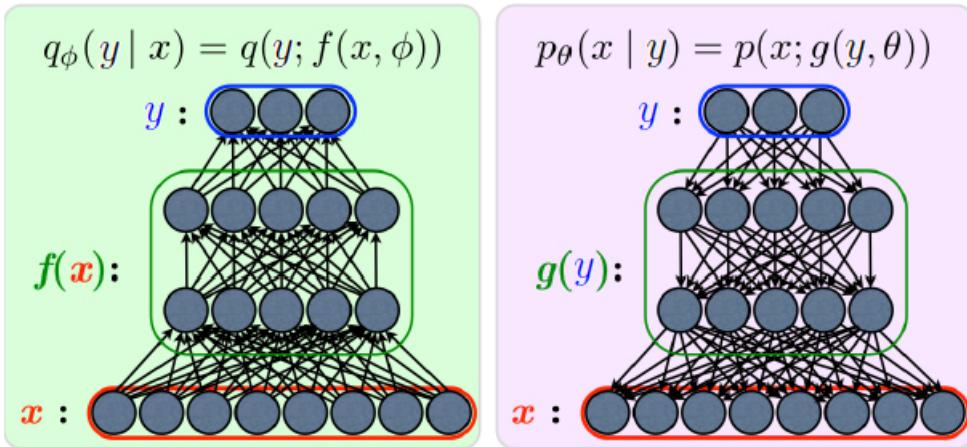


Figura 2.7: Arquitetura do *encoder* e do *decoder* probabilísticos. Fonte: [1]

variável latente  $y$ . Esse termo funciona como um termo regularizador, uma vez que força a distribuição a ser similar à distribuição anterior. E o segundo termo da soma da equação 2.10 pode ser entendido em termos da reconstrução de  $x$  através da distribuição posterior  $q_\phi(y|x)$  e a probabilidade  $p_\theta(x|y)$ . [1]

Para obter a reconstrução  $\hat{x}$  dada a amostra  $y$ , o parâmetro de  $p_\theta(x|y)$  é obtido por  $g(y, \theta)$  onde a reconstrução  $\hat{x}$  é amostrado de  $p_\theta(x; g(y, \theta))$ . A escolha pelas distribuições estão abertas a qualquer tipo de distribuição paramétrica. Para a distribuição da variável latente  $y$ , que são  $p_\theta(y)$  e  $q_\phi(y|x)$ , a escolha comum é a distribuição normal. Já para  $p_\theta(x|y)$ , é comum se utilizar a distribuição de Bernoulli ou a gaussiana multivariável. [1]

O *variational autoencoder* também é treinado utilizando o algoritmo de *Backpropagation*. E na figura 2.8, é possível ver a arquitetura completa do *variational autoencoder*, com o *encoder* e o *decoder* conectados pela variável latente  $y$ .

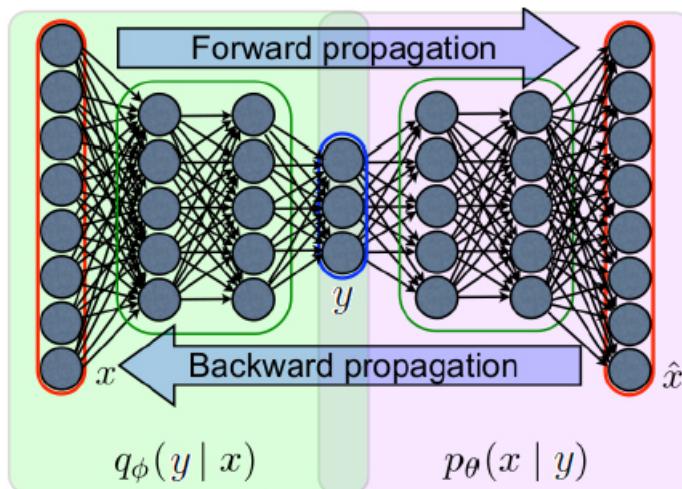


Figura 2.8: Arquitetura do *variational autoencoder*. Fonte: [1]

## 2.4 Resumo do Capítulo

Nesse capítulo foram apresentado os conceitos básicos de Redes Neurais Artificiais, *Principal Component Analysis* e sua relação com o *Autoencoder*. Além disso, foram apresentados os conceitos de *denoising autoencoder*, *stacked autoencoder*, *variational autoencoder* e *sparse autoencoder*, algumas das variações do algoritmo original do *autoencoder*.



# Capítulo 3

## Metodologia

### 3.1 Aplicação de *autoencoder* em problemas de detecção de falhas

[14, Tagawa 2014] trabalha com uma extensão do *denoising autoencoder* para conseguir utilizar informações iniciais incompletas. Esta decisão veio do fato de que modelos convencionais baseados nos dados sofrem com *overfitting* (sobreajuste) das funções e de que abordagens baseadas em modelos físicos sofrem com a falta de informações específicas de sistemas complexos.

Para tanto, o método proposto utiliza informações sobre a estrutura dos dados, como a relação entre as variáveis, e também conhecimento sobre as possíveis falhas e isto é contabilizado na função objetiva. A partir disso, o *structured denoising autoencoder*, como é chamado no trabalho, foi aplicado em dados sintéticos para detecção e análise de falhas e foi o único método capaz de extrair as causas reais dos dados falhos dentre os testados.

[19, Zhang 2016] utiliza uma abordagem baseada em *autoencoder* para detectar falhas em um processo siderúrgico, em que picos ocorrem por conta de trocas arbitrárias entre dois fornos. Monitorar esse tipo de processo é particularmente difícil por ser um ambiente com muitos ruídos e que obriga o uso de medições indiretas das variáveis. E, exatamente por conta disso, se optou por utilizar um método conhecido por extrair características robustas do sistema: o *denoising autoencoder*. Nesse trabalho, um *deep autoencoder* foi construído com base em 4 camadas de *denoising autoencoder* treinadas previamente. Para validar o método, foram utilizados dados de falha de forno frio e, em comparação com um PCA de duas etapas, o modelo construído foi efetivo em extrair características robustas do processo e mostrou potencial para monitorar processos.

[13, Sun 2014] focou seu trabalho em inspecionar a vibração de componentes e sistemas eletromecânicos de forma a aumentar a confiabilidade do processo e garantir sua qualidade. Para tanto, construiu um método denominado *Tilear* baseado em *autoencoders*. Para treinamento, apenas dados de motores elétricos sem falhas foram utilizados para se extrair as principais características do sistema. Em uma segunda parte, o *Tilear* apresenta uma tomada de decisão, comparando o sinal gravado para teste e o sinal reconstruído para calcular quão bem o motor testado se adequa ao modelo calculado. A partir da detecção de anomalias, uma decisão confiável pode ser tomada com performance semelhante à técnica de Máquina de Vetores de Suporte (*Support Vector Machine*).

[16, Wang 2016] desenvolve um método aplicado para reconhecimento de falhas de um

transformador. É composto por duas camadas de *sparse autoencoder* contínuo para obter as características do processo e depois uma camada para classificação da falha. O modelo de *sparse autoencoder* tem as vantagens de reconhecimento de dados contínuos, capacidade de aprendizagem não-supervisionada, alta precisão e robustez. Em contrapartida, o modelo exige um longo tempo para treinamento e um computador de alto desempenho. Como conclusão, os experimentos comparativos mostram que o método proposto pode extrair características dos dados originais e alcançar uma taxa de acerto superior na detecção de falhas.

[4, Chopra 2015] aplica uma técnica baseada em *sparse autoencoders* para detecção e análise de falhas de motores de combustão interna utilizando seus sinais acústicos. Por ser baseado em um *autoencoder*, esse método dispensa extração manual de recursos e seleção prévia dos dados de falha. Após extrair as características do sistema, os dados são usados na regressão softmax para classificá-los em defeituosos ou não. Nesse trabalho, um conjunto de dados de motores de combustão interna foi utilizado para testes com quatro classes diferentes de falhas. Uma das vantagens no uso de *sparse autoencoder* para aprender características de falha é a melhora o desempenho de classificação a partir de um pequeno número de dados, obtendo resultados superiores a 98% de acerto.

[1, An 2015] traz um método de detecção de falhas baseado na probabilidade de reconstrução do *variational autoencoder*. E uma vez que a probabilidade de reconstrução leva em conta a variabilidade da distribuição das variáveis, é possível obter uma pontuação de anomalia mais objetiva do que o erro de reconstrução dos métodos baseados em *autoencoder* simples. E de acordo com resultados experimentais, concluiu-se que a performance do *variational autoencoder* foi superior que a de outros métodos. Além disso, o método proposto permite que seja feita uma análise da causa da anomalia a partir da reconstrução dos dados.

## 3.2 Metodologia de desenvolvimento do projeto

Para treinamento do modelo baseado em *autoencoder*, amostras com informações sem falhas são utilizadas primeiramente. Assim, depois de treinada, a rede irá reconstruir os dados normais muito bem, enquanto irá falhar ao fazer o mesmo com dados anômalos que o *autoencoder* não havia conhecido.[1]

Assim, o erro de reconstrução é utilizado como escore de anomalia, detectando se uma amostra foi bem reconstruída pelo modelo ou não. Amostras que apresentarem erro de reconstrução maior que o limite definido são consideradas com falhas [1]. Esse procedimento pode ser acompanhado no algoritmo 2.

**Algoritmo 2:** Algoritmo de detecção de falhas baseado em *Autoencoder*. Fonte: [1]

---

**Data:** Dados normais  $X$ , dados anômalos  $x^{(i)} \quad i = 1, \dots, N$ , limite  $\alpha$

**Result:** Definição de falha ou não

```

1  $\phi, \theta \leftarrow$  treina um autoencoder utilizando os dados normais  $X$ 
2 for  $i=1$  to  $N$  do
3   erro de reconstrução ( $i$ ) =  $RMSE(x^{(i)}, g_\theta(f_\phi(x^{(i)})))$ 
4   if erro de reconstrução ( $i$ ) >  $\alpha$  then
5     |  $x^{(i)}$  é uma falha
6   end
7   else
8     |  $x^{(i)}$  não é uma falha
9   end
10 end

```

---

### 3.2.1 Indicadores de Performance

Para que se compare o desempenho dos diferentes modelos desenvolvidos, diversos cálculos podem ser realizados. A partir dos valores obtidos, é possível definir qual modelo pode ser mais indicado para a aplicação desejada. A seguir, estão listados os índices utilizados nesse projeto como indicadores de performance.

#### Erro médio quadrático (RMSE)

O erro médio quadrático (Root Mean Square Error) é um índice que traz a média da amplitude do erro do resultado do modelo em relação ao real. Geralmente o erro quadrático médio é utilizado quando pretende-se destacar os erros absolutos maiores em amplitude, uma vez que o erro é elevado ao quadrado no cálculo. Seu cálculo é dado por:

$$RMSE = \sqrt{\frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2} \quad (3.1)$$

Neste trabalho, o RMSE será utilizado no cálculo do erro de reconstrução para ajuste do *autoencoder*.

#### Área sob a curva ROC (AUROC)

A análise do Receptor operacional de características (ROC) é originalmente um método para medição da performance de diagnósticos médicos. É então utilizado para problemas de classificação entre duas classes [17]. No caso desse projeto, trata-se da existência ou não de falha na amostra de dados analisada.

Uma forma de medir a performance de um classificador é calcular o índice de verdadeiros positivos (TP) e falsos positivos (FP) para um conjunto de dados. Para este trabalho, a taxa de TP é a porcentagem de amostras com falhas que foram corretamente apontadas como tal. Já a taxa de FP é a porcentagem de amostras que são normais, mas que tiveram falha apontada. [17]

Com base na relação entre a classificação de verdadeiros positivos e falsos positivos é construída a curva ROC. Na figura 3.1 são colocados dois exemplos de curvas ROC. Quanto maior a área sob a curva (AUROC), melhor é a classificação do modelo, visto que gera mais verdadeiros positivos e menos falsos positivos. [17]

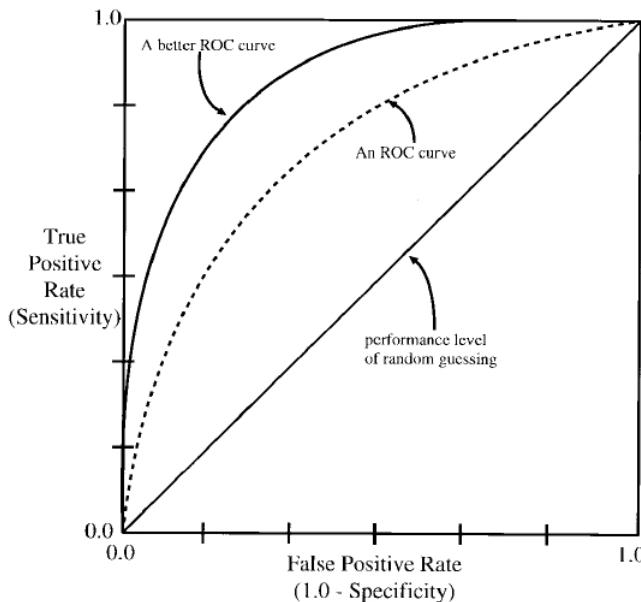


Figura 3.1: Dois exemplos de curvas ROC e a curva esperada para tentativa aleatória. Quanto maior a área sob a curva ROC, melhor a classificação realizada pelo modelo. Fonte:[17]

### Tempo de execução

Por fim, uma falha em um sistema ou em uma planta industrial pode trazer consequências graves se não for atendida rapidamente. Por conta disso, o tempo de execução também é um importante índice a ser considerado ao comparar a performance de um modelo. De acordo com a aplicação do modelo, pode-se inclusive considerar mais indicado um modelo que possua maior taxa de erros mas que gere resultados em menor tempo.

## 3.3 Resumo do Capítulo

Nesse capítulo foram apresentados trabalhos já realizados para detecção de falhas utilizando *autoencoders* e suas variações. Posteriormente, foi apresentada qual a metodologia utilizada para o desenvolvimento do projeto. Por fim, explicou-se quais parâmetros poderiam ser utilizados para fins de comparação entre os modelos desenvolvidos.

# Capítulo 4

## Resultados

## 4.1 Tennessee Eastman Process

Nesse projeto, os dados do *Tennessee Eastman Process* (TEP) foram utilizados como base para testar os diferentes tipos de *autoencoder*. O TEP é um problema criado pela Eastman Chemical Company para prover um processo industrial realista para avaliação de métodos de controle e monitoramento de processo e é comumente utilizado pela comunidade científica para esses fins. [6]

O problema de Tennessee Eastman requer a coordenação de cinco unidades principais de operação: Um reator, um separador, um condensador, um removedor e um compressor. O fluxo do processo pode ser acompanhado na figura 4.1. [6]

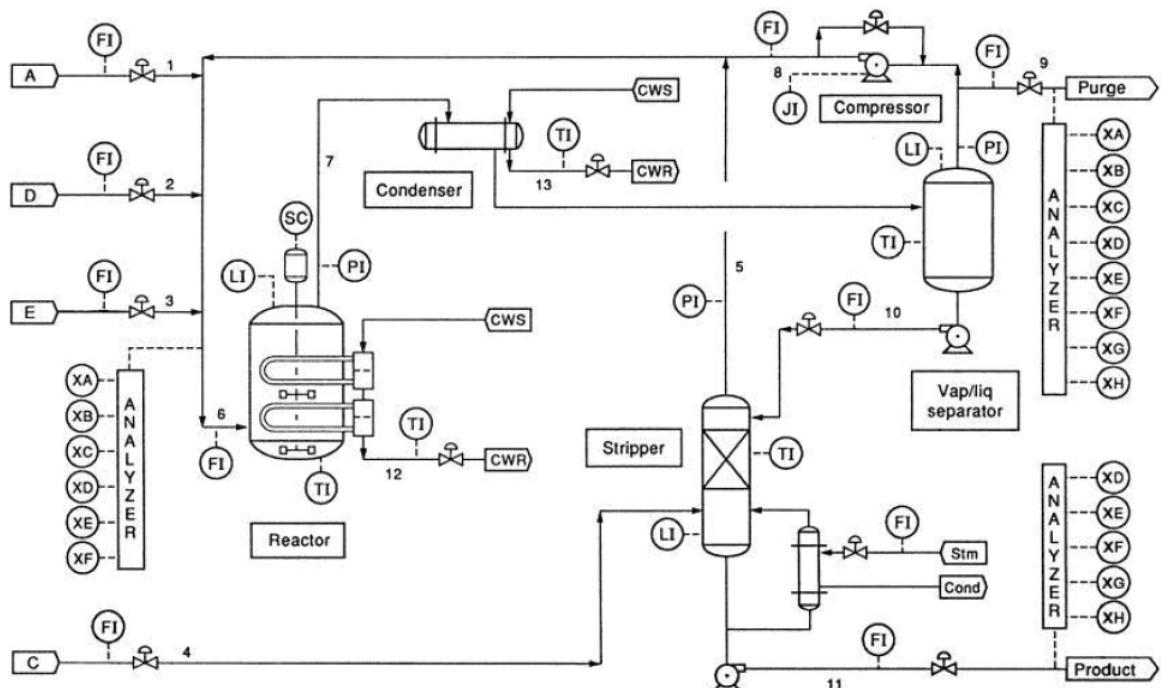


Figura 4.1: Esquema do Tennessee Eastman Process Fonte:[5]

A planta possui oito componentes: A, B, C, D, E, F, G e H. Os componentes A, C, D e E são reagentes gasosos e B é um gás inerte que é injetado no reator. Então os líquidos G e H

Tabela 4.1: Variáveis do TEP utilizadas nas análises. Fonte: [9]

Variable	Description
$t(1)$	Feed A (Stream 1)
$t(2)$	Feed D (Stream 2)
$t(3)$	Feed E (Stream 3)
$t(4)$	Total Feed (Stream 4)
$t(5)$	Recycle Flow (Stream 8)
$t(6)$	Reactor Feed Rate (Stream 6)
$t(7)$	Reactor Pressure
$t(8)$	Reactor Level
$t(9)$	Reactor Temperature
$t(10)$	Purge Rate (Stream 9)
$t(11)$	Product Separator Temperature
$t(12)$	Product Separator Level
$t(13)$	Product Separator Pressure
$t(14)$	Product Separator Underflow (Stream 10)
$t(15)$	Stripper Level
$t(16)$	Stripper Pressure
$t(17)$	Stripper Underflow (Stream 11)
$t(18)$	Stripper Temperature
$t(19)$	Stripper Steam Flow
$t(20)$	Compressor Work
$t(21)$	Reactor Cooling Water Outlet Temperature
$t(22)$	Separator Cooling Water Outlet Temperature

são produzidos. O componente F é um subproduto das reações. [9]

Os dados utilizados foram os mesmos de [9, Nakamura], obtidos utilizando o modelo *Simulink* do TEP [5] sob uma estratégia de controle descentralizada [11]. O simulador pode ser baixado em <http://depts.washington.edu/control/LARRY/TE/download.html>. [18]

O processo possui 41 variáveis medidas e 12 variáveis manipuladas. Mas, por conta da uniformidade no período de amostragem de 6 minutos, apenas as primeiras 22 variáveis medidas foram utilizadas nos testes. As descrições delas estão na tabela 4.1. As outras 19 variáveis medidas possuem diferentes frequências de amostragem e são medições dos componentes da planta, tornando seu uso direto inviável. [9]

O modelo *Simulink* possui 21 modos de operação, sendo um normal e o restante situações de falhas. As descrições das falhas podem ser vistas na tabela 4.2. [9]

E a título de demonstração, na figura 4.2 há a série de dados da variável  $t(1)$  (Feed A) no modo de operação normal e também em três diferentes introduções de falhas.

## 4.2 Testes Realizados

Para comparação do desempenho dos diferentes tipos de *autoencoder* listados na Revisão Bibliográfica, foram realizados testes utilizando os dados do TEP. Nos dados com falhas utilizados, a falha ocorre sempre na amostra de número 100. E, por se tratar de uma série

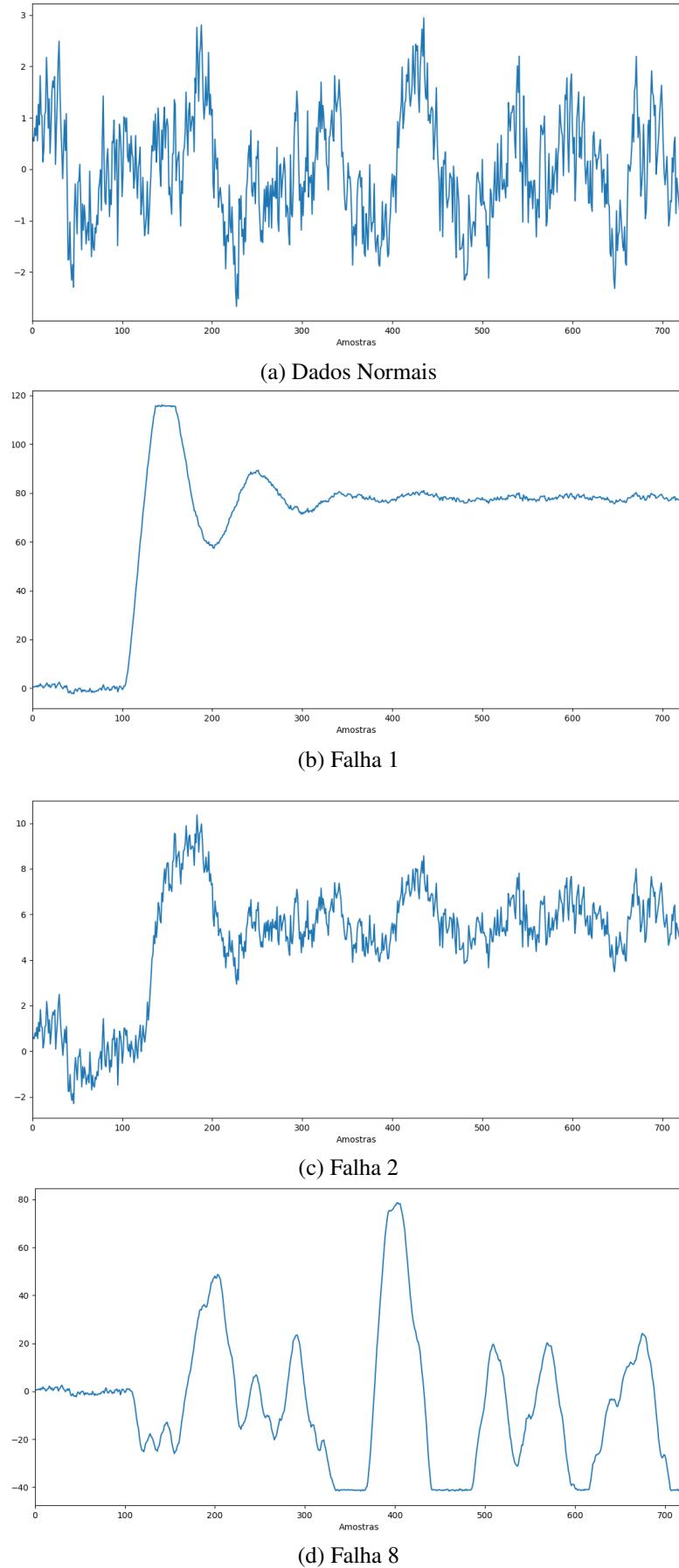
Figura 4.2: Exemplo dos dados da variável  $t(1)$ .

Tabela 4.2: Descrição das situações de falhas do TEP. Fonte: [12]

Variable	Description
IDV(1)	A/C feed ratio, B composition constant (Stream 4)
IDV(2)	B composition, A/C ratio constant (Stream 4)
IDV(3)	D feed temperature (Stream 2)
IDV(4)	Reactor cooling water inlet temperature
IDV(5)	Condenser cooling water inlet temperature
IDV(6)	A feed loss (Stream 1)
IDV(7)	C header pressure loss - reduced availability (Stream 4)
IDV(8)	A, B, C feed composition (Stream 4)
IDV(9)	D feed temperature (Stream 2)
IDV(10)	C feed temperature (Stream 4)
IDV(11)	Reactor cooling water inlet temperature
IDV(12)	Condenser cooling water inlet temperature
IDV(13)	Reaction kinetics
IDV(14)	Reactor cooling water valve
IDV(15)	Condenser cooling water valve
IDV(16)	Unknown
IDV(17)	Unknown
IDV(18)	Unknown
IDV(19)	Unknown
IDV(20)	Unknown

temporal, o *encoder* e o *decoder* devem ser modelos capazes de trabalhar com estruturas temporais. Como sugerido pela biblioteca [3, Keras], foi utilizado o LSTM (*Long short-term memory*). Mais informações sobre o LSTM podem ser encontradas em [7].

Para cada um dos tipos de *autoencoder*, um modelo foi treinado para os dados sem falhas. E para gerar o melhor modelo de cada tipo, a quantidade de épocas variou de 1 a 20; a dimensão da *Bottleneck Layer* variou de 1 a 10; a quantidade de pontos anteriores utilizados para a construção da série temporal variou de 1 a 8; e o limite admitido para erro de reconstrução variou de 1.0 a 2.0 vezes o erro de reconstrução dos dados sem falhas. Para o *denoising autoencoder*, o fator de ruído variou de 0.1 a 0.9. Para o *sparse autoencoder*, o regulador do número de ativações em cada camada (*regularizer*) variou de  $10^{-1}$  a  $10^{-10}$ . Já para o *variational autoencoder*, o tamanho da camada latente variou de 1 a 5 e o valor de  $\epsilon$  utilizado para gerar amostras a partir da camada latente variou de 0.1 a 1.0.

Na tabela 4.3, há a compilação dos parâmetros utilizados em cada um dos modelos para gerar os melhores resultados.

Então cada um dos conjuntos de dados que apresentavam falhas foi apresentado ao modelo para que o erro de reconstrução fosse analisado. Caso esse erro fosse maior que o limite, o dado era considerado falha. E para cada modelo construído e para cada conjunto de dados aplicado, foi construída uma curvas ROC e calculada a respectiva áreas sob a curva. Além disso, foi calculado o tempo total gasto por cada modelo para seu treinamento e teste.

Os modelos foram desenvolvidos em Python 3.5.3 utilizando a biblioteca [3, Keras] 2.0.4 e TensorFlow 1.0.1. O ambiente de teste foi um notebook HP Probook 4530s, com processador Intel Core i3 2.10GHz e memória RAM de 4.00 GB.

Tabela 4.3: Parâmetros utilizados para treinar os diferentes modelos

Modelo	Épocas	Bottleneck	Pontos ant.	Limite	Ruído	Regulador	Latente	$\epsilon$
Simples	16	2	2	1.2	-	-	-	-
Stacked	4	3	3	1.1	-	-	-	-
Denoising	25	6	3	1.0	0.5	-	-	-
Sparse	16	9	6	1.1	-	$10^{-5}$	-	-
Variational	7	4	3	1.8	-	-	1	3

Tabela 4.4: Resultados obtidos nos testes

Modelo	Soma 20 AUROCs	Soma 12 AUROCs	Tempo treino (s)	Tempo testes (s)
Simples	12.10	8.19	26.1	29.8
Stacked	12.45	8.36	18.1	4.1
Denoising	11.48	6.38	34.6	27.0
Sparse	12.56	8.57	33.9	37.7
Variational	11.56	7.61	24.6	13.4

## 4.3 Resultados Obtidos

As falhas 3, 4, 13, 16, 17, 18 e 20 não apresentam variações suficientes ao longo do tempo quando comparado com os dados normais. Por conta disso, nenhum modelo apresentou bons resultados para detecção dessas falhas. Já a falha 6 possui um comportamento instável e também não apresentou bons resultados em nenhum dos testes.

Na tabela 4.4, podem ser vistos os resultados obtidos nos testes com os modelos desenvolvidos. O primeiro índice indica a soma de todas as áreas sob as curvas ROCs construídas para cada uma das 20 entradas com falhas. O segundo índice é a soma das AUROCs para as entradas com falhas 1, 2, 5, 7, 8, 9, 10, 11, 12, 14, 15, 19. O terceiro índice apresentado é o tempo de treinamento do modelo. E o quarto índice é o tempo total gasto testando o modelo com as 20 entradas com falhas. Quanto maior a AUROC, melhor o modelo e quanto menor o tempo gasto, melhor o modelo.

Na figura 4.3, há as curvas ROC apresentando o desempenho de cada modelo para os testes contendo as falhas 1, 2, 5, 7, 8, 9, 10, 11, 12, 14, 15, 19.

De acordo com os resultados obtidos, então, podemos notar que:

- O *sparse autoencoder* apresentou o melhor resultado entre os modelos testados em relação à soma das AUROCs. Em contrapartida, foi o que levou mais tempo para ser treinado e testado. Sendo assim, não se mostra a melhor solução para casos em que o tempo é uma restrição importante. A restrição de ativações dos neurônios nesse tipo de rede neural força o modelo a aprender uma representação consistente dos dados de entrada. Sendo assim, quando amostras com falhas são apresentadas à rede, ela nota com maior precisão que não se tratam de amostras parecidas com as normais.
- O *stacked autoencoder* apresentou o segundo melhor resultado segundo a soma das AUROCs, além de ter levado o menor tempo para treino e testes. Esse baixo tempo de treino ocorre porque esse tipo de *autoencoder* treina camadas intermediárias de neurônios antes da camada mais oculta. Assim, é possível extraír uma representação consistente dos dados sem a necessidade de um grande número de épocas.

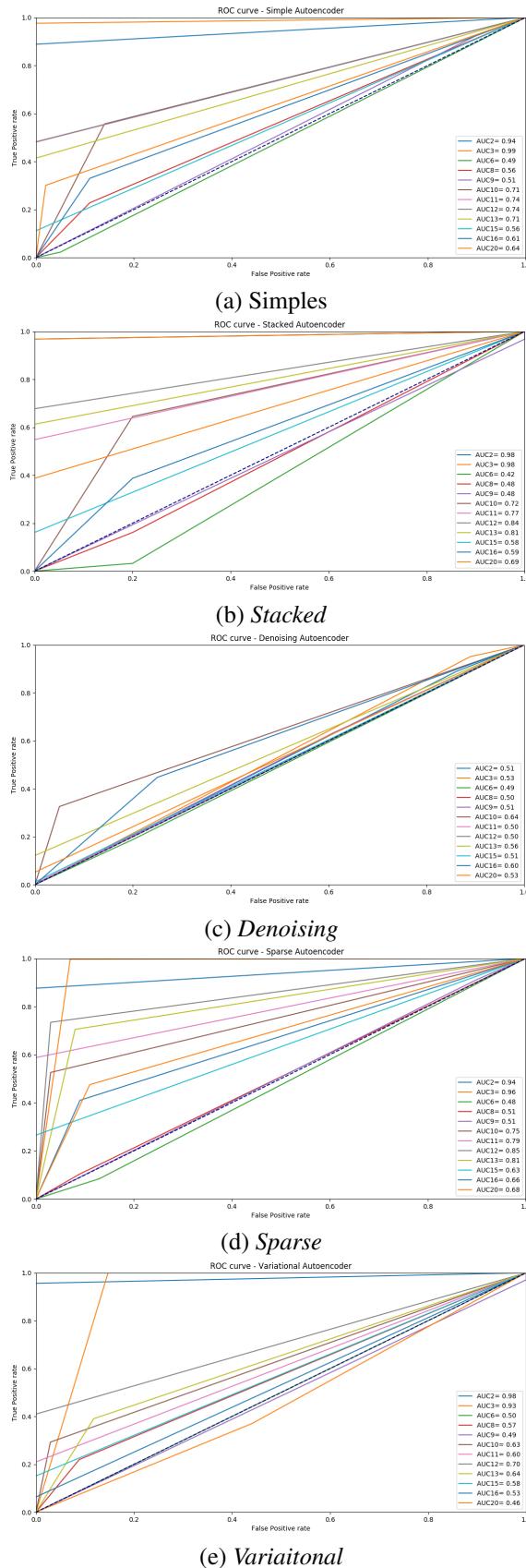


Figura 4.3: Curvas ROC para os testes com as falhas 1, 2, 5, 7, 8, 9, 10, 11, 12, 14, 15, 19.

- O *Variational Autoencoder* apresentou o segundo pior resultado de soma de AUROCs, mas também apresentou o segundo melhor tempo de treino e testes.
- Já o *denoising Autoencoder* apresentou o menor resultado de soma de AUROCs e também o maior tempo de teste e treino. Dessa forma, se mostrou como o modelo menos eficiente para detecção de falhas no TEP. Isso se deve ao fato de que esse tipo de *autoencoder* é capaz de gerar dados sem falhas a partir de dados corrompidos, dessa forma, pode receber amostras com falhas e obter resultados próximos de dados normais ao decodificar a representação e não conseguir detectar a falha presente.

## 4.4 Resumo do Capítulo

Nesse capítulo, foram apresentados os dados utilizados para realização dos testes. Também foi explicado como os modelos foram ajustados e foram descritos os testes realizados para análise do desempenho dos diferentes tipos de *autoencoder*. Por fim, são apresentados os resultados comparativos dos testes.



# Capítulo 5

## Conclusões

### 5.1 Considerações Finais

Nos últimos anos, a confiabilidade do sistemas têm sido cada vez mais exigida pelos diversos tipos de empresas. Assim, cada falha ocorrida deve ser analisada para encontrar as causas e as soluções. Para que não se dependa somente dos diferentes níveis de experiências dos especialistas do processo, técnicas de inteligência computacional estão sendo utilizadas para detecção e análise de falhas como uma forma de apoio à decisão. Tendo isso em vista, este trabalho trouxe a comparação de desempenho de cinco diferentes tipos de *autoencoder* para analisar o quanto capazes cada um dos modelos são em detectar falhas e o quanto rápido conseguem ser treinados e aplicados.

A base de dados utilizada foi gerada a partir do *Tennessee Eastman Process* e constitui em uma série temporal sem a presença de erros e mais vinte séries com a presença de uma falha. Ao desenvolver e treinar os tipos de *autoencoder* propostos utilizando esses dados, o *sparse autoencoder* apresentou a maior área sob a curva ROC. Apesar disso, esse modelo levou o maior tempo entre todos para ser treinado. Já o *stacked autoencoder* foi o que apresentou melhor desempenho em relação ao tempo de treinamento e a segunda maior área sob a curva ROC, portanto é mais indicado para situações que o tempo é uma restrição importante. E dentre todos, o *denoising autoencoder* foi o que apresentou maior custo computacional e os piores resultados nos testes.

### 5.2 Propostas de Continuidade

Nesse trabalho foram construídos cinco modelos, cada um com base em apenas um tipo de *autoencoder*. Propõe-se então o estudo do desempenho de modelos que apresentem tipos associados de *autoencoder*, como o *sparse* associado ao *stacked*, por exemplo.

É também proposto que se analise o desempenho dos *autoencoders* na detecção de falhas em dados reais. Situações possíveis são a aplicação do modelo proposto em sistemas embarcados em automóveis ou então em sistema de automação residencial, por exemplo.

Por fim, propõe-se como próximo passo fazer também a análise de falhas, além da sua detecção. Assim o sistema poderá não só apontar uma falha, mas também definir características específicas de cada falha.



# Referências Bibliográficas

- [1] J. AN and S. CHO. Variational autoencoder based anomaly detection using reconstruction probability. 2015.
- [2] D. BLEI. Cos 424: Interacting with data. 2008.
- [3] F. CHOLLET. Keras. <https://github.com/fchollet/keras>, 2015.
- [4] P. CHOPRA and S. YADAV. Fault detection and classification by unsupervised feature extraction and dimensionality reduction. In *Complex Intell. Syst.*, 2015.
- [5] J. J. DOWNS and E. F. VOGEL. A plant-wide industrial process control problem. In *Computers chem. Engng.*, 1993.
- [6] R. ESLAMLOUEYAN. Designing a hierarchical neural network based on fuzzy clustering for fault diagnosis of the tennessee eastman process. In *Applied Soft Computing*, 2011.
- [7] SCHRAUDOLPH N. N. GERS, F. A. and J. SCHMIDHUBER. Learning precise timing with lstm recurrent networks. In *Journal of Machine Learning Research 3*, 2002.
- [8] J. R. JANG. *Neuro-Fuzzy and Soft Computing: A Computational Approach to Learning and Machine Intelligence*. Prentice-Hall, upper saddle river, NJ, 1997.
- [9] T. NAKAMURA et. al. Adaptive fault detection and diagnosis using parsimonious gaussian mixture models trained with distributed computing techniques. In *Journal of the Franklin Institute*.
- [10] L. C. OLIVEIRA. Utilização de inteligência computacional no controle estático de convertedores LD (Linz-Donawitz) - versão preliminar, 2007.
- [11] N. L. RICKER. Decentralized control of the tennessee eastman challenge process. In *Journal of Process Control*, 1995.
- [12] CHIANG L. H. RUSSELL, E. L. and R. D. BRAATZ. Fault detection in industrial processes using canonical variate analysis and dynamic principal component analysis. In *Chemometrics and Intelligent Laboratory Systems 51*, 2000.
- [13] J. SUN et al. Automated fault detection using deep belief networks for the quality inspection of electromotors. In *Technisches Messen*, 2014.
- [14] T. TAGAWA and T. YARI. Denoising autoencoder for fault detection and analysis. In *JMLR: WORKSHOP AND CONFERENCE*, 2014.
- [15] P. VINCENT et al. Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion. In *Journal of Machine Learning Research*, 2010.

- [16] L. WANG et al. Transformer fault diagnosis using continuous sparse autoencoder, 2016.
- [17] K. WOODS and K. BOWYER. Generating roc curves for artificial neural netorks. *IEEE Transactions on medical imaging*, 16(3):329–337, 1997.
- [18] S. Yin et al. A comparison study of basic data-driven fault diagnosis and process monitoring methods on the benchmark tennessee eastman process. In *Journal of Process Control*, 2012.
- [19] T. ZHANG et al. Fault detection for ironmaking process based on stacked denoising autoencoders. In *American Control Conference (ACC)*, 2016.