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, fevereiro 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, fevereiro de 2017

Lista de Figuras

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 [16].

Na maioria dos casos, essa detecção de falhas é feita por técnicos capacitados, cuja decisão por ser influenciada por seus diferentes treinamentos e níveis de experiência. Assim, esse trabalho pode não ser tão confiável quanto se exige [2]. Além disso, o aumento da complexidade dos sistemas implica também em uma análise ainda mais custosa [16].

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 [12].

Uma dessas técnicas é o *autoencoder*, uma rede neural onde a seleção de características e sua extração são não supervisionadas [2]. 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 a sua última camada, a *bottleneck layer* (camada gargalo) [2]. Nesse processo, características dos dados de entrada são extraídas e depois são armazenadas nessa última camada [11].

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 [14][11]. Então ao ser treinado, o modelo baseado em *autoencoder* busca a minimização do erro de reconstrução [2], 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. Nisso, a *bottleneck layer* obtida é a melhor representação comprimida possível para o vetor de entrada [11].

Uma das vantagens em se utilizar os *autoencoder*s 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 *loss function* (função de perda) [16].

Além dos *autoencoder*s 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 generativo 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 [2]. Para tanto, sua diferença do *autoencoder* simples é a adição de escassez na função de custo do modelo [2]. Assim, o número de ativações nos neurônio na camada oculta é limitado, fazendo com que o espaço de características selecionadas seja ainda mais comprimido [2]. Além disso, [14, 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 [12]. 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 [16].

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*: a necessidade de poucas informações sobre os possíveis ruídos e distúrbios [16]. Além disso, segundo [16, 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 analisar 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 três capítulos. O primeiro traz a introdução ao problema e os objetivos que se pretende alcançar ao final de sua construção. O capítulo 2 contempla

conceitos básicos de redes neurais e *autoencoders*, de formas que o leitor compreenda melhor o que foi utilizado no trabalho. No capítulo 3 está a descrição da metodologia utilizada para o desenvolvimento do projeto. Além disso, esse capítulo traz uma breve visão do que já foi desenvolvido na área por outros pesquisadores.

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 [5]. E o neurônio 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 sob nesse aspecto. [8]

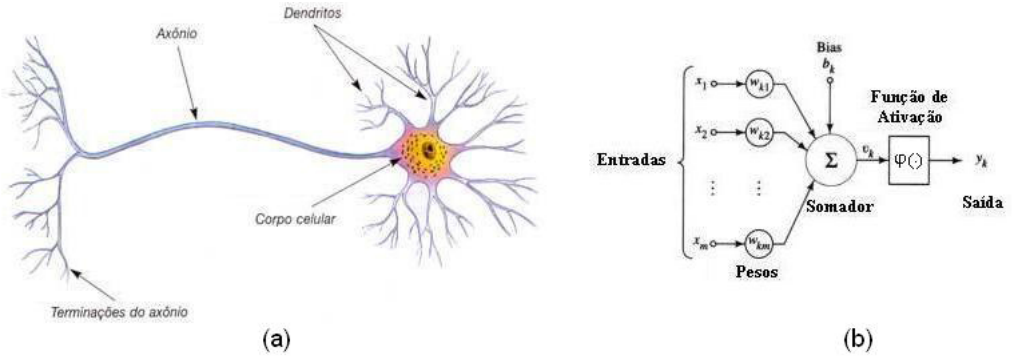

Figura 2.1: Comparação da estrutura do neurônio biológico (a) e do neurônio artificial (b). Fonte: [8]

As redes neurais artificiais podem ser classificadas segundo diversos critérios, como método de aprendizado, arquitetura, tipos de saída, tipos de nós ou implementação. Os principais critérios para o desenvolvimento deste trabalho estão melhor explicados nos itens a seguir.[5]

Arquitetura - tipos de conexão

1. Feedforwad: 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.[5]

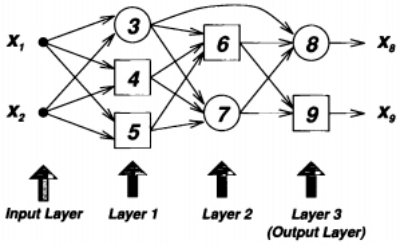

Figura 2.2: Exemplo de rede neural com arquitetura feedforward. Fonte: [5]

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. [5]

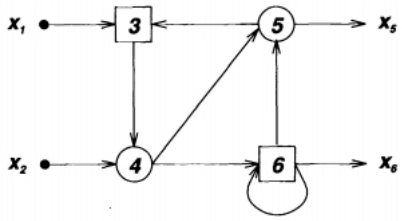

Figura 2.3: Exemplo de rede neural com arquitetura recorrente. Fonte: [5]

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. Nesse caso, uma vez definida, a rede não será alterada até que um novo conjunto de dados seja apresentado. [5]

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 constante de características.[5]

2.2 Análise de Componentes Principais (PCA)

O PCA (*Principal Component Analysis*) é um modelo geralmente utilizado em conexões com análise de contribuição. Com N dados de entrada x, o PCA gera a matriz de transformação

linear $W = w_1, ..., {w_d}^T$ 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: [12]

$$x_r = W^T W x = W^T y, \tag{2.1}$$

com um vetor de erro de reconstrução e dado por:

$$e = x - W^T W x = x - x_r, \tag{2.2}$$

Sendo $e = e_1, ..., e_N$, ele pode ser utilizado para calcular o RMSE (*Root mean square error*). A partir disso, pode-se calcular o escore de anomalias e com isso encontrar possíveis falhas. [12]

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.[12]

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 [16]. A partir disso, pode-se entender o *autoencoder* como um PCA não-linear.

Portanto, um *autoencoder* é um modelo de rede neural construído para aprender uma representação comprimida dos dados de entrada e então reconstruí-los o mais próximo possível a partir dessa representação [11][1][2]. 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 $f(.)$ afim seguida de uma não-linearidade:

  $$y = f_\theta(x) = s(Wx + b), \tag{2.3}$$

  onde $\theta = W, b$; $s(.)$ é uma função não linear, como a sigmoidal por exemplo; W é uma matriz de pesos de dimensão dxd; e b é um vetor de offset de dimensão d. [13]

- **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 $g(.)$ afim seguida de uma não-linearidade:

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

  sendo $\theta' = W', b'$; e opcionalmente W' pode ser definida por $W' = W^T$, indicando a situação de "pesos ligados".

  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. [16]

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 ver 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). [11]

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 perceber as matrizes de peso W no *encoder* e no *decoder* e os valores do vetor de *offset* b.

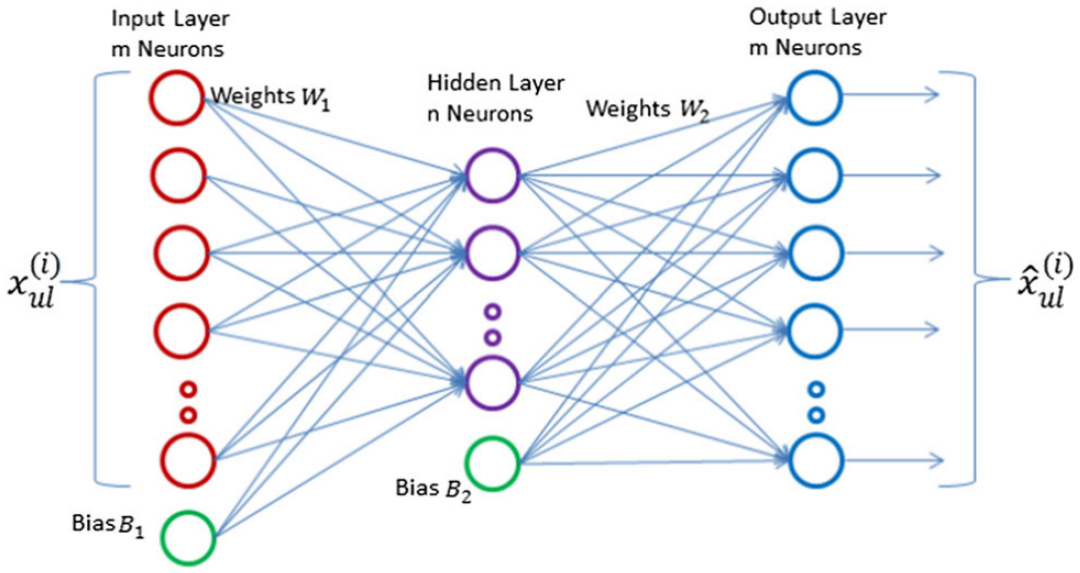

Figura 2.4: Estrutura do *autoencoder* com a camada de entrada, a camada oculta e a camada de saída. Fonte: [2]

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* de forma a minimizar esse erro dado por:

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

sendo que o erro $L(X, Z(\tilde{X}))$ pode ser definido de diversas formas, como por exemplo o erro quadrado: [16]

$$L(x, z) = ||x - z||^2 . \tag{2.6}$$

Em síntese, treinar um *autoencoder* implica em aprender os pesos e *offsets* utilizados nas camadas da rede neural para que as amostras de entrada possas ser reconstruídas o mais próximo possível do original. Para isso, os pesos das matrizes W e W' podem ser aperfeiçoados por *backpropagation* [11].

Sendo assim, um algoritmo de treinamento de um *autoencoder* pode ser dado por:

Algoritmo 1: Algoritmo de treinamento do Autoencoder. Fonte: [1]

Data: Dataset: $x^{(1)}, x^{(2)}, ..., x^{(N)}$
Result: encoder f_θ, decoder g_θ
$\phi, \theta \leftarrow$ Inicia parâmetros
while *convergência dos parâmetros* ϕ,θ **do**
    Calcula a soma do erro de reconstrução
    $\phi, \theta \leftarrow$ atualiza parâmetros utilizando gradiente de E
end

2.3.1 *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][12][16]. E há vários métodos para corrupção, como ruído de mascaramento, ruído "sal e pimenta", ruído gaussiano aditivo [16]. O mapeamento das características na camada oculta, nesse caso, pode ser expresso pela equação:

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

onde $\theta = W, b$; $s(.)$ 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* [16].

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 [16]. Isso significa ter z tão próximo quanto possível da entrada não corrompida [13].

É importante colocar que, para cada amostra de treinamento x apresentada ao modeloa, uma versão diferente de $\tilde{x}$ é gerada. Os parâmetros do *encoder* e do *decoder* são inicializado aleatoriamente e então otimizados por gradiente descendente [13].

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 [13]. E ainda assim a saída z deve ser uma reconstrução dos dados x não corrompidos [12].

Em síntese, a extração das características dos dados deve ser robusta às corrupções dos dados de entrada e preservar as informações da entrada original [16]. Isso então força o aprendizado de um mapeamento que extraia somente os padrões significativos dos dados [1].

Na figura 2.5, 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. [16]

Stacked Denoising autoencoder

Os *stacked denoising autoencoders* é uma rede neural profunda criada segundo algumas etapas descritas a seguir. Na figura 2.6, é possível observar a arquitetura de um *stacked*

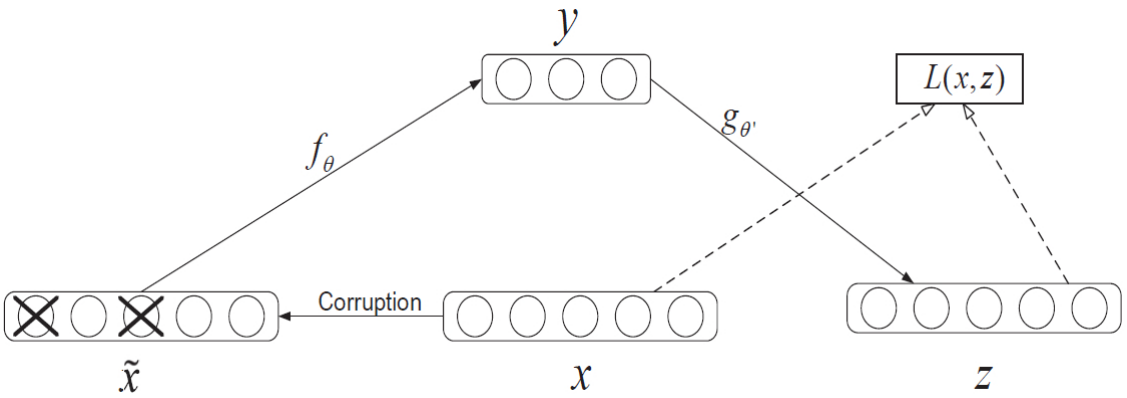

Figura 2.5: Arquitetura do *denoising autoencoder*. Alguns elementos de x são zerados aleatoriamente e a entrada corrompida $\tilde{x}$ é utilizada para mapear as características y e então reconstruir x produzindo z. A função de perda $L(x, z)$ é calculada segundo a saída e a entrada original. Fonte: [13]

autoencoder. Nela, 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. [16]

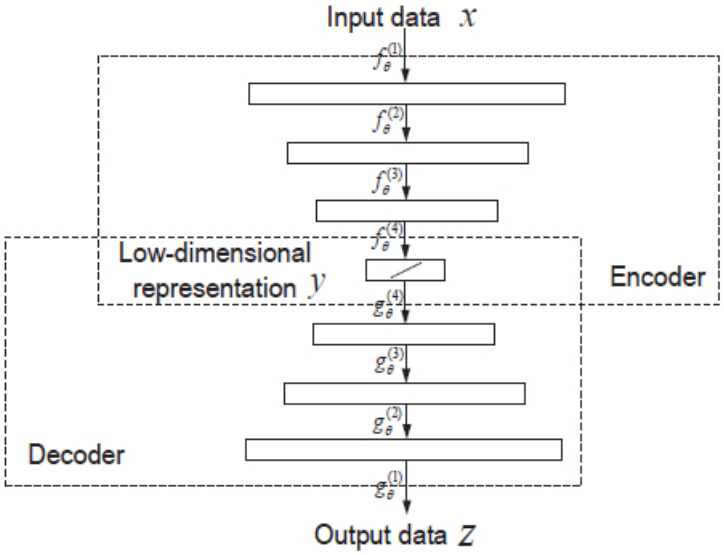

Figura 2.6: Arquitetura de um *stacked autoencoder*. Fonte: [16]

No empilhamento (*stacking*) do *denoising autoencoder*, a entrada corrompida é utilizada somente para o treinamento inicial de cada camada individualmente. E uma vez com a função de mapeamento f já definida, ela vai então ser utilizada em entradas não corrompidas. [13]

A representação de uma entrada limpa na camada anterior é calculada pelo *encoder*, e então ela é utilizada como entrada limpa para a próxima camada. Ao repetir esse procedimento, a representação final da entrada original pode ser obtida. Para reconstruir essa entrada original a partir da representação final, os *decoders* correspondentes devem ser empilhados em ordem reversa dos *encoders*. [16]

Na figura 2.7 é possível acompanhar, como exemplo, o empilhamento dos *encoders*: primeiramente foi encontrada uma função de mapeamento f_θ para a primeira camada utilizando do algoritmo do *denoising autoencoder*. O vetor de características encontrado a partir desse mapeamento é então corrompido e é encontrada a função de mapeamento $f_\theta^{(2)}$ que trará um novo vetor de características. Depois, as funções de mapeamento são utilizadas apenas com entradas limpas e empilhadas para formar o *stacked denoising autoencoder*.

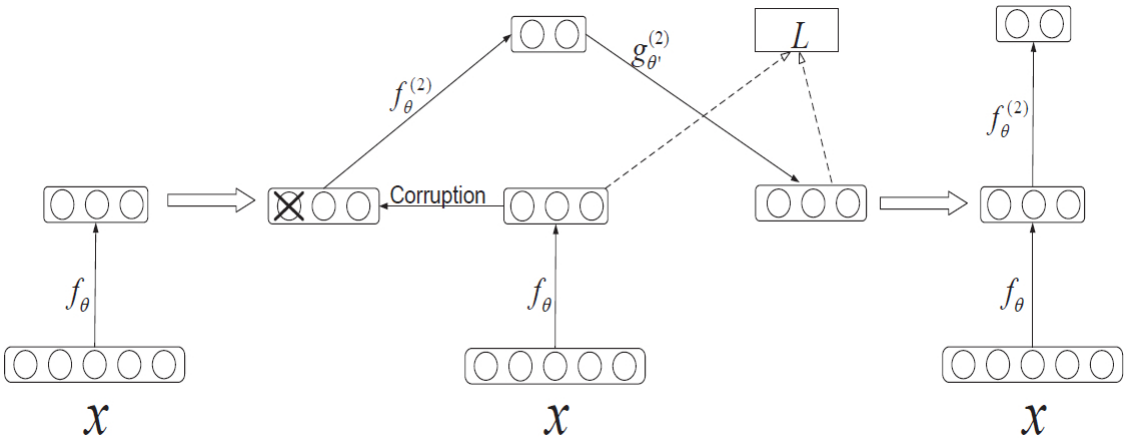

Figura 2.7: Treinamento inicial de um *stacked denoising autoencoder*. Fonte: [16]

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 descente. [16]

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. [13]

2.3.2 *Variational autoencoder*

A principal diferença entre o *variational autoencoder* e o *autoencoder* simples é que o *variational autoencoder* é um modelo generativo 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 *decorders* 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 2.8, a arquitetura descrita do *encoder* e do *decoder* podem ser vistas.[1]

A função objetivo de uma *variational autoencoder* é a restrição variacional da probabilidade marginal de dados dada pela soma da equação 2.8. Já a probabilidade marginal de cada ponto é dada pela equação 2.9. [1]

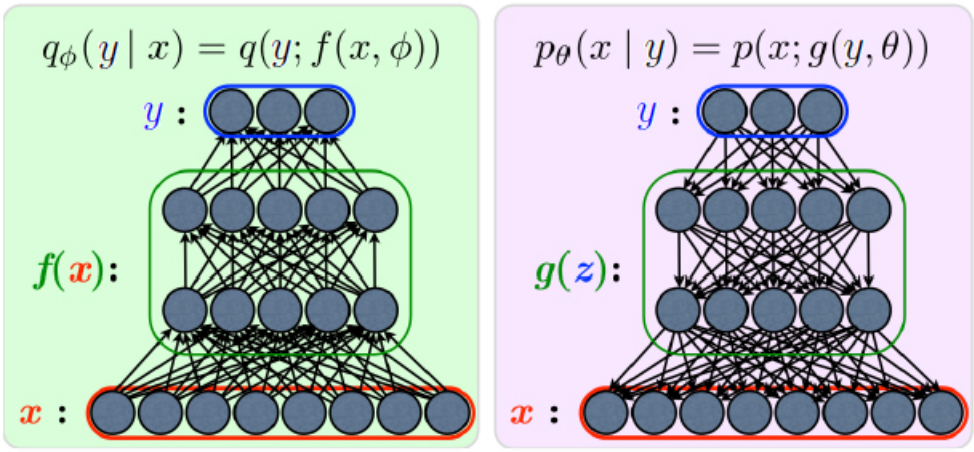

Figura 2.8: Arquitetura do *Encoder* e do *decoder* probabilísticos. Fonte: [1]

$$\log p_\theta(x^{(i)}, .., x^{(N)}) = \sum_{i=1}^{N} \log p_\theta(x^{(i)}) \tag{2.8}$$

$$\log p_\theta(x^{(i)}) = -D_{KL}(q_\phi(y|x)||p_\theta(y)) + E_{q_\phi(y|x^{(i)})}[\log p_\theta(x|y)] \tag{2.9}$$

onde $p_\theta(x|y)$ é a probabilidade de se obter x dada a variável latente y. O primeiro termo da soma da equação 2.9 é a divergência KL entre a aproximada posterior e a prévia da 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 2.9 pode ser entendido em termos ta 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*. Para que o segundo termo da equação 2.9 não precise ser calculado através do método de Monte Carlo, é feita uma reparametrização que utiliza uma variável aleatória de uma distribuição normal. Assim, a variável aleatória y $q_\phi(y|x)$ é representada por uma transformação determinística $h_\phi(\epsilon, x)$ onde ϵ é de uma distribuição normal:

$$\tilde{y} = h_\phi(\epsilon, x), \quad \epsilon\, \mathcal{N}(0, 1), \tag{2.10}$$

e a reparametrização deve garantir que $\tilde{y}$ siga a distribuição de $q_\phi(y|x)$. [1]

Na figura 2.9, é possível ver a arquitetura completa do *variational autoencoder*, com o *encoder* e o *decoder* conectados pela variável latente y.

Tendo isso em vista, um algoritmo para treinar o *variational autoencoder* pode ser visto a seguir:

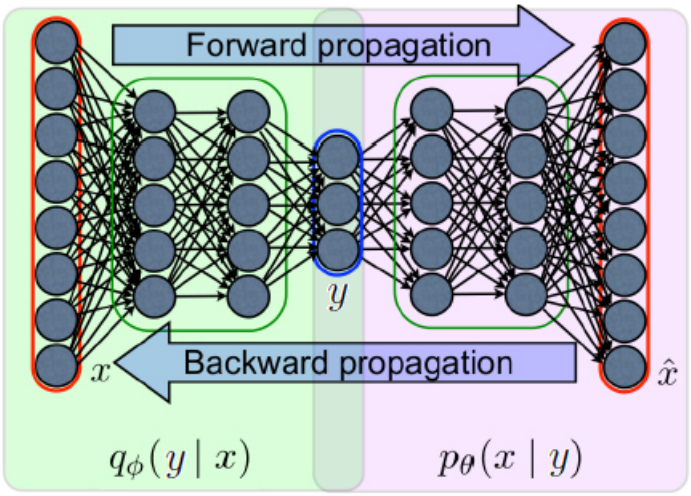

Figura 2.9: Arquitetura do *variational autoencoder*. Fonte: [1]

Algoritmo 2: Algoritmo de treinamento do *Variational Autoencoder*. Fonte: [1]

Data: Dataset: $x^{(1)}, x^{(2)}, ..., x^{(N)}$
Result: probabilistic encoder f_θ, probabilistic decoder g_θ

$\phi, \theta \leftarrow$ Inicia parâmetros
while *convergência dos parâmetros* ϕ,θ **do**
  for *i=1 to N* **do**
    desenhe L amostras de $\epsilon\, \mathcal{N}(0, 1)$
    $y^{i,l} = h_\phi(\epsilon^{(i)}, x^{(i)}) \quad i = 1, ..., N$
  end
  $E = \sum_{i=1}^{N}[-D_{KL}(q_\phi(y|x^{(i)})||p_\theta(y)) + \frac{1}{L}\sum_{l=1}^{L}(\log p_\theta(x^{(i)}|y^{(i,l)}))]$
  $\phi, \theta \leftarrow$ atualiza parâmetros utilizando gradiente de E
end

2.3.3 *Sparse autoencoder*

O *sparse autoencoder* é uma variação do *autoencoder* com uma dispersão adicionada à sua função de custo. E seu objetivo é resolver o seguinte problema de otimização para aprender as características dos dados de entrada: [2]

$$\min ||x - z||^2 + \lambda||W||^2, \tag{2.11}$$

sendo o termo z a reconstrução aproximada da entrada x. O vetor W é a ativação do vetor de entrada x e o y é o vetor de características aprendidas. Durante o treinamento, os pesos da matriz W e o vetor bias b são atualizados para reduzir o erro de reconstrução de x. [2]

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. [2]

A dispersão 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. Esta

dispersão é aplicada na função de custo do *sparse autoencoder* em termos de divergência de Kullback-Leibler (KL). Com isso, a função de custo total a ser minimizada com a dispersão é: [2]

$$C(W, B) = \frac{1}{2v} \sum_{i=1}^{v} ||x^{(i)} - z^{(i)}||^2 + \lambda ||W||^2 + \beta \sum_{j=1}^{m} KL(\rho||\hat{\rho}_j), \tag{2.12}$$

onde W é a soma dos pesos das duas camadas e o termo $KL(\rho||\hat{\rho}_j)$ é definido como:

$$KL(\rho||\hat{\rho}_j) = \rho \log \frac{\rho}{\hat{\rho}_j} + (1 - \rho) \log \frac{1 - \rho}{1 - \hat{\rho}_j}, \tag{2.13}$$

onde o parâmetro ρ é a dispersão desejada e é ela quem controla a ativação dos neurônios ocultos; $\hat{\rho}_j$ é a ativação média da camada oculta de neurônios; β controla o peso da penalidade de dispersão; e λ regulariza o decaimento do peso. [2]

2.4 Resumo do Capítulo

Nesse capítulo foram apresentado os conceitos básico de Redes Neurais Artificiais, *Principal Component Analysis* e sua relação com o *Autoencoder*. Além disso, foram apresentados os conceitos de *denoising autoencoder*, *stacked denoising 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

[12, 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 do 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 análise de falhas e foi o único método capaz de extrair as causas reais dos dados falhos dentre os testados.

[16, 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 método já existente de um PCA de duas etapas, o modelo construído foi efetivo em extrair características robustas do processo e mostrou potencial para monitorar processos.

[11, Sun 2014] focou seu trabalho em inspecionar a vibração de componentes e sistemas eletromecânicos de formas 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 de formas a 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*).

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

um transformador. É composto por duas camadas de *sparse autoencoder* contínuo para treinamento não-supervisionado para obter as características do processo e depois uma camada de *back propagation* para classificação da falha. O modelo de *sparse autoencoder* tem as vantagens de reconhecimento de dados contínuos, capacidade de aprendizagem não-supervisionado, 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.

[2, 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 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 anomalias 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 muito mais objetiva e baseada em princípios 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]. Caso uma anomalia seja detectada, um novo *autoencoder* é treinado. Então os novos dados são apresentados aos dois *autoencoders* construídos, caso o *autoencoder* original apresente um menor erro de reconstrução que o modelo construído com dados anômalos, os novos dados são considerados como normais. Já se o erro de reconstrução for menor, os novos dados são uma falha. E se ainda os dois erros de reconstrução forem maiores que o limite determinado, um novo *autoencoder* é treinado e assim sucessivamente. Esse procedimento pode ser acompanhado no algoritmo 3. [6]

Algoritmo 3: Algoritmo de detecção de anomalias baseado em *Autoencoder*. Fonte: [1], [7]

Data: Dados normais X, dados anômalos $x^{(i)} \quad i = 1, ..., N, \quad$ limite α
Result: Modelos utilizados por amostra
$\phi, \theta \leftarrow$ treina um *autoencoder* utilizando os dados normais X
for *i=1 to N* **do**
  for *modelo=0 to M* **do**
    erro de reconstrução $(i) = RMSE(x^{(i)}, g_\theta(f_\phi(x^{(i)})))$
  end
  if *min(erro de reconstrução* (i)*)*$> \alpha$ **then**
    $x^{(i)}$ é uma anomalia
    $\phi, \theta \leftarrow$ treina um novo *autoencoder* utilizando os dados com falha $x^{(i)}$
  end
  else
    $x^{(i)}$ não é uma anomalia
  end
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:

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

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

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 análise 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 comumente utilizado pela comunidade para comparação de abordagens no monitoramento e controle de processos. É um problema criado pela Eastman Chemical Company para prover um processo industrial realista para avaliação de métodos de controle e monitoramento de processo. [4]

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. [4]

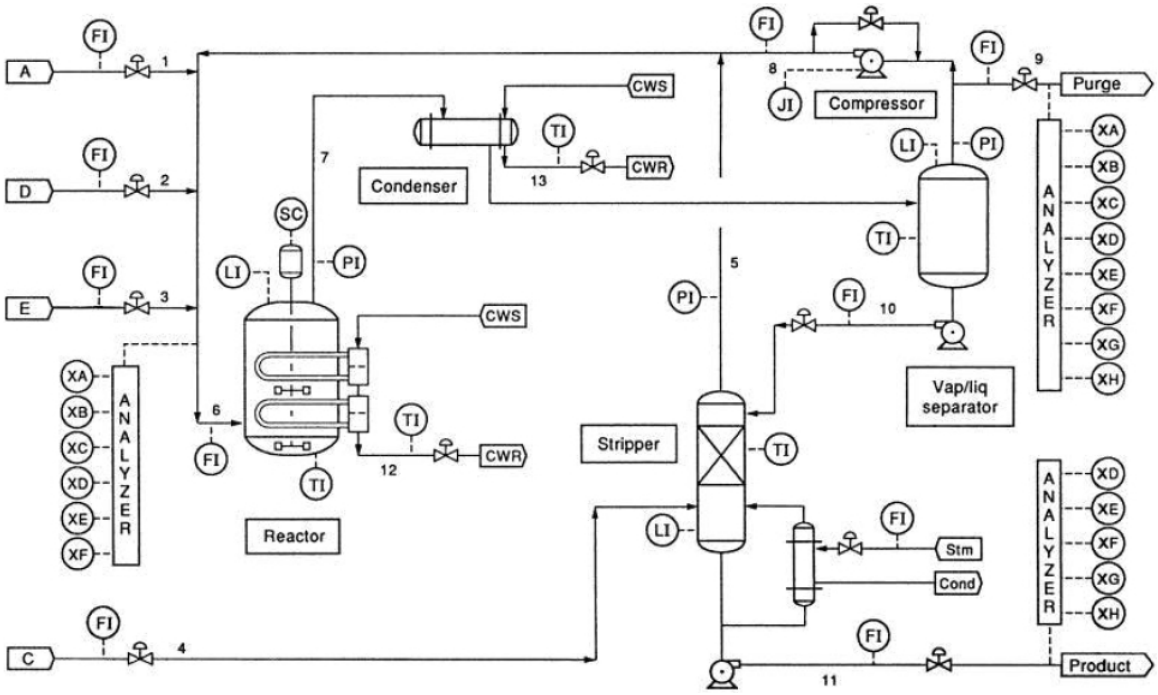

Figura 4.1: Esquema do Tennessee Eastman Process Fonte:[3]

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: [7]

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 Preassure
$t(8)$	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. [7]

Os dados utilizados foram os mesmos de [7, Nakamura], obtidos utilizando o modelo *Simulink* do TEP [3] sob uma estratégia de controle descentralizada [9]. O simulador pode ser baixado em http://depts.washington.edu/control/LARRY/TE/download.html. [15]

O processo possui 41 variáveis medidas e 12 variáveis manipuladas. Por conta da uniformidade no período de amostragem de 6 minutos, apenas as primeiras 22 variáveis medidas foram utilizadas. A descrição 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. [7]

O modelo *Simulink* possui 21 modos de operação, sendo um normal e o restante situações de falhas. A descrição das falhas pode ser vista na tabela 4.2. [7]

4.2 Testes Realizados e Resultados Obtidos

Como feito em [7, Nakamura], foram realizados dois tipos de testes para analisar o desempenho dos diferentes modelos: os com falhas isoladas e os com falhas sequenciais.

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 4.2: Descrição das situações de falhas do TEP. Fonte: [10]

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

4.2.1 Falhas Isoladas

Nesse teste, primeiramente foi treinado um modelo para os dados normais, então uma falha foi aplicada para que o sistema pudesse aprender o novo comportamento e então ela foi aplicada novamente para que o sistema diagnosticasse a falha. Seguindo isto, todas as falhas foram aplicadas e testadas isoladamente [7].

Para cada um dos testes, foram utilizados todos os tipos de *autoencoders* listados na Revisão Bibliográfica. E para gerar o melhor modelo, a quantidade de épocas variou de X a Y; a quantidade de amostras passadas levadas em consideração variou de X a Y; e a dimensão da *Bottleneck Layer* variou de X a Y. Sendo assim, foram treinados X modelos para cada tipo de autoencoder em cada cenário de falha apresentado. Em cada caso, o menor valor de RMSE definia o melhor modelo a ser utilizado. [7]

4.2.2 Falhas Sequenciais

Em cenários reais, as falhas podem não ocorrer isoladamente. Sendo assim, foi construído um conjunto de dados que aplica em sequência ao sistema algumas falhas para que ele possa aprender os comportamentos falhos e então detectá-los posteriormente. Entre cada falha, um conjunto de dados normais foi aplicado, visto que é provável que uma planta industrial não passe de uma situação de falha para outro logo em seguida.

Seguindo o proposto por [7, Nakamura], as falhas 13, 14 e 11 foram escolhidas por cobrir diferentes tipos de falhas (abrupta, incremental, gradual, recorrente ou *outlier*). A sequência aplicada foi: 0, 13, 0, 14, 0, 13, 0, 11, 0, 11, 0, 14; sendo 0 a amostra normal.

4.3 Resumo do Capítulo

Nesse capítulos foram apresentados os dados utilizados para realização dos testes, bem como a descrição dos testes realizados para análise do desempenho dos diferentes tipos de *autoencoder* para a detecção de falhas. Por fim, são apresentados os resultados comparativos dos testes.

Referências Bibliográficas

[1] J. AN and S. CHO. Variational autoencoder based anomaly detection using reconstruction probability. 2015.

[2] P. CHOPRA and S. YADAV. Fault detection and classification by unsupervised feature extraction and dimensionality reduction. In *Complex Intell. Syst.*, 2015.

[3] J. J. DOWNS and E. F. VOGEL. A plant-wide insdustrial process control problem. In *Computers chem. Engng.*, 1993.

[4] R. ESLAMLOUEYAN. Designing a hierarchical neural network based on fuzzy clustering for fault diagnosis of the tennessee eastman process. In *Applied Soft Computing*, 2011.

[5] J. R. JANG. *Neuro-Fuzzy and Soft Computing: A Computational Approach to Learning and Machine Intelligence*. Prentice-Hall, upper saddle river, NJ, 1997.

[6] T. NAKAMURA and A. LEMOS. A batch-incremental process fault detection and diagnosis using mixtures of probablistic pca. 2014.

[7] 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*.

[8] L. C. OLIVEIRA. Utilização de inteligência computacional no controle estático de convertedores LD (Linz-Donawitz) - versão preliminar, 2007.

[9] N. L. RICKER. Decentralized control of the tennessee eastman challenge process. In *Journal of Process Control*, 1995.

[10] 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.

[11] J. SUN et al. Automated fault detection using deep belief networks for the quality inspection of electromotors. In *Technisches Messen*, 2014.

[12] T. TAGAWA and T. YARI. Denoising autoencoder for fault detection and analysis. In *JMLR: WORKSHOP AND CONFERENCE*, 2014.

[13] 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.

[14] L. WANG et al. Transformer fault diagnosis using continuous sparse autoencoder, 2016.

[15] 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.

[16] T. ZHANG et al. Fault detection for ironmaking process based on stacked denoising autoencoders. In *American Control Conference (ACC)*, 2016.