Rafael Bastos Scisinio Dias

Estimativa da vazão de produção de poços utilizando técnicas de Inteligência Artificial

Projeto de Graduação

Projeto de Graduação apresentado ao Departamento de Engenharia Mecânica da PUC-Rio

Orientador: Márcio da Silveira Carvalho

Coorientador: Sergio Ribeiro

Rio de Janeiro

Junho de 2022

AGRADECIMENTOS

Agradeço a minha família, especialmente aos meus pais Fabio e Adriana e ao meu irmão Bernardo, por todo suporte, apoio, incentivo que recebi e por sempre tentarem me proporcionar as melhores condições.

Aos meus amigos da PUC-Rio, em especial ao Alexandre da Matta, Felipe Marsiglia e Leonardo Leite, por estarem comigo desde meu 1º dia no curso de Engenharia Mecânica, compartilhando experiências, dificuldades e alegrias. Sem eles, minha trajetória na PUC não teria sido tão divertida.

Aos meus demais amigos, que também torceram pelo meu sucesso, em especial a minha namorada Karina, com quem compartilho tristezas e felicidades, e que me apoiou muito na parte final do curso.

Por fim agradeço ao meu orientador Marcio e ao meu coorientador Sergio, por me auxiliarem com minhas dúvidas e dificuldades, assim como todo o suporte prestado ao longo da realização deste trabalho.

RESUMO

Estimativa da vazão de produção de poços utilizando técnicas de Inteligência Artificial

A utilização de técnicas de inteligência artificial para estimação da vazão de produção de poços tem sido aplicada como substituto dos métodos tradicionais de caracterização de poços. Entretanto, ainda não é claro que as técnicas de inteligência artificial são capazes de fato de fornecer uma estimativa da vazão de produção, com um erro reduzido. Desse modo, o objetivo deste trabalho é desenvolver uma metodologia baseada em técnicas de inteligência artificial e implementar a mesma em um código computacional capaz de estimar a vazão de produção, com um erro baixo. Para atingir o objetivo, foram realizadas algumas simulações com diferentes modelos de rede neural e comparadas as previsões geradas, com os dados reais. E os resultados obtidos apresentaram comportamento satisfatório, assim como um erro consideravelmente reduzido.

Palavras-chave: Inteligência artificial; rede neural; PDG

ABSTRACT

Estimation of well production flow rate using Artificial Intelligence techniques

The use of artificial intelligence techniques to estimate the production flow rate of wells has been applied as a substitute for the traditional methods of well characterization. However, it is not clear that artificial intelligence techniques are able to actually provide an estimate of the production flow, with a reduced error. Thus, the objective of this work is to develop a methodology and a computational code capable of estimating the production flow, with a low error. To achieve the objective, some simulations were carried out with different models of neural network and the generated predictions were compared with the real data. And the results obtained showed satisfactory behavior, as well as a considerably reduced error.

Key words: Artificial intelligence; neural network; PDG

LISTA DE FIGURAS

LISTA DE TABELAS

SUMÁRIO

1 INTRODUÇÃO

Segundo o Instituto Brasileiro de Petróleo e Gás (IBP) (2022), 46% da oferta interna de energia do país é originária do setor de óleo e gás. Além disso, o IBP projeta que no período de 2023 a 2030, nos setores de exploração e produção, 570 mil postos de trabalho (diretos e indiretos) serão gerados a cada ano. De acordo com a Federação única dos Petroleiros (FUP), em 2021 o Brasil possuía um efetivo de aproximadamente 155 mil trabalhadores.

Atualmente a produção de petróleo e gás no Brasil é decorrente de 6.143 poços (466 marítimos e 5.677 terrestres), de acordo com o Boletim da Produção de Petróleo e Gás Natural, divulgado pela ANP (Agência Nacional do Petróleo, Gás Natural e Biocombustíveis) em março de 2022.

Dessa forma, a caracterização das propriedades de fluxo de reservatórios, assim como a estimação da vazão de produção dos poços são informações importantes, pois auxiliam na otimização das estratégias de exploração do campo (TIAN, 2018; GARUZZI e ROMERO, 2014). Porém, por serem dados de difícil obtenção e de alto custo com os métodos atuais, a aplicação de técnicas de inteligência artificial para estimação da vazão de produção de poços tem se tornado uma alternativa.

Entretanto, ainda não é clara a capacidade das diferentes técnicas baseadas em inteligência artificial, como o aprendizado de máquina, aprendizado profundo e redes neurais, de estimar a vazão de produção de poços.

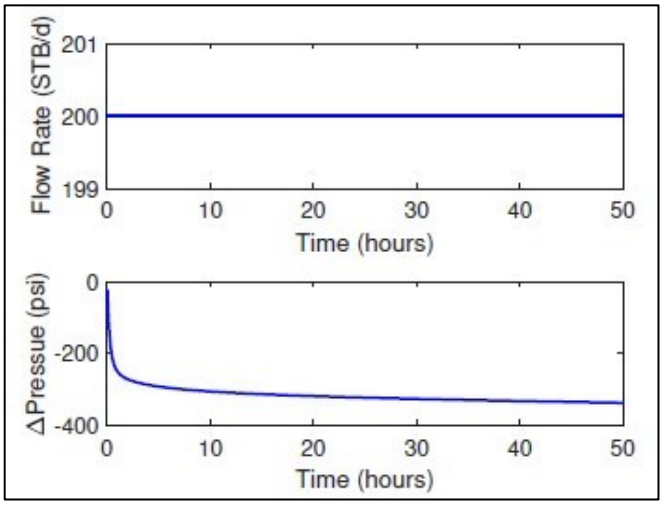
Diante disso, o presente trabalho tem como objetivo apresentar uma formulação e elaborar um código computacional, utilizando as técnicas de inteligência artificial, que seja capaz de estimar a vazão de produção de poços, com um nível de erro baixo. Para isso serão testados alguns modelos de rede neural, e comparando as previsões geradas, com os dados reais.

2 TESTE DE POÇO

A caracterização de reservatórios é uma tarefa muito importante para o setor de óleo e gás, e uma das abordagens mais utilizadas para isso é o teste de poços. Durante o período de testes, a evolução da pressão é medida durante um período que o poço é operado ou a uma vazão constante (*Drawdown Test* – Teste de abaixamento da pressão) ou é fechado (*Build-up Test* – Teste de acúmulo de pressão) e a duração do teste pode durar de algumas horas até alguns dias (TIAN, 2018; GARUZZI e ROMERO, 2014).

- *Drawdown Test*: nesse teste, o poço é aberto para produção, a uma vazão constante diferente de zero, ocasionando a queda da pressão no reservatório ao longo do tempo, conforme Figura 1. Os objetivos desse teste são: obtenção da permeabilidade média da rocha, determinar o volume de poros, detectar a heterogeneidade do reservatório entre outros (TIAN, 2018; GARUZZI e ROMERO, 2014).

Figura 1 - Exemplo de um *Drawdown Test*

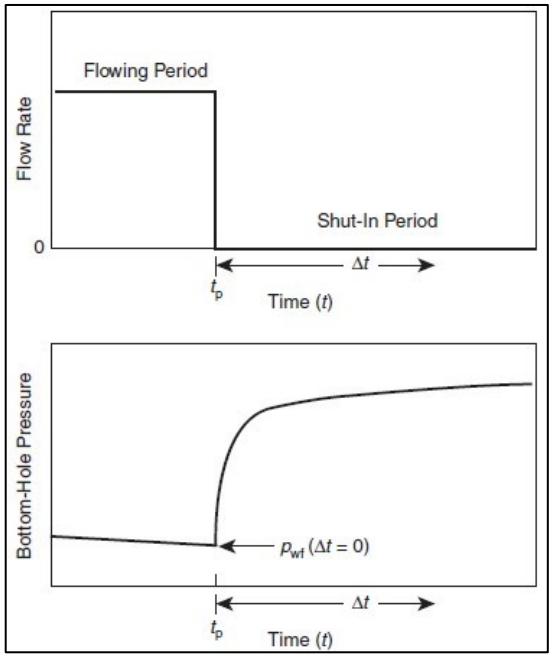


Fonte: (TIAN, 2018, p. 2)

- *Build-up Test*: nesse teste ocorre o fechamento do poço (*shut-in*) (taxa de vazão nula), ocasionando no aumento da pressão do reservatório ao longo do tempo, conforme Figura 2. Os objetivos do teste são: determinar a pressão estática do reservatório, determinar a

permeabilidade efetiva do reservatório, os limites do reservatório entre outros (TIAN, 2018; GARUZZI e ROMERO, 2014).

Figura 2 - Exemplo de *Build-up Test*



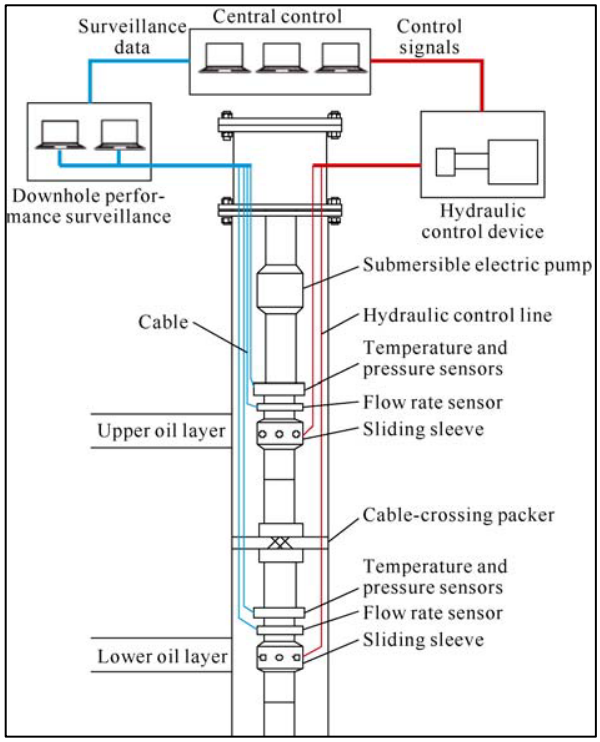
Fonte: (AHMED e MEEHAN, 2012, p. 79)

Dessa forma, os dados de pressão coletados nos testes são utilizados para caracterizar diversos parâmetros do reservatório como: fronteiras do reservatório, permeabilidade, armazenamento do poço entre outros.

2.1 Medidor Permanente de Fundo de Poço (PDG)

Para medição da pressão, temperatura e em alguns casos a vazão de produção, é instalado no poço um dispositivo chamado PDG (*Permanent Downhole Gauge*), representado na Figura 3, o qual fornece dados contínuos das grandezas mencionadas e inicialmente era usado apenas para o monitoramento da produção. Porém, com o passar do tempo, percebeu-se que o PDG poderia ser utilizado para caracterização do reservatório ao redor do poço (TIAN, 2018).

Figura 3 – Modelo de sistema de monitoramento permanente de fundo de poço



Fonte: (HE *et al.*, 2020, p. 1105)

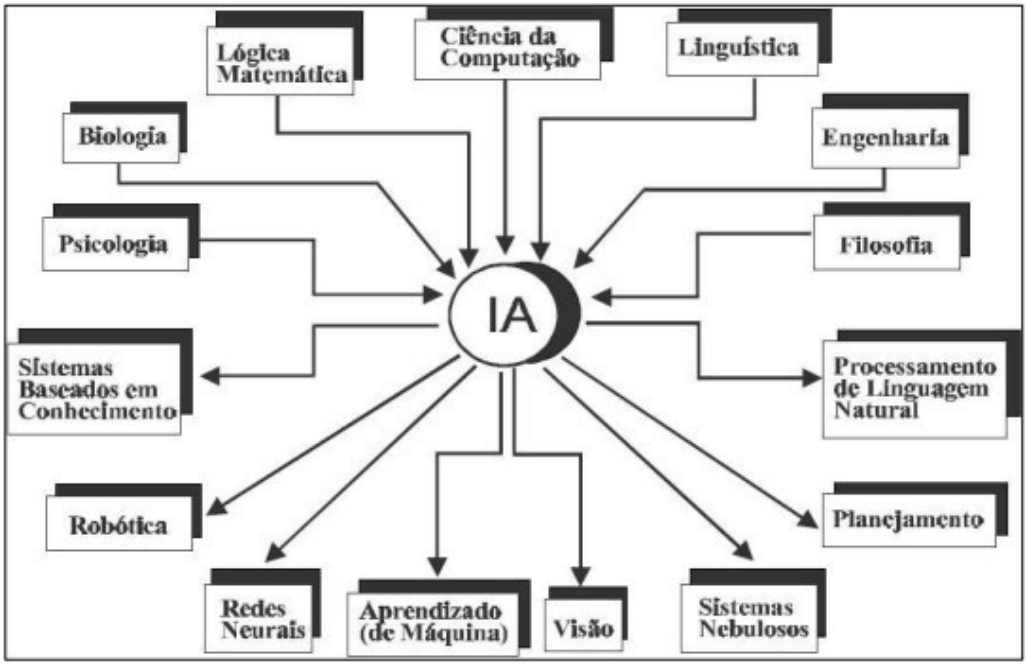
No entanto, diferentemente da testagem de poços, os PDGs obtêm pressão e temperatura em condições de vazão variável durante o longo período da produção (TIAN, 2018). Além disso, nos PDGs os valores de vazão não são verificados, fazendo com que as técnicas aplicadas nos testes de poços não sejam adequadas para a análise de dados do PDG. Diante desse problema, técnicas de inteligência artificial, como o aprendizado de máquina e redes neurais, vem sendo utilizadas para análise desses dados (TIAN, 2018; TIAN e HORNE, 2019).

3 INTELIGÊNCIA ARTIFICIAL

O termo Inteligência Artificial (IA) foi introduzido na década de 1950 e refere-se a máquinas que apresentam comportamento similar ao humano. Alguns dos objetivos nas pesquisas em IA são: o aprendizado, o raciocínio, a comunicação (processamento de linguagem natural), a manipulação e a movimentação de objetos (ONGSULEE, 2017; GOMES, 2010).

A inteligência artificial, apesar de ser área de estudo da ciência da computação, atualmente vem sendo aplicada em diversos outros setores como os de robótica, biologia, engenharia entre outros (ONGSULEE, 2017; GOMES, 2010), como visto na Figura 4.

Figura 4 - Aplicações da Inteligência Artificial



Fonte: (MONARD; BARANAUKAS, 2000, p.2)

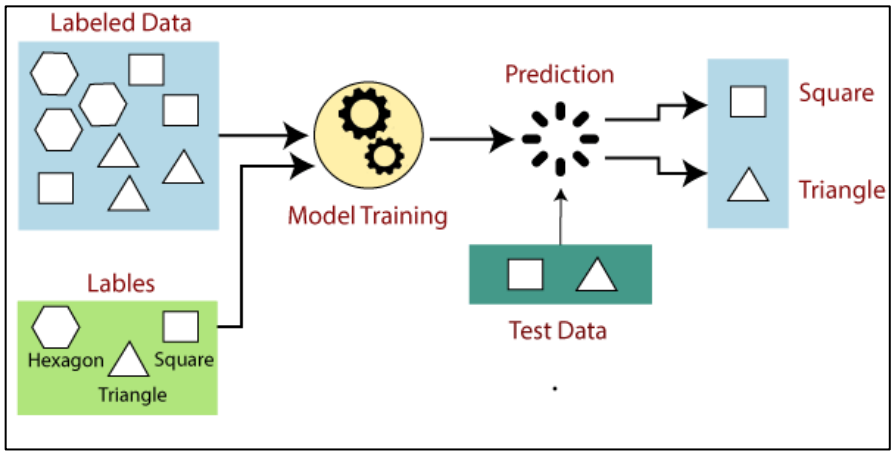
3.1 Aprendizado de máquina (Machine learning)

O aprendizado de máquina (*Machine Learning,* em inglês ML) é uma subárea da inteligência artificial, em que os computadores são levados a construir correlações matemáticas complexas que se assemelham ao aprendizado, sem a introdução de uma instrução direta dada pelo programador, como por exemplo as redes neurais

artificiais (*Artificial Neural Network*, do inglês ANN). No geral, os algoritmos usados no ML são capazes de utilizar os próprios erros para evoluir essas correlações, detectar padrões e aprimorar continuamente as previsões do modelo. São comumente utilizados em situações nas quais a elaboração de um código de programação seria muito complexa, ou até mesmo impraticável (ONGSULEE, 2017; LEMLEY, BAZRAFKAN e CORCORAN, 2017).

Os métodos do aprendizado de máquina podem ser divididos em supervisionado, não supervisionado e semissupervisionado (NASTESKI, 2017; LEMLEY, BAZRAFKAN e CORCORAN, 2017). Neste trabalho, o método utilizado será do aprendizado supervisionado, no qual durante o treinamento são fornecidos ao modelo os dados de entrada (*Inputs*) e saída (*Targets*) conhecidos, para que o algoritmo possa encontrar as correlações e assim inferir valores de saída para um conjunto de dados de entrada nunca visto, conforme visto na Figura 5.

Figura 5 - Aprendizado de Máquina Supervisionado



Fonte: (JAVATPOINT, 2022, p.2)

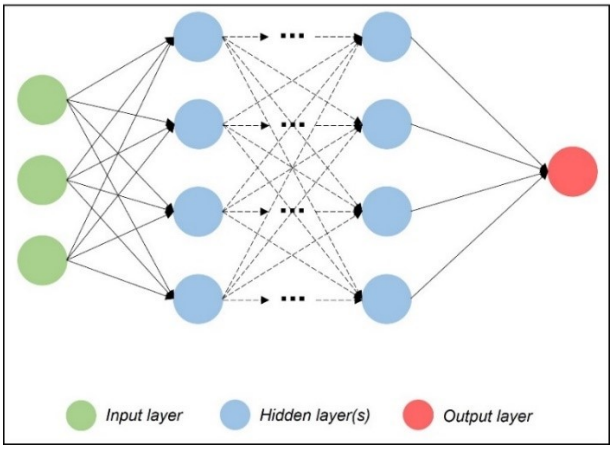
Disponível em: https://www.javatpoint.com/supervised-machine-learning

3.1.1 Redes Neurais

As redes neurais foram projetadas, com o objetivo de modelar a forma de atuação do cérebro na realização de uma tarefa, sendo o comportamento dos neurônios a principal inspiração para o modelo das redes (FLECK *et al.*, 2016; BI *et al.*, 2019). Estruturalmente as redes são compostas de camadas de neurônios artificiais, divididas em 3 grupos: camada de entrada (*Input layer*), camada escondida

(*Hidden layer*) e camada de saída (*Output layer*) (LEMLEY, BAZRAFKAN e CORCORAN, 2017; BI *et al*., 2019; LECUN, BENGIO e HINTON, 2015; FLECK *et al*., 2016; PACHECO e PEREIRA, 2018), como representado na Figura 6.

Figura 6 - Representação de uma ANN

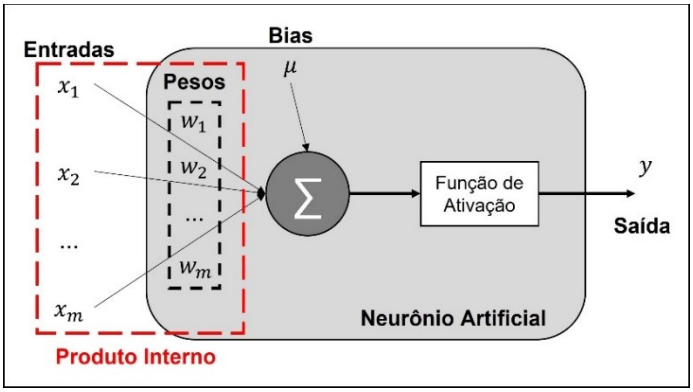


Fonte: Autoria própria.

A conversão dos *inputs* (x) para os *outputs* (y) é feita nas camadas escondidas, na qual o somatório do produto interno, elemento a elemento, dos *inputs* e dos pesos (w) é somado a um *bias* (μ) (limiar) e aplicado a uma função de ativação (f), como visto em (1) e ilustrado na Figura 7.

$$y = f\left(\sum_{i=1}^{m} w_i x_i - \mu\right) \tag{1}$$

Figura 7 - Representação de um Neurônio Artificial



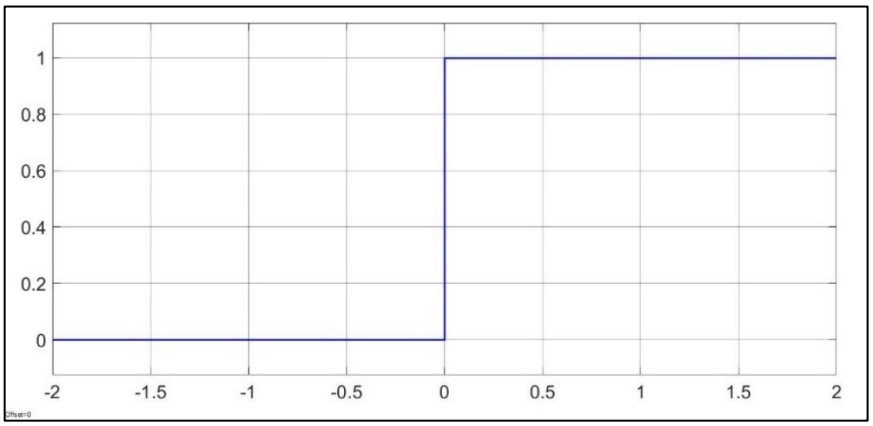
Fonte: Modificado de Rauber (2005)

A função de ativação processa o sinal gerado, da combinação linear dos pesos e das entradas, gerando o sinal de saída do neurônio artificial. A escolha da função varia de acordo com a aplicação, mas alguns dos tipos de função de ativação mais utilizados são (FLECK *et al*., 2016; RAUBER, 2005; SILVA, 2010):

- Função de limiar: restringe o sinal de saída em valores binários, sendo 0 quando o valor for negativo e 1 quando for positivo, conforme representado na Figura 8.

$$f(z) = \begin{cases} 0, & z < 0 \\ 1, & z \geq 0 \end{cases} \tag{2}$$

Figura 8 - Função Limiar

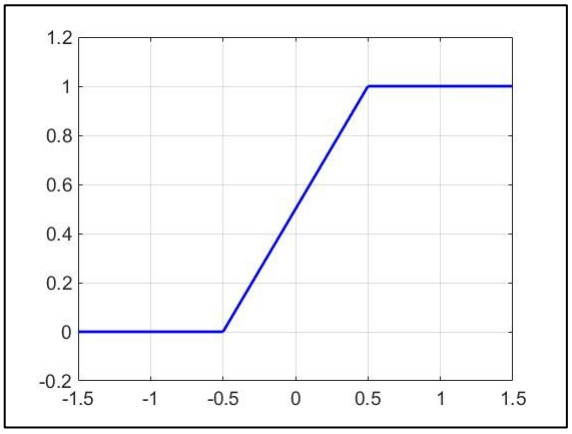


Fonte: Autoria própria.

- Função de limiar por partes: pode ser identificado como uma aproximação de um amplificador não linear, conforme Figura 9.

$$f(z) = \begin{cases} 0, & z \leq -0{,}5/a \\ a \cdot z + 0{,}5, & 0{,}5/a > a \cdot z > -0{,}5/a \\ 1, & z \geq 0{,}5/a \end{cases} \tag{3}$$

Figura 9 - Função de Limiar por Partes

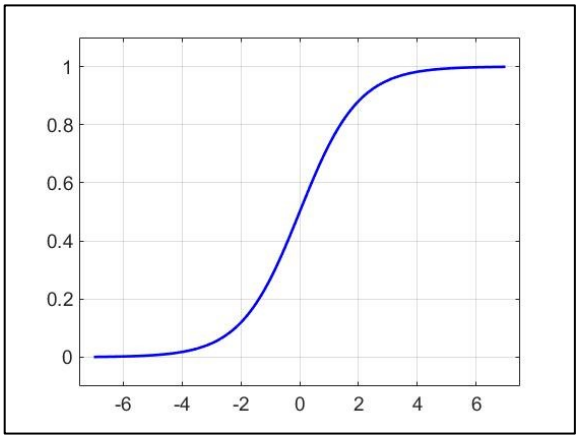


Fonte: Autoria própria.

- Função sigmoidal: esse tipo de função é o mais utilizado em ANNs, visto que é uma função contínua, com balanceamento adequado entre o comportamento linear e não linear. Um exemplo de função sigmoidal é a função logística (4), que possui um intervalo de variação entre 0 e 1, como representado na Figura 10.

$$f(z) = \frac{1}{1 + e^{-z}} \tag{4}$$

Figura 10 - Função Sigmoidal



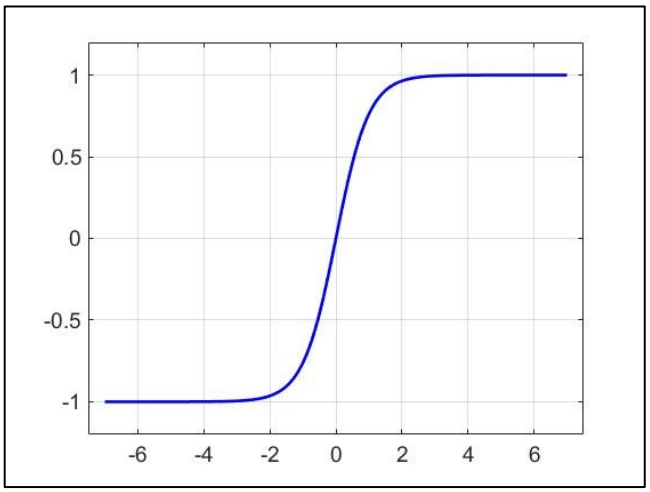
Fonte: Autoria própria.

- Função tangente hiperbólica: outro tipo de função sigmoidal que, diferente da função logística, assume valores positivos e negativos para

um mesmo intervalo, como visto na Figura 11, podendo trazer benefícios dependendo da aplicação.

$$f(z) = \tanh(z) \tag{5}$$

Figura 11 - Função Tangente Hiperbólica



Fonte: Autoria própria.

Além da escolha da função de ativação que melhor se adeque a aplicação, avaliar o desempenho da rede neural é um dado importante, pois é um parâmetro que indica se a ANN performa dentro das tolerâncias desejadas, ou se ajustes são necessários. A forma de avaliação mais utilizada é a do erro médio quadrático (*Mean Square Error*, em inglês MSE), representada em (6) (SILVA, 2010).

$$MSE = \frac{1}{N}\sum_{j=1}^{M}\sum_{i=1}^{N}\left(y_{p_{ji}} - y_{r_{ji}}\right)^2 \tag{6}$$

Onde *N* é o número de amostras de treinamento, *M* é o número de *outputs* da rede, $y_{p_{ji}}$ é a saída prevista pela rede, $y_{r_{ji}}$ é o valor real da saída e *MSE* é o erro médio quadrático entre a saída prevista e o valor real. Idealmente, o valor do *MSE* é minimizado durante o treinamento, até que ele fique dentro dos limites toleráveis do projeto (SILVA, 2010).

3.1.2 Aprendizagem profunda (Deep Learning)

Aprendizado profundo (*Deep Learning*, em inglês DL) é uma área do *machine learning* que se refere a redes neurais artificiais com mais de 1 camada escondida (*hidden layer*). A vantagem do *deep learning* em comparação com o *machine learning* é a capacidade de trabalhar com dados brutos (*raw data*), ou seja, o pré-processamento dos dados não é necessário (ONGSULEE, 2017; RUSK, 2016; LECUN, BENGIO e HINTON, 2015; PACHECO e PEREIRA, 2018).

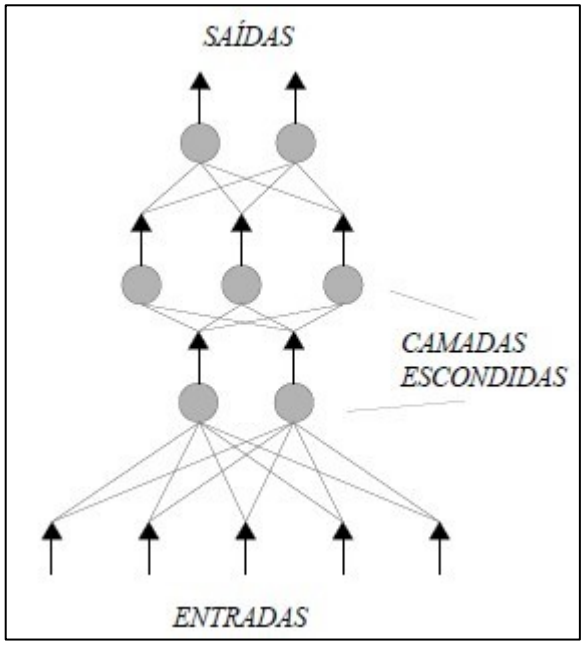
Assim como os algoritmos de *machine learning*, os algoritmos de *deep learning* são utilizados para o reconhecimento de imagens, reconhecimento facial, processamento de linguagem natural entre outros. Aplicações estas que vem sendo utilizadas por diversas empresas como a Google, para o desenvolvimento de carros autônomos e serviços de tradução (PACHECO e PEREIRA, 2018; LECUN, BENGIO e HINTON, 2015; ONGSULEE, 2017).

3.1.2.1 Redes neurais recorrentes (RNN)

As redes neurais recorrentes (*Recurrent Neural Networks*, em inglês RNN), são um tipo de rede neural que possui uma realimentação dos dados, fazendo com que essa estrutura de rede seja muito útil para se trabalhar com dados temporais ou sequenciais (SILVA, 2010; RAUBER, 2005; LEMLEY, BAZRAFKAN e CORCORAN, 2017; LECUN, BENGIO e HINTON, 2015).

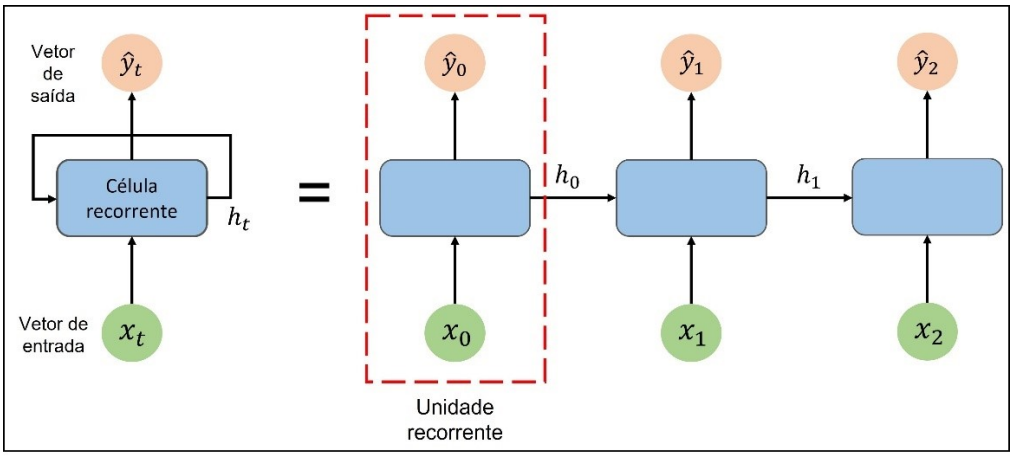
Diferente de uma rede de propagação para frente (*Feed Forward Network*), na qual o fluxo de informação é unidirecional, conforme Figura 12, nas RNNs as camadas escondidas possuem um *loop* de alimentação, permitindo que a informação seja passada de volta para o neurônio, como visto em Figura 13 (SILVA, 2010; RAUBER, 2005; LEMLEY, BAZRAFKAN e CORCORAN, 2017; LECUN, BENGIO e HINTON, 2015, TIAN, 2018).

Figura 12 - Feed Forward Network



Fonte: (RAUBER, 2005, p.7)

Figura 13 - Representação de uma RNN



Fonte: Modificado de Lecun, Bengio e Hinton (2015)

Conforme a Figura 13, cada unidade recorrente, para cada instante t, calcula o estado oculto (h_t) levando em consideração a entrada (x_t) e o estado escondido do tempo anterior (h_{t-1}), e calcula a saída (y_t). O cálculo do estado escondido e da saída da célula recorrente estão representados nas equações (7) e (8). Sendo W_x, W_h e W_y as matrizes de pesos da entrada, estado oculto e saída respectivamente, μ_h e μ_y os vetores de *bias* e f_1 e f_2 funções de ativação (TIAN, 2018).

$$h_t = f_1(x_t \cdot W_x + h_{t-1} \cdot W_h + \mu_h) \quad (7)$$

$$y_t = f_2(W_y \cdot h_t + \mu_y) \quad (8)$$

Os parâmetros W_x, W_h, W_y, μ_h e μ_y geralmente são inicializados pelo usuário, e o objetivo é encontrar os valores dos parâmetros, de modo que o erro seja minimizado. Um dos métodos utilizados é o da descida do gradiente no qual, através de um processo iterativo, busca-se encontrar Θ, vetor dos parâmetros do modelo, que minimiza o erro. E Θ é calculado pela diferença entre Θ e o produto entre a taxa de aprendizado (α) e o gradiente da função de erro ($\cdot \nabla J(\Theta)$) com respeito a Θ, conforme (9) (TIAN, 2018).

$$\Theta = \Theta - \alpha \cdot \nabla J(\Theta) \quad (9)$$

Um problema que ocorre ao treinar uma RNN é do desaparecimento (ou explosão) do gradiente pois, para calculá-lo, uma matriz de pesos é multiplicada por ela mesma várias vezes, devido a aplicação da regra da cadeia. Com isso, caso os elementos da matriz sejam próximos de zero, o gradiente vai para zero e caso eles sejam muito grandes, o gradiente vai para infinito (TIAN, 2018). Uma solução para esse problema, é a utilização de arquiteturas de RNN mais robustas e avançadas como a *Long Short-Term Memory* (LSTM) ou a *Gated Recurrent Unit* (GRU).

3.1.2.1.1 Long Short-Term Memory (LSTM)

Diferente das redes recorrentes simples, que não são capazes de manter dependências a longo prazo (memória curta), as unidades de memória de curto e longo prazo (LSTM), introduzidas em 1997 por Hochreiter e Schmidhuber, são um tipo de RNN capaz de armazenar informações de longo prazo. Uma unidade LSTM é capaz de adicionar ou remover informações ao estado da célula, através de 3 portões internos (HOCHREITER e SCHMIDHUBER, 1997; DUTTA, KUMAR e BASU, 2020; CHUNG *et al*., 2014; CHUNG *et al*., 2015; RODRIGUES, 2021).

- Portão de esquecimento (*Forget Gate*): determina quais informações serão esquecidas (removidas) do estado da célula. Recebe duas entradas, o *input* no instante atual (x_t) e a saída da célula anterior (h_{t-1}), que são multiplicadas por matrizes de peso e adicionado um *bias*. O resultado é aplicado numa função de ativação sigmoidal com saída entre 0 e 1, conforme x, na qual um resultado próximo de 0 representa que a informação será esquecida, e um resultado próximo de 1 a informação será mantida.

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + \mu_f) \tag{10}$$

- Portão de entrada (*Input Gate*): é usado para atualizar o estado da célula com novas informações. O estado oculto da célula anterior (h_{t-1}) e a entrada no instante atual (x_t) são aplicados simultaneamente a uma função sigmóide e a tangente hiperbólica, resultando em i_t (conforme (11)) e $\tilde{c}_t$ (candidato ao novo estado de célula). Em seguida i_t e $\tilde{c}_t$ são multiplicados e somados a multiplicação de f_t com c_{t-1} (estado da célula anterior), para gerar o novo estado de célula c_t, conforme (12).

$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + \mu_i) \tag{11}$$

$$c_t = f_t \cdot c_{t-1} + i_t \cdot \tilde{c}_t \tag{12}$$

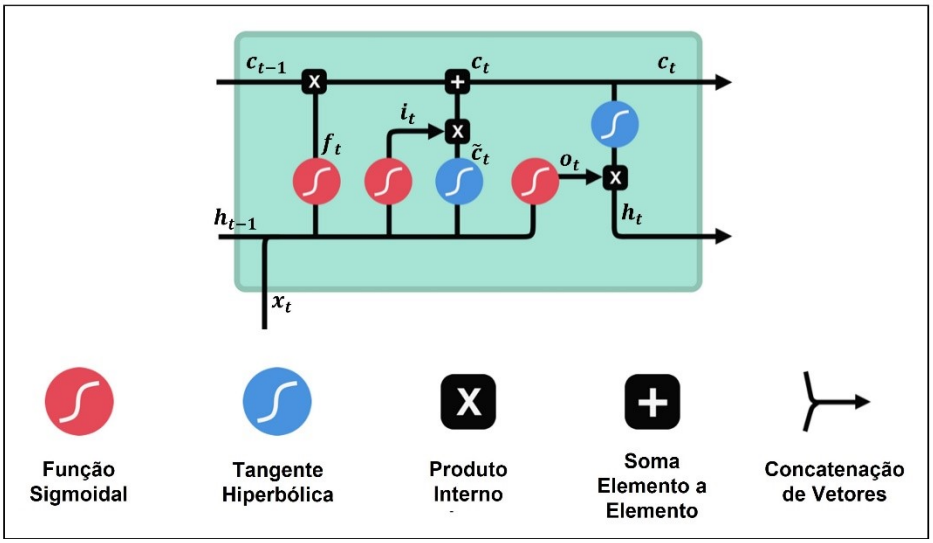
- Portão de saída (*Output Gate*): determina que informações devem ir para o novo estado oculto. Essa operação consiste em 2 passos: primeiro o estado oculto da célula anterior e a entrada no instante atual são aplicados a função sigmoidal, conforme (13), segundo o estado da célula atual é aplicado a tangente hiperbólica e então os resultados são multiplicados para gerar h_t, conforme (14).

$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + \mu_o) \tag{13}$$

$$h_t = o_t \cdot \tanh(c_t) \tag{14}$$

A representação de uma unidade LSTM está conforme Figura 14.

Figura 14 - Representação de uma unidade LSTM



Fonte: Modificado de Phi (2018, p.8)
Disponível em: https://towardsdatascience.com/illustrated-guide-to-lstms-and-gru-s-a-step-by-step-explanation-44e9eb85bf21

3.1.2.1.2 Gated Recurrent Unit (GRU)

A unidade recorrente com portas (*Gated Recurrent Unit*, em inglês GRU), é outro tipo de RNN capaz de lidar com dependências de longo prazo. Sua estrutura foi proposta por Cho *et al*. em 2014 é similar a uma unidade LSTM, a GRU também possui uma estrutura interna contendo portões, mas diferente da LSTM que possui 3 portões a GRU possui apenas 2 portões (CHO *et al*., 2014; DUTTA, KUMAR e BASU, 2020; CHUNG *et al*., 2014; CHUNG *et al*., 2015).

- Portão de atualização (*Update Gate*): é uma combinação do *forget gate* e do *input gate* da LSTM num único portão. Ele define quanto da memória passada deve ser descartada, e quanta informação nova deve ser adicionada. O *update gate* (z_t), recebe a entrada no instante atual (x_t) e o estado oculto passado (h_{t-1}), que são multiplicados por matrizes de pesos e aplicados a função de ativação sigmoidal, conforme (15).

$$z_t = \sigma(W_z \cdot [h_{t-1}, x_t]) \quad (15)$$

- Portão de redefinição (*Reset Gate*): determina quanta informação do passado esquecer. Similar ao *update gate*, o *reset gate* recebe x_t e h_{t-1}, multiplica-os por uma matriz de pesos e aplica o resultado a função sigmoidal, conforme (16).

$$r_t = \sigma(W_r \cdot [h_{t-1}, x_t]) \quad (16)$$

Com isso, para cada instante de tempo, o estado oculto (h_t) é calculado conforme (17).

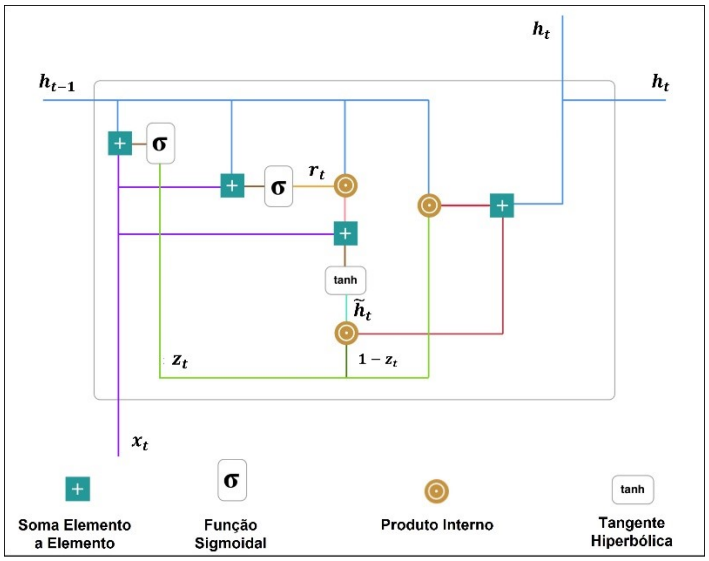
$$h_t = (1 - z_t) \cdot h_{t-1} + z_t \cdot \tilde{h}_t \quad (17)$$

Sendo $\tilde{h}_t$ o novo conteúdo da memória, que é calculado conforme (18). Onde W_1 e W_2 são matrizes de pesos, e $\odot$ representa a multiplicação elemento a elemento.

$$\tilde{h}_t = \tanh(W_1 \cdot x_t + r_t \odot W_2 \cdot h_{t-1}) \quad (18)$$

A representação de uma GRU está conforme Figura 15.

Figura 15 - Representação de uma GRU



Fonte: Modificado de Kostadinov (2017, p.3)

Disponível em: https://towardsdatascience.com/understanding-gru-networks-2ef37df6c9be

Uma vantagem da GRU em comparação com a LSTM é que, por possuir menos operações internas, o treinamento de RNNs do tipo GRU são mais rápidas. Entretanto a GRU expõe totalmente a sua memória sem nenhum controle, enquanto a LSTM, através do *output gate*, controla quanta memória será vista ou usada por outras unidades na rede. Diante das semelhanças e diferenças entre ambas, é difícil concluir que tipo de rede (GRU e LSTM) performa melhor de maneira geral (CHO *et al.*, 2014; DUTTA, KUMAR e BASU, 2020; CHUNG *et al.*, 2014; CHUNG *et al.*, 2015; HOCHREITER e SCHMIDHUBER, 1997; RODRIGUES, 2021).

4 METODOLOGIA

Para as simulações realizadas neste trabalho, o código do Anexo A (Pedersen, 2018) foi modificado, alterando-se a fonte dos dados, a estrutura da rede neural, o valor de determinados parâmetros do código, para construção dos códigos computacionais dos Apêndices B, C, D e E.

Os códigos foram escritos na linguagem Python e utilizada a biblioteca *Tensorflow*, para implementação da RNN. Outras bibliotecas como: *Matplotlib*, *Numpy*, *Pandas* e *Scikit-learn* foram usadas para o processamento dos dados, visualização dos gráficos entre outros.

4.1 Dados

Foram utilizados 2 conjuntos de dados neste trabalho, os conjuntos 1 e 2 foram gerados a partir de 2 simulações feitas no simulador desenvolvido no LMMP PUC-Rio (Laboratório de Micro hidrodinâmica e Escoamento em Meios Porosos), que utiliza o método de diferenças finitas para resolver as equações de conservação de massa, energia e momento.

Para as simulações, considerou-se um reservatório homogêneo, com porosidade única e que a lei de Darcy governa o escoamento. Com isso, foi imposto ao simulador um regime de vazão assim como as propriedades da rocha e do fluído (permeabilidade, porosidade, viscosidade etc.), e como resposta o simulador retorna os perfis de pressão e temperatura.

Cada conjunto possui 105.121 valores referentes ao tempo (1 ano de duração), a vazão de produção, a temperatura e a pressão. Como visto na Figura 16, os valores de tempo estão em segundos, e representam o tempo de simulação de cada dado.

Entretanto, por não estarem igualmente intervalados, houve a necessidade de fazer uma reamostragem dos dados, utilizando o código do Apêndice A, de modo que os dados tivessem um intervalo de 5 minutos e os novos valores de vazão, temperatura e pressão foram atualizados pela média dos valores dentro dos 5 minutos, como visto na Figura 17.

Figura 16 - Pedaço do conjunto de dados 1

	A	B	C	D
1	tempo (s)	vazao (m^3/dia)	temperatura (K)	pressao (Pa)
2	0	0	334	49033000
3	0.000102564	0.009259259	334	49033000
4	0.000719816	0.009259259	334	49032999.97
5	0.002320509	0.009259259	334	49032999.83
6	0.005373392	0.009259259	334	49032999.3
7	0.010347218	0.009259259	333.9999999	49032997.43
8	0.017710737	0.009259259	333.9999998	49032991.11
9	0.027932701	0.009259259	333.9999995	49032973.91
10	0.041481861	0.009259259	333.9999987	49032934.89
11	0.058826968	0.009259259	333.9999971	49032854.87
12	0.080436773	0.009259259	333.9999943	49032710.5
13	0.106780027	0.009259259	333.9999905	49032521.61
14	0.138325482	0.009259259	333.999987	49032347.45
15	0.175541889	0.009259259	333.999982	49032092.98
16	0.218897998	0.009259259	333.9999731	49031642.36
17	0.268862561	0.009259259	333.9999593	49030949.34
18	0.32590433	0.009259259	333.999939	49029926.28
19	0.390492055	0.009259259	333.9999101	49028471.53
20	0.463094487	0.009259259	333.9998707	49026482.24
21	0.544180378	0.009259259	333.9998173	49023791.46

Fonte: Autoria própria.

Figura 17 - Pedaço do conjunto de dados 1 reamostrado

	A	B	C	D
1	tempo	vazao (m^3/dia)	temperatura (K)	pressao (Pa)
2	01/01/2021 00:00	0.009200284	333.9938963	48670544.52
3	01/01/2021 00:05	0.009259274	333.9931975	48424272.55
4	01/01/2021 00:10	0.0092593	333.9945609	48367386.8
5	01/01/2021 00:15	0.009259338	333.9957613	48328308.56
6	01/01/2021 00:20	0.00925939	333.9968175	48298087.55
7	01/01/2021 00:25	0.009259455	333.9977473	48273572.85
8	01/01/2021 00:30	0.009259533	333.99858	48252778.64
9	01/01/2021 00:35	0.009259625	333.9993329	48234691.69
10	01/01/2021 00:40	0.009259728	334.0000032	48219046.74
11	01/01/2021 00:45	0.009259844	334.0006156	48205068.43
12	01/01/2021 00:50	0.009259977	334.0011959	48192061.77
13	01/01/2021 00:55	0.009260118	334.0017192	48180506.19
14	01/01/2021 01:00	0.009260274	334.0022112	48169774.76
15	01/01/2021 01:05	0.009260444	334.0026727	48159813.33
16	01/01/2021 01:10	0.009260625	334.0031043	48150581.38
17	01/01/2021 01:15	0.009260817	334.0035069	48142035.56
18	01/01/2021 01:20	0.009261018	334.0038813	48134143.71
19	01/01/2021 01:25	0.009261239	334.0042511	48126392.11
20	01/01/2021 01:30	0.009261468	334.0045938	48119246.93
21	01/01/2021 01:35	0.0092617	334.0049101	48112683.57

Fonte: Autoria própria.

4.2 Estrutura da rede neural

Para definir a estrutura da rede neural que seria usada para os treinamentos de cada modelo, primeiramente foram realizadas algumas simulações utilizando o código do Apêndice B (referente ao modelo de rede LSTM 1), variando os parâmetros

da rede a fim de obter os valores os quais, maximizam o uso da GPU disponível (Nvidia GTX 1070). Dessa forma, os modelos LSTM 2 (Apêndice C), GRU 1 (Apêndice D) e GRU 2 (Apêndice E) foram construídos utilizando a mesma estrutura do modelo LSTM 1 (Apêndice B), conforme Tabela 1.

Tabela 1 - Estrutura da rede neural

Camada	Tipo RNN	Unidades
1ª	LSTM/GRU	375
2ª	LSTM/GRU	375
3ª	Densa	200
4ª	Densa	1

Fonte: Autoria própria.

4.3 Parâmetros utilizados

Para realização das simulações, além da estrutura da rede, alguns parâmetros importantes foram definidos, para garantir uma padronização das simulações em todos os modelos (LSTM 1, LSTM 2, GRU1 e GRU 2). Os valores dos parâmetros estão conforme Tabela 2.

Onde o número de passos deslocados representa quantos passos no futuro os modelos irão prever, o tamanho do *batch* representa quantas sequências de dados (entrada e saída) serão usadas para treinamento, o tamanho da sequência representa quantos valores cada *batch* possuirá, o número de épocas representa quantas iterações serão feitas nos *batches* e os passos por época são o número de passos necessários para considerar que uma época terminou.

Tabela 2 - Parâmetros importantes do código

Parâmetro	Valor	Parâmetro	Valor
Passos deslocados	288 (1 dia)	Fator de redução de α	0.1
Divisão de treinamento	85%	Taxa de aprendizado mínima	1×10^{-6}
Tamanho do *batch*	150	Épocas	25
Tamanho da sequência	1440 (5 dias)	Passos por época	300
Taxa de aprendizado (α)	1×10^{-3}		

Fonte: Autoria própria.

4.4 Treinamento

Todos os modelos utilizados (Apêndices B, C, D e E) foram treinados utilizando 85% do conjunto de dados, sendo que os modelos dos Apêndices B e D, utilizaram o conjunto de dados 1 como fonte, e os modelos dos Apêndices C e E utilizaram o conjunto de dados 2.

O treinamento da rede utiliza o método do gradiente descendente, de modo a encontrar os pesos que minimizam o erro (MSE), entretanto, o esforço computacional para calcular o gradiente utilizando todos os dados de treinamento (89.000 dados) é muito alto. Diante disso, foram criados 150 pacotes (*batches*) de entrada e saída, começando em índices aleatórios e contendo 1440 dados, de modo que o método de descida do gradiente fosse aplicado nos *batches*, reduzindo o esforço computacional.

Além disso, para evitar que ao longo do treinamento dos modelos, eles passassem a “desaprender”, ou seja, que o erro aumentasse ou parasse de diminuir, foi utilizado o método de parada antecipada (*Early Stopping*), o qual interrompe o treinamento caso o erro não diminua em 5 épocas. Outro método aplicado, foi o da redução da taxa de aprendizado (α), a qual seria multiplicada for um fator de 0,1 toda vez que o erro não diminua em 2 épocas, sendo que essa redução é feita até α atingir o valor de 1×10^{-6}.

Para garantir que os modelos utilizem sempre os pesos mais otimizados até o momento, o método do ponto de retorno (*Checkpoint*) foi utilizado, de modo que apenas os pesos que produzem o melhor desempenho (menor erro) são salvos. Dessa forma, mesmo que o erro aumente entre 2 épocas, os pesos ótimos não serão alterados.

4.5 Teste e abrangência

Para validação dos modelos treinados, os 15% restantes de cada conjunto de dados foram utilizados como dados de teste. Esses valores não foram utilizados, nem visualizados pelos modelos durante o treinamento. Diante disso, cada modelo recebeu os respectivos dados de teste de entrada, gerou a saída de cada modelo que foram comparadas com os dados de teste de saída, resultando em um valor de MSE dos dados de teste.

Para verificação da capacidade de abrangência dos modelos, os conjuntos de dados opostos aos utilizados na etapa de treinamento foram utilizados para tal. E da mesma forma para os dados de teste, as entradas dos dados de abrangência foram passadas para os respectivos modelos, e a saída gerada por cada modelo comparada com a saída real dos dados de abrangência, resultando em um valor de MSE dos dados de abrangência.

4.6 Previsão retroalimentada

Para realização da previsão retroalimentada, foi implementado um procedimento iterativo, no qual uma janela móvel de 1440 dados foi utilizada como entrada para o método de previsão dos modelos. Na primeira iteração, foram utilizados os últimos 1440 dados de treinamento e o último valor de previsão da vazão gerado, foi inserido no lugar do 1º dado de vazão de entrada do conjunto de testes, assim como adicionado num vetor para análise das previsões. Em seguida, a janela avança um índice e o processo de geração da previsão, substituição do último dado de vazão na entrada de índice i+1 e adição no vetor das previsões se repete, até que o vetor de análise das previsões tenha a mesma quantidade de dados que os dados de teste de saída.

5 RESULTADOS

Para comparação dos resultados, as saídas (previsões) dos modelos, assim como os valores de MSE foram divididos em 3 categorias: dados de validação (teste), dados de abrangência e dados da previsão retroalimentada.

5.1 Dados de validação (teste)

Os resultados (MSE) das previsões de cada modelo, em relação aos respectivos dados de teste (validação) estão conforme Tabela 3. Sendo possível perceber que, todos os resultados estão na mesma ordem de grandeza, mas os modelos do tipo LSTM apresentaram um desempenho ligeiramente melhor.

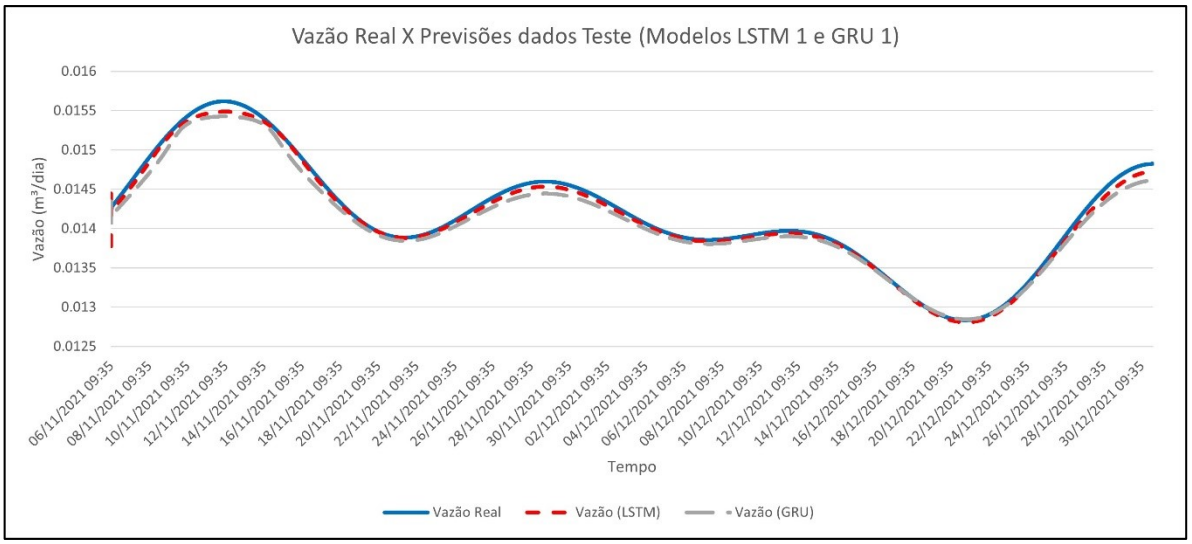
Tabela 3 – Resultados (MSE) de cada modelo para os dados de validação

Modelo	**MSE (Dados Teste)**
LSTM 1	6.178×10^{-4}
LSTM 2	6.434×10^{-4}
GRU 1	7.726×10^{-4}
GRU 2	7.150×10^{-4}

Fonte: Autoria própria

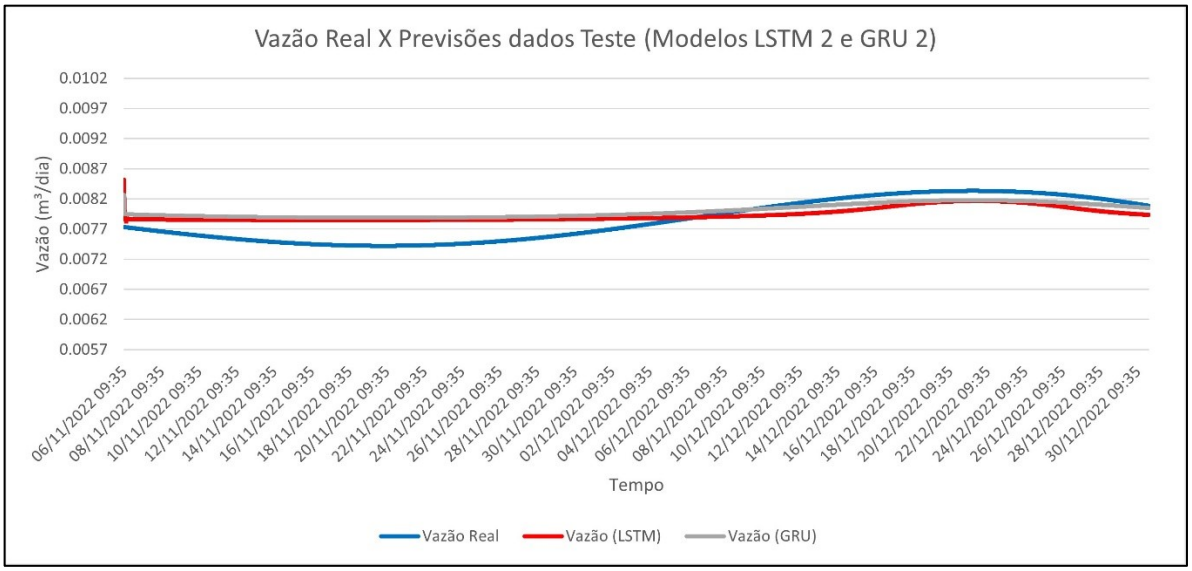
A visualização da previsão de cada modelo pode ser vista na Figura 18 e na Figura 19. Na qual é possível observar que, todos os modelos conseguiram reproduzir o comportamento dos dados de validação.

Figura 18 - Gráfico da Vazão Real X Previsões dos dados Teste dos modelos LSTM 1 e GRU 1



Fonte: Autoria própria

Figura 19 - Gráfico da Vazão Real X Previsões dos dados Teste dos modelos LSTM 2 e GRU 2



Fonte: Autoria própria

5.2 Dados de abrangência

A verificação da abrangência de uma rede neural indica se ela é capaz de, ao receber dados desconhecidos e diferentes dos dados de teste, manter o desempenho quanto aos dados de validação. Os resultados das previsões de cada modelo, em relação aos respectivos dados de abrangência estão conforme Tabela 4.

Tabela 4 - Resultados (MSE) de cada modelo para os dados de abrangência

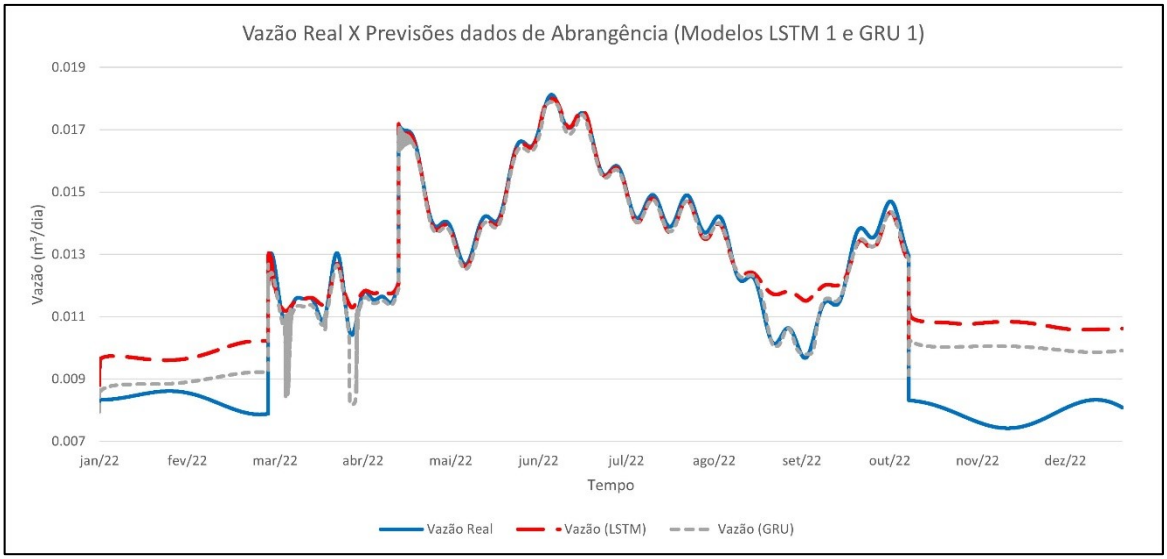
Modelo	MSE (Dados Abrangência)
LSTM 1	2.119×10^{-2}
LSTM 2	2.655×10^{-3}
GRU 1	1.146×10^{-2}
GRU 2	2.846×10^{-3}

Fonte: Autoria própria

A partir dos dados da Tabela 4 é possível identificar que, os modelos treinados com o 2º conjunto de dados tiveram um desempenho melhor do que os treinados com o 1º conjunto, ainda sim todos os modelos apresentaram um desempenho considerado bom.

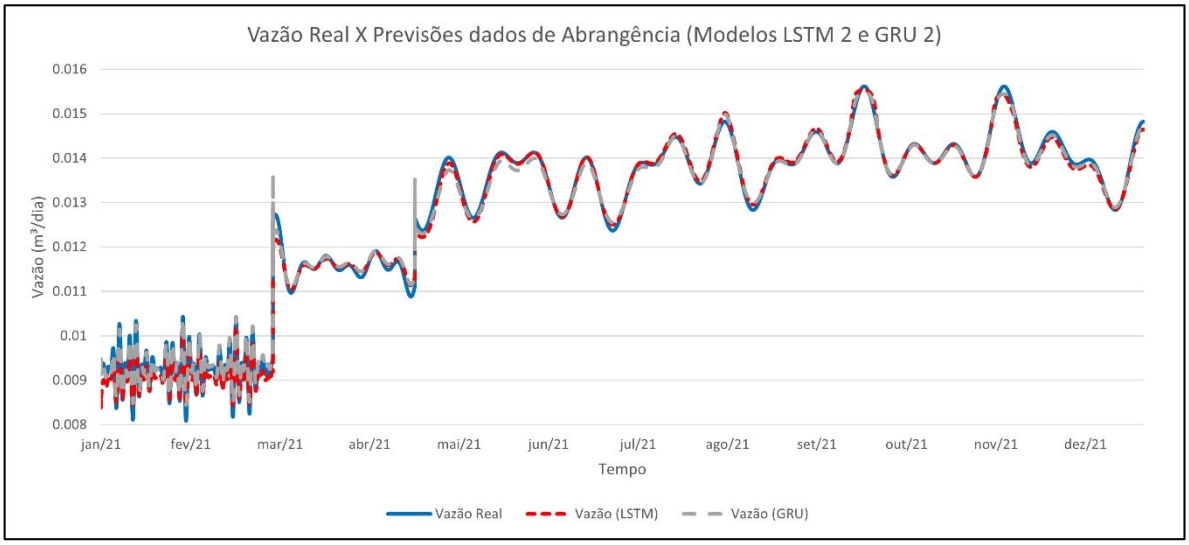
As saídas (previsões) para os dados de abrangência são apresentadas nas Figura 20 e Figura 21. Sendo possível perceber que os modelos LSTM 1 e GRU 1, apresentaram algumas discrepâncias no comportamento inicial e final das previsões, como visto na Figura 20. O que não ocorreu com as saídas dos modelos LSTM 2 e GRU 2, conforme Figura 21.

Figura 20 - Gráfico da Vazão Real X Previsões dos dados de Abrangência dos modelos LSTM 1 e GRU 1



Fonte: Autoria própria

Figura 21 - Gráfico da Vazão Real X Previsões dos dados de Abrangência dos modelos LSTM 2 e GRU 2



Fonte: Autoria própria

5.3 Dados da previsão retroalimentada

Os resultados (MSE) das previsões retroalimentadas de cada modelo, em relação aos respectivos dados de teste (validação) estão conforme Tabela 5.

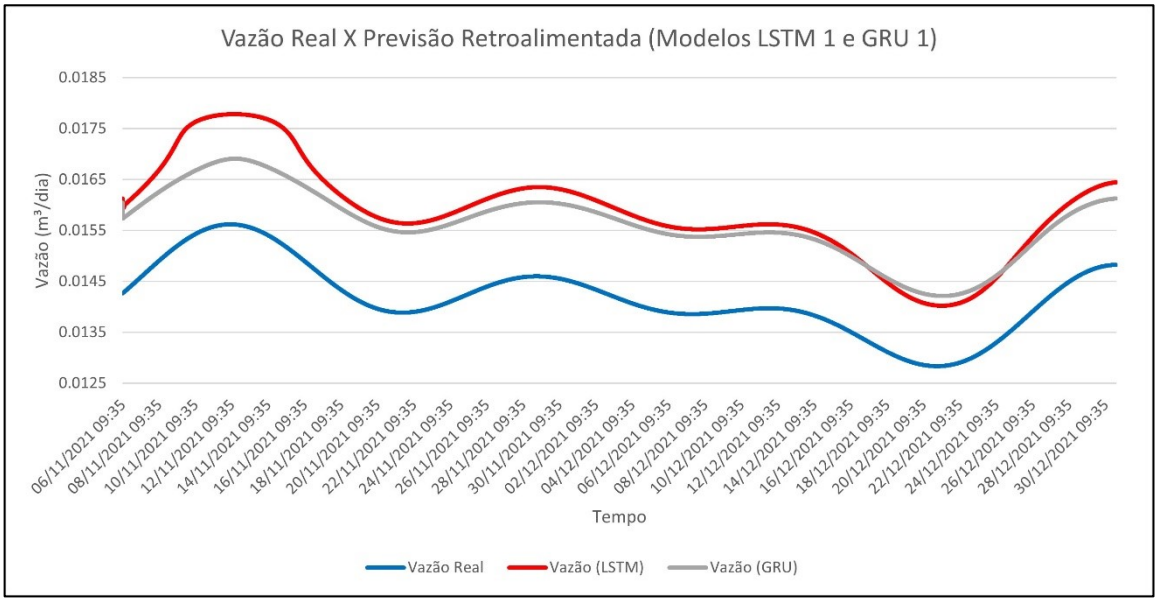
Tabela 5 - Resultados (MSE) de cada modelo para a previsão retroalimentada

Modelo	MSE (Previsão Retroalimentada)
LSTM 1	2.228×10^{-2}
LSTM 2	5.754×10^{-5}
GRU 1	1.804×10^{-4}
GRU 2	4.559×10^{-5}

Fonte: Autoria própria

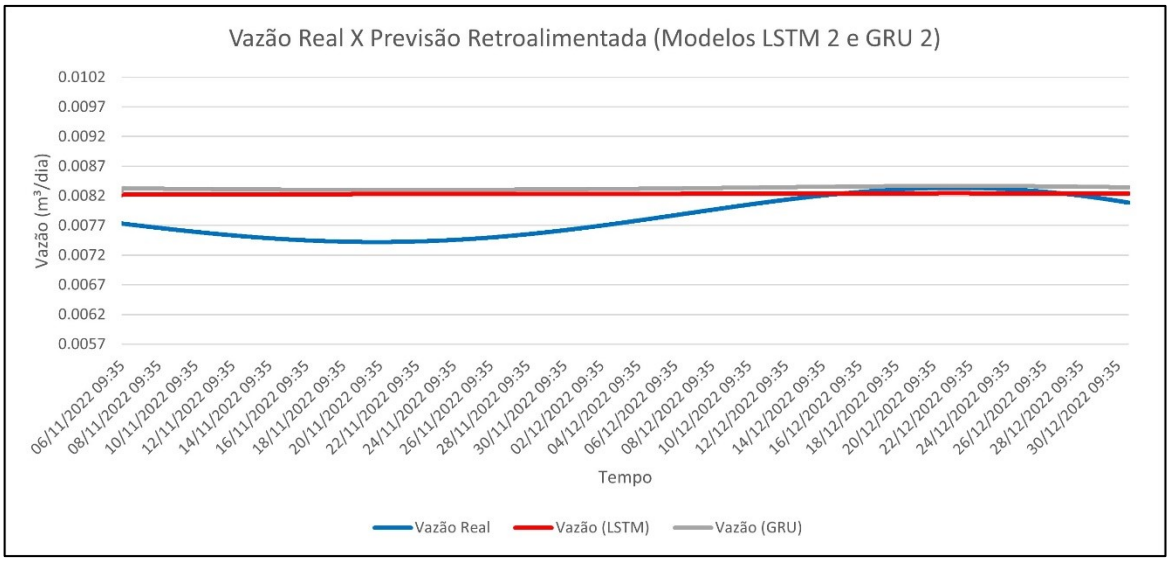
A visualização das previsões retroalimentadas estão conforme Figura 22 e Figura 23. E é possível perceber que, todos os modelos foram capazes de reproduzir o comportamento dos dados reais, entretanto, o modelo LSTM 1 foi o que apresentou pior desempenho, como visto também na Tabela 5.

Figura 22 - Gráfico da Vazão Real X Previsão Retroalimentada dos modelos LSTM 1 e GRU 1



Fonte: Autoria própria

Figura 23 - Gráfico da Vazão Real X Previsão Retroalimentada dos modelos LSTM 2 e GRU 2



Fonte: Autoria própria

6 CONCLUSÃO

O presente trabalho buscou elaborar um modelo, capaz de estimar a vazão de produção de poços utilizando técnicas de inteligência artificial. Diante dos resultados obtidos, é possível concluir que os 4 modelos de rede elaborados (LSTM 1, LSTM 2, GRU 1 e GRU 2), foram capazes de estimar a vazão de produção com um nível de erro consideravelmente reduzido.

Além disso, a performance das redes foi muito parecida, indicando que para esse caso, o tipo de RNN (LSTM ou GRU) pouco afetou o desempenho. Mas é válido ressaltar que, como o treinamento foi feito utilizando pontos iniciais aleatórios, que ao realizar outras simulações com os mesmos códigos, os resultados podem variar, tanto positivamente quanto negativamente.

REFERÊNCIAS

BI, Q. *et al*. **What is machine learning? A primer for the epidemiologist**. American journal of epidemiology, v. 188, n. 12, p. 2222-2239, 2019.

CHEN, M. *et al*. **Machine learning for wireless networks with artificial intelligence**: A tutorial on neural networks. arXiv preprint arXiv:1710.02913, v. 9, 2017.

CHO, K. *et al*. **Learning phrase representations using RNN encoder-decoder for statistical machine translation**. arXiv preprint arXiv:1406.1078, 2014.

CHUNG, J. *et al*. **Empirical evaluation of gated recurrent neural networks on sequence modeling**. arXiv preprint arXiv:1412.3555, 2014.

CHUNG, J. *et al*. **Gated feedback recurrent neural networks**. In: International conference on machine learning. PMLR, 2015. p. 2067-2075.

DUTTA, A.; KUMAR, S.; BASU, M. **A gated recurrent unit approach to bitcoin price prediction**. Journal of Risk and Financial Management, v. 13, n. 2, p. 23, 2020.

FLECK, L. *et al*. **Redes neurais artificiais: Princípios básicos**. Revista Eletrônica Científica Inovação e Tecnologia, v. 1, n. 13, p. 47-57, 2016.

GARUZZI, R. P.; ROMERO, O. J. **Abordagem analítica e computacional do teste drawdown**. Latin American Journal of Energy Research, v. 1, n. 1, p. 39-45, 2014.

GHAHRAMANI, Z. **Unsupervised learning**. In: Summer school on machine learning. Springer, Berlin, Heidelberg, 2003. p. 72-112.

GOMES, D. D. S. **Inteligência Artificial: conceitos e aplicações**. Olhar Científico. v1, n. 2, p. 234-246, 2010.

HE, L. et al. **Development and prospect of separated zone oil production technology**. Petroleum Exploration and Development, v. 47, n. 5, p. 1103-1116, 2020.

HOCHREITER, S.; SCHMIDHUBER, J. **Long short-term memory**. Neural computation, v. 9, n. 8, p. 1735-1780, 1997.

LECUN, Y.; BENGIO, Y.; HINTON, G. **Deep learning**. nature, v. 521, n. 7553, p. 436-444, 2015.

LEMLEY, J.; BAZRAFKAN, S.; CORCORAN, P. **Deep Learning for Consumer Devices and Services: Pushing the limits for machine learning, artificial intelligence, and computer vision**. IEEE Consumer Electronics Magazine, v. 6, n. 2, p. 48-56, 2017.

NASTESKI, V. **An overview of the supervised machine learning methods**. Horizons. b, v. 4, p. 51-62, 2017.

ONGSULEE, P. **Artificial intelligence, machine learning and deep learning**. In: 2017 15th International Conference on ICT and Knowledge Engineering (ICT&KE). IEEE, 2017. p. 1-6.

PACHECO, C. A. R.; PEREIRA, N. S. **Deep learning conceitos e utilização nas diversas Áreas do conhecimento**. Revista Ada Lovelace, v. 2, p. 34-49, 2018.

RAUBER, T. W. **Redes neurais artificiais**. Universidade Federal do Espírito Santo, v. 29, 2005.

RODRIGUES, J. N. **Inflação no Brasil: uma aplicação de Séries Temporais e Redes Neurais Recorrentes**. Orientadora: Nádia Giaretta Biase, 2021, 64 f. TCC (Graduação) – Curso de Estatística, Faculdade de Matemática, Universidade Federal de Uberlândia, Minas Gerais, 2021.

RUSK, N. **Deep learning**. Nature Methods, v. 13, n. 1, p. 35-35, 2016.

SILVA, M. P. E. **Aplicação de redes neurais e artificiais no diagnóstico de falhas de turbinas a gás**. Dissertação (Mestrado) — PUC-RIO - PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO, p. 33-42, 2010.

TIAN, C. **Machine learning approaches for permanent downhole gauge data interpretation**. Stanford University, 2018.

TIAN, C.; HORNE, R. N. **Applying machine-learning techniques to interpret flow-rate, pressure, and temperature data from permanent downhole gauges**. SPE Reservoir Evaluation & Engineering, v. 22, n. 02, p. 386-401, 2019.

APÊNDICE A – Código de reamostragem dos dados

In [ ]:
```
#Import das Bibliotecas Necessárias e do diretório dos arquivos
import pandas as pd
import numpy as np
from pathlib import Path
from datetime import timedelta
path_dir = "C:/Users/Rafael/Desktop/hello/"
```

Definição dos DataFrames com base nos dados de simulação

In [ ]:
```
df = pd.read_csv (path_dir + "Resp_simulador_vazao1.csv",sep=';')
df2 = pd.read_csv (path_dir + "Resp_simulador_vazao2.csv",sep=';')
```

Verificação do DataFrame 1

In [ ]:
```
df.head()
```

Out[ ]:

	tempo	vazao	temp	pressao
0	0.000000	0.000000	334.0	49033000.00
1	0.000103	0.009259	334.0	49033000.00
2	0.000720	0.009259	334.0	49032999.97
3	0.002321	0.009259	334.0	49032999.83
4	0.005373	0.009259	334.0	49032999.30

Verificação do DataFrame 2

In [ ]:
```
df2.head()
```

Out[ ]:

	tempo	vazao	temp	pressao
0	0.000000	0.000000	334.0	49033000.00
1	0.000103	0.008333	334.0	49033000.00
2	0.000720	0.008333	334.0	49032999.97
3	0.002321	0.008333	334.0	49032999.83
4	0.005373	0.008333	334.0	49032999.30

Definição das datas iniciais de cada simulação e construção dos vetores de data/tempo a partir delas

In [ ]:
```
data_ini1 = pd.date_range('1/1/2021', periods=1, freq='s') #Data inicial da simulação 1
data_ini2 = pd.date_range('1/1/2022', periods=1, freq='s') #Data inicial da simulação 2

tf = [] #Inicialização do timeframe 1
for i in range(0,len(df)):

  tf.append(data_ini1 + timedelta(seconds=df['tempo'][i]))

tf2 = [] #Inicialização do timeframe 2
for i in range(0,len(df2)):

  tf2.append(data_ini2 + timedelta(seconds=df2['tempo'][i]))
```

Criação do TimeFrame 1 e visualização do cabeçalho

In [ ]:
```
T=pd.DataFrame(tf,columns=['tempo'])
T.head()
```

Out[ ]:

	tempo
0	2021-01-01 00:00:00.000000
1	2021-01-01 00:00:00.000103
2	2021-01-01 00:00:00.000720
3	2021-01-01 00:00:00.002321
4	2021-01-01 00:00:00.005373

Definição da coluna index e preenchimento do TimeFrame com os dados de Vazão, Temperatura e Pressão da simulação 1

In [ ]:
```
T_aux=T.set_index("tempo") #Definição da coluna index sendo a data/tempo
vazao = df['vazao'].to_numpy() #Vetor com os dados de vazão
temp = df['temp'].to_numpy() #Vetor com os dados de temperatura
pressao = df['pressao'].to_numpy() #Vetor com os dados de pressão
T_aux["vazao"]=vazao #Preenchimento dos dados de vazão, temperatura e pressão no TimeFrame 1
T_aux["temp"]=temp
T_aux["pressao"]=pressao
T_aux.head() #Visualização do cabeçalho
```

Out[ ]:

	vazao	temp	pressao
tempo			
2021-01-01 00:00:00.000000	0.000000	334.0	49033000.00
2021-01-01 00:00:00.000103	0.009259	334.0	49033000.00
2021-01-01 00:00:00.000720	0.009259	334.0	49032999.97
2021-01-01 00:00:00.002321	0.009259	334.0	49032999.83
2021-01-01 00:00:00.005373	0.009259	334.0	49032999.30

Criação do TimeFrame 2 e visualização do cabeçalho

In [ ]:
```
T2=pd.DataFrame(tf2,columns=['tempo'])
T2.head()
```

Out[ ]:

	tempo
0	2022-01-01 00:00:00.000000
1	2022-01-01 00:00:00.000103
2	2022-01-01 00:00:00.000720
3	2022-01-01 00:00:00.002321
4	2022-01-01 00:00:00.005373

Definição da coluna index e preenchimento do TimeFrame com os dados de Vazão, Temperatura e Pressão da simulação 1

In [ ]:
```
T2_aux=T2.set_index("tempo") #Definição da coluna index sendo a data/tempo
vazao2 = df2['vazao'].to_numpy() #Vetor com os dados de vazão
temp2 = df2['temp'].to_numpy() #Vetor com os dados de temperatura
pressao2 = df2['pressao'].to_numpy() #Vetor com os dados de pressão
T2_aux["vazao"]=vazao2 #Preenchimento dos dados de vazão, temperatura e pressão no TimeFrame
T2_aux["temp"]=temp2
T2_aux["pressao"]=pressao2
T2_aux.head() #Visualização do cabeçalho
```

Out[ ]:

tempo	vazao	temp	pressao
2022-01-01 00:00:00.000000	0.000000	334.0	49033000.00
2022-01-01 00:00:00.000103	0.008333	334.0	49033000.00
2022-01-01 00:00:00.000720	0.008333	334.0	49032999.97
2022-01-01 00:00:00.002321	0.008333	334.0	49032999.83
2022-01-01 00:00:00.005373	0.008333	334.0	49032999.30

Reorganização dos dados utilizando a média dos valores para intervalos de 5min

In [ ]:
```
T_resampled = T_aux.resample('5min').mean()
T2_resampled = T2_aux.resample('5min').mean()
```

Exportação dos dados reamostrados para arquivos .csv

In [ ]:
```
filepath = Path("C:/Users/Rafael/Desktop/hello/Simulador_vazao1_resampled.csv")
filepath.parent.mkdir(parents=True, exist_ok=True)
T_resampled.to_csv(filepath)
filepath2 = Path("C:/Users/Rafael/Desktop/hello/Simulador_vazao2_resampled.csv")
filepath2.parent.mkdir(parents=True, exist_ok=True)
T2_resampled.to_csv(filepath2)
```

APÊNDICE B – Código Do Modelo LSTM 1

In [ ]:
```
#Import das Bibliotecas Necessárias
%matplotlib inline
import matplotlib.pyplot as plt
import tensorflow as tf
import numpy as np
import pandas as pd
import os
from pathlib import Path
from sklearn.preprocessing import MinMaxScaler

from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, GRU, LSTM
from tensorflow.keras.optimizers import RMSprop
from tensorflow.keras.callbacks import EarlyStopping, ModelCheckpoint
from tensorflow.keras.callbacks import TensorBoard, ReduceLROnPlateau
from tensorflow.keras.backend import square, mean
```

In [ ]:
```
#Versões das Bibliotecas Principais
print("Versão Tensorflow:",tf.__version__)
print("Versão Keras:",tf.keras.__version__)
print("Versão Pandas:",pd.__version__)
print("Versão Numpy:",np.__version__)
```

```
Versão Tensorflow: 2.7.0
Versão Keras: 2.7.0
Versão Pandas: 1.3.5
Versão Numpy: 1.22.1
```

Criação dos DataFrames a partir dos conjuntos de dados

In [ ]:
```
path_dir = "C:/Users/Rafael/Desktop/hello/" #Diretório do arquivo
df = pd.read_csv(path_dir + "Simulador_vazao1_resampled.csv",sep=';',header=0,
                 names=["Tempo","Vazão","Temperatura","Pressão"],
                 index_col="Tempo")
df.head(n=5) #Visualização do cabeçalho
```

Out[ ]:

Tempo	Vazão	Temperatura	Pressão
01/01/2021 00:00	0.009200	333.993896	48670544.52
01/01/2021 00:05	0.009259	333.993198	48424272.55
01/01/2021 00:10	0.009259	333.994561	48367386.80
01/01/2021 00:15	0.009259	333.995761	48328308.56
01/01/2021 00:20	0.009259	333.996818	48298087.55

In [ ]:
```
df2 = pd.read_csv(path_dir + "Simulador_vazao2_resampled.csv",sep=';',header=0,
                 names=["Tempo","Vazão","Temperatura","Pressão"],
                 index_col="Tempo")
df2.head(n=5) #Visualização do cabeçalho
```

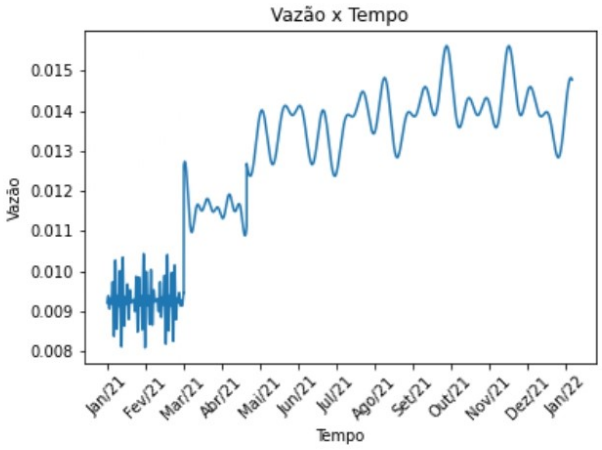
Out[ ]:

Tempo	Vazão	Temperatura	Pressão
01/01/2022 00:00	0.008280	333.994414	48706532.88
01/01/2022 00:05	0.008333	333.993486	48485096.35
01/01/2022 00:10	0.008333	333.994575	48433867.60
01/01/2022 00:15	0.008333	333.995567	48398652.38
01/01/2022 00:20	0.008333	333.996456	48371396.34

Plot da Vazão X Tempo

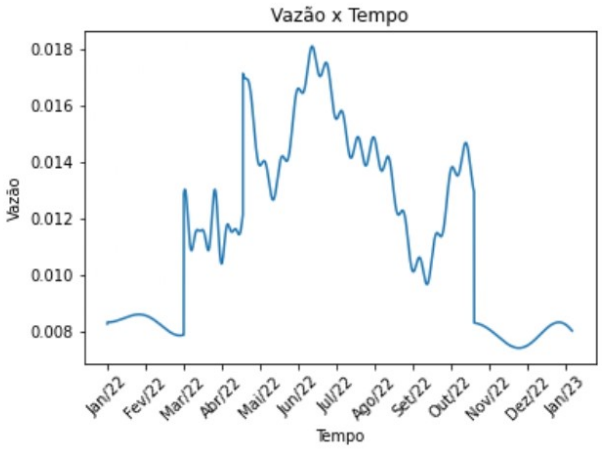
In [ ]:
```
ax1 = df['Vazão'].plot()
ax1.set_xticks([288*30*i for i in range(13)],["Jan/21", "Fev/21", "Mar/21",
            "Abr/21", "Mai/21", "Jun/21", "Jul/21", "Ago/21", "Set/21", "Out/21",
            "Nov/21", "Dez/21", "Jan/22"])
plt.xticks(rotation=45)
plt.ylabel("Vazão")
plt.title("Vazão x Tempo")
```

Out[ ]:
```
Text(0.5, 1.0, 'Vazão x Tempo')
```



In [ ]:
```
ax2 = df2['Vazão'].plot()
ax2.set_xticks([288*30*i for i in range(13)],["Jan/22", "Fev/22", "Mar/22",
            "Abr/22", "Mai/22", "Jun/22", "Jul/22", "Ago/22", "Set/22", "Out/22",
            "Nov/22", "Dez/22", "Jan/23"])
plt.xticks(rotation=45)
plt.ylabel("Vazão")
plt.title("Vazão x Tempo")
```

Out[ ]:
```
Text(0.5, 1.0, 'Vazão x Tempo')
```



In [ ]:
```
#Formato dos DataFrames
print("Formato DataFrame 1: ",df.values.shape)
print("Formato DataFrame 2: ",df2.values.shape)
```

```
Formato DataFrame 1:  (105121, 3)
Formato DataFrame 2:  (105121, 3)
```

In [ ]:
```
#Parâmetro Alvo
target_names = ["Vazão"]
```

In [ ]:
```
#Número de passos deslocados
shift_steps = 288 #Equivale a 1 dia
```

Criação dos DataFrame Targets, deslocando do parâmetro escolhido o número de passos

In [ ]:
```
df_targets = df[target_names].shift(-shift_steps)
df_targets2 = df2[target_names].shift(-shift_steps)
```

In [ ]:
```
df[target_names].tail(shift_steps+5)
```

Out[ ]:

	Vazão
Tempo	
30/12/2021 23:40	0.014823
30/12/2021 23:45	0.014823
30/12/2021 23:50	0.014823
30/12/2021 23:55	0.014823
31/12/2021 00:00	0.014823
...	...
31/12/2021 23:40	0.014776
31/12/2021 23:45	0.014776
31/12/2021 23:50	0.014775
31/12/2021 23:55	0.014775
01/01/2022 00:00	0.014775

293 rows × 1 columns

In [ ]:
```
df_targets.tail(shift_steps + 5)
```

Out[ ]:

Tempo	Vazão
30/12/2021 23:40	0.014776
30/12/2021 23:45	0.014776
30/12/2021 23:50	0.014775
30/12/2021 23:55	0.014775
31/12/2021 00:00	0.014775
...	...
31/12/2021 23:40	NaN
31/12/2021 23:45	NaN
31/12/2021 23:50	NaN
31/12/2021 23:55	NaN
01/01/2022 00:00	NaN

293 rows × 1 columns

In [ ]:
```
df2[target_names].tail(shift_steps+5)
```

Out[ ]:

Tempo	Vazão
30/12/2022 23:40	0.008085
30/12/2022 23:45	0.008085
30/12/2022 23:50	0.008085
30/12/2022 23:55	0.008085
31/12/2022 00:00	0.008084
...	...
31/12/2022 23:40	0.008030
31/12/2022 23:45	0.008030
31/12/2022 23:50	0.008030
31/12/2022 23:55	0.008030
01/01/2023 00:00	0.008030

293 rows × 1 columns

In [ ]:
```
df_targets2.tail(shift_steps + 5)
```

Out[ ]:

Tempo	Vazão
30/12/2022 23:40	0.00803
30/12/2022 23:45	0.00803
30/12/2022 23:50	0.00803
30/12/2022 23:55	0.00803
31/12/2022 00:00	0.00803
...	...
31/12/2022 23:40	NaN
31/12/2022 23:45	NaN
31/12/2022 23:50	NaN
31/12/2022 23:55	NaN
01/01/2023 00:00	NaN

293 rows × 1 columns

Definição dos vetores de entrada

In [ ]:
```
x_data = df.values[0:-shift_steps]
x_data2 = df2.values[0:-shift_steps]
```

In [ ]:
```
print(type(x_data))
print("Formato inputs 1:", x_data.shape)
print(type(x_data2))
print("Formato inputs 2:", x_data2.shape)
```

```
<class 'numpy.ndarray'>
Formato inputs 1: (104833, 3)
<class 'numpy.ndarray'>
Formato inputs 2: (104833, 3)
```

Definição dos vetores de saída

In [ ]:
```
y_data = df_targets.values[:-shift_steps]
y_data2 = df_targets2.values[:-shift_steps]
```

In [ ]:
```
print(type(y_data))
print("Formato outputs 1:", y_data.shape)
print(type(y_data2))
print("Formato outputs 2:", y_data2.shape)
```

```
<class 'numpy.ndarray'>
Formato outputs 1: (104833, 1)
<class 'numpy.ndarray'>
Formato outputs 2: (104833, 1)
```

Definição da quantidade de dados de treinamento e teste

In [ ]:
```
num_data = len(x_data)
train_split = 0.85
num_train = int(train_split * num_data)
print("Número de dados de treinamento:",num_train)
```

```
Número de dados de treinamento: 89108
```

In [ ]:
```
num_test = num_data - num_train
print("Número de dados de teste:",num_test)
```

```
Número de dados de teste: 15725
```

Separação da entrada nos dados de treinamento e teste

In [ ]:
```
x_train = x_data[0:num_train]
x_test = x_data[num_train:]
```

Separação da saída nos dados de treinamento e teste

In [ ]:
```
y_train = y_data[0:num_train]
y_test = y_data[num_train:]
```

In [ ]:
```
#Quantidade de parâmetros de entrada
num_x_signals = x_data.shape[1]
print("Parâmetros de entrada:",num_x_signals)
```

```
Parâmetros de entrada: 3
```

In [ ]:
```
#Quantidade de parâmetros de saída
num_y_signals = y_data.shape[1]
print("Parâmetros de saída:",num_y_signals)
```

```
Parâmetros de saída: 1
```

In [ ]:
```
#Valores mínimo e máximo dos dados de treinamento
print("Min:", np.min(x_train))
print("Max:", np.max(x_train))
```

```
Min: 0.00808754
Max: 48670544.52
```

In [ ]:
```
#Método para escalar os dados
x_scaler = MinMaxScaler()
```

In [ ]:
```
#Variável com os dados de entrada de treinamento escalados entre [0,1]
x_train_scaled = x_scaler.fit_transform(x_train)
print("Min:", np.min(x_train_scaled))
print("Max:", np.max(x_train_scaled))
```

```
Min: 0.0
Max: 1.0000000000000018
```

In [ ]:
```
#Variável com os dados de entrada de teste escalados entre [0,1]
x_test_scaled = x_scaler.transform(x_test)
```

In [ ]:
```
y_scaler = MinMaxScaler()
#Variável com os dados de saída de treinamento escalados entre [0,1]
y_train_scaled = y_scaler.fit_transform(y_train)

#Variável com os dados de saída de teste escalados entre [0,1]
y_test_scaled = y_scaler.transform(y_test)
```

Função para criação de batchs de treinamento, utilizando pontos iniciais aleatórios

In [ ]:
```
def batch_generator(batch_size, sequence_length):
    """
    Generator function for creating random batches of training-data.
    """

    # Infinite loop.
    while True:
        # Allocate a new array for the batch of input-signals.
        x_shape = (batch_size, sequence_length, num_x_signals)
        x_batch = np.zeros(shape=x_shape, dtype=np.float64)

        # Allocate a new array for the batch of output-signals.
        y_shape = (batch_size, sequence_length, num_y_signals)
```

```
        y_batch = np.zeros(shape=y_shape, dtype=np.float64)

        # Fill the batch with random sequences of data.
        for i in range(batch_size):
            # Get a random start-index.
            # This points somewhere into the training-data.
            idx = np.random.randint(num_train - sequence_length)

            # Copy the sequences of data starting at this index.
            x_batch[i] = x_train_scaled[idx:idx+sequence_length]
            y_batch[i] = y_train_scaled[idx:idx+sequence_length]

        yield (x_batch, y_batch)
```

In [ ]:
```
#Quantidade de batchs
batch_size = 150
```

In [ ]:
```
#Tamanho da sequência de dados
sequence_length = 288 * 5 #Corresponde a 5 dias
print("Tamanho da sequência:",sequence_length)
```

```
Tamanho da sequência: 1440
```

In [ ]:
```
generator = batch_generator(batch_size=batch_size,
                            sequence_length=sequence_length)
```

In [ ]:
```
#Criação dos batchs
x_batch, y_batch = next(generator)
```

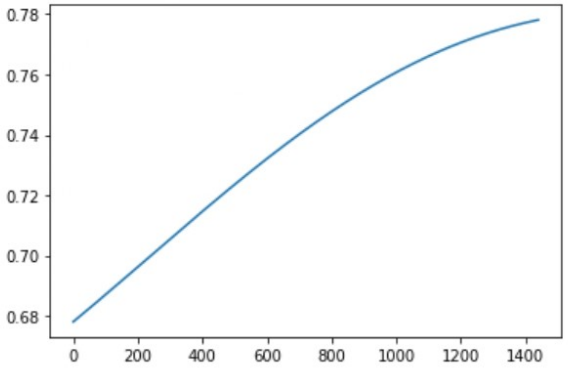
In [ ]:
```
print("Formato do x_batch:",x_batch.shape)
print("Formato do y_batch:",y_batch.shape)
```

```
Formato do x_batch: (150, 1440, 3)
Formato do y_batch: (150, 1440, 1)
```

Visualização da vazão, para a sequência do 1º batch dos dados de entrada

In [ ]:
```
batch = 0   # First sequence in the batch.
signal = 0  # First signal from the 3 input-signals.
seq = x_batch[batch, :, signal]
plt.plot(seq)
```

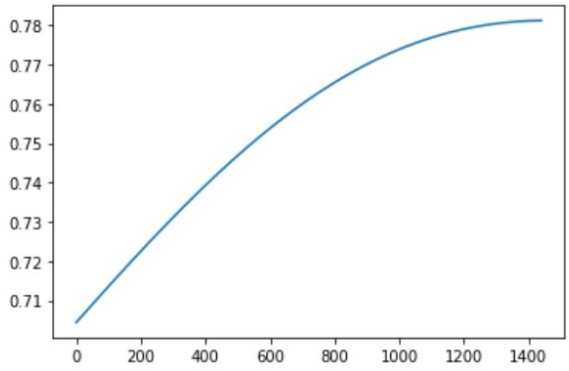
Out[ ]:
```
[<matplotlib.lines.Line2D at 0x21e26e3f310>]
```



Visualização da vazão, para a sequência do 1º batch dos dados de saída

In [ ]:
```
seq = y_batch[batch, :, signal]
plt.plot(seq)
```

Out[ ]:
```
[<matplotlib.lines.Line2D at 0x21e25ca2d60>]
```



In [ ]:
```
#Definição dos dados de validação
validation_data = (np.expand_dims(x_test_scaled, axis=0),
                   np.expand_dims(y_test_scaled, axis=0))
```

Criação da RNN do tipo LSTM

In [ ]:
```
model = Sequential()
```

In [ ]:
```
#1ª camada da rede - LSTM
model.add(LSTM(units=375,
               return_sequences=True,
               input_shape=(None, num_x_signals,)))
```

In [ ]:
```
#2ª camada da rede - LSTM
model.add(LSTM(units=375,
               return_sequences=True))
```

In [ ]:
```
#3ª camada da rede - Densa - Função de ativação: sigmoidal
model.add(Dense(200, activation='sigmoid'))
```

In [ ]:
```
#4ª camada da rede - Densa - Função de ativação: sigmoidal
model.add(Dense(num_y_signals, activation='sigmoid'))
```

In [ ]:
```
#Otimizador
optimizer = RMSprop(learning_rate=1e-3)
```

In [ ]:
```
#Compilação do modelo
model.compile(loss="mse", optimizer=optimizer)
```

In [ ]:
```
#Visualização do sumário da rede, com as camadas,
# número de unidades e parâmetros treináveis
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 lstm (LSTM)                 (None, None, 375)         568500

 lstm_1 (LSTM)               (None, None, 375)         1126500

 dense (Dense)               (None, None, 200)         75200

 dense_1 (Dense)             (None, None, 1)           201

=================================================================
Total params: 1,770,401
Trainable params: 1,770,401
Non-trainable params: 0
_________________________________________________________________
```

In [ ]:
```
#Checkpoint para salvamento dos pesos
path_checkpoint = 'LSTM_Modelo1_checkpoint.keras'
callback_checkpoint = ModelCheckpoint(filepath=path_checkpoint,
                                      monitor='val_loss',
                                      verbose=1,
                                      save_weights_only=True,
                                      save_best_only=True)
```

In [ ]:
```
#Método para interrupção antecipada do treinamento, caso a rede pare de aprender
callback_early_stopping = EarlyStopping(monitor='val_loss',
                                        patience=5, verbose=1)
```

In [ ]:
```
callback_tensorboard = TensorBoard(log_dir='./LSTM_Modelo1_logs/',
                                   histogram_freq=0,
                                   write_graph=False)
```

In [ ]:
```
#Método para redução da taxa de aprendizado
callback_reduce_lr = ReduceLROnPlateau(monitor='val_loss',
                                       factor=0.1,
                                       min_lr=1e-6,
                                       patience=2,
                                       verbose=1)
```

In [ ]:
```
callbacks = [callback_early_stopping,
             callback_checkpoint,
             callback_tensorboard,
             callback_reduce_lr]
```

Treinamento da RNN

In [ ]:
```
%%time
model.fit(x=generator,
          epochs=25,
          steps_per_epoch=300,
          validation_data=validation_data,
          verbose=2,
          callbacks=callbacks)
```

```
Epoch 1/25

Epoch 00001: val_loss improved from inf to 0.00361, saving model to LSTM_Modelo1_checkpoint.k
eras
300/300 - 1068s - loss: 0.0129 - val_loss: 0.0036 - lr: 0.0010 - 1068s/epoch - 4s/step
Epoch 2/25

Epoch 00002: val_loss improved from 0.00361 to 0.00083, saving model to LSTM_Modelo1_checkpoi
nt.keras
300/300 - 1063s - loss: 0.0032 - val_loss: 8.3137e-04 - lr: 0.0010 - 1063s/epoch - 4s/step
Epoch 3/25

Epoch 00003: val_loss did not improve from 0.00083
300/300 - 1060s - loss: 0.0024 - val_loss: 0.0022 - lr: 0.0010 - 1060s/epoch - 4s/step
Epoch 4/25

Epoch 00004: val_loss did not improve from 0.00083

Epoch 00004: ReduceLROnPlateau reducing learning rate to 0.00010000000474974513.
300/300 - 1059s - loss: 0.0022 - val_loss: 0.0011 - lr: 0.0010 - 1059s/epoch - 4s/step
Epoch 5/25

Epoch 00005: val_loss improved from 0.00083 to 0.00062, saving model to LSTM_Modelo1_checkpoi
nt.keras
300/300 - 1041s - loss: 0.0014 - val_loss: 6.1785e-04 - lr: 1.0000e-04 - 1041s/epoch - 3s/ste
p
Epoch 6/25

Epoch 00006: val_loss did not improve from 0.00062
300/300 - 1040s - loss: 0.0013 - val_loss: 6.8395e-04 - lr: 1.0000e-04 - 1040s/epoch - 3s/ste
p
Epoch 7/25

Epoch 00007: val_loss did not improve from 0.00062

Epoch 00007: ReduceLROnPlateau reducing learning rate to 1.0000000474974514e-05.
300/300 - 1034s - loss: 0.0013 - val_loss: 6.5333e-04 - lr: 1.0000e-04 - 1034s/epoch - 3s/ste
p
Epoch 8/25

Epoch 00008: val_loss did not improve from 0.00062
300/300 - 1054s - loss: 0.0012 - val_loss: 6.6536e-04 - lr: 1.0000e-05 - 1054s/epoch - 4s/ste
p
Epoch 9/25

Epoch 00009: val_loss did not improve from 0.00062

Epoch 00009: ReduceLROnPlateau reducing learning rate to 1.0000000656873453e-06.
300/300 - 1046s - loss: 0.0011 - val_loss: 6.5430e-04 - lr: 1.0000e-05 - 1046s/epoch - 3s/ste
p
Epoch 10/25

Epoch 00010: val_loss did not improve from 0.00062
300/300 - 1046s - loss: 9.5717e-04 - val_loss: 6.5444e-04 - lr: 1.0000e-06 - 1046s/epoch - 3
s/step
Epoch 00010: early stopping
CPU times: total: 2h 29min 7s
Wall time: 2h 55min 12s
```

Out[ ]: <keras.callbacks.History at 0x16454fd1160>

In [ ]:
```python
#Carregamento dos pesos
try:
    model.load_weights(path_checkpoint)
except Exception as error:
    print("Error trying to load checkpoint.")
    print(error)
```

In [ ]:
```
#Avaliação da rede com os dados de teste
resultado_modelo1_LSTM = model.evaluate(x=np.expand_dims(x_test_scaled, axis=0),
                        y=np.expand_dims(y_test_scaled, axis=0))
```

```
1/1 [==============================] - 5s 5s/step - loss: 6.1785e-04
```

In [ ]:
```
print("MSE (Dados de Teste): %10.3e" % (resultado_modelo1_LSTM))
```

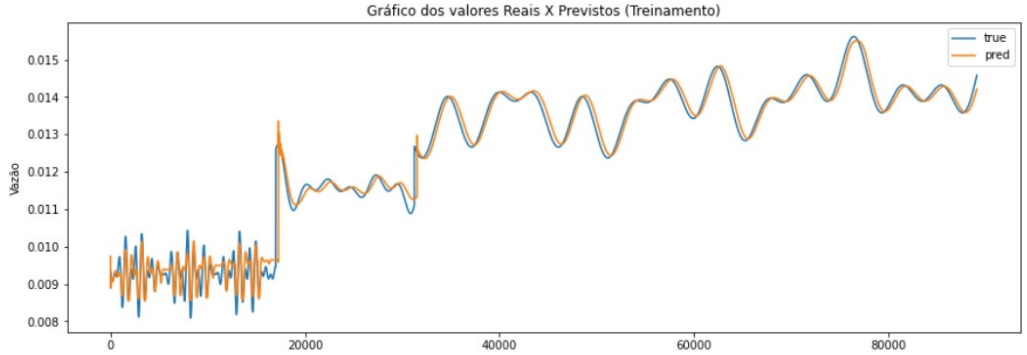
```
MSE (Dados de Teste):  6.178e-04
```

Função para plotar os dados reais e os dados previstos pela rede

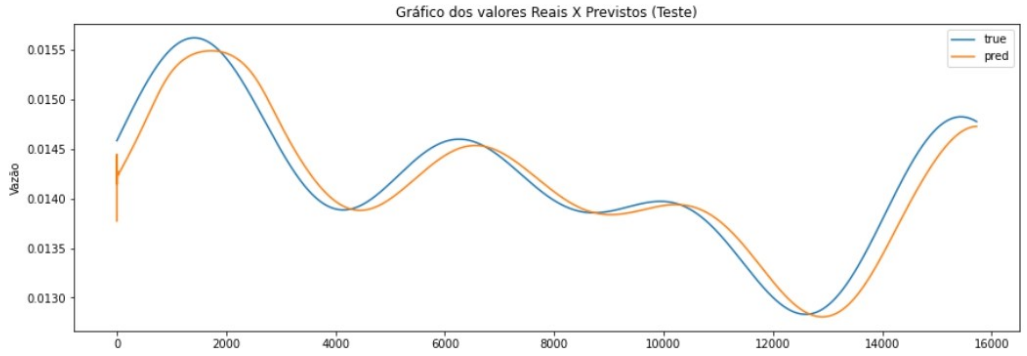
In [ ]:
```
def plot_comparison(start_idx, length=100, train=True):
    """
    Plot the predicted and true output-signals.

    :param start_idx: Start-index for the time-series.
    :param length: Sequence-length to process and plot.
    :param train: Boolean whether to use training- or test-set.
    """

    if train:
        # Use training-data.
        x = x_train_scaled
        y_true = y_train
        titulo = "Gráfico dos valores Reais X Previstos (Treinamento)"
    else:
        # Use test-data.
        x = x_test_scaled
        y_true = y_test
        titulo = "Gráfico dos valores Reais X Previstos (Teste)"

    # End-index for the sequences.
    end_idx = start_idx + length

    # Select the sequences from the given start-index and
    # of the given length.
    x = x[start_idx:end_idx]
    y_true = y_true[start_idx:end_idx]

    # Input-signals for the model.
    x = np.expand_dims(x, axis=0)

    # Use the model to predict the output-signals.
    y_pred = model.predict(x)

    # The output of the model is between 0 and 1.
    # Do an inverse map to get it back to the scale
    # of the original data-set.
    y_pred_rescaled = y_scaler.inverse_transform(y_pred[0])

    # For each output-signal.
    for signal in range(len(target_names)):
        # Get the output-signal predicted by the model.
        signal_pred = y_pred_rescaled[:, signal]

        # Get the true output-signal from the data-set.
        signal_true = y_true[:, signal]

        # Make the plotting-canvas bigger.
        plt.figure(figsize=(15,5))

        # Plot and compare the two signals.
        plt.plot(signal_true, label='true')
        plt.plot(signal_pred, label='pred')

        # Plot labels etc.
        plt.ylabel(target_names[signal])
        plt.title(titulo)
```

```
        plt.legend()
        plt.show()
```

In [ ]:
```
plot_comparison(start_idx=0, length=num_train, train=True)
```



In [ ]:
```
plot_comparison(start_idx=0, length=num_test, train=False)
```



In [ ]:
```
#Exportação dos dados de treinamento previstos para csv
y_prev_model1LSTM_dados1_train = model.predict(np.expand_dims(x_train_scaled,axis=0))
y_prev_model1LSTM_dados1_train_rescaled = y_scaler.inverse_transform(
    y_prev_model1LSTM_dados1_train[0])
y_prev_model1LSTM_dados1_train_df = pd.DataFrame(y_prev_model1LSTM_dados1_train_rescaled)
filepath = Path(path_dir + 'y_prev_model1LSTM_dados1_train.csv')
filepath.parent.mkdir(parents=True, exist_ok=True)
y_prev_model1LSTM_dados1_train_df.to_csv(filepath)
```

In [ ]:
```
#Exportação dos dados de teste previstos para csv
y_prev_model1LSTM_dados1_test = model.predict(np.expand_dims(x_test_scaled,axis=0))
y_prev_model1LSTM_dados1_test_rescaled = y_scaler.inverse_transform(
    y_prev_model1LSTM_dados1_test[0])
y_prev_model1LSTM_dados1_test_df = pd.DataFrame(y_prev_model1LSTM_dados1_test_rescaled)
filepath = Path(path_dir + 'y_prev_model1LSTM_dados1_test.csv')
filepath.parent.mkdir(parents=True, exist_ok=True)
y_prev_model1LSTM_dados1_test_df.to_csv(filepath)
```

Avaliação da abrangência do Modelo

In [ ]:
```
x_data2_scaled = x_scaler.fit_transform(x_data2)
y_data2_scaled = y_scaler.fit_transform(y_data2)
```

In [ ]:
```
resultado_modelo1LSTM_dados2 = model.evaluate(
                               x=np.expand_dims(x_data2_scaled, axis=0),
                               y=np.expand_dims(y_data2_scaled, axis=0))
```

```
1/1 [==============================] - 18s 18s/step - loss: 0.0212
```

In [ ]:
```
print("MSE (Conjunto de dados 2): %10.3e" % (resultado_modelo1LSTM_dados2))
```

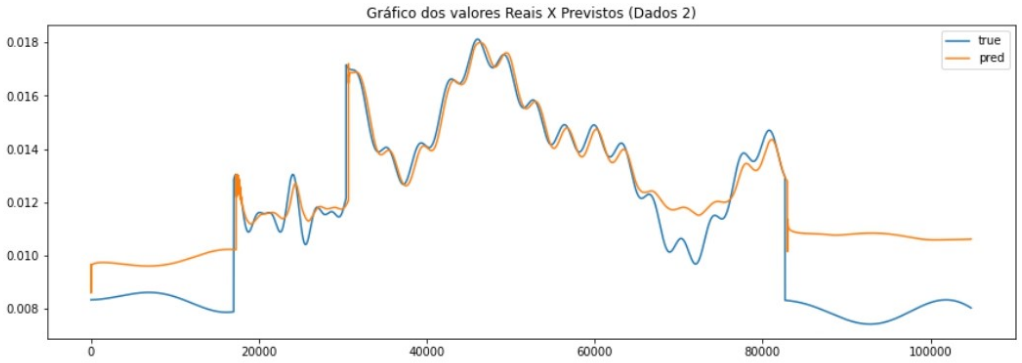
```
MSE (Conjunto de dados 2):  2.119e-02
```

Gráfico dos valores Reais X Previsão do Modelo 1, usando o conjunto de dados 2 como entrada

In [ ]:
```
#Exportação dos dados de abrangência previstos para csv
y_previsao_modelo1LSTM_dados2 = model.predict(np.expand_dims(x_data2_scaled, axis=0))
y_previsao_modelo1LSTM_dados2_rescaled = y_scaler.inverse_transform(
    y_previsao_modelo1LSTM_dados2[0])
y_previsao_modelo1LSTM_dados2_df = pd.DataFrame(y_previsao_modelo1LSTM_dados2_rescaled)
filepath = Path(path_dir + 'y_previsao_modelo1LSTM_dados2.csv')
filepath.parent.mkdir(parents=True, exist_ok=True)
y_previsao_modelo1LSTM_dados2_df.to_csv(filepath)
```

In [ ]:
```
plt.figure(figsize=(15,5))

# Plot and compare the two signals.
plt.plot(y_data2, label='true')
plt.plot(y_previsao_modelo1LSTM_dados2_rescaled, label='pred')
plt.title("Gráfico dos valores Reais X Previstos (Dados 2)")
plt.legend()
```

Out[ ]:
```
<matplotlib.legend.Legend at 0x21e4a2b9d60>
```



Previsão Retroalimentada

In [ ]:
```
def vector3d_generator(num_test, sequence_length):
    """
    Generator function for creating a 3D vector.
    """

    # Infinite loop.
    while True:
        # Allocate a new array for the batch of input-signals.
        x_shape = (1, num_test + sequence_length, num_x_signals)
        x_prev = np.zeros(shape=x_shape, dtype=np.float64)

        # Fill the batch with random sequences of data.
        for i in range(num_test + sequence_length):
            if i < sequence_length:
                x_prev[0:1,i] = x_train_scaled[len(x_train_scaled)-sequence_length+i:
                                               len(x_train_scaled)-sequence_length+i+1]
            else:
                x_prev[0:1,i] = x_test_scaled[i-sequence_length:i-sequence_length+1]

        yield (x_prev)
```

In [ ]:
```
vetor_generator = vector3d_generator(num_test=num_test,
                            sequence_length=sequence_length)
```

In [ ]:
```
x_prev = next(vetor_generator)
```

In [ ]:
```
y_previsao_retroalimentada = []
for i in range(0, num_test):
    previsao = model.predict(x_prev[0:,i:i+sequence_length])
    y_previsao_retroalimentada.append(previsao[0,-1,0])
    x_prev[0:,i+sequence_length,0]=previsao[0,-1,0]
```

In [ ]:
```
x_prev_rescaled = x_scaler.inverse_transform(x_prev[0])
```

In [ ]:
```
#Exportação dos dados da previsão retroalimentada para csv
y_previsao_retroalimentada_rescaled = y_scaler.inverse_transform(
    np.expand_dims(y_previsao_retroalimentada,axis=1))
y_previsao_retroalimentada_df = pd.DataFrame(y_previsao_retroalimentada_rescaled)
filepath = Path(path_dir + 'y_prev_retroalimentada_model1LSTM.csv')
filepath.parent.mkdir(parents=True, exist_ok=True)
y_previsao_retroalimentada_df.to_csv(filepath)
```

Avaliação da Previsão Retroalimentada

In [ ]:
```
resultado_previsao_retroalimentada = model.evaluate(
    x=np.expand_dims(x_prev_rescaled[1440:], axis=0),
y=np.expand_dims(y_previsao_retroalimentada_rescaled, axis=0))
```

```
1/1 [==============================] - 3s 3s/step - loss: 0.0223
```

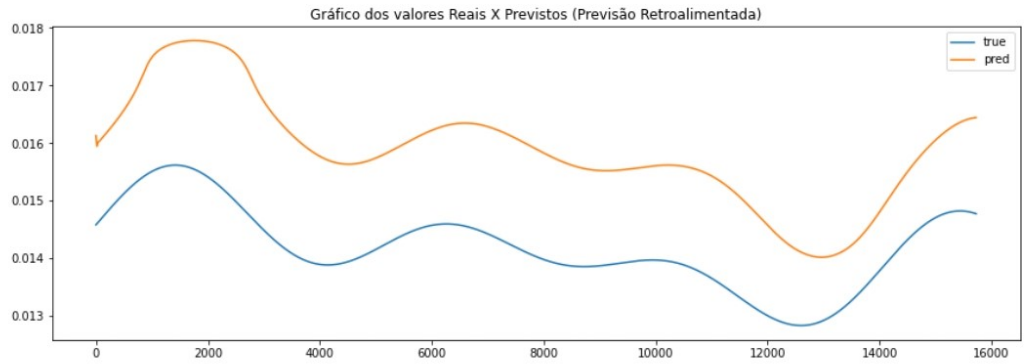
In [ ]:
```
print("MSE (Previsão Retroalimentada): %10.3e" % (resultado_previsao_retroalimentada))
```

```
MSE (Previsão Retroalimentada):  2.228e-02
```

Gráfico dos valores Reais X Previstos, utilizando a previsão da vazão como entrada no índice i+1

In [ ]:
```
plt.figure(figsize=(15,5))

# Plot and compare the two signals.
plt.plot(y_test, label='true')
plt.plot(y_previsao_retroalimentada_rescaled, label='pred')
plt.title("Gráfico dos valores Reais X Previstos (Previsão Retroalimentada)")
plt.legend()
```

Out[ ]:
```
<matplotlib.legend.Legend at 0x2202a3e01c0>
```



APÊNDICE C – Código do Modelo LSTM 2

In [ ]:
```
#Import das Bibliotecas Necessárias
%matplotlib inline
import matplotlib.pyplot as plt
import tensorflow as tf
import numpy as np
import pandas as pd
import os
from pathlib import Path
from sklearn.preprocessing import MinMaxScaler

from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, GRU, LSTM
from tensorflow.keras.optimizers import RMSprop
from tensorflow.keras.callbacks import EarlyStopping, ModelCheckpoint
from tensorflow.keras.callbacks import TensorBoard, ReduceLROnPlateau
from tensorflow.keras.backend import square, mean
```

In [ ]:
```
#Versões das Bibliotecas Principais
print("Versão Tensorflow:",tf.__version__)
print("Versão Keras:",tf.keras.__version__)
print("Versão Pandas:",pd.__version__)
print("Versão Numpy:",np.__version__)
```

```
Versão Tensorflow: 2.7.0
Versão Keras: 2.7.0
Versão Pandas: 1.3.5
Versão Numpy: 1.22.1
```

Criação dos DataFrames a partir dos conjuntos de dados

In [ ]:
```
path_dir = "C:/Users/Rafael/Desktop/hello/" #Diretório do arquivo
df = pd.read_csv(path_dir + "Simulador_vazao1_resampled.csv",sep=';',header=0,
                 names=["Tempo","Vazão","Temperatura","Pressão"],
                 index_col="Tempo")
df.head(n=5) #Visualização do cabeçalho
```

Out[ ]:

Tempo	Vazão	Temperatura	Pressão
01/01/2021 00:00	0.009200	333.993896	48670544.52
01/01/2021 00:05	0.009259	333.993198	48424272.55
01/01/2021 00:10	0.009259	333.994561	48367386.80
01/01/2021 00:15	0.009259	333.995761	48328308.56
01/01/2021 00:20	0.009259	333.996818	48298087.55

In [ ]:
```
df2 = pd.read_csv(path_dir + "Simulador_vazao2_resampled.csv",sep=';',header=0,
                 names=["Tempo","Vazão","Temperatura","Pressão"],
                 index_col="Tempo")
df2.head(n=5) #Visualização do cabeçalho
```

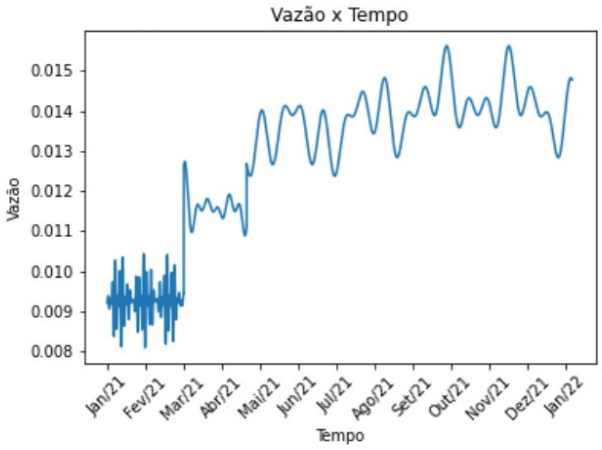
Out[ ]:

Tempo	Vazão	Temperatura	Pressão
01/01/2022 00:00	0.008280	333.994414	48706532.88
01/01/2022 00:05	0.008333	333.993486	48485096.35
01/01/2022 00:10	0.008333	333.994575	48433867.60
01/01/2022 00:15	0.008333	333.995567	48398652.38
01/01/2022 00:20	0.008333	333.996456	48371396.34

Plot da Vazão X Tempo

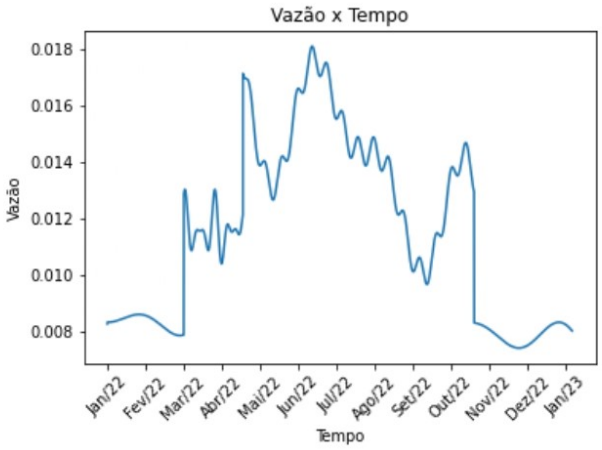
In [ ]:
```
ax1 = df['Vazão'].plot()
ax1.set_xticks([288*30*i for i in range(13)],["Jan/21", "Fev/21", "Mar/21",
            "Abr/21", "Mai/21", "Jun/21", "Jul/21", "Ago/21", "Set/21", "Out/21",
            "Nov/21", "Dez/21", "Jan/22"])
plt.xticks(rotation=45)
plt.ylabel("Vazão")
plt.title("Vazão x Tempo")
```

Out[ ]:
```
Text(0.5, 1.0, 'Vazão x Tempo')
```



In [ ]:
```
ax2 = df2['Vazão'].plot()
ax2.set_xticks([288*30*i for i in range(13)],["Jan/22", "Fev/22", "Mar/22",
            "Abr/22", "Mai/22", "Jun/22", "Jul/22", "Ago/22", "Set/22", "Out/22",
            "Nov/22", "Dez/22", "Jan/23"])
plt.xticks(rotation=45)
plt.ylabel("Vazão")
plt.title("Vazão x Tempo")
```

Out[ ]:
```
Text(0.5, 1.0, 'Vazão x Tempo')
```



In [ ]:
```
#Formato dos DataFrames
print("Formato DataFrame 1: ",df.values.shape)
print("Formato DataFrame 2: ",df2.values.shape)
```

```
Formato DataFrame 1:  (105121, 3)
Formato DataFrame 2:  (105121, 3)
```

In [ ]:
```
#Parâmetro Alvo
target_names = ["Vazão"]
```

In [ ]:
```
#Número de passos deslocados
shift_steps = 288 #Equivale a 1 dia
```

Criação dos DataFrame Targets, deslocando do parâmetro escolhido o número de passos

In [ ]:
```
df_targets = df[target_names].shift(-shift_steps)
df_targets2 = df2[target_names].shift(-shift_steps)
```

In [ ]:
```
df[target_names].tail(shift_steps+5)
```

Out[ ]:

	Vazão
Tempo	
30/12/2021 23:40	0.014823
30/12/2021 23:45	0.014823
30/12/2021 23:50	0.014823
30/12/2021 23:55	0.014823
31/12/2021 00:00	0.014823
...	...
31/12/2021 23:40	0.014776
31/12/2021 23:45	0.014776
31/12/2021 23:50	0.014775
31/12/2021 23:55	0.014775
01/01/2022 00:00	0.014775

293 rows × 1 columns

In [ ]:
```
df_targets.tail(shift_steps + 5)
```

Out[ ]:

Tempo	Vazão
30/12/2021 23:40	0.014776
30/12/2021 23:45	0.014776
30/12/2021 23:50	0.014775
30/12/2021 23:55	0.014775
31/12/2021 00:00	0.014775
...	...
31/12/2021 23:40	NaN
31/12/2021 23:45	NaN
31/12/2021 23:50	NaN
31/12/2021 23:55	NaN
01/01/2022 00:00	NaN

293 rows × 1 columns

In [ ]:
```
df2[target_names].tail(shift_steps+5)
```

Out[ ]:

Tempo	Vazão
30/12/2022 23:40	0.008085
30/12/2022 23:45	0.008085
30/12/2022 23:50	0.008085
30/12/2022 23:55	0.008085
31/12/2022 00:00	0.008084
...	...
31/12/2022 23:40	0.008030
31/12/2022 23:45	0.008030
31/12/2022 23:50	0.008030
31/12/2022 23:55	0.008030
01/01/2023 00:00	0.008030

293 rows × 1 columns

In [ ]:
```
df_targets2.tail(shift_steps + 5)
```

Out[ ]:

Tempo	Vazão
30/12/2022 23:40	0.00803
30/12/2022 23:45	0.00803
30/12/2022 23:50	0.00803
30/12/2022 23:55	0.00803
31/12/2022 00:00	0.00803
...	...
31/12/2022 23:40	NaN
31/12/2022 23:45	NaN
31/12/2022 23:50	NaN
31/12/2022 23:55	NaN
01/01/2023 00:00	NaN

293 rows × 1 columns

Definição dos vetores de entrada

In [ ]:
```
x_data = df.values[0:-shift_steps]
x_data2 = df2.values[0:-shift_steps]
```

In [ ]:
```
print(type(x_data))
print("Formato inputs 1:", x_data.shape)
print(type(x_data2))
print("Formato inputs 2:", x_data2.shape)
```

```
<class 'numpy.ndarray'>
Formato inputs 1: (104833, 3)
<class 'numpy.ndarray'>
Formato inputs 2: (104833, 3)
```

Definição dos vetores de saída

In [ ]:
```
y_data = df_targets.values[:-shift_steps]
y_data2 = df_targets2.values[:-shift_steps]
```

In [ ]:
```
print(type(y_data))
print("Formato outputs 1:", y_data.shape)
print(type(y_data2))
print("Formato outputs 2:", y_data2.shape)
```

```
<class 'numpy.ndarray'>
Formato outputs 1: (104833, 1)
<class 'numpy.ndarray'>
Formato outputs 2: (104833, 1)
```

Definição da quantidade de dados de treinamento e teste

In [ ]:
```
num_data = len(x_data2)
train_split = 0.85
num_train = int(train_split * num_data)
print("Número de dados de treinamento:",num_train)
```

```
Número de dados de treinamento: 89108
```

In [ ]:
```
num_test = num_data - num_train
print("Número de dados de teste:",num_test)
```

```
Número de dados de teste: 15725
```

Separação da entrada nos dados de treinamento e teste

In [ ]:
```
x_train = x_data2[0:num_train]
x_test = x_data2[num_train:]
```

Separação da saída nos dados de treinamento e teste

In [ ]:
```
y_train = y_data2[0:num_train]
y_test = y_data2[num_train:]
```

In [ ]:
```
#Quantidade de parâmetros de entrada
num_x_signals = x_data2.shape[1]
print("Parâmetros de entrada:",num_x_signals)
```

```
Parâmetros de entrada: 3
```

In [ ]:
```
#Quantidade de parâmetros de saída
num_y_signals = y_data2.shape[1]
print("Parâmetros de saída:",num_y_signals)
```

```
Parâmetros de saída: 1
```

In [ ]:
```
#Valores mínimo e máximo dos dados de treinamento
print("Min:", np.min(x_train))
print("Max:", np.max(x_train))
```

```
Min: 0.007733355
Max: 48706532.88
```

In [ ]:
```
#Método para escalar os dados
x_scaler = MinMaxScaler()
```

In [ ]:
```
#Variável com os dados de entrada de treinamento escalados entre [0,1]
x_train_scaled = x_scaler.fit_transform(x_train)
print("Min:", np.min(x_train_scaled))
print("Max:", np.max(x_train_scaled))
```

```
Min: 0.0
Max: 1.0
```

In [ ]:
```
#Variável com os dados de entrada de teste escalados entre [0,1]
x_test_scaled = x_scaler.transform(x_test)
```

In [ ]:
```
y_scaler = MinMaxScaler()
#Variável com os dados de saída de treinamento escalados entre [0,1]
y_train_scaled = y_scaler.fit_transform(y_train)

#Variável com os dados de saída de teste escalados entre [0,1]
y_test_scaled = y_scaler.transform(y_test)
```

Função para criação de batchs de treinamento, utilizando pontos iniciais aleatórios

In [ ]:
```
def batch_generator(batch_size, sequence_length):
    """
    Generator function for creating random batches of training-data.
    """

    # Infinite loop.
    while True:
        # Allocate a new array for the batch of input-signals.
        x_shape = (batch_size, sequence_length, num_x_signals)
        x_batch = np.zeros(shape=x_shape, dtype=np.float64)

        # Allocate a new array for the batch of output-signals.
        y_shape = (batch_size, sequence_length, num_y_signals)
```

```
        y_batch = np.zeros(shape=y_shape, dtype=np.float64)

        # Fill the batch with random sequences of data.
        for i in range(batch_size):
            # Get a random start-index.
            # This points somewhere into the training-data.
            idx = np.random.randint(num_train - sequence_length)

            # Copy the sequences of data starting at this index.
            x_batch[i] = x_train_scaled[idx:idx+sequence_length]
            y_batch[i] = y_train_scaled[idx:idx+sequence_length]

        yield (x_batch, y_batch)
```

In [ ]:
```
#Quantidade de batchs
batch_size = 150
```

In [ ]:
```
#Tamanho da sequência de dados
sequence_length = 288 * 5 #Corresponde a 5 dias
print("Tamanho da sequência:",sequence_length)
```

```
Tamanho da sequência: 1440
```

In [ ]:
```
generator = batch_generator(batch_size=batch_size,
                            sequence_length=sequence_length)
```

In [ ]:
```
#Criação dos batchs
x_batch, y_batch = next(generator)
```

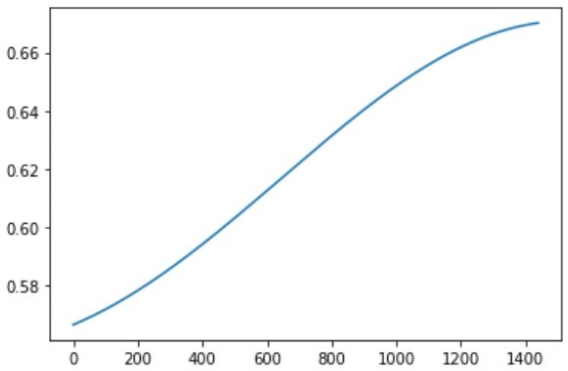
In [ ]:
```
print("Formato do x_batch:",x_batch.shape)
print("Formato do y_batch:",y_batch.shape)
```

```
Formato do x_batch: (150, 1440, 3)
Formato do y_batch: (150, 1440, 1)
```

Visualização da vazão, para a sequência do 1º batch dos dados de entrada

In [ ]:
```
batch = 0   # First sequence in the batch.
signal = 0  # First signal from the 3 input-signals.
seq = x_batch[batch, :, signal]
plt.plot(seq)
```

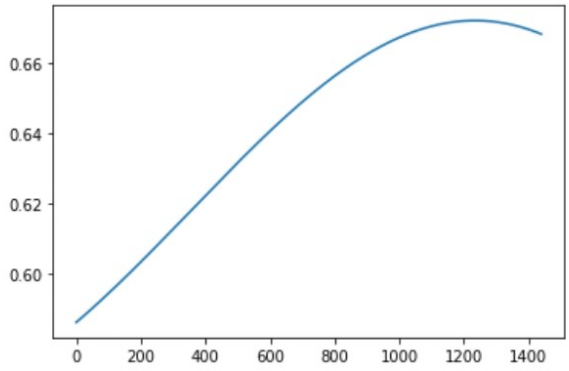
Out[ ]:
```
[<matplotlib.lines.Line2D at 0x1bbf9c82e50>]
```



Visualização da vazão, para a sequência do 1º batch dos dados de saída

In [ ]:
```
seq = y_batch[batch, :, signal]
plt.plot(seq)
```

Out[ ]:
```
[<matplotlib.lines.Line2D at 0x1bbebe589d0>]
```



In [ ]:
```
#Definição dos dados de validação
validation_data = (np.expand_dims(x_test_scaled, axis=0),
                   np.expand_dims(y_test_scaled, axis=0))
```

Criação da RNN do tipo LSTM

In [ ]:
```
model = Sequential()
```

In [ ]:
```
#1ª camada da rede - LSTM
model.add(LSTM(units=375,
               return_sequences=True,
               input_shape=(None, num_x_signals,)))
```

In [ ]:
```
#2ª camada da rede - LSTM
model.add(LSTM(units=375,
               return_sequences=True))
```

In [ ]:
```
#3ª camada da rede - Densa - Função de ativação: sigmoidal
model.add(Dense(200, activation='sigmoid'))
```

In [ ]:
```
#4ª camada da rede - Densa - Função de ativação: sigmoidal
model.add(Dense(num_y_signals, activation='sigmoid'))
```

In [ ]:
```
#Otimizador
optimizer = RMSprop(learning_rate=1e-3)
```

In [ ]:
```
#Compilação do modelo
model.compile(loss="mse", optimizer=optimizer)
```

In [ ]:
```
#Visualização do sumário da rede, com as camadas,
# número de unidades e parâmetros treináveis
model.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 lstm_2 (LSTM)               (None, None, 375)         568500

 lstm_3 (LSTM)               (None, None, 375)         1126500

 dense_2 (Dense)             (None, None, 200)         75200

 dense_3 (Dense)             (None, None, 1)           201

=================================================================
Total params: 1,770,401
Trainable params: 1,770,401
Non-trainable params: 0
_________________________________________________________________
```

In [ ]:
```
#Checkpoint para salvamento dos pesos
path_checkpoint = 'LSTM_Modelo2_checkpoint.keras'
callback_checkpoint = ModelCheckpoint(filepath=path_checkpoint,
                                      monitor='val_loss',
                                      verbose=1,
                                      save_weights_only=True,
                                      save_best_only=True)
```

In [ ]:
```
#Método para interrupção antecipada do treinamento, caso a rede pare de aprender
callback_early_stopping = EarlyStopping(monitor='val_loss',
                                        patience=5, verbose=1)
```

In [ ]:
```
callback_tensorboard = TensorBoard(log_dir='./LSTM_Modelo2_logs/',
                                   histogram_freq=0,
                                   write_graph=False)
```

In [ ]:
```
#Método para redução da taxa de aprendizado
callback_reduce_lr = ReduceLROnPlateau(monitor='val_loss',
                                       factor=0.1,
                                       min_lr=1e-6,
                                       patience=2,
                                       verbose=1)
```

In [ ]:
```
callbacks = [callback_early_stopping,
             callback_checkpoint,
             callback_tensorboard,
             callback_reduce_lr]
```

Treinamento da RNN

In [ ]:
```
%%time
model.fit(x=generator,
          epochs=25,
          steps_per_epoch=300,
          validation_data=validation_data,
          verbose=2,
          callbacks=callbacks)
```

```
Epoch 1/25

Epoch 00001: val_loss improved from inf to 0.00168, saving model to LSTM_Modelo2_checkpoint.k
eras
300/300 - 1252s - loss: 0.0110 - val_loss: 0.0017 - lr: 0.0010 - 1252s/epoch - 4s/step
Epoch 2/25

Epoch 00002: val_loss improved from 0.00168 to 0.00144, saving model to LSTM_Modelo2_checkpoi
nt.keras
300/300 - 1392s - loss: 0.0039 - val_loss: 0.0014 - lr: 0.0010 - 1392s/epoch - 5s/step
Epoch 3/25

Epoch 00003: val_loss did not improve from 0.00144
300/300 - 1869s - loss: 0.0034 - val_loss: 0.0016 - lr: 0.0010 - 1869s/epoch - 6s/step
Epoch 4/25

Epoch 00004: val_loss improved from 0.00144 to 0.00095, saving model to LSTM_Modelo2_checkpoi
nt.keras
300/300 - 1258s - loss: 0.0032 - val_loss: 9.5484e-04 - lr: 0.0010 - 1258s/epoch - 4s/step
Epoch 5/25

Epoch 00005: val_loss did not improve from 0.00095
300/300 - 1137s - loss: 0.0031 - val_loss: 0.0014 - lr: 0.0010 - 1137s/epoch - 4s/step
Epoch 6/25

Epoch 00006: val_loss did not improve from 0.00095

Epoch 00006: ReduceLROnPlateau reducing learning rate to 0.00010000000474974513.
300/300 - 1066s - loss: 0.0030 - val_loss: 0.0012 - lr: 0.0010 - 1066s/epoch - 4s/step
Epoch 7/25

Epoch 00007: val_loss improved from 0.00095 to 0.00076, saving model to LSTM_Modelo2_checkpoi
nt.keras
300/300 - 1602s - loss: 0.0024 - val_loss: 7.5583e-04 - lr: 1.0000e-04 - 1602s/epoch - 5s/ste
p
Epoch 8/25

Epoch 00008: val_loss improved from 0.00076 to 0.00072, saving model to LSTM_Modelo2_checkpoi
nt.keras
300/300 - 1250s - loss: 0.0022 - val_loss: 7.2403e-04 - lr: 1.0000e-04 - 1250s/epoch - 4s/ste
p
Epoch 9/25

Epoch 00009: val_loss improved from 0.00072 to 0.00064, saving model to LSTM_Modelo2_checkpoi
nt.keras
300/300 - 1361s - loss: 0.0021 - val_loss: 6.4337e-04 - lr: 1.0000e-04 - 1361s/epoch - 5s/ste
p
Epoch 10/25

Epoch 00010: val_loss did not improve from 0.00064
300/300 - 1128s - loss: 0.0019 - val_loss: 6.7014e-04 - lr: 1.0000e-04 - 1128s/epoch - 4s/ste
p
Epoch 11/25

Epoch 00011: val_loss did not improve from 0.00064

Epoch 00011: ReduceLROnPlateau reducing learning rate to 1.0000000474974514e-05.
300/300 - 1079s - loss: 0.0019 - val_loss: 6.8634e-04 - lr: 1.0000e-04 - 1079s/epoch - 4s/ste
p
Epoch 12/25

Epoch 00012: val_loss did not improve from 0.00064
300/300 - 1055s - loss: 0.0019 - val_loss: 7.0692e-04 - lr: 1.0000e-05 - 1055s/epoch - 4s/ste
p
Epoch 13/25

Epoch 00013: val_loss did not improve from 0.00064

Epoch 00013: ReduceLROnPlateau reducing learning rate to 1.0000000656873453e-06.
```

```
300/300 - 1259s - loss: 0.0019 - val_loss: 6.9574e-04 - lr: 1.0000e-05 - 1259s/epoch - 4s/ste
p
Epoch 14/25

Epoch 00014: val_loss did not improve from 0.00064
300/300 - 1052s - loss: 0.0018 - val_loss: 7.0116e-04 - lr: 1.0000e-06 - 1052s/epoch - 4s/ste
p
Epoch 00014: early stopping
CPU times: total: 4h 7min 41s
Wall time: 4h 56min 2s
```

Out[ ]:
```
<keras.callbacks.History at 0x1bbf90a0d00>
```

In [ ]:
```python
#Carregamento dos pesos
try:
    model.load_weights(path_checkpoint)
except Exception as error:
    print("Error trying to load checkpoint.")
    print(error)
```

In [ ]:
```python
#Avaliação da rede com os dados de teste
resultado_modelo2_LSTM = model.evaluate(x=np.expand_dims(x_test_scaled, axis=0),
                        y=np.expand_dims(y_test_scaled, axis=0))
```

```
1/1 [==============================] - 1s 673ms/step - loss: 6.4337e-04
```

In [ ]:
```python
print("MSE (Dados de Teste): %10.3e" % (resultado_modelo2_LSTM))
```

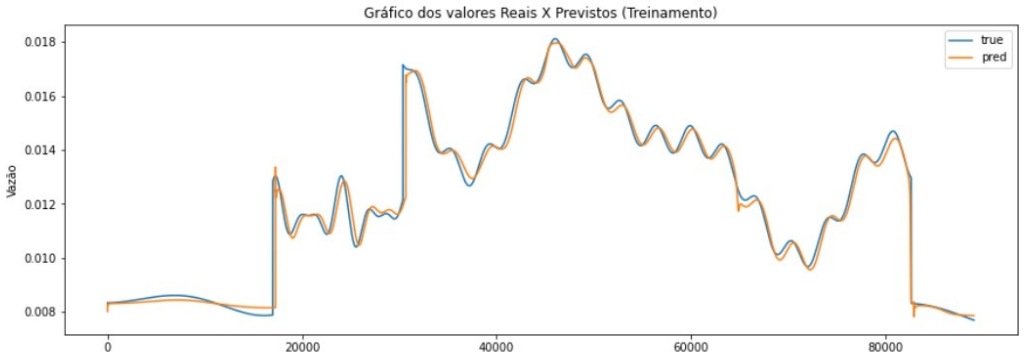
```
MSE (Dados de Teste):  6.434e-04
```

Função para plotar os dados reais e os dados previstos pela rede

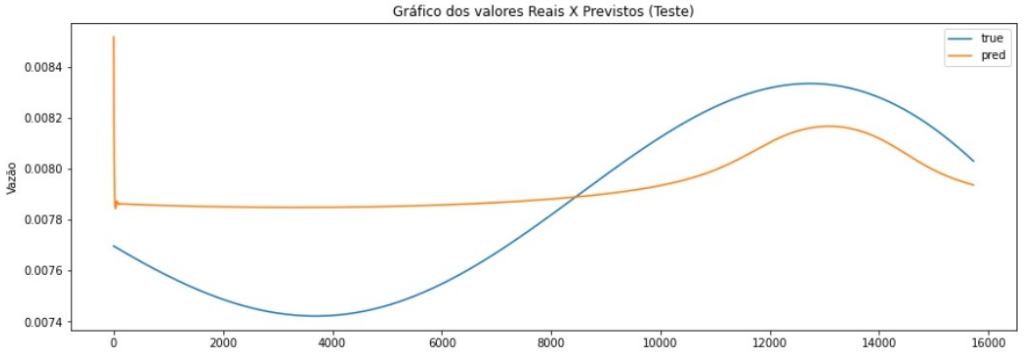
In [ ]:
```python
def plot_comparison(start_idx, length=100, train=True):
    """
    Plot the predicted and true output-signals.

    :param start_idx: Start-index for the time-series.
    :param length: Sequence-length to process and plot.
    :param train: Boolean whether to use training- or test-set.
    """

    if train:
        # Use training-data.
        x = x_train_scaled
        y_true = y_train
        titulo = "Gráfico dos valores Reais X Previstos (Treinamento)"
    else:
        # Use test-data.
        x = x_test_scaled
        y_true = y_test
        titulo = "Gráfico dos valores Reais X Previstos (Teste)"

    # End-index for the sequences.
    end_idx = start_idx + length

    # Select the sequences from the given start-index and
    # of the given length.
    x = x[start_idx:end_idx]
    y_true = y_true[start_idx:end_idx]

    # Input-signals for the model.
    x = np.expand_dims(x, axis=0)

    # Use the model to predict the output-signals.
    y_pred = model.predict(x)

    # The output of the model is between 0 and 1.
    # Do an inverse map to get it back to the scale
    # of the original data-set.
```

```
    y_pred_rescaled = y_scaler.inverse_transform(y_pred[0])

    # For each output-signal.
    for signal in range(len(target_names)):
        # Get the output-signal predicted by the model.
        signal_pred = y_pred_rescaled[:, signal]

        # Get the true output-signal from the data-set.
        signal_true = y_true[:, signal]

        # Make the plotting-canvas bigger.
        plt.figure(figsize=(15,5))

        # Plot and compare the two signals.
        plt.plot(signal_true, label='true')
        plt.plot(signal_pred, label='pred')

        # Plot labels etc.
        plt.ylabel(target_names[signal])
        plt.title(titulo)
        plt.legend()
        plt.show()
```

In [ ]:
```
plot_comparison(start_idx=0, length=num_train, train=True)
```



In [ ]:
```
plot_comparison(start_idx=0, length=num_test, train=False)
```



In [ ]:
```
#Exportação dos dados de treinamento previstos para csv
y_prev_model2LSTM_dados2_train = model.predict(np.expand_dims(x_train_scaled,axis=0))
y_prev_model2LSTM_dados2_train_rescaled = y_scaler.inverse_transform(
    y_prev_model2LSTM_dados2_train[0])
y_prev_model2LSTM_dados2_train_df = pd.DataFrame(y_prev_model2LSTM_dados2_train_rescaled)
filepath = Path(path_dir + 'y_prev_model2LSTM_dados2_train.csv')
filepath.parent.mkdir(parents=True, exist_ok=True)
y_prev_model2LSTM_dados2_train_df.to_csv(filepath)
```

In [ ]:
```
#Exportação dos dados de teste previstos para csv
```

```
y_prev_model2LSTM_dados2_test = model.predict(np.expand_dims(x_test_scaled,axis=0))
y_prev_model2LSTM_dados2_test_rescaled = y_scaler.inverse_transform(
    y_prev_model2LSTM_dados2_test[0])
y_prev_model2LSTM_dados2_test_df = pd.DataFrame(y_prev_model2LSTM_dados2_test_rescaled)
filepath = Path(path_dir + 'y_prev_model2LSTM_dados2_test.csv')
filepath.parent.mkdir(parents=True, exist_ok=True)
y_prev_model2LSTM_dados2_test_df.to_csv(filepath)
```

Avaliação da abrangência do Modelo

In [ ]:
```
x_data1_scaled = x_scaler.fit_transform(x_data)
y_data1_scaled = y_scaler.fit_transform(y_data)
```

In [ ]:
```
resultado_modelo2LSTM_dados1 = model.evaluate(
                                x=np.expand_dims(x_data1_scaled, axis=0),
                                y=np.expand_dims(y_data1_scaled, axis=0))
```

```
1/1 [==============================] - 20s 20s/step - loss: 0.0027
```

In [ ]:
```
print("MSE (Conjunto de dados 1): %10.3e" % (resultado_modelo2LSTM_dados1))
```

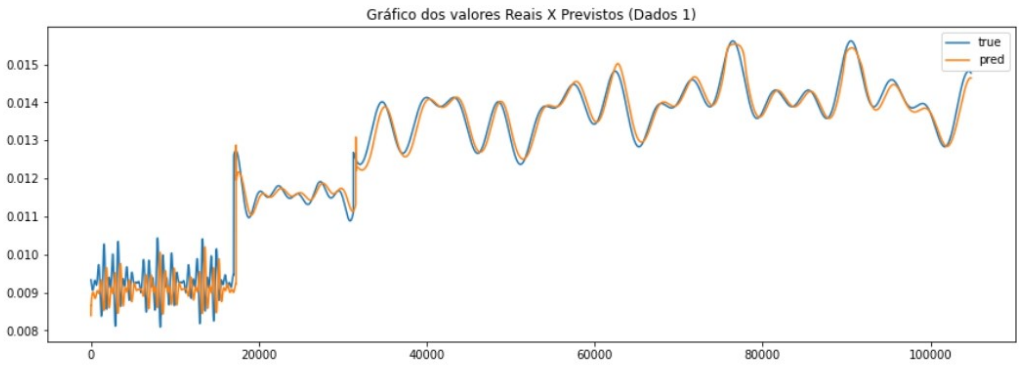
```
MSE (Conjunto de dados 1):  2.655e-03
```

Gráfico dos valores Reais X Previsão do Modelo 2, usando o conjunto de dados 1 como entrada

In [ ]:
```
#Exportação dos dados de abrangência previstos para csv
y_previsao_modelo2LSTM_dados1 = model.predict(np.expand_dims(x_data1_scaled, axis=0))
y_previsao_modelo2LSTM_dados1_rescaled = y_scaler.inverse_transform(
    y_previsao_modelo2LSTM_dados1[0])
y_previsao_modelo2LSTM_dados1_df = pd.DataFrame(y_previsao_modelo2LSTM_dados1_rescaled)
filepath = Path(path_dir + 'y_previsao_modelo2LSTM_dados1.csv')
filepath.parent.mkdir(parents=True, exist_ok=True)
y_previsao_modelo2LSTM_dados1_df.to_csv(filepath)
```

In [ ]:
```
plt.figure(figsize=(15,5))

# Plot and compare the two signals.
plt.plot(y_data, label='true')
plt.plot(y_previsao_modelo2LSTM_dados1_rescaled, label='pred')
plt.title("Gráfico dos valores Reais X Previstos (Dados 1)")
plt.legend()
```

Out[ ]:
```
<matplotlib.legend.Legend at 0x1bdd06533a0>
```



Previsão Retroalimentada

In [ ]:
```
def vector3d_generator(num_test, sequence_length):
    """
    Generator function for creating a 3D vector.
    """
```

```
    # Infinite loop.
    while True:
        # Allocate a new array for the batch of input-signals.
        x_shape = (1, num_test + sequence_length, num_x_signals)
        x_prev = np.zeros(shape=x_shape, dtype=np.float64)

        # Fill the batch with random sequences of data.
        for i in range(num_test + sequence_length):
            if i < sequence_length:
                x_prev[0:1,i] = x_train_scaled[len(x_train_scaled)-sequence_length+i:
                                               len(x_train_scaled)-sequence_length+i+1]
            else:
                x_prev[0:1,i] = x_test_scaled[i-sequence_length:i-sequence_length+1]

        yield (x_prev)
```

In [ ]:
```
vetor_generator = vector3d_generator(num_test=num_test,
                           sequence_length=sequence_length)
```

In [ ]:
```
x_prev = next(vetor_generator)
```

In [ ]:
```
y_previsao_retroalimentada = []
for i in range(0, num_test):
    previsao = model.predict(x_prev[0:,i:i+sequence_length])
    y_previsao_retroalimentada.append(previsao[0,-1,0])
    x_prev[0:,i+sequence_length,0]=previsao[0,-1,0]
```

In [ ]:
```
x_prev_rescaled = x_scaler.inverse_transform(x_prev[0])
```

In [ ]:
```
#Exportação dos dados da previsão retroalimentada para csv
y_previsao_retroalimentada_rescaled = y_scaler.inverse_transform(
    np.expand_dims(y_previsao_retroalimentada,axis=1))
y_previsao_retroalimentada_df = pd.DataFrame(y_previsao_retroalimentada_rescaled)
filepath = Path(path_dir + 'y_prev_retroalimentada_model2LSTM.csv')
filepath.parent.mkdir(parents=True, exist_ok=True)
y_previsao_retroalimentada_df.to_csv(filepath)
```

Avaliação da Previsão Retroalimentada

In [ ]:
```
resultado_previsao_retroalimentada = model.evaluate(
x=np.expand_dims(x_prev_rescaled[1440:], axis=0),
y=np.expand_dims(y_previsao_retroalimentada_rescaled, axis=0))
```

```
1/1 [==============================] - 1s 680ms/step - loss: 5.7538e-05
```

In [ ]:
```
print("MSE (Previsão Retroalimentada): %10.3e" % (resultado_previsao_retroalimentada))
```

```
MSE (Previsão Retroalimentada):  5.754e-05
```

Gráfico dos valores Reais X Previstos, utilizando a previsão da vazão como entrada no índice i+1

In [ ]:
```
plt.figure(figsize=(15,5))

# Plot and compare the two signals.
plt.plot(y_test, label='true')
plt.plot(y_previsao_retroalimentada_rescaled, label='pred')
plt.title("Gráfico dos valores Reais X Previstos (Previsão Retroalimentada)")
plt.legend()
```

Out[ ]:
```
<matplotlib.legend.Legend at 0x1bdd8a3d100>
```

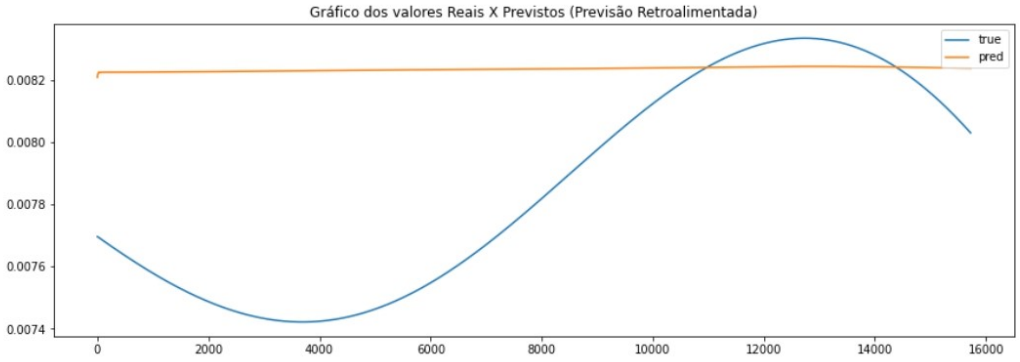


Gráfico dos valores Reais X Previstos (Previsão Retroalimentada)
true
pred
0.0082
0.0080
0.0078
0.0076
0.0074
0
2000
4000
6000
8000
10000
12000
14000
16000

APÊNDICE D – Código do Modelo GRU 1

In [ ]:
```
#Import das Bibliotecas Necessárias
%matplotlib inline
import matplotlib.pyplot as plt
import tensorflow as tf
import numpy as np
import pandas as pd
import os
from pathlib import Path
from sklearn.preprocessing import MinMaxScaler

from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, GRU, LSTM
from tensorflow.keras.optimizers import RMSprop
from tensorflow.keras.callbacks import EarlyStopping, ModelCheckpoint
from tensorflow.keras.callbacks import TensorBoard, ReduceLROnPlateau
from tensorflow.keras.backend import square, mean
```

In [ ]:
```
#Versões das Bibliotecas Principais
print("Versão Tensorflow:",tf.__version__)
print("Versão Keras:",tf.keras.__version__)
print("Versão Pandas:",pd.__version__)
print("Versão Numpy:",np.__version__)
```

```
Versão Tensorflow: 2.7.0
Versão Keras: 2.7.0
Versão Pandas: 1.3.5
Versão Numpy: 1.22.1
```

Criação dos DataFrames a partir dos conjuntos de dados

In [ ]:
```
path_dir = "C:/Users/Rafael/Desktop/hello/" #Diretório do arquivo
df = pd.read_csv(path_dir + "Simulador_vazao1_resampled.csv",sep=';',header=0,
                 names=["Tempo","Vazão","Temperatura","Pressão"],
                 index_col="Tempo")
df.head(n=5) #Visualização do cabeçalho
```

Out[ ]:

Tempo	Vazão	Temperatura	Pressão
01/01/2021 00:00	0.009200	333.993896	48670544.52
01/01/2021 00:05	0.009259	333.993198	48424272.55
01/01/2021 00:10	0.009259	333.994561	48367386.80
01/01/2021 00:15	0.009259	333.995761	48328308.56
01/01/2021 00:20	0.009259	333.996818	48298087.55

In [ ]:
```
df2 = pd.read_csv(path_dir + "Simulador_vazao2_resampled.csv",sep=';',header=0,
                 names=["Tempo","Vazão","Temperatura","Pressão"],
                 index_col="Tempo")
df2.head(n=5) #Visualização do cabeçalho
```

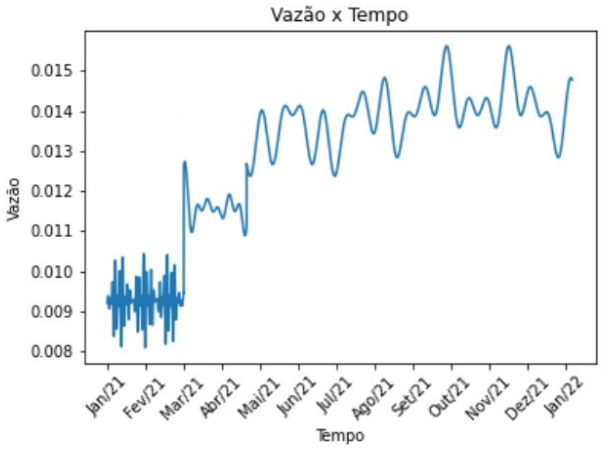
Out[ ]:

Tempo	Vazão	Temperatura	Pressão
01/01/2022 00:00	0.008280	333.994414	48706532.88
01/01/2022 00:05	0.008333	333.993486	48485096.35
01/01/2022 00:10	0.008333	333.994575	48433867.60
01/01/2022 00:15	0.008333	333.995567	48398652.38
01/01/2022 00:20	0.008333	333.996456	48371396.34

Plot da Vazão X Tempo

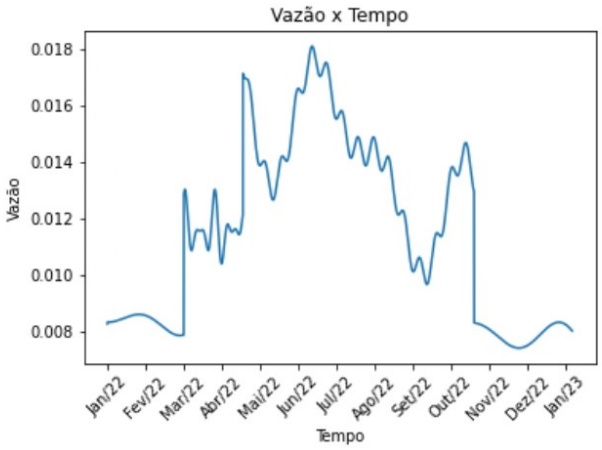
In [ ]:
```
ax1 = df['Vazão'].plot()
ax1.set_xticks([288*30*i for i in range(13)],["Jan/21", "Fev/21", "Mar/21",
            "Abr/21", "Mai/21", "Jun/21", "Jul/21", "Ago/21", "Set/21", "Out/21",
            "Nov/21", "Dez/21", "Jan/22"])
plt.xticks(rotation=45)
plt.ylabel("Vazão")
plt.title("Vazão x Tempo")
```

Out[ ]:
```
Text(0.5, 1.0, 'Vazão x Tempo')
```



In [ ]:
```
ax2 = df2['Vazão'].plot()
ax2.set_xticks([288*30*i for i in range(13)],["Jan/22", "Fev/22", "Mar/22",
            "Abr/22", "Mai/22", "Jun/22", "Jul/22", "Ago/22", "Set/22", "Out/22",
            "Nov/22", "Dez/22", "Jan/23"])
plt.xticks(rotation=45)
plt.ylabel("Vazão")
plt.title("Vazão x Tempo")
```

Out[ ]:
```
Text(0.5, 1.0, 'Vazão x Tempo')
```



In [ ]:
```
#Formato dos DataFrames
print("Formato DataFrame 1: ",df.values.shape)
print("Formato DataFrame 2: ",df2.values.shape)
```

```
Formato DataFrame 1:  (105121, 3)
Formato DataFrame 2:  (105121, 3)
```

In [ ]:
```
#Parâmetro Alvo
target_names = ["Vazão"]
```

In [ ]:
```
#Número de passos deslocados
shift_steps = 288 #Equivale a 1 dia
```

Criação dos DataFrame Targets, deslocando do parâmetro escolhido o número de passos

In [ ]:
```
df_targets = df[target_names].shift(-shift_steps)
df_targets2 = df2[target_names].shift(-shift_steps)
```

In [ ]:
```
df[target_names].tail(shift_steps+5)
```

Out[ ]:

	Vazão
Tempo	
30/12/2021 23:40	0.014823
30/12/2021 23:45	0.014823
30/12/2021 23:50	0.014823
30/12/2021 23:55	0.014823
31/12/2021 00:00	0.014823
...	...
31/12/2021 23:40	0.014776
31/12/2021 23:45	0.014776
31/12/2021 23:50	0.014775
31/12/2021 23:55	0.014775
01/01/2022 00:00	0.014775

293 rows × 1 columns

In [ ]:
```
df_targets.tail(shift_steps + 5)
```

Out[ ]:

Tempo	Vazão
30/12/2021 23:40	0.014776
30/12/2021 23:45	0.014776
30/12/2021 23:50	0.014775
30/12/2021 23:55	0.014775
31/12/2021 00:00	0.014775
...	...
31/12/2021 23:40	NaN
31/12/2021 23:45	NaN
31/12/2021 23:50	NaN
31/12/2021 23:55	NaN
01/01/2022 00:00	NaN

293 rows × 1 columns

In [ ]:
```
df2[target_names].tail(shift_steps+5)
```

Out[ ]:

Tempo	Vazão
30/12/2022 23:40	0.008085
30/12/2022 23:45	0.008085
30/12/2022 23:50	0.008085
30/12/2022 23:55	0.008085
31/12/2022 00:00	0.008084
...	...
31/12/2022 23:40	0.008030
31/12/2022 23:45	0.008030
31/12/2022 23:50	0.008030
31/12/2022 23:55	0.008030
01/01/2023 00:00	0.008030

293 rows × 1 columns

In [ ]:
```
df_targets2.tail(shift_steps + 5)
```

Out[ ]:

Tempo	Vazão
30/12/2022 23:40	0.00803
30/12/2022 23:45	0.00803
30/12/2022 23:50	0.00803
30/12/2022 23:55	0.00803
31/12/2022 00:00	0.00803
...	...
31/12/2022 23:40	NaN
31/12/2022 23:45	NaN
31/12/2022 23:50	NaN
31/12/2022 23:55	NaN
01/01/2023 00:00	NaN

293 rows × 1 columns

Definição dos vetores de entrada

In [ ]:
```
x_data = df.values[0:-shift_steps]
x_data2 = df2.values[0:-shift_steps]
```

In [ ]:
```
print(type(x_data))
print("Formato inputs 1:", x_data.shape)
print(type(x_data2))
print("Formato inputs 2:", x_data2.shape)
```

```
<class 'numpy.ndarray'>
Formato inputs 1: (104833, 3)
<class 'numpy.ndarray'>
Formato inputs 2: (104833, 3)
```

Definição dos vetores de saída

In [ ]:
```
y_data = df_targets.values[:-shift_steps]
y_data2 = df_targets2.values[:-shift_steps]
```

In [ ]:
```
print(type(y_data))
print("Formato outputs 1:", y_data.shape)
print(type(y_data2))
print("Formato outputs 2:", y_data2.shape)
```

```
<class 'numpy.ndarray'>
Formato outputs 1: (104833, 1)
<class 'numpy.ndarray'>
Formato outputs 2: (104833, 1)
```

Definição da quantidade de dados de treinamento e teste

In [ ]:
```
num_data = len(x_data)
train_split = 0.85
num_train = int(train_split * num_data)
print("Número de dados de treinamento:",num_train)
```

```
Número de dados de treinamento: 89108
```

In [ ]:
```
num_test = num_data - num_train
print("Número de dados de teste:",num_test)
```

```
Número de dados de teste: 15725
```

Separação da entrada nos dados de treinamento e teste

In [ ]:
```
x_train = x_data[0:num_train]
x_test = x_data[num_train:]
```

Separação da saída nos dados de treinamento e teste

In [ ]:
```
y_train = y_data[0:num_train]
y_test = y_data[num_train:]
```

In [ ]:
```
#Quantidade de parâmetros de entrada
num_x_signals = x_data.shape[1]
print("Parâmetros de entrada:",num_x_signals)
```

```
Parâmetros de entrada: 3
```

In [ ]:
```
#Quantidade de parâmetros de saída
num_y_signals = y_data.shape[1]
print("Parâmetros de saída:",num_y_signals)
```

```
Parâmetros de saída: 1
```

In [ ]:
```
#Valores mínimo e máximo dos dados de treinamento
print("Min:", np.min(x_train))
print("Max:", np.max(x_train))
```

```
Min: 0.00808754
Max: 48670544.52
```

In [ ]:
```
#Método para escalar os dados
x_scaler = MinMaxScaler()
```

In [ ]:
```
#Variável com os dados de entrada de treinamento escalados entre [0,1]
x_train_scaled = x_scaler.fit_transform(x_train)
print("Min:", np.min(x_train_scaled))
print("Max:", np.max(x_train_scaled))
```

```
Min: 0.0
Max: 1.0000000000000018
```

In [ ]:
```
#Variável com os dados de entrada de teste escalados entre [0,1]
x_test_scaled = x_scaler.transform(x_test)
```

In [ ]:
```
y_scaler = MinMaxScaler()
#Variável com os dados de saída de treinamento escalados entre [0,1]
y_train_scaled = y_scaler.fit_transform(y_train)

#Variável com os dados de saída de teste escalados entre [0,1]
y_test_scaled = y_scaler.transform(y_test)
```

Função para criação de batchs de treinamento, utilizando pontos iniciais aleatórios

In [ ]:
```
def batch_generator(batch_size, sequence_length):
    """
    Generator function for creating random batches of training-data.
    """

    # Infinite loop.
    while True:
        # Allocate a new array for the batch of input-signals.
        x_shape = (batch_size, sequence_length, num_x_signals)
        x_batch = np.zeros(shape=x_shape, dtype=np.float64)

        # Allocate a new array for the batch of output-signals.
        y_shape = (batch_size, sequence_length, num_y_signals)
```

```
        y_batch = np.zeros(shape=y_shape, dtype=np.float64)

        # Fill the batch with random sequences of data.
        for i in range(batch_size):
            # Get a random start-index.
            # This points somewhere into the training-data.
            idx = np.random.randint(num_train - sequence_length)

            # Copy the sequences of data starting at this index.
            x_batch[i] = x_train_scaled[idx:idx+sequence_length]
            y_batch[i] = y_train_scaled[idx:idx+sequence_length]

        yield (x_batch, y_batch)
```

In [ ]:
```
#Quantidade de batchs
batch_size = 150
```

In [ ]:
```
#Tamanho da sequência de dados
sequence_length = 288 * 5 #Corresponde a 5 dias
print("Tamanho da sequência:",sequence_length)
```

```
Tamanho da sequência: 1440
```

In [ ]:
```
generator = batch_generator(batch_size=batch_size,
                            sequence_length=sequence_length)
```

In [ ]:
```
#Criação dos batchs
x_batch, y_batch = next(generator)
```

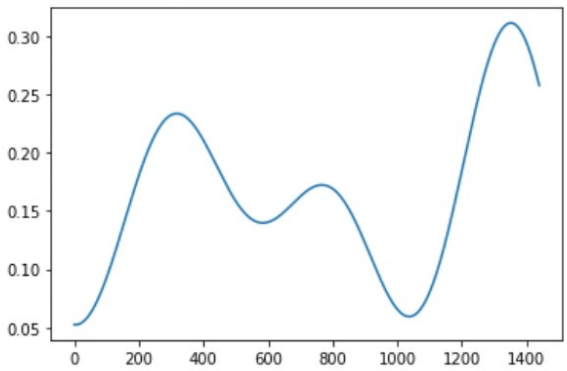
In [ ]:
```
print("Formato do x_batch:",x_batch.shape)
print("Formato do y_batch:",y_batch.shape)
```

```
Formato do x_batch: (150, 1440, 3)
Formato do y_batch: (150, 1440, 1)
```

Visualização da vazão, para a sequência do 1º batch dos dados de entrada

In [ ]:
```
batch = 0   # First sequence in the batch.
signal = 0  # First signal from the 3 input-signals.
seq = x_batch[batch, :, signal]
plt.plot(seq)
```

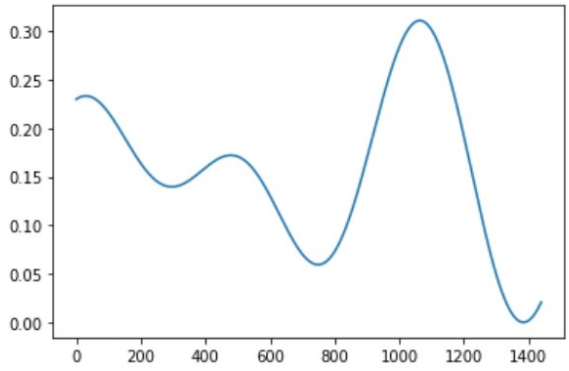
Out[ ]:
```
[<matplotlib.lines.Line2D at 0x16206dbb190>]
```



Visualização da vazão, para a sequência do 1º batch dos dados de saída

In [ ]:
```
seq = y_batch[batch, :, signal]
plt.plot(seq)
```

Out[ ]:
```
[<matplotlib.lines.Line2D at 0x16205c0cb50>]
```



In [ ]:
```
#Definição dos dados de validação
validation_data = (np.expand_dims(x_test_scaled, axis=0),
                   np.expand_dims(y_test_scaled, axis=0))
```

Criação da RNN do tipo GRU

In [ ]:
```
model = Sequential()
```

In [ ]:
```
#1ª camada da rede - GRU
model.add(GRU(units=375,
              return_sequences=True,
              input_shape=(None, num_x_signals,)))
```

In [ ]:
```
#2ª camada da rede - GRU
model.add(GRU(units=375,
              return_sequences=True))
```

In [ ]:
```
#3ª camada da rede - Densa - Função de ativação: sigmoidal
model.add(Dense(200, activation='sigmoid'))
```

In [ ]:
```
#4ª camada da rede - Densa - Função de ativação: sigmoidal
model.add(Dense(num_y_signals, activation='sigmoid'))
```

In [ ]:
```
#Otimizador
optimizer = RMSprop(learning_rate=1e-3)
```

In [ ]:
```
#Compilação do modelo
model.compile(loss="mse", optimizer=optimizer)
```

In [ ]:
```
#Visualização do sumário da rede, com as camadas,
# número de unidades e parâmetros treináveis
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 gru (GRU)                   (None, None, 375)         427500

 gru_1 (GRU)                 (None, None, 375)         846000

 dense (Dense)               (None, None, 200)         75200

 dense_1 (Dense)             (None, None, 1)           201

=================================================================
Total params: 1,348,901
Trainable params: 1,348,901
Non-trainable params: 0
_________________________________________________________________
```

In [ ]:
```
#Checkpoint para salvamento dos pesos
path_checkpoint = 'GRU_Modelo1_checkpoint.keras'
callback_checkpoint = ModelCheckpoint(filepath=path_checkpoint,
                                      monitor='val_loss',
                                      verbose=1,
                                      save_weights_only=True,
                                      save_best_only=True)
```

In [ ]:
```
#Método para interrupção antecipada do treinamento, caso a rede pare de aprender
callback_early_stopping = EarlyStopping(monitor='val_loss',
                                        patience=5, verbose=1)
```

In [ ]:
```
callback_tensorboard = TensorBoard(log_dir='./GRU_Modelo1_logs/',
                                   histogram_freq=0,
                                   write_graph=False)
```

In [ ]:
```
#Método para redução da taxa de aprendizado
callback_reduce_lr = ReduceLROnPlateau(monitor='val_loss',
                                       factor=0.1,
                                       min_lr=1e-6,
                                       patience=2,
                                       verbose=1)
```

In [ ]:
```
callbacks = [callback_early_stopping,
             callback_checkpoint,
             callback_tensorboard,
             callback_reduce_lr]
```

Treinamento da RNN

In [ ]:
```
%%time
model.fit(x=generator,
          epochs=25,
          steps_per_epoch=300,
          validation_data=validation_data,
          verbose=2,
          callbacks=callbacks)
```

```
Epoch 1/25

Epoch 00001: val_loss improved from inf to 0.00238, saving model to GRU_Modelo1_checkpoint.ke
ras
300/300 - 250s - loss: 0.0056 - val_loss: 0.0024 - lr: 0.0010 - 250s/epoch - 833ms/step
Epoch 2/25

Epoch 00002: val_loss improved from 0.00238 to 0.00130, saving model to GRU_Modelo1_checkpoin
t.keras
300/300 - 246s - loss: 0.0035 - val_loss: 0.0013 - lr: 0.0010 - 246s/epoch - 821ms/step
Epoch 3/25

Epoch 00003: val_loss did not improve from 0.00130
300/300 - 250s - loss: 0.0027 - val_loss: 0.0017 - lr: 0.0010 - 250s/epoch - 833ms/step
Epoch 4/25

Epoch 00004: val_loss improved from 0.00130 to 0.00104, saving model to GRU_Modelo1_checkpoin
t.keras
300/300 - 252s - loss: 0.0022 - val_loss: 0.0010 - lr: 0.0010 - 252s/epoch - 839ms/step
Epoch 5/25

Epoch 00005: val_loss did not improve from 0.00104
300/300 - 253s - loss: 0.0021 - val_loss: 0.0023 - lr: 0.0010 - 253s/epoch - 844ms/step
Epoch 6/25

Epoch 00006: val_loss improved from 0.00104 to 0.00077, saving model to GRU_Modelo1_checkpoin
t.keras
300/300 - 254s - loss: 0.0021 - val_loss: 7.7257e-04 - lr: 0.0010 - 254s/epoch - 848ms/step
Epoch 7/25

Epoch 00007: val_loss did not improve from 0.00077
300/300 - 255s - loss: 0.0020 - val_loss: 8.9969e-04 - lr: 0.0010 - 255s/epoch - 849ms/step
Epoch 8/25

Epoch 00008: val_loss did not improve from 0.00077

Epoch 00008: ReduceLROnPlateau reducing learning rate to 0.00010000000474974513.
300/300 - 255s - loss: 0.0026 - val_loss: 0.0010 - lr: 0.0010 - 255s/epoch - 851ms/step
Epoch 9/25

Epoch 00009: val_loss did not improve from 0.00077
300/300 - 254s - loss: 0.0027 - val_loss: 9.5126e-04 - lr: 1.0000e-04 - 254s/epoch - 848ms/st
ep
Epoch 10/25

Epoch 00010: val_loss did not improve from 0.00077

Epoch 00010: ReduceLROnPlateau reducing learning rate to 1.0000000474974514e-05.
300/300 - 255s - loss: 0.0025 - val_loss: 9.3427e-04 - lr: 1.0000e-04 - 255s/epoch - 849ms/st
ep
Epoch 11/25

Epoch 00011: val_loss did not improve from 0.00077
300/300 - 255s - loss: 0.0025 - val_loss: 9.3698e-04 - lr: 1.0000e-05 - 255s/epoch - 851ms/st
ep
Epoch 00011: early stopping
CPU times: total: 32min 22s
Wall time: 46min 19s
```

Out[ ]: <keras.callbacks.History at 0x1e77853fc10>

In [ ]:
```python
#Carregamento dos pesos
try:
    model.load_weights(path_checkpoint)
except Exception as error:
    print("Error trying to load checkpoint.")
    print(error)
```

In [ ]:
```
#Avaliação da rede com os dados de teste
resultado_modelo1 = model.evaluate(x=np.expand_dims(x_test_scaled, axis=0),
                        y=np.expand_dims(y_test_scaled, axis=0))
```

```
1/1 [==============================] - 7s 7s/step - loss: 7.7257e-04
```

In [ ]:
```
print("MSE (Dados de Teste): %10.3e" % (resultado_modelo1))
```

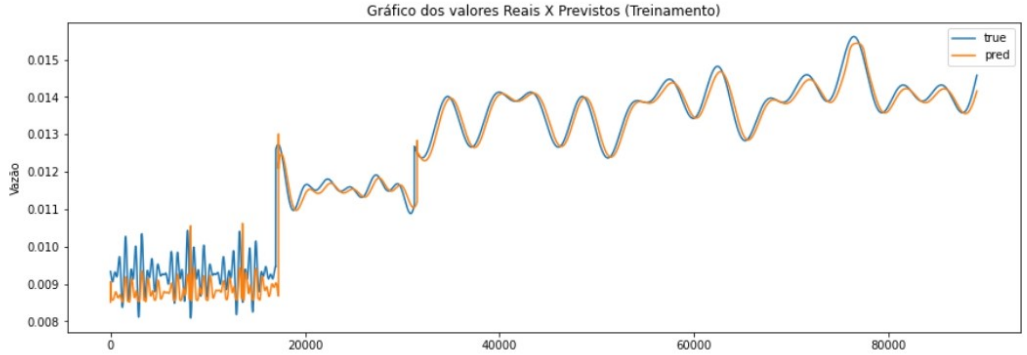
```
MSE (Dados de Teste):  7.726e-04
```

Função para plotar os dados reais e os dados previstos pela rede

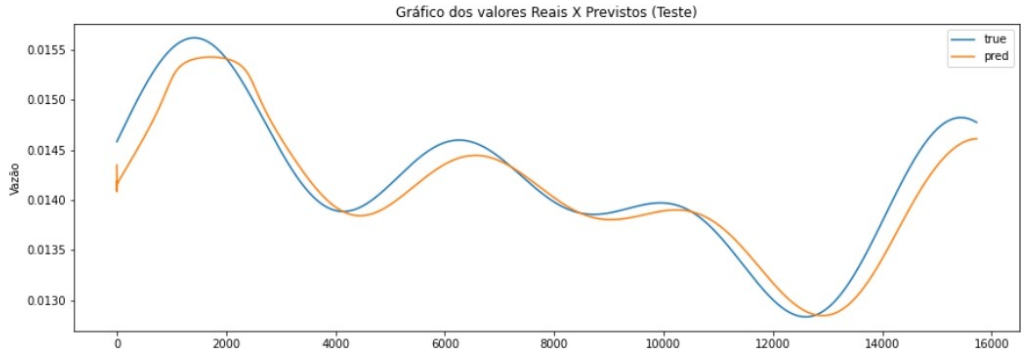
In [ ]:
```
def plot_comparison(start_idx, length=100, train=True):
    """
    Plot the predicted and true output-signals.

    :param start_idx: Start-index for the time-series.
    :param length: Sequence-length to process and plot.
    :param train: Boolean whether to use training- or test-set.
    """

    if train:
        # Use training-data.
        x = x_train_scaled
        y_true = y_train
        titulo = "Gráfico dos valores Reais X Previstos (Treinamento)"
    else:
        # Use test-data.
        x = x_test_scaled
        y_true = y_test
        titulo = "Gráfico dos valores Reais X Previstos (Teste)"

    # End-index for the sequences.
    end_idx = start_idx + length

    # Select the sequences from the given start-index and
    # of the given length.
    x = x[start_idx:end_idx]
    y_true = y_true[start_idx:end_idx]

    # Input-signals for the model.
    x = np.expand_dims(x, axis=0)

    # Use the model to predict the output-signals.
    y_pred = model.predict(x)

    # The output of the model is between 0 and 1.
    # Do an inverse map to get it back to the scale
    # of the original data-set.
    y_pred_rescaled = y_scaler.inverse_transform(y_pred[0])

    # For each output-signal.
    for signal in range(len(target_names)):
        # Get the output-signal predicted by the model.
        signal_pred = y_pred_rescaled[:, signal]

        # Get the true output-signal from the data-set.
        signal_true = y_true[:, signal]

        # Make the plotting-canvas bigger.
        plt.figure(figsize=(15,5))

        # Plot and compare the two signals.
        plt.plot(signal_true, label='true')
        plt.plot(signal_pred, label='pred')

        # Plot labels etc.
        plt.ylabel(target_names[signal])
        plt.title(titulo)
```

```
        plt.legend()
        plt.show()
```

In [ ]:
```
plot_comparison(start_idx=0, length=num_train, train=True)
```



In [ ]:
```
plot_comparison(start_idx=0, length=num_test, train=False)
```



In [ ]:
```
#Exportação dos dados de treinamento previstos para csv
y_prev_model1GRU_dados1_train = model.predict(np.expand_dims(x_train_scaled,axis=0))
y_prev_model1GRU_dados1_train_rescaled = y_scaler.inverse_transform(
    y_prev_model1GRU_dados1_train[0])
y_prev_model1GRU_dados1_train_df = pd.DataFrame(y_prev_model1GRU_dados1_train_rescaled)
filepath = Path(path_dir + 'y_prev_model1GRU_dados1_train.csv')
filepath.parent.mkdir(parents=True, exist_ok=True)
y_prev_model1GRU_dados1_train_df.to_csv(filepath)
```

In [ ]:
```
#Exportação dos dados de teste previstos para csv
y_prev_model1GRU_dados1_test = model.predict(np.expand_dims(x_test_scaled,axis=0))
y_prev_model1GRU_dados1_test_rescaled = y_scaler.inverse_transform(
    y_prev_model1GRU_dados1_test[0])
y_prev_model1GRU_dados1_test_df = pd.DataFrame(y_prev_model1GRU_dados1_test_rescaled)
filepath = Path(path_dir + 'y_prev_model1GRU_dados1_test.csv')
filepath.parent.mkdir(parents=True, exist_ok=True)
y_prev_model1GRU_dados1_test_df.to_csv(filepath)
```

Avaliação da abrangência do Modelo

In [ ]:
```
x_data2_scaled = x_scaler.fit_transform(x_data2)
y_data2_scaled = y_scaler.fit_transform(y_data2)
```

In [ ]:
```
resultado_modelo1_dados2 = model.evaluate(x=np.expand_dims(x_data2_scaled, axis=0),
                        y=np.expand_dims(y_data2_scaled, axis=0))
```

```
1/1 [==============================] - 7s 7s/step - loss: 0.0115
```

In [ ]:
```
print("MSE (Conjunto de dados 2): %10.3e" % (resultado_modelo1_dados2))
```

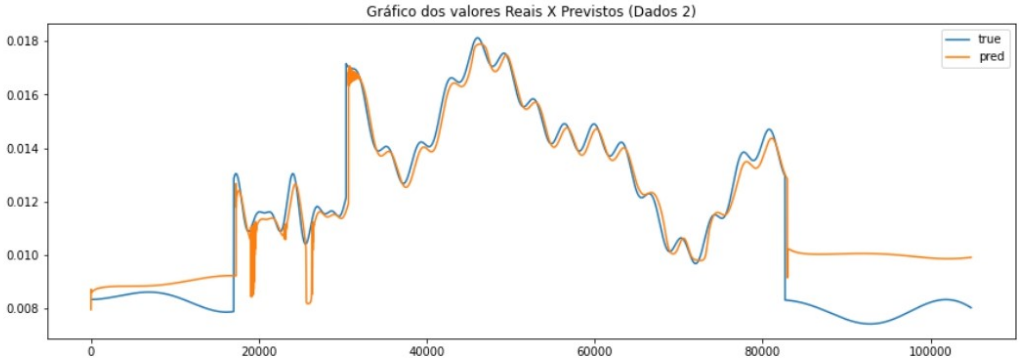
```
MSE (Conjunto de dados 2):  1.146e-02
```

Gráfico dos valores Reais X Previsão do Modelo 1, usando o conjunto de dados 2 como entrada

In [ ]:
```
#Exportação dos dados de abrangência previstos para csv
y_previsao_modelo1_dados2 = model.predict(np.expand_dims(x_data2_scaled, axis=0))
y_previsao_modelo1_dados2_rescaled = y_scaler.inverse_transform(y_previsao_modelo1_dados2[0])
y_previsao_modelo1GRU_dados2_df = pd.DataFrame(y_previsao_modelo1_dados2_rescaled)
filepath = Path(path_dir + 'y_previsao_modelo1GRU_dados2.csv')
filepath.parent.mkdir(parents=True, exist_ok=True)
y_previsao_modelo1GRU_dados2_df.to_csv(filepath)
```

In [ ]:
```
plt.figure(figsize=(15,5))

# Plot and compare the two signals.
plt.plot(y_data2, label='true')
plt.plot(y_previsao_modelo1_dados2_rescaled, label='pred')
plt.title("Gráfico dos valores Reais X Previstos (Dados 2)")
plt.legend()
```

Out[ ]:
```
<matplotlib.legend.Legend at 0x1622e3cfbe0>
```



Previsão Retroalimentada

In [ ]:
```
def vector3d_generator(num_test, sequence_length):
    """
    Generator function for creating a 3D vector.
    """

    # Infinite loop.
    while True:
        # Allocate a new array for the batch of input-signals.
        x_shape = (1, num_test + sequence_length, num_x_signals)
        x_prev = np.zeros(shape=x_shape, dtype=np.float64)

        # Fill the batch with random sequences of data.
        for i in range(num_test + sequence_length):
            if i < sequence_length:
                x_prev[0:1,i] = x_train_scaled[len(x_train_scaled)-sequence_length+i:
                                               len(x_train_scaled)-sequence_length+i+1]
            else:
                x_prev[0:1,i] = x_test_scaled[i-sequence_length:i-sequence_length+1]

        yield (x_prev)
```

In [ ]:
```
vetor_generator = vector3d_generator(num_test=num_test,
                            sequence_length=sequence_length)
```

In [ ]:
```
x_prev = next(vetor_generator)
```

In [ ]:
```
y_previsao_retroalimentada = []
for i in range(0, num_test):
    previsao = model.predict(x_prev[0:,i:i+sequence_length])
    y_previsao_retroalimentada.append(previsao[0,-1,0])
    x_prev[0:,i+sequence_length,0]=previsao[0,-1,0]
```

In [ ]:
```
x_prev_rescaled = x_scaler.inverse_transform(x_prev[0])
```

In [ ]:
```
#Exportação dos dados da previsão retroalimentada para csv
y_previsao_retroalimentada_rescaled = y_scaler.inverse_transform(
    np.expand_dims(y_previsao_retroalimentada,axis=1))
y_previsao_retroalimentada_df = pd.DataFrame(y_previsao_retroalimentada_rescaled)
filepath = Path(path_dir + 'y_prev_retroalimentada_model1GRU.csv')
filepath.parent.mkdir(parents=True, exist_ok=True)
y_previsao_retroalimentada_df.to_csv(filepath)
```

Avaliação da Previsão Retroalimentada

In [ ]:
```
resultado_previsao_retroalimentada = model.evaluate(
    x=np.expand_dims(x_prev_rescaled[1440:], axis=0),
y=np.expand_dims(y_previsao_retroalimentada_rescaled, axis=0))
```

```
1/1 [==============================] - 1s 1s/step - loss: 1.8037e-04
```

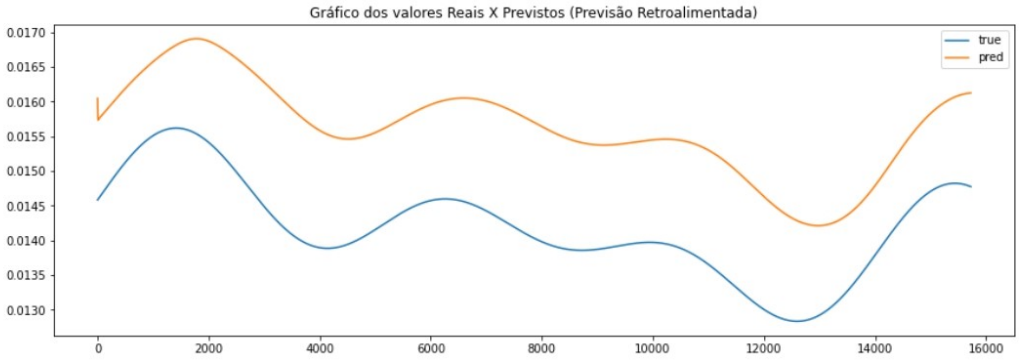
In [ ]:
```
print("MSE (Previsão Retroalimentada): %10.3e" % (resultado_previsao_retroalimentada))
```

```
MSE (Previsão Retroalimentada):  1.804e-04
```

Gráfico dos valores Reais X Previstos, utilizando a previsão da vazão como entrada no índice i+1

In [ ]:
```
plt.figure(figsize=(15,5))

# Plot and compare the two signals.
plt.plot(y_test, label='true')
plt.plot(y_previsao_retroalimentada_rescaled, label='pred')
plt.title("Gráfico dos valores Reais X Previstos (Previsão Retroalimentada)")
plt.legend()
```

Out[ ]:
```
<matplotlib.legend.Legend at 0x16353ca1550>
```



APÊNDICE E – Código do Modelo GRU 2

In [ ]:
```
#Import das Bibliotecas Necessárias
%matplotlib inline
import matplotlib.pyplot as plt
import tensorflow as tf
import numpy as np
import pandas as pd
import os
from pathlib import Path
from sklearn.preprocessing import MinMaxScaler

from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, GRU, LSTM
from tensorflow.keras.optimizers import RMSprop
from tensorflow.keras.callbacks import EarlyStopping, ModelCheckpoint
from tensorflow.keras.callbacks import TensorBoard, ReduceLROnPlateau
from tensorflow.keras.backend import square, mean
```

In [ ]:
```
#Versões das Bibliotecas Principais
print("Versão Tensorflow:",tf.__version__)
print("Versão Keras:",tf.keras.__version__)
print("Versão Pandas:",pd.__version__)
print("Versão Numpy:",np.__version__)
```

```
Versão Tensorflow: 2.7.0
Versão Keras: 2.7.0
Versão Pandas: 1.3.5
Versão Numpy: 1.22.1
```

Criação dos DataFrames a partir dos conjuntos de dados

In [ ]:
```
path_dir = "C:/Users/Rafael/Desktop/hello/" #Diretório do arquivo
df = pd.read_csv(path_dir + "Simulador_vazao1_resampled.csv",sep=';',header=0,
                 names=["Tempo","Vazão","Temperatura","Pressão"],
                 index_col="Tempo")
df.head(n=5) #Visualização do cabeçalho
```

Out[ ]:

Tempo	Vazão	Temperatura	Pressão
01/01/2021 00:00	0.009200	333.993896	48670544.52
01/01/2021 00:05	0.009259	333.993198	48424272.55
01/01/2021 00:10	0.009259	333.994561	48367386.80
01/01/2021 00:15	0.009259	333.995761	48328308.56
01/01/2021 00:20	0.009259	333.996818	48298087.55

In [ ]:
```
df2 = pd.read_csv(path_dir + "Simulador_vazao2_resampled.csv",sep=';',header=0,
                 names=["Tempo","Vazão","Temperatura","Pressão"],
                 index_col="Tempo")
df2.head(n=5) #Visualização do cabeçalho
```

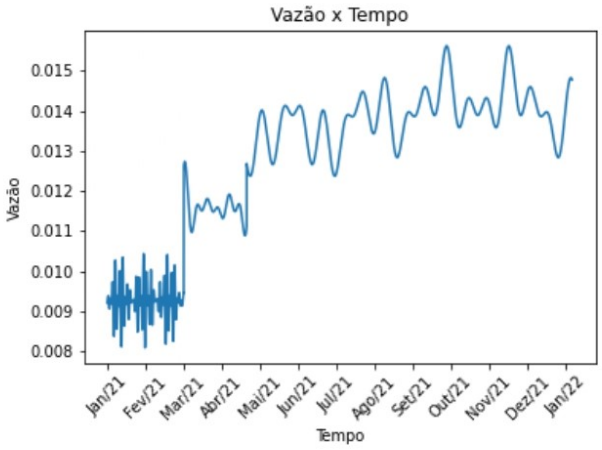
Out[ ]:

Tempo	Vazão	Temperatura	Pressão
01/01/2022 00:00	0.008280	333.994414	48706532.88
01/01/2022 00:05	0.008333	333.993486	48485096.35
01/01/2022 00:10	0.008333	333.994575	48433867.60
01/01/2022 00:15	0.008333	333.995567	48398652.38
01/01/2022 00:20	0.008333	333.996456	48371396.34

Plot da Vazão X Tempo

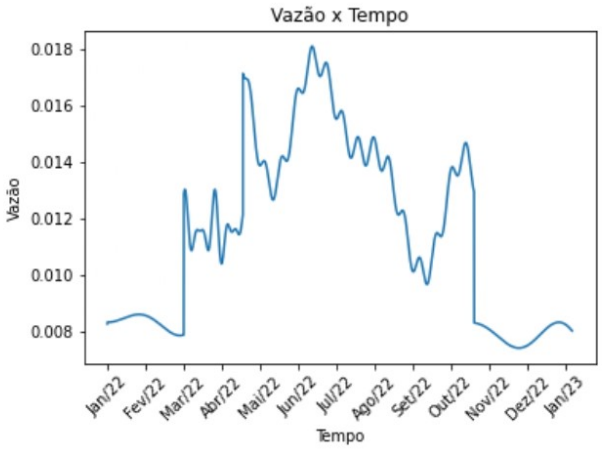
In [ ]:
```
ax1 = df['Vazão'].plot()
ax1.set_xticks([288*30*i for i in range(13)],["Jan/21", "Fev/21", "Mar/21",
            "Abr/21", "Mai/21", "Jun/21", "Jul/21", "Ago/21", "Set/21", "Out/21",
            "Nov/21", "Dez/21", "Jan/22"])
plt.xticks(rotation=45)
plt.ylabel("Vazão")
plt.title("Vazão x Tempo")
```

Out[ ]:
```
Text(0.5, 1.0, 'Vazão x Tempo')
```



In [ ]:
```
ax2 = df2['Vazão'].plot()
ax2.set_xticks([288*30*i for i in range(13)],["Jan/22", "Fev/22", "Mar/22",
            "Abr/22", "Mai/22", "Jun/22", "Jul/22", "Ago/22", "Set/22", "Out/22",
            "Nov/22", "Dez/22", "Jan/23"])
plt.xticks(rotation=45)
plt.ylabel("Vazão")
plt.title("Vazão x Tempo")
```

Out[ ]:
```
Text(0.5, 1.0, 'Vazão x Tempo')
```



In [ ]:
```
#Formato dos DataFrames
print("Formato DataFrame 1: ",df.values.shape)
print("Formato DataFrame 2: ",df2.values.shape)
```

```
Formato DataFrame 1:  (105121, 3)
Formato DataFrame 2:  (105121, 3)
```

In [ ]:
```
#Parâmetro Alvo
target_names = ["Vazão"]
```

In [ ]:
```
#Número de passos deslocados
shift_steps = 288 #Equivale a 1 dia
```

Criação dos DataFrame Targets, deslocando do parâmetro escolhido o número de passos

In [ ]:
```
df_targets = df[target_names].shift(-shift_steps)
df_targets2 = df2[target_names].shift(-shift_steps)
```

In [ ]:
```
df[target_names].tail(shift_steps+5)
```

Out[ ]:

	Vazão
Tempo	
30/12/2021 23:40	0.014823
30/12/2021 23:45	0.014823
30/12/2021 23:50	0.014823
30/12/2021 23:55	0.014823
31/12/2021 00:00	0.014823
...	...
31/12/2021 23:40	0.014776
31/12/2021 23:45	0.014776
31/12/2021 23:50	0.014775
31/12/2021 23:55	0.014775
01/01/2022 00:00	0.014775

293 rows × 1 columns

In [ ]:
```
df_targets.tail(shift_steps + 5)
```

Out[ ]:

Tempo	Vazão
30/12/2021 23:40	0.014776
30/12/2021 23:45	0.014776
30/12/2021 23:50	0.014775
30/12/2021 23:55	0.014775
31/12/2021 00:00	0.014775
...	...
31/12/2021 23:40	NaN
31/12/2021 23:45	NaN
31/12/2021 23:50	NaN
31/12/2021 23:55	NaN
01/01/2022 00:00	NaN

293 rows × 1 columns

In [ ]:
```
df2[target_names].tail(shift_steps+5)
```

Out[ ]:

Tempo	Vazão
30/12/2022 23:40	0.008085
30/12/2022 23:45	0.008085
30/12/2022 23:50	0.008085
30/12/2022 23:55	0.008085
31/12/2022 00:00	0.008084
...	...
31/12/2022 23:40	0.008030
31/12/2022 23:45	0.008030
31/12/2022 23:50	0.008030
31/12/2022 23:55	0.008030
01/01/2023 00:00	0.008030

293 rows × 1 columns

In [ ]:
```
df_targets2.tail(shift_steps + 5)
```

Out[ ]:

Tempo	Vazão
30/12/2022 23:40	0.00803
30/12/2022 23:45	0.00803
30/12/2022 23:50	0.00803
30/12/2022 23:55	0.00803
31/12/2022 00:00	0.00803
...	...
31/12/2022 23:40	NaN
31/12/2022 23:45	NaN
31/12/2022 23:50	NaN
31/12/2022 23:55	NaN
01/01/2023 00:00	NaN

293 rows × 1 columns

Definição dos vetores de entrada

In [ ]:
```
x_data = df.values[0:-shift_steps]
x_data2 = df2.values[0:-shift_steps]
```

In [ ]:
```
print(type(x_data))
print("Formato inputs 1:", x_data.shape)
print(type(x_data2))
print("Formato inputs 2:", x_data2.shape)
```

```
<class 'numpy.ndarray'>
Formato inputs 1: (104833, 3)
<class 'numpy.ndarray'>
Formato inputs 2: (104833, 3)
```

Definição dos vetores de saída

In [ ]:
```
y_data = df_targets.values[:-shift_steps]
y_data2 = df_targets2.values[:-shift_steps]
```

In [ ]:
```
print(type(y_data))
print("Formato outputs 1:", y_data.shape)
print(type(y_data2))
print("Formato outputs 2:", y_data2.shape)
```

```
<class 'numpy.ndarray'>
Formato outputs 1: (104833, 1)
<class 'numpy.ndarray'>
Formato outputs 2: (104833, 1)
```

Definição da quantidade de dados de treinamento e teste

In [ ]:
```
num_data = len(x_data2)
train_split = 0.85
num_train = int(train_split * num_data)
print("Número de dados de treinamento:",num_train)
```

```
Número de dados de treinamento: 89108
```

In [ ]:
```
num_test = num_data - num_train
print("Número de dados de teste:",num_test)
```

```
Número de dados de teste: 15725
```

Separação da entrada nos dados de treinamento e teste

In [ ]:
```
x_train = x_data2[0:num_train]
x_test = x_data2[num_train:]
```

Separação da saída nos dados de treinamento e teste

In [ ]:
```
y_train = y_data2[0:num_train]
y_test = y_data2[num_train:]
```

In [ ]:
```
#Quantidade de parâmetros de entrada
num_x_signals = x_data2.shape[1]
print("Parâmetros de entrada:",num_x_signals)
```

```
Parâmetros de entrada: 3
```

In [ ]:
```
#Quantidade de parâmetros de saída
num_y_signals = y_data2.shape[1]
print("Parâmetros de saída:",num_y_signals)
```

```
Parâmetros de saída: 1
```

In [ ]:
```
#Valores mínimo e máximo dos dados de treinamento
print("Min:", np.min(x_train))
print("Max:", np.max(x_train))
```

```
Min: 0.007733355
Max: 48706532.88
```

In [ ]:
```
#Método para escalar os dados
x_scaler = MinMaxScaler()
```

In [ ]:
```
#Variável com os dados de entrada de treinamento escalados entre [0,1]
x_train_scaled = x_scaler.fit_transform(x_train)
print("Min:", np.min(x_train_scaled))
print("Max:", np.max(x_train_scaled))
```

```
Min: 0.0
Max: 1.0
```

In [ ]:
```
#Variável com os dados de entrada de teste escalados entre [0,1]
x_test_scaled = x_scaler.transform(x_test)
```

In [ ]:
```
y_scaler = MinMaxScaler()
#Variável com os dados de saída de treinamento escalados entre [0,1]
y_train_scaled = y_scaler.fit_transform(y_train)

#Variável com os dados de saída de teste escalados entre [0,1]
y_test_scaled = y_scaler.transform(y_test)
```

Função para criação de batchs de treinamento, utilizando pontos iniciais aleatórios

In [ ]:
```
def batch_generator(batch_size, sequence_length):
    """
    Generator function for creating random batches of training-data.
    """

    # Infinite loop.
    while True:
        # Allocate a new array for the batch of input-signals.
        x_shape = (batch_size, sequence_length, num_x_signals)
        x_batch = np.zeros(shape=x_shape, dtype=np.float64)

        # Allocate a new array for the batch of output-signals.
        y_shape = (batch_size, sequence_length, num_y_signals)
```

```
        y_batch = np.zeros(shape=y_shape, dtype=np.float64)

        # Fill the batch with random sequences of data.
        for i in range(batch_size):
            # Get a random start-index.
            # This points somewhere into the training-data.
            idx = np.random.randint(num_train - sequence_length)

            # Copy the sequences of data starting at this index.
            x_batch[i] = x_train_scaled[idx:idx+sequence_length]
            y_batch[i] = y_train_scaled[idx:idx+sequence_length]

        yield (x_batch, y_batch)
```

In [ ]:
```
#Quantidade de batchs
batch_size = 150
```

In [ ]:
```
#Tamanho da sequência de dados
sequence_length = 288 * 5 #Corresponde a 5 dias
print("Tamanho da sequência:",sequence_length)
```

```
Tamanho da sequência: 1440
```

In [ ]:
```
generator = batch_generator(batch_size=batch_size,
                            sequence_length=sequence_length)
```

In [ ]:
```
#Criação dos batchs
x_batch, y_batch = next(generator)
```

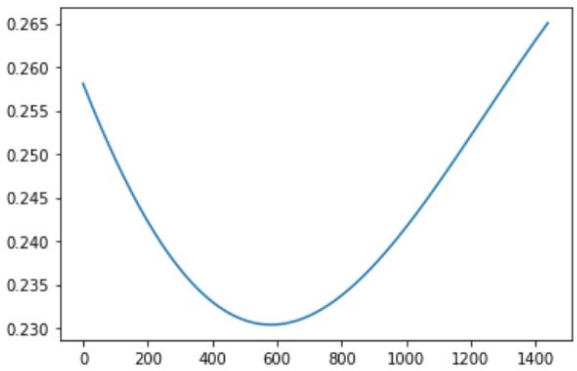
In [ ]:
```
print("Formato do x_batch:",x_batch.shape)
print("Formato do y_batch:",y_batch.shape)
```

```
Formato do x_batch: (150, 1440, 3)
Formato do y_batch: (150, 1440, 1)
```

Visualização da vazão, para a sequência do 1º batch dos dados de entrada

In [ ]:
```
batch = 0   # First sequence in the batch.
signal = 0  # First signal from the 3 input-signals.
seq = x_batch[batch, :, signal]
plt.plot(seq)
```

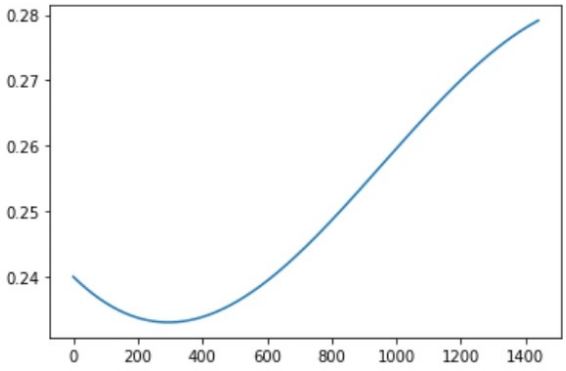
Out[ ]:
```
[<matplotlib.lines.Line2D at 0x29590f99220>]
```



Visualização da vazão, para a sequência do 1º batch dos dados de saída

In [ ]:
```
seq = y_batch[batch, :, signal]
plt.plot(seq)
```

Out[ ]:
```
[<matplotlib.lines.Line2D at 0x2958fdef8b0>]
```



In [ ]:
```
#Definição dos dados de validação
validation_data = (np.expand_dims(x_test_scaled, axis=0),
                   np.expand_dims(y_test_scaled, axis=0))
```

Criação da RNN do tipo GRU

In [ ]:
```
model = Sequential()
```

In [ ]:
```
#1ª camada da rede - GRU
model.add(GRU(units=375,
              return_sequences=True,
              input_shape=(None, num_x_signals,)))
```

In [ ]:
```
#2ª camada da rede - GRU
model.add(GRU(units=375,
              return_sequences=True))
```

In [ ]:
```
#3ª camada da rede - Densa - Função de ativação: sigmoidal
model.add(Dense(200, activation='sigmoid'))
```

In [ ]:
```
#4ª camada da rede - Densa - Função de ativação: sigmoidal
model.add(Dense(num_y_signals, activation='sigmoid'))
```

In [ ]:
```
#Otimizador
optimizer = RMSprop(learning_rate=1e-3)
```

In [ ]:
```
#Compilação do modelo
model.compile(loss="mse", optimizer=optimizer)
```

In [ ]:
```
#Visualização do sumário da rede, com as camadas,
# número de unidades e parâmetros treináveis
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 gru (GRU)                   (None, None, 375)         427500

 gru_1 (GRU)                 (None, None, 375)         846000

 dense (Dense)               (None, None, 200)         75200

 dense_1 (Dense)             (None, None, 1)           201

=================================================================
Total params: 1,348,901
Trainable params: 1,348,901
Non-trainable params: 0
_________________________________________________________________
```

In [ ]:
```
#Checkpoint para salvamento dos pesos
path_checkpoint = 'GRU_Modelo2_checkpoint.keras'
callback_checkpoint = ModelCheckpoint(filepath=path_checkpoint,
                                      monitor='val_loss',
                                      verbose=1,
                                      save_weights_only=True,
                                      save_best_only=True)
```

In [ ]:
```
#Método para interrupção antecipada do treinamento, caso a rede pare de aprender
callback_early_stopping = EarlyStopping(monitor='val_loss',
                                        patience=5, verbose=1)
```

In [ ]:
```
callback_tensorboard = TensorBoard(log_dir='./GRU_Modelo2_logs/',
                                   histogram_freq=0,
                                   write_graph=False)
```

In [ ]:
```
#Método para redução da taxa de aprendizado
callback_reduce_lr = ReduceLROnPlateau(monitor='val_loss',
                                       factor=0.1,
                                       min_lr=1e-6,
                                       patience=2,
                                       verbose=1)
```

In [ ]:
```
callbacks = [callback_early_stopping,
             callback_checkpoint,
             callback_tensorboard,
             callback_reduce_lr]
```

Treinamento da RNN

In [ ]:
```
%%time
model.fit(x=generator,
          epochs=25,
          steps_per_epoch=300,
          validation_data=validation_data,
          verbose=2,
          callbacks=callbacks)
```

```
Epoch 1/25

Epoch 00001: val_loss improved from inf to 0.00219, saving model to GRU_Modelo2_checkpoint.ke
ras
300/300 - 248s - loss: 0.0072 - val_loss: 0.0022 - lr: 0.0010 - 248s/epoch - 828ms/step
Epoch 2/25

Epoch 00002: val_loss improved from 0.00219 to 0.00183, saving model to GRU_Modelo2_checkpoin
t.keras
300/300 - 248s - loss: 0.0037 - val_loss: 0.0018 - lr: 0.0010 - 248s/epoch - 827ms/step
Epoch 3/25

Epoch 00003: val_loss did not improve from 0.00183
300/300 - 252s - loss: 0.0035 - val_loss: 0.0019 - lr: 0.0010 - 252s/epoch - 839ms/step
Epoch 4/25

Epoch 00004: val_loss improved from 0.00183 to 0.00157, saving model to GRU_Modelo2_checkpoin
t.keras
300/300 - 253s - loss: 0.0033 - val_loss: 0.0016 - lr: 0.0010 - 253s/epoch - 845ms/step
Epoch 5/25

Epoch 00005: val_loss improved from 0.00157 to 0.00138, saving model to GRU_Modelo2_checkpoin
t.keras
300/300 - 254s - loss: 0.0031 - val_loss: 0.0014 - lr: 0.0010 - 254s/epoch - 847ms/step
Epoch 6/25

Epoch 00006: val_loss did not improve from 0.00138
300/300 - 255s - loss: 0.0031 - val_loss: 0.0015 - lr: 0.0010 - 255s/epoch - 851ms/step
Epoch 7/25

Epoch 00007: val_loss did not improve from 0.00138

Epoch 00007: ReduceLROnPlateau reducing learning rate to 0.00010000000474974513.
300/300 - 256s - loss: 0.0031 - val_loss: 0.0016 - lr: 0.0010 - 256s/epoch - 852ms/step
Epoch 8/25

Epoch 00008: val_loss improved from 0.00138 to 0.00082, saving model to GRU_Modelo2_checkpoin
t.keras
300/300 - 255s - loss: 0.0026 - val_loss: 8.2123e-04 - lr: 1.0000e-04 - 255s/epoch - 851ms/st
ep
Epoch 9/25

Epoch 00009: val_loss improved from 0.00082 to 0.00078, saving model to GRU_Modelo2_checkpoin
t.keras
300/300 - 255s - loss: 0.0025 - val_loss: 7.7720e-04 - lr: 1.0000e-04 - 255s/epoch - 851ms/st
ep
Epoch 10/25

Epoch 00010: val_loss improved from 0.00078 to 0.00076, saving model to GRU_Modelo2_checkpoin
t.keras

Epoch 00010: ReduceLROnPlateau reducing learning rate to 1.0000000474974514e-05.
300/300 - 256s - loss: 0.0025 - val_loss: 7.5653e-04 - lr: 1.0000e-04 - 256s/epoch - 852ms/st
ep
Epoch 11/25

Epoch 00011: val_loss improved from 0.00076 to 0.00074, saving model to GRU_Modelo2_checkpoin
t.keras
300/300 - 256s - loss: 0.0020 - val_loss: 7.4235e-04 - lr: 1.0000e-05 - 256s/epoch - 852ms/st
ep
Epoch 12/25

Epoch 00012: val_loss improved from 0.00074 to 0.00073, saving model to GRU_Modelo2_checkpoin
t.keras

Epoch 00012: ReduceLROnPlateau reducing learning rate to 1.0000000656873453e-06.
300/300 - 256s - loss: 0.0020 - val_loss: 7.3240e-04 - lr: 1.0000e-05 - 256s/epoch - 853ms/st
ep
Epoch 13/25
```

```
Epoch 00013: val_loss improved from 0.00073 to 0.00073, saving model to GRU_Modelo2_checkpoin
t.keras
300/300 - 256s - loss: 0.0020 - val_loss: 7.3159e-04 - lr: 1.0000e-06 - 256s/epoch - 854ms/st
ep
Epoch 14/25

Epoch 00014: val_loss improved from 0.00073 to 0.00073, saving model to GRU_Modelo2_checkpoin
t.keras

Epoch 00014: ReduceLROnPlateau reducing learning rate to 1e-06.
300/300 - 256s - loss: 0.0019 - val_loss: 7.2890e-04 - lr: 1.0000e-06 - 256s/epoch - 852ms/st
ep
Epoch 15/25

Epoch 00015: val_loss improved from 0.00073 to 0.00073, saving model to GRU_Modelo2_checkpoin
t.keras
300/300 - 255s - loss: 0.0019 - val_loss: 7.2586e-04 - lr: 1.0000e-06 - 255s/epoch - 851ms/st
ep
Epoch 16/25

Epoch 00016: val_loss improved from 0.00073 to 0.00072, saving model to GRU_Modelo2_checkpoin
t.keras
300/300 - 256s - loss: 0.0019 - val_loss: 7.2487e-04 - lr: 1.0000e-06 - 256s/epoch - 852ms/st
ep
Epoch 17/25

Epoch 00017: val_loss improved from 0.00072 to 0.00072, saving model to GRU_Modelo2_checkpoin
t.keras
300/300 - 255s - loss: 0.0019 - val_loss: 7.2465e-04 - lr: 1.0000e-06 - 255s/epoch - 851ms/st
ep
Epoch 18/25

Epoch 00018: val_loss improved from 0.00072 to 0.00072, saving model to GRU_Modelo2_checkpoin
t.keras
300/300 - 256s - loss: 0.0019 - val_loss: 7.2342e-04 - lr: 1.0000e-06 - 256s/epoch - 854ms/st
ep
Epoch 19/25

Epoch 00019: val_loss improved from 0.00072 to 0.00072, saving model to GRU_Modelo2_checkpoin
t.keras
300/300 - 256s - loss: 0.0020 - val_loss: 7.2103e-04 - lr: 1.0000e-06 - 256s/epoch - 853ms/st
ep
Epoch 20/25

Epoch 00020: val_loss improved from 0.00072 to 0.00072, saving model to GRU_Modelo2_checkpoin
t.keras
300/300 - 257s - loss: 0.0018 - val_loss: 7.1830e-04 - lr: 1.0000e-06 - 257s/epoch - 855ms/st
ep
Epoch 21/25

Epoch 00021: val_loss improved from 0.00072 to 0.00072, saving model to GRU_Modelo2_checkpoin
t.keras
300/300 - 256s - loss: 0.0019 - val_loss: 7.1825e-04 - lr: 1.0000e-06 - 256s/epoch - 855ms/st
ep
Epoch 22/25

Epoch 00022: val_loss improved from 0.00072 to 0.00072, saving model to GRU_Modelo2_checkpoin
t.keras
300/300 - 256s - loss: 0.0019 - val_loss: 7.1777e-04 - lr: 1.0000e-06 - 256s/epoch - 854ms/st
ep
Epoch 23/25

Epoch 00023: val_loss improved from 0.00072 to 0.00072, saving model to GRU_Modelo2_checkpoin
t.keras
300/300 - 256s - loss: 0.0018 - val_loss: 7.1502e-04 - lr: 1.0000e-06 - 256s/epoch - 855ms/st
ep
Epoch 24/25
```

```
Epoch 00024: val_loss did not improve from 0.00072
300/300 - 255s - loss: 0.0020 - val_loss: 7.1641e-04 - lr: 1.0000e-06 - 255s/epoch - 852ms/st
ep
Epoch 25/25

Epoch 00025: val_loss improved from 0.00072 to 0.00071, saving model to GRU_Modelo2_checkpoin
t.keras
300/300 - 256s - loss: 0.0019 - val_loss: 7.1498e-04 - lr: 1.0000e-06 - 256s/epoch - 854ms/st
ep
CPU times: total: 1h 13min 36s
Wall time: 1h 46min 11s
```

Out[ ]:
```
<keras.callbacks.History at 0x1611250d8b0>
```

In [ ]:
```
#Carregamento dos pesos
try:
    model.load_weights(path_checkpoint)
except Exception as error:
    print("Error trying to load checkpoint.")
    print(error)
```

In [ ]:
```
#Avaliação da rede com os dados de teste
resultado_modelo2 = model.evaluate(x=np.expand_dims(x_test_scaled, axis=0),
                        y=np.expand_dims(y_test_scaled, axis=0))
```

```
1/1 [==============================] - 26s 26s/step - loss: 7.1498e-04
```

In [ ]:
```
print("MSE (Dados de Teste): %10.3e" % (resultado_modelo2))
```

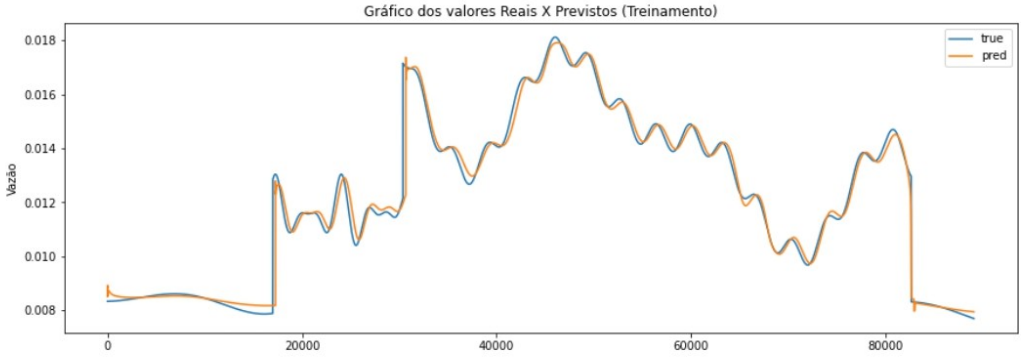
```
MSE (Dados de Teste):  7.150e-04
```

Função para plotar os dados reais e os dados previstos pela rede

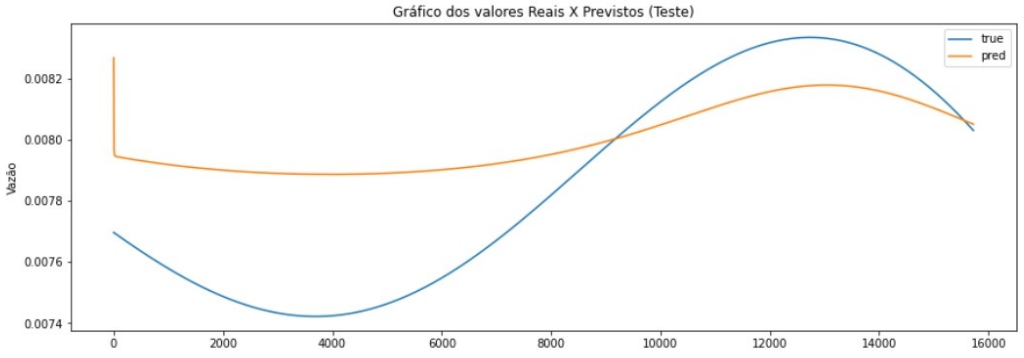
In [ ]:
```
def plot_comparison(start_idx, length=100, train=True):
    """
    Plot the predicted and true output-signals.

    :param start_idx: Start-index for the time-series.
    :param length: Sequence-length to process and plot.
    :param train: Boolean whether to use training- or test-set.
    """

    if train:
        # Use training-data.
        x = x_train_scaled
        y_true = y_train
        titulo = "Gráfico dos valores Reais X Previstos (Treinamento)"
    else:
        # Use test-data.
        x = x_test_scaled
        y_true = y_test
        titulo = "Gráfico dos valores Reais X Previstos (Teste)"

    # End-index for the sequences.
    end_idx = start_idx + length

    # Select the sequences from the given start-index and
    # of the given length.
    x = x[start_idx:end_idx]
    y_true = y_true[start_idx:end_idx]

    # Input-signals for the model.
    x = np.expand_dims(x, axis=0)

    # Use the model to predict the output-signals.
    y_pred = model.predict(x)

    # The output of the model is between 0 and 1.
    # Do an inverse map to get it back to the scale
```

```
    # of the original data-set.
    y_pred_rescaled = y_scaler.inverse_transform(y_pred[0])

    # For each output-signal.
    for signal in range(len(target_names)):
        # Get the output-signal predicted by the model.
        signal_pred = y_pred_rescaled[:, signal]

        # Get the true output-signal from the data-set.
        signal_true = y_true[:, signal]

        # Make the plotting-canvas bigger.
        plt.figure(figsize=(15,5))

        # Plot and compare the two signals.
        plt.plot(signal_true, label='true')
        plt.plot(signal_pred, label='pred')

        # Plot labels etc.
        plt.ylabel(target_names[signal])
        plt.title(titulo)
        plt.legend()
        plt.show()
```

In [ ]:
```
plot_comparison(start_idx=0, length=num_train, train=True)
```



In [ ]:
```
plot_comparison(start_idx=0, length=num_test, train=False)
```



In [ ]:
```
#Exportação dos dados de treinamento previstos para csv
y_prev_model2GRU_dados2_train = model.predict(np.expand_dims(x_train_scaled,axis=0))
y_prev_model2GRU_dados2_train_rescaled = y_scaler.inverse_transform(
    y_prev_model2GRU_dados2_train[0])
y_prev_model2GRU_dados2_train_df = pd.DataFrame(y_prev_model2GRU_dados2_train_rescaled)
filepath = Path(path_dir + 'y_prev_model2GRU_dados2_train.csv')
filepath.parent.mkdir(parents=True, exist_ok=True)
y_prev_model2GRU_dados2_train_df.to_csv(filepath)
```

In [ ]:
```
#Exportação dos dados de teste previstos para csv
y_prev_model2GRU_dados2_test = model.predict(np.expand_dims(x_test_scaled,axis=0))
y_prev_model2GRU_dados2_test_rescaled = y_scaler.inverse_transform(
    y_prev_model2GRU_dados2_test[0])
y_prev_model2GRU_dados2_test_df = pd.DataFrame(y_prev_model2GRU_dados2_test_rescaled)
filepath = Path(path_dir + 'y_prev_model2GRU_dados2_test.csv')
filepath.parent.mkdir(parents=True, exist_ok=True)
y_prev_model2GRU_dados2_test_df.to_csv(filepath)
```

Avaliação da abrangência do Modelo

In [ ]:
```
x_data1_scaled = x_scaler.fit_transform(x_data)
y_data1_scaled = y_scaler.fit_transform(y_data)
```

In [ ]:
```
resultado_modelo2_dados1 = model.evaluate(x=np.expand_dims(x_data1_scaled, axis=0),
                        y=np.expand_dims(y_data1_scaled, axis=0))
```

```
1/1 [==============================] - 3s 3s/step - loss: 0.0028
```

In [ ]:
```
print("MSE (Conjunto de dados 1): %10.3e" % (resultado_modelo2_dados1))
```

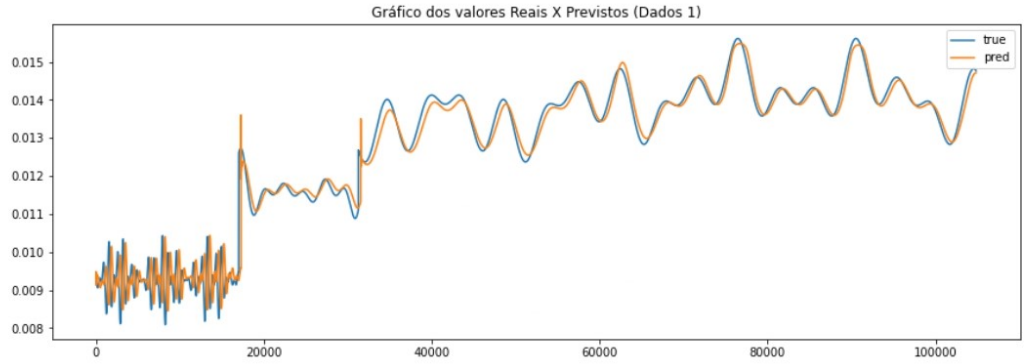
```
MSE (Conjunto de dados 1):  2.846e-03
```

Gráfico dos valores Reais X Previsão do Modelo 2, usando o conjunto de dados 1 como entrada

In [ ]:
```
#Exportação dos dados de abrangência previstos para csv
y_previsao_modelo2_dados1 = model.predict(np.expand_dims(x_data1_scaled, axis=0))
y_previsao_modelo2_dados1_rescaled = y_scaler.inverse_transform(y_previsao_modelo2_dados1[0])
y_previsao_modelo2GRU_dados1_df = pd.DataFrame(y_previsao_modelo2_dados1_rescaled)
filepath = Path(path_dir + 'y_previsao_modelo2GRU_dados1.csv')
filepath.parent.mkdir(parents=True, exist_ok=True)
y_previsao_modelo2GRU_dados1_df.to_csv(filepath)
```

In [ ]:
```
plt.figure(figsize=(15,5))

# Plot and compare the two signals.
plt.plot(y_data, label='true')
plt.plot(y_previsao_modelo2_dados1_rescaled, label='pred')
plt.title("Gráfico dos valores Reais X Previstos (Dados 1)")
plt.legend()
```

Out[ ]:
```
<matplotlib.legend.Legend at 0x295b3c33b80>
```



Previsão Retroalimentada

In [ ]:
```
def vector3d_generator(num_test, sequence_length):
    """
    Generator function for creating a 3D vector.
    """

    # Infinite loop.
```

```
    while True:
        # Allocate a new array for the batch of input-signals.
        x_shape = (1, num_test + sequence_length, num_x_signals)
        x_prev = np.zeros(shape=x_shape, dtype=np.float64)

        # Fill the batch with random sequences of data.
        for i in range(num_test + sequence_length):
            if i < sequence_length:
                x_prev[0:1,i] = x_train_scaled[len(x_train_scaled)-sequence_length+i:
                                               len(x_train_scaled)-sequence_length+i+1]
            else:
                x_prev[0:1,i] = x_test_scaled[i-sequence_length:i-sequence_length+1]

        yield (x_prev)
```

In [ ]:
```
vetor_generator = vector3d_generator(num_test=num_test,
                            sequence_length=sequence_length)
```

In [ ]:
```
x_prev = next(vetor_generator)
```

In [ ]:
```
y_previsao_retroalimentada = []
for i in range(0, num_test):
    previsao = model.predict(x_prev[0:,i:i+sequence_length])
    y_previsao_retroalimentada.append(previsao[0,-1,0])
    x_prev[0:,i+sequence_length,0]=previsao[0,-1,0]
```

In [ ]:
```
x_prev_rescaled = x_scaler.inverse_transform(x_prev[0])
```

In [ ]:
```
#Exportação dos dados da previsão retroalimentada para csv
y_previsao_retroalimentada_rescaled = y_scaler.inverse_transform(
    np.expand_dims(y_previsao_retroalimentada,axis=1))
y_previsao_retroalimentada_df = pd.DataFrame(y_previsao_retroalimentada_rescaled)
filepath = Path(path_dir + 'y_prev_retroalimentada_model2GRU.csv')
filepath.parent.mkdir(parents=True, exist_ok=True)
y_previsao_retroalimentada_df.to_csv(filepath)
```

Avaliação da Previsão Retroalimentada

In [ ]:
```
resultado_previsao_retroalimentada = model.evaluate(
    x=np.expand_dims(x_prev_rescaled[1440:], axis=0),
    y=np.expand_dims(y_previsao_retroalimentada_rescaled, axis=0))
```

```
1/1 [==============================] - 1s 962ms/step - loss: 4.5592e-05
```

In [ ]:
```
print("MSE (Previsão Retroalimentada): %10.3e" % (resultado_previsao_retroalimentada))
```

```
MSE (Previsão Retroalimentada):  4.559e-05
```

Gráfico dos valores Reais X Previstos, utilizando a previsão da vazão como entrada no índice i+1

In [ ]:
```
plt.figure(figsize=(15,5))

# Plot and compare the two signals.
plt.plot(y_test, label='true')
plt.plot(y_previsao_retroalimentada_rescaled, label='pred')
plt.title("Gráfico dos valores Reais X Previstos (Previsão Retroalimentada)")
plt.legend()
```

Out[ ]:
```
<matplotlib.legend.Legend at 0x297c39bf3d0>
```

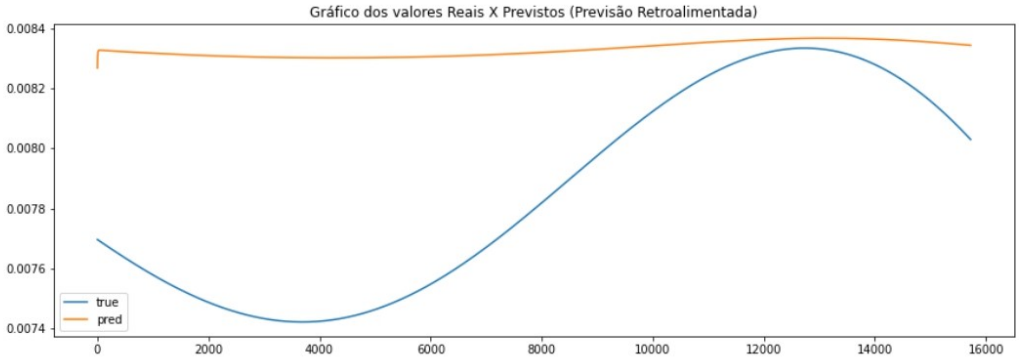


Gráfico dos valores Reais X Previstos (Previsão Retroalimentada)
0.0084
0.0082
0.0080
0.0078
0.0076
0.0074
true
pred
0
2000
4000
6000
8000
10000
12000
14000
16000

ANEXO A – TUTORIAL PREVISÃO DE SÉRIES TEMPORAIS

TensorFlow Tutorial #23

Time-Series Prediction

by Magnus Erik Hvass Pedersen / GitHub / Videos on YouTube

Introduction

This tutorial tries to predict the future weather of a city using weather-data from several other cities.

Because we will be working with sequences of arbitrary length, we will use a Recurrent Neural Network (RNN).

You should be familiar with TensorFlow and Keras in general, see Tutorials #01 and #03-C, and the basics of Recurrent Neural Networks as explained in Tutorial #20.

Location

We will use weather-data from the period 1980-2018 for five cities in Denmark:

- **Aalborg** The weather-data is actually from an airforce base which is also home to The Hunter Corps (Jægerkorps)).
- **Aarhus** is the city where the inventor of C++ studied and the Google V8 JavaScript Engine was developed.
- **Esbjerg** has a large fishing-port.
- **Odense** is the birth-city of the fairytale author H. C. Andersen.
- **Roskilde** has an old cathedral housing the tombs of the Danish royal family.

The following map shows the location of the cities in Denmark:

The following map shows the location of Denmark within Europe:

Flowchart

In this tutorial, we are trying to predict the weather for the Danish city "Odense" 24 hours into the future, given the current and past weather-data from 5 cities (although the flowchart below only shows 2 cities).

We use a Recurrent Neural Network (RNN) because it can work on sequences of arbitrary length. During training we will use sub-sequences of 1344 data-points (8 weeks) from the training-set, with each data-point or observation having 20 input-signals for the temperature, pressure, etc. for each of the 5 cities. We then want to train the neural network so it outputs the 3 signals for tomorrow's temperature, pressure and wind-speed.

Imports

In [ ]:
```
%matplotlib inline
import matplotlib.pyplot as plt
import tensorflow as tf
import numpy as np
import pandas as pd
import os
from sklearn.preprocessing import MinMaxScaler
```

We need to import several things from Keras.

In [ ]:
```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Input, Dense, GRU, Embedding
from tensorflow.keras.optimizers import RMSprop
from tensorflow.keras.callbacks import EarlyStopping, ModelCheckpoint
from tensorflow.keras.callbacks import TensorBoard, ReduceLROnPlateau
from tensorflow.keras.backend import square, mean
```

This was developed using Python 3.6 (Anaconda) and package versions:

In [ ]:
```
tf.__version__
```

Out[ ]:
```
'2.1.0'
```

In [ ]:
```
tf.keras.__version__
```

Out[ ]:
```
'2.2.4-tf'
```

In [ ]:
```
pd.__version__
```

Out[ ]:
```
'1.0.3'
```

Load Data

Weather-data for 5 cities in Denmark will be downloaded automatically below.

The raw weather-data was originally obtained from the National Climatic Data Center (NCDC), USA. Their web-site and database-access is very confusing and may change soon. Furthermore, the raw data-file had to be manually edited before it could be read. So you should expect some challenges if you want to download weather-data for another region. The following Python-module provides some functionality that may be helpful if you want to use new weather-data, but you will have to modify the source-code to fit your data-format.

In [ ]:
```
import weather
```

Download the data-set if you don't have it already. It is about 35 MB.

In [ ]:
```
weather.maybe_download_and_extract()
```

```
Data has apparently already been downloaded and unpacked.
```

List of the cities used in the data-set.

In [ ]:
```
cities = weather.cities
cities
```

Out[ ]:
```
['Aalborg', 'Aarhus', 'Esbjerg', 'Odense', 'Roskilde']
```

Load and resample the data so it has observations at regular time-intervals for every 60 minutes. Missing data-points are linearly interpolated. This takes about 30 seconds to run the first time but uses a cache-file so it loads very quickly the next time.

In [ ]:
```
%%time
df = weather.load_resampled_data()
```

```
CPU times: user 13.9 ms, sys: 55.4 ms, total: 69.3 ms
Wall time: 157 ms
```

These are the top rows of the data-set.

In [ ]:
```
df.head()
```

Out[ ]:

	Aalborg				Aarhus				
	Temp	Pressure	WindSpeed	WindDir	Temp	Pressure	WindSpeed	WindDir	Temp
DateTime									
1980-03-01 11:00:00	5.000000	1007.766667	10.2	280.000000	5.0	1008.300000	15.4	290.0	6.083333
1980-03-01 12:00:00	5.000000	1008.000000	10.3	290.000000	5.0	1008.600000	13.4	280.0	6.583333
1980-03-01 13:00:00	5.000000	1008.066667	9.7	290.000000	5.0	1008.433333	15.4	280.0	6.888889
1980-03-01 14:00:00	4.333333	1008.133333	11.1	283.333333	5.0	1008.266667	14.9	300.0	6.222222
1980-03-01 15:00:00	4.000000	1008.200000	11.3	280.000000	5.0	1008.100000	17.0	290.0	5.555556

Missing Data

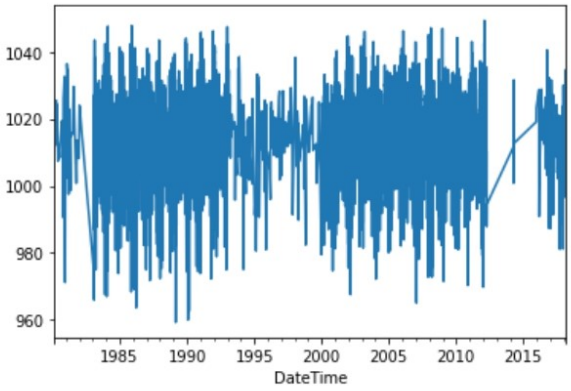
The two cities Esbjerg and Roskilde have missing data for the atmospheric pressure, as can be seen in the following two plots.

Because we are using resampled data, we have filled in the missing values with new values that are linearly interpolated from the neighbouring values, which appears as long straight lines in these plots.

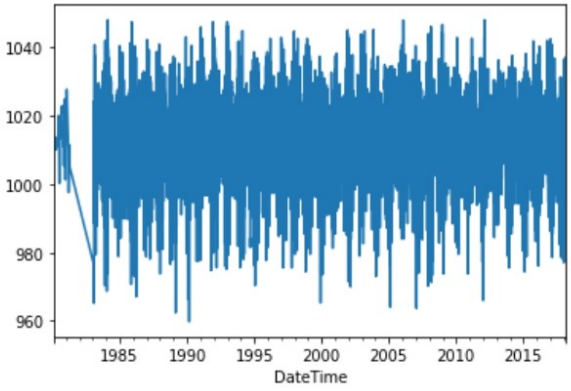
This may confuse the neural network. For simplicity, we will simply remove these two signals from the data.

But it is only short periods of data that are missing, so you could actually generate this data by creating a predictive model that generates the missing data from all the other input signals. Then you could add these generated values back into the data-set to fill the gaps.

In [ ]:
```
df['Esbjerg']['Pressure'].plot();
```



In [ ]:
```
df['Roskilde']['Pressure'].plot();
```



Before removing these two signals, there are 20 input-signals in the data-set.

In [ ]:
```
df.values.shape
```

Out[ ]:
```
(333109, 20)
```

Then we remove the two signals that have missing data.

In [ ]:
```
df.drop(('Esbjerg', 'Pressure'), axis=1, inplace=True)
df.drop(('Roskilde', 'Pressure'), axis=1, inplace=True)
```

Now there are only 18 input-signals in the data.

In [ ]:
```
df.values.shape
```

Out[ ]:
```
(333109, 18)
```

We can verify that these two data-columns have indeed been removed.

In [ ]:
```
df.head(1)
```

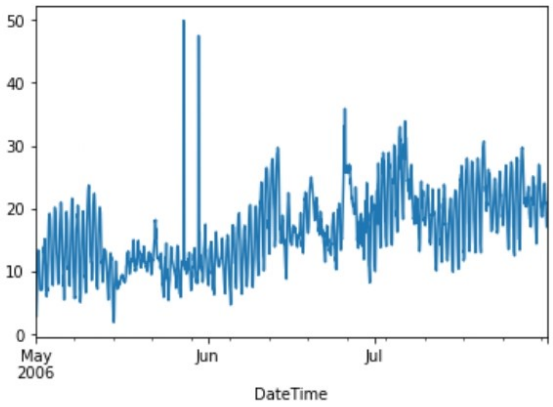
Out[ ]:

	Aalborg				Aarhus					
	Temp	Pressure	WindSpeed	WindDir	Temp	Pressure	WindSpeed	WindDir	Temp	WindS
DateTime										
1980-03-01 11:00:00	5.0	1007.766667	10.2	280.0	5.0	1008.3	15.4	290.0	6.083333	12.38

Data Errors

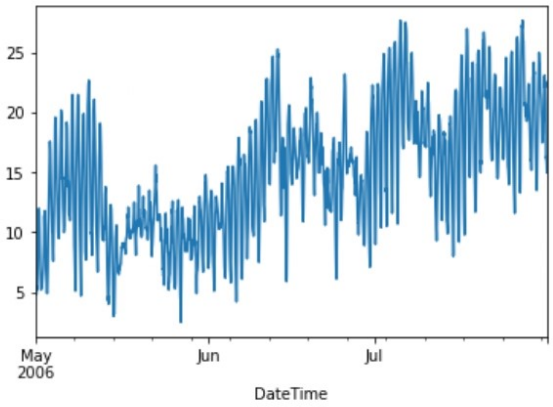
There are some errors in this data. As shown in the plot below, the temperature in the city of Odense suddenly jumped to almost 50 degrees C. But the highest temperature ever measured in Denmark was only 36.4 degrees Celcius and the lowest was -31.2 C. So this is clearly a data error. However, we will not correct any data-errors in this tutorial.

In [ ]:
```
df['Odense']['Temp']['2006-05':'2006-07'].plot();
```

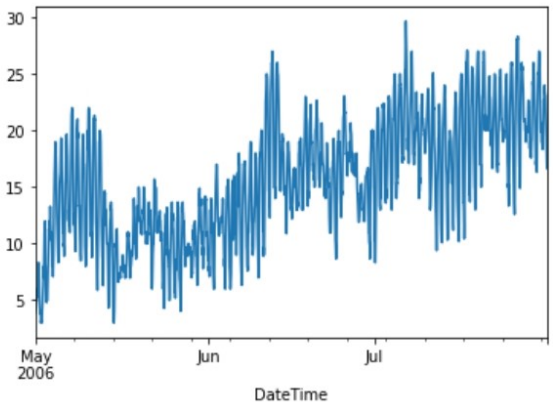


This can also be confirmed to be an error by considering the temperatures in some of the other cities in Denmark for that period, which was only around 10 degrees. Because the country is so small, it is not possible for one city in Denmark to have 50 degrees while another city only has 10 degrees.

In [ ]:
```
df['Aarhus']['Temp']['2006-05':'2006-07'].plot();
```



In [ ]:
```
df['Roskilde']['Temp']['2006-05':'2006-07'].plot();
```



Add Data

We can add some input-signals to the data that may help our model in making predictions.

For example, given just a temperature of 10 degrees Celcius the model wouldn't know whether that temperature was measured during the day or the night, or during summer or winter. The model would have to infer this from the surrounding data-points which might not be very accurate for determining whether it's an abnormally warm winter, or an abnormally cold summer, or whether it's day or night. So having this information could make a big difference in how accurately the model can predict the next output.

Although the data-set does contain the date and time information for each observation, it is only used in the index so as to order the data. We will therefore add separate input-signals to the data-set for the day-of-year (between 1 and 366) and the hour-of-day (between 0 and 23).

In [ ]:
```
df['Various', 'Day'] = df.index.dayofyear
df['Various', 'Hour'] = df.index.hour
```

Target Data for Prediction

We will try and predict the future weather-data for this city.

In [ ]:
```
target_city = 'Odense'
```

We will try and predict these signals.

In [ ]:
```
target_names = ['Temp', 'WindSpeed', 'Pressure']
```

The following is the number of time-steps that we will shift the target-data. Our data-set is resampled to have an observation for each hour, so there are 24 observations for 24 hours.

If we want to predict the weather 24 hours into the future, we shift the data 24 time-steps. If we want to predict the weather 7 days into the future, we shift the data 7 * 24 time-steps.

In [ ]:
```
shift_days = 1
shift_steps = shift_days * 24  # Number of hours.
```

Create a new data-frame with the time-shifted data.

Note the negative time-shift!

In [ ]:
```
df_targets = df[target_city][target_names].shift(-shift_steps)
```

WARNING! You should double-check that you have shifted the data in the right direction! We want to predict the future, not the past!

The shifted data-frame is confusing because Pandas keeps the original time-stamps even though we have shifted the data. You can check the time-shift is correct by comparing the original and time-shifted data-frames.

This is the first `shift_steps + 5` rows of the original data-frame:

In [ ]:
```
df[target_city][target_names].head(shift_steps + 5)
```

Out[ ]:

DateTime	Temp	WindSpeed	Pressure
1980-03-01 11:00:00	6.142857	12.585714	1011.066667
1980-03-01 12:00:00	7.000000	11.300000	1011.200000
1980-03-01 13:00:00	7.000000	12.118182	1011.300000
1980-03-01 14:00:00	6.857143	12.742857	1011.400000
1980-03-01 15:00:00	6.000000	12.400000	1011.500000
1980-03-01 16:00:00	4.909091	12.618182	1011.688889
1980-03-01 17:00:00	3.953488	12.646512	1011.877778
1980-03-01 18:00:00	3.674419	11.725581	1012.066667
1980-03-01 19:00:00	3.395349	10.804651	1012.255556
1980-03-01 20:00:00	3.116279	9.883721	1012.444444
1980-03-01 21:00:00	2.837209	8.962791	1012.633333
1980-03-01 22:00:00	2.558140	8.041860	1012.822222
1980-03-01 23:00:00	2.279070	7.120930	1013.011111
1980-03-02 00:00:00	2.000000	6.200000	1013.200000
1980-03-02 01:00:00	2.076923	7.738462	1012.366667
1980-03-02 02:00:00	2.538462	7.969231	1011.533333
1980-03-02 03:00:00	3.000000	8.200000	1010.700000
1980-03-02 04:00:00	3.000000	7.927273	1010.100000
1980-03-02 05:00:00	2.916667	7.658333	1009.500000
1980-03-02 06:00:00	2.416667	7.408333	1008.900000
1980-03-02 07:00:00	2.000000	7.100000	1008.300000
1980-03-02 08:00:00	2.142857	6.542857	1007.700000
1980-03-02 09:00:00	3.000000	6.200000	1007.100000
1980-03-02 10:00:00	2.833333	8.350000	1006.466667
1980-03-02 11:00:00	2.000000	6.828571	1005.833333
1980-03-02 12:00:00	2.000000	8.200000	1005.200000
1980-03-02 13:00:00	0.166667	9.216667	1004.766667
1980-03-02 14:00:00	1.000000	11.885714	1004.333333
1980-03-02 15:00:00	1.000000	12.400000	1003.900000

The following is the first 5 rows of the time-shifted data-frame. This should be identical to the last 5 rows shown above from the original data, except for the time-stamp.

In [ ]:
```
df_targets.head(5)
```

Out[ ]:

DateTime	Temp	WindSpeed	Pressure
1980-03-01 11:00:00	2.000000	6.828571	1005.833333
1980-03-01 12:00:00	2.000000	8.200000	1005.200000
1980-03-01 13:00:00	0.166667	9.216667	1004.766667
1980-03-01 14:00:00	1.000000	11.885714	1004.333333
1980-03-01 15:00:00	1.000000	12.400000	1003.900000

The time-shifted data-frame has the same length as the original data-frame, but the last observations are `NaN` (not a number) because the data has been shifted backwards so we are trying to shift data that does not exist in the original data-frame.

In [ ]:
```
df_targets.tail()
```

Out[ ]:

DateTime	Temp	WindSpeed	Pressure
2018-03-01 19:00:00	NaN	NaN	NaN
2018-03-01 20:00:00	NaN	NaN	NaN
2018-03-01 21:00:00	NaN	NaN	NaN
2018-03-01 22:00:00	NaN	NaN	NaN
2018-03-01 23:00:00	NaN	NaN	NaN

NumPy Arrays

We now convert the Pandas data-frames to NumPy arrays that can be input to the neural network. We also remove the last part of the numpy arrays, because the target-data has `NaN` for the shifted period, and we only want to have valid data and we need the same array-shapes for the input- and output-data.

These are the input-signals:

In [ ]:
```
x_data = df.values[0:-shift_steps]
```

In [ ]:
```
print(type(x_data))
print("Shape:", x_data.shape)
```

```
<class 'numpy.ndarray'>
Shape: (333085, 20)
```

These are the output-signals (or target-signals):

In [ ]:
```
y_data = df_targets.values[:-shift_steps]
```

In [ ]:
```
print(type(y_data))
print("Shape:", y_data.shape)
```

```
<class 'numpy.ndarray'>
Shape: (333085, 3)
```

This is the number of observations (aka. data-points or samples) in the data-set:

In [ ]:
```
num_data = len(x_data)
num_data
```

Out[ ]:
```
333085
```

This is the fraction of the data-set that will be used for the training-set:

In [ ]:
```
train_split = 0.9
```

This is the number of observations in the training-set:

In [ ]:
```
num_train = int(train_split * num_data)
num_train
```

Out[ ]:
```
299776
```

This is the number of observations in the test-set:

In [ ]:
```
num_test = num_data - num_train
num_test
```

Out[ ]:
```
33309
```

These are the input-signals for the training- and test-sets:

In [ ]:
```
x_train = x_data[0:num_train]
x_test = x_data[num_train:]
len(x_train) + len(x_test)
```

Out[ ]:
```
333085
```

These are the output-signals for the training- and test-sets:

In [ ]:
```
y_train = y_data[0:num_train]
y_test = y_data[num_train:]
len(y_train) + len(y_test)
```

Out[ ]:
```
333085
```

This is the number of input-signals:

In [ ]:
```
num_x_signals = x_data.shape[1]
num_x_signals
```

Out[ ]:
```
20
```

This is the number of output-signals:

In [ ]:
```
num_y_signals = y_data.shape[1]
num_y_signals
```

Out[ ]:
```
3
```

Scaled Data

The data-set contains a wide range of values:

In [ ]:
```
print("Min:", np.min(x_train))
print("Max:", np.max(x_train))
```

```
Min: -27.0
Max: 1050.8
```

The neural network works best on values roughly between -1 and 1, so we need to scale the data before it is being input to the neural network. We can use `scikit-learn` for this.

We first create a scaler-object for the input-signals.

In [ ]:
```
x_scaler = MinMaxScaler()
```

We then detect the range of values from the training-data and scale the training-data.

In [ ]:
```
x_train_scaled = x_scaler.fit_transform(x_train)
```

Apart from a small rounding-error, the data has been scaled to be between 0 and 1.

In [ ]:
```
print("Min:", np.min(x_train_scaled))
print("Max:", np.max(x_train_scaled))
```

```
Min: 0.0
Max: 1.0000000000000002
```

We use the same scaler-object for the input-signals in the test-set.

In [ ]:
```
x_test_scaled = x_scaler.transform(x_test)
```

The target-data comes from the same data-set as the input-signals, because it is the weather-data for one of the cities that is merely time-shifted. But the target-data could be from a different source with different value-ranges, so we create a separate scaler-object for the target-data.

In [ ]:
```
y_scaler = MinMaxScaler()
y_train_scaled = y_scaler.fit_transform(y_train)
y_test_scaled = y_scaler.transform(y_test)
```

Data Generator

The data-set has now been prepared as 2-dimensional numpy arrays. The training-data has almost 300k observations, consisting of 20 input-signals and 3 output-signals.

These are the array-shapes of the input and output data:

In [ ]:
```
print(x_train_scaled.shape)
print(y_train_scaled.shape)
```

```
(299776, 20)
(299776, 3)
```

Instead of training the Recurrent Neural Network on the complete sequences of almost 300k observations, we will use the following function to create a batch of shorter sub-sequences picked at random from the training-data.

In [ ]:
```
def batch_generator(batch_size, sequence_length):
    """
    Generator function for creating random batches of training-data.
    """

    # Infinite loop.
    while True:
        # Allocate a new array for the batch of input-signals.
        x_shape = (batch_size, sequence_length, num_x_signals)
        x_batch = np.zeros(shape=x_shape, dtype=np.float16)

        # Allocate a new array for the batch of output-signals.
```

```
        y_shape = (batch_size, sequence_length, num_y_signals)
        y_batch = np.zeros(shape=y_shape, dtype=np.float16)

        # Fill the batch with random sequences of data.
        for i in range(batch_size):
            # Get a random start-index.
            # This points somewhere into the training-data.
            idx = np.random.randint(num_train - sequence_length)

            # Copy the sequences of data starting at this index.
            x_batch[i] = x_train_scaled[idx:idx+sequence_length]
            y_batch[i] = y_train_scaled[idx:idx+sequence_length]

        yield (x_batch, y_batch)
```

We will use a large batch-size so as to keep the GPU near 100% work-load. You may have to adjust this number depending on your GPU, its RAM and your choice of `sequence_length` below.

In [ ]:
```
batch_size = 256
```

We will use a sequence-length of 1344, which means that each random sequence contains observations for 8 weeks. One time-step corresponds to one hour, so 24 x 7 time-steps corresponds to a week, and 24 x 7 x 8 corresponds to 8 weeks.

In [ ]:
```
sequence_length = 24 * 7 * 8
sequence_length
```

Out[ ]:
```
1344
```

We then create the batch-generator.

In [ ]:
```
generator = batch_generator(batch_size=batch_size,
                            sequence_length=sequence_length)
```

We can then test the batch-generator to see if it works.

In [ ]:
```
x_batch, y_batch = next(generator)
```

This gives us a random batch of 256 sequences, each sequence having 1344 observations, and each observation having 20 input-signals and 3 output-signals.

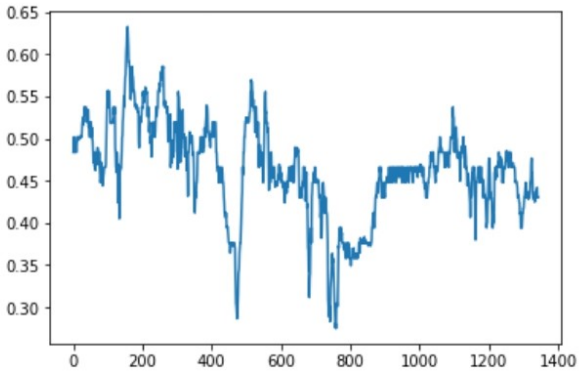
In [ ]:
```
print(x_batch.shape)
print(y_batch.shape)
```

```
(256, 1344, 20)
(256, 1344, 3)
```

We can plot one of the 20 input-signals as an example.

In [ ]:
```
batch = 0   # First sequence in the batch.
signal = 0  # First signal from the 20 input-signals.
seq = x_batch[batch, :, signal]
plt.plot(seq)
```

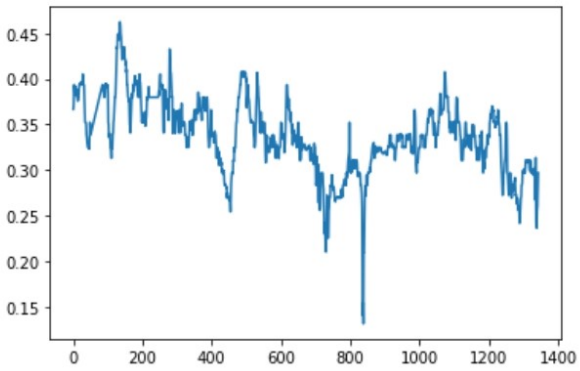
Out[ ]:
```
[<matplotlib.lines.Line2D at 0x7fcb3fc12898>]
```



We can also plot one of the output-signals that we want the model to learn how to predict given all those 20 input signals.

In [ ]:
```
seq = y_batch[batch, :, signal]
plt.plot(seq)
```

Out[ ]:
```
[<matplotlib.lines.Line2D at 0x7fcb3fb54cf8>]
```



Validation Set

The neural network trains quickly so we can easily run many training epochs. But then there is a risk of overfitting the model to the training-set so it does not generalize well to unseen data. We will therefore monitor the model's performance on the test-set after each epoch and only save the model's weights if the performance is improved on the test-set.

The batch-generator randomly selects a batch of short sequences from the training-data and uses that during training. But for the validation-data we will instead run through the entire sequence from the test-set and measure the prediction accuracy on that entire sequence.

In [ ]:
```
validation_data = (np.expand_dims(x_test_scaled, axis=0),
                   np.expand_dims(y_test_scaled, axis=0))
```

Create the Recurrent Neural Network

We are now ready to create the Recurrent Neural Network (RNN). We will use the Keras API for this because of its simplicity. See Tutorial #03-C for a tutorial on Keras and Tutorial #20 for more information on Recurrent Neural Networks.

```
```

In [ ]:
```
model = Sequential()
```

We can now add a Gated Recurrent Unit (GRU) to the network. This will have 512 outputs for each time-step in the sequence.

Note that because this is the first layer in the model, Keras needs to know the shape of its input, which is a batch of sequences of arbitrary length (indicated by `None`), where each observation has a number of input-signals (`num_x_signals`).

In [ ]:
```
model.add(GRU(units=512,
              return_sequences=True,
              input_shape=(None, num_x_signals,)))
```

The GRU outputs a batch of sequences of 512 values. We want to predict 3 output-signals, so we add a fully-connected (or dense) layer which maps 512 values down to only 3 values.

The output-signals in the data-set have been limited to be between 0 and 1 using a scaler-object. So we also limit the output of the neural network using the Sigmoid activation function, which squashes the output to be between 0 and 1.

In [ ]:
```
model.add(Dense(num_y_signals, activation='sigmoid'))
```

A problem with using the Sigmoid activation function, is that we can now only output values in the same range as the training-data.

For example, if the training-data only has temperatures between -20 and +30 degrees, then the scaler-object will map -20 to 0 and +30 to 1. So if we limit the output of the neural network to be between 0 and 1 using the Sigmoid function, this can only be mapped back to temperature values between -20 and +30.

We can use a linear activation function on the output instead. This allows for the output to take on arbitrary values. It might work with the standard initialization for a simple network architecture, but for more complicated network architectures e.g. with more layers, it might be necessary to initialize the weights with smaller values to avoid `NaN` values during training. You may need to experiment with this to get it working.

In [ ]:
```
if False:
    from tensorflow.python.keras.initializers import RandomUniform

    # Maybe use lower init-ranges.
    init = RandomUniform(minval=-0.05, maxval=0.05)

    model.add(Dense(num_y_signals,
                    activation='linear',
                    kernel_initializer=init))
```

Loss Function

We will use Mean Squared Error (MSE) as the loss-function that will be minimized. This measures how closely the model's output matches the true output signals.

However, at the beginning of a sequence, the model has only seen input-signals for a few time-steps, so its generated output may be very inaccurate. Using the loss-value for the early time-steps may cause the model to distort its later output. We therefore give the model a "warmup-period" of 50 time-steps where we don't use its accuracy in the loss-function, in hope of improving the accuracy for later time-steps.

In [ ]:
```
warmup_steps = 50
```

In [ ]:
```
def loss_mse_warmup(y_true, y_pred):
    """
    Calculate the Mean Squared Error between y_true and y_pred,
    but ignore the beginning "warmup" part of the sequences.

    y_true is the desired output.
    y_pred is the model's output.
    """

    # The shape of both input tensors are:
    # [batch_size, sequence_length, num_y_signals].

    # Ignore the "warmup" parts of the sequences
    # by taking slices of the tensors.
    y_true_slice = y_true[:, warmup_steps:, :]
    y_pred_slice = y_pred[:, warmup_steps:, :]

    # These sliced tensors both have this shape:
    # [batch_size, sequence_length - warmup_steps, num_y_signals]

    # Calculat the Mean Squared Error and use it as loss.
    mse = mean(square(y_true_slice - y_pred_slice))

    return mse
```

Compile Model

This is the optimizer and the beginning learning-rate that we will use.

In [ ]:
```
optimizer = RMSprop(lr=1e-3)
```

We then compile the Keras model so it is ready for training.

In [ ]:
```
model.compile(loss=loss_mse_warmup, optimizer=optimizer)
```

This is a very small model with only two layers. The output shape of `(None, None, 3)` means that the model will output a batch with an arbitrary number of sequences, each of which has an arbitrary number of observations, and each observation has 3 signals. This corresponds to the 3 target signals we want to predict.

In [ ]:
```
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
gru (GRU)                    (None, None, 512)         820224
_________________________________________________________________
dense (Dense)                (None, None, 3)           1539
=================================================================
Total params: 821,763
Trainable params: 821,763
Non-trainable params: 0
_________________________________________________________________
```

Callback Functions

During training we want to save checkpoints and log the progress to TensorBoard so we create the appropriate callbacks for Keras.

This is the callback for writing checkpoints during training.

In [ ]:
```
path_checkpoint = '23_checkpoint.keras'
callback_checkpoint = ModelCheckpoint(filepath=path_checkpoint,
                                      monitor='val_loss',
                                      verbose=1,
                                      save_weights_only=True,
                                      save_best_only=True)
```

This is the callback for stopping the optimization when performance worsens on the validation-set.

In [ ]:
```
callback_early_stopping = EarlyStopping(monitor='val_loss',
                                        patience=5, verbose=1)
```

This is the callback for writing the TensorBoard log during training.

In [ ]:
```
callback_tensorboard = TensorBoard(log_dir='./23_logs/',
                                   histogram_freq=0,
                                   write_graph=False)
```

This callback reduces the learning-rate for the optimizer if the validation-loss has not improved since the last epoch (as indicated by `patience=0`). The learning-rate will be reduced by multiplying it with the given factor. We set a start learning-rate of 1e-3 above, so multiplying it by 0.1 gives a learning-rate of 1e-4. We don't want the learning-rate to go any lower than this.

In [ ]:
```
callback_reduce_lr = ReduceLROnPlateau(monitor='val_loss',
                                       factor=0.1,
                                       min_lr=1e-4,
                                       patience=0,
                                       verbose=1)
```

In [ ]:
```
callbacks = [callback_early_stopping,
             callback_checkpoint,
             callback_tensorboard,
             callback_reduce_lr]
```

Train the Recurrent Neural Network

We can now train the neural network.

Note that a single "epoch" does not correspond to a single processing of the training-set, because of how the batch-generator randomly selects sub-sequences from the training-set. Instead we have selected `steps_per_epoch` so that one "epoch" is processed in a few minutes.

With these settings, each "epoch" took about 2.5 minutes to process on a GTX 1070. After 14 "epochs" the optimization was stopped because the validation-loss had not decreased for 5 "epochs". This optimization took about 35 minutes to finish.

Also note that the loss sometimes becomes `NaN` (not-a-number). This is often resolved by restarting and running the Notebook again. But it may also be caused by your neural network architecture, learning-rate, batch-size, sequence-length, etc. in which case you may have to modify those settings.

In [ ]:
```
%%time
model.fit(x=generator,
          epochs=20,
          steps_per_epoch=100,
          validation_data=validation_data,
          callbacks=callbacks)
```

```
WARNING:tensorflow:sample_weight modes were coerced from
  ...
    to
  ['...']
Train for 100 steps, validate on 1 samples
Epoch 1/20
 99/100 [============================>.] - ETA: 0s - loss: 0.0086
Epoch 00001: val_loss improved from inf to 0.00398, saving model to 23_checkpoint.keras
100/100 [==============================] - 68s 684ms/step - loss: 0.0085 - val_loss: 0.0040
Epoch 2/20
 99/100 [============================>.] - ETA: 0s - loss: 0.0048
Epoch 00002: val_loss did not improve from 0.00398

Epoch 00002: ReduceLROnPlateau reducing learning rate to 0.00010000000474974513.
100/100 [==============================] - 71s 713ms/step - loss: 0.0048 - val_loss: 0.0043
Epoch 3/20
 99/100 [============================>.] - ETA: 0s - loss: 0.0031
Epoch 00003: val_loss improved from 0.00398 to 0.00258, saving model to 23_checkpoint.keras
100/100 [==============================] - 71s 712ms/step - loss: 0.0031 - val_loss: 0.0026
Epoch 4/20
 99/100 [============================>.] - ETA: 0s - loss: 0.0029
Epoch 00004: val_loss improved from 0.00258 to 0.00250, saving model to 23_checkpoint.keras

Epoch 00004: ReduceLROnPlateau reducing learning rate to 0.0001.
100/100 [==============================] - 67s 670ms/step - loss: 0.0029 - val_loss: 0.0025
Epoch 5/20
 99/100 [============================>.] - ETA: 0s - loss: 0.0028
Epoch 00005: val_loss improved from 0.00250 to 0.00248, saving model to 23_checkpoint.keras
100/100 [==============================] - 71s 713ms/step - loss: 0.0028 - val_loss: 0.0025
Epoch 6/20
 99/100 [============================>.] - ETA: 0s - loss: 0.0028
Epoch 00006: val_loss improved from 0.00248 to 0.00243, saving model to 23_checkpoint.keras
100/100 [==============================] - 68s 678ms/step - loss: 0.0028 - val_loss: 0.0024
Epoch 7/20
 99/100 [============================>.] - ETA: 0s - loss: 0.0027
Epoch 00007: val_loss did not improve from 0.00243
100/100 [==============================] - 65s 651ms/step - loss: 0.0027 - val_loss: 0.0024
Epoch 8/20
 99/100 [============================>.] - ETA: 0s - loss: 0.0027
Epoch 00008: val_loss did not improve from 0.00243
100/100 [==============================] - 65s 650ms/step - loss: 0.0027 - val_loss: 0.0024
Epoch 9/20
 99/100 [============================>.] - ETA: 0s - loss: 0.0027
Epoch 00009: val_loss improved from 0.00243 to 0.00239, saving model to 23_checkpoint.keras
100/100 [==============================] - 65s 652ms/step - loss: 0.0027 - val_loss: 0.0024
Epoch 10/20
 99/100 [============================>.] - ETA: 0s - loss: 0.0026
Epoch 00010: val_loss improved from 0.00239 to 0.00239, saving model to 23_checkpoint.keras
100/100 [==============================] - 65s 650ms/step - loss: 0.0026 - val_loss: 0.0024
Epoch 11/20
 99/100 [============================>.] - ETA: 0s - loss: 0.0026
Epoch 00011: val_loss improved from 0.00239 to 0.00231, saving model to 23_checkpoint.keras
100/100 [==============================] - 65s 652ms/step - loss: 0.0026 - val_loss: 0.0023
Epoch 12/20
 99/100 [============================>.] - ETA: 0s - loss: 0.0026
Epoch 00012: val_loss improved from 0.00231 to 0.00229, saving model to 23_checkpoint.keras
100/100 [==============================] - 65s 651ms/step - loss: 0.0026 - val_loss: 0.0023
Epoch 13/20
 99/100 [============================>.] - ETA: 0s - loss: 0.0026
Epoch 00013: val_loss improved from 0.00229 to 0.00228, saving model to 23_checkpoint.keras
100/100 [==============================] - 65s 652ms/step - loss: 0.0026 - val_loss: 0.0023
Epoch 14/20
 99/100 [============================>.] - ETA: 0s - loss: 0.0026
Epoch 00014: val_loss did not improve from 0.00228
100/100 [==============================] - 65s 653ms/step - loss: 0.0026 - val_loss: 0.0023
Epoch 15/20
 99/100 [============================>.] - ETA: 0s - loss: 0.0026
Epoch 00015: val_loss did not improve from 0.00228
100/100 [==============================] - 65s 653ms/step - loss: 0.0026 - val_loss: 0.0023
```

```
Epoch 16/20
 99/100 [============================>.] - ETA: 0s - loss: 0.0025
Epoch 00016: val_loss did not improve from 0.00228
100/100 [==============================] - 66s 657ms/step - loss: 0.0025 - val_loss: 0.0023
Epoch 17/20
 99/100 [============================>.] - ETA: 0s - loss: 0.0025
Epoch 00017: val_loss did not improve from 0.00228
100/100 [==============================] - 67s 665ms/step - loss: 0.0025 - val_loss: 0.0023
Epoch 18/20
 99/100 [============================>.] - ETA: 0s - loss: 0.0025
Epoch 00018: val_loss did not improve from 0.00228
100/100 [==============================] - 69s 685ms/step - loss: 0.0025 - val_loss: 0.0023
Epoch 00018: early stopping
CPU times: user 15min 17s, sys: 4min 8s, total: 19min 26s
Wall time: 20min 4s
```

Out[ ]:
```
<tensorflow.python.keras.callbacks.History at 0x7fcb3e686940>
```

Load Checkpoint

Because we use early-stopping when training the model, it is possible that the model's performance has worsened on the test-set for several epochs before training was stopped. We therefore reload the last saved checkpoint, which should have the best performance on the test-set.

In [ ]:
```
try:
    model.load_weights(path_checkpoint)
except Exception as error:
    print("Error trying to load checkpoint.")
    print(error)
```

Performance on Test-Set

We can now evaluate the model's performance on the test-set. This function expects a batch of data, but we will just use one long time-series for the test-set, so we just expand the array-dimensionality to create a batch with that one sequence.

In [ ]:
```
result = model.evaluate(x=np.expand_dims(x_test_scaled, axis=0),
                        y=np.expand_dims(y_test_scaled, axis=0))
```

```
1/1 [==============================] - 1s 729ms/sample - loss: 0.0023
```

In [ ]:
```
print("loss (test-set):", result)
```

```
loss (test-set): 0.002279780339449644
```

In [ ]:
```
# If you have several metrics you can use this instead.
if False:
    for res, metric in zip(result, model.metrics_names):
        print("{0}: {1:.3e}".format(metric, res))
```

Generate Predictions

This helper-function plots the predicted and true output-signals.

In [ ]:
```
def plot_comparison(start_idx, length=100, train=True):
    """
    Plot the predicted and true output-signals.

    :param start_idx: Start-index for the time-series.
    :param length: Sequence-length to process and plot.
    :param train: Boolean whether to use training- or test-set.
```

```
    """

    if train:
        # Use training-data.
        x = x_train_scaled
        y_true = y_train
    else:
        # Use test-data.
        x = x_test_scaled
        y_true = y_test

    # End-index for the sequences.
    end_idx = start_idx + length

    # Select the sequences from the given start-index and
    # of the given length.
    x = x[start_idx:end_idx]
    y_true = y_true[start_idx:end_idx]

    # Input-signals for the model.
    x = np.expand_dims(x, axis=0)

    # Use the model to predict the output-signals.
    y_pred = model.predict(x)

    # The output of the model is between 0 and 1.
    # Do an inverse map to get it back to the scale
    # of the original data-set.
    y_pred_rescaled = y_scaler.inverse_transform(y_pred[0])

    # For each output-signal.
    for signal in range(len(target_names)):
        # Get the output-signal predicted by the model.
        signal_pred = y_pred_rescaled[:, signal]

        # Get the true output-signal from the data-set.
        signal_true = y_true[:, signal]

        # Make the plotting-canvas bigger.
        plt.figure(figsize=(15,5))

        # Plot and compare the two signals.
        plt.plot(signal_true, label='true')
        plt.plot(signal_pred, label='pred')

        # Plot grey box for warmup-period.
        p = plt.axvspan(0, warmup_steps, facecolor='black', alpha=0.15)

        # Plot labels etc.
        plt.ylabel(target_names[signal])
        plt.legend()
        plt.show()
```

We can now plot an example of predicted output-signals. It is important to understand what these plots show, as they are actually a bit more complicated than you might think.

These plots only show the output-signals and not the 20 input-signals used to predict the output-signals. The time-shift between the input-signals and the output-signals is held fixed in these plots. The model **always** predicts the output-signals e.g. 24 hours into the future (as defined in the `shift_steps` variable above). So the plot's x-axis merely shows how many time-steps of the input-signals have been seen by the predictive model so far.

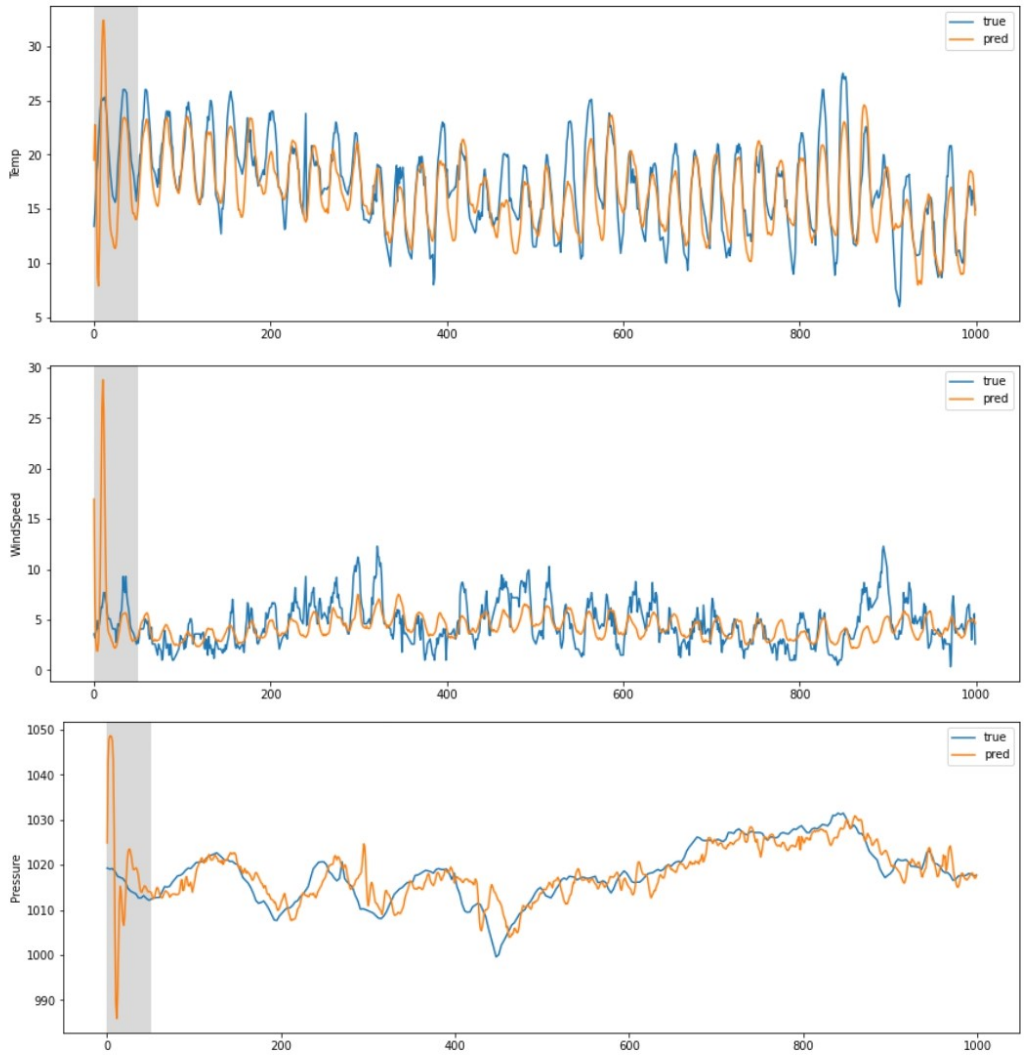
The prediction is not very accurate for the first 30-50 time-steps because the model has seen very little input-data at this point. The model generates a single time-step of output data for each time-step of the input-data, so when the model has only run for a few time-steps, it knows very little of the history of the

input-signals and cannot make an accurate prediction. The model needs to "warm up" by processing perhaps 30-50 time-steps before its predicted output-signals can be used.

That is why we ignore this "warmup-period" of 50 time-steps when calculating the mean-squared-error in the loss-function. The "warmup-period" is shown as a grey box in these plots.

Let us start with an example from the training-data. This is data that the model has seen during training so it should perform reasonably well on this data.

In [ ]:
```
plot_comparison(start_idx=100000, length=1000, train=True)
```



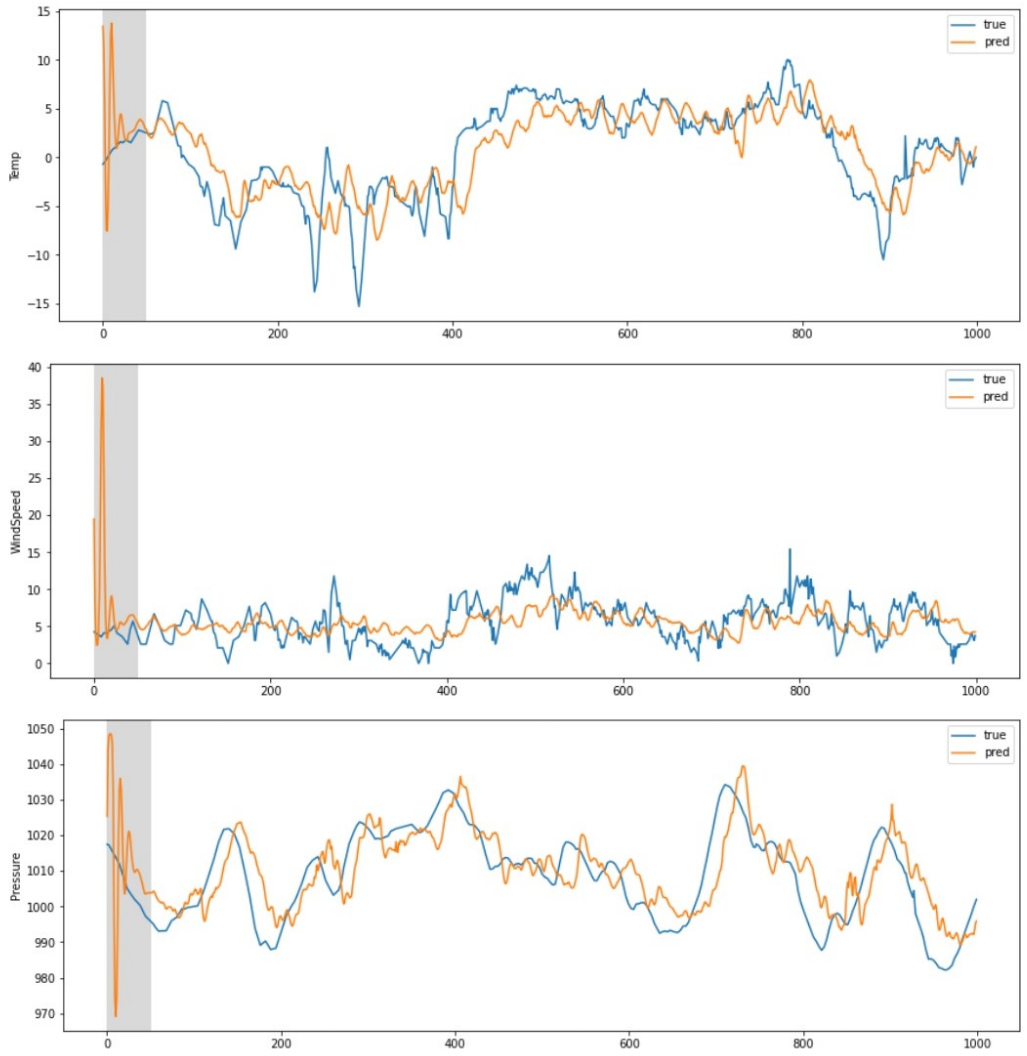
The model was able to predict the overall oscillations of the temperature quite well but the peaks were sometimes inaccurate. For the wind-speed, the overall oscillations are predicted reasonably well but the peaks are quite inaccurate. For the atmospheric pressure, the overall curve-shape has been predicted although there seems to be a slight lag and the predicted curve has a lot of noise compared to the smoothness of the original signal.

Strange Example

The following is another example from the training-set.

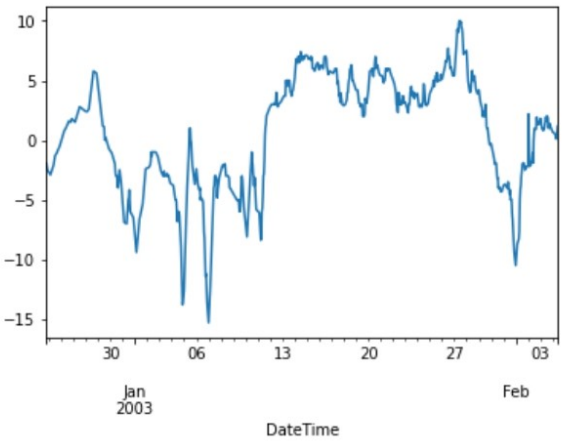
Note how the temperature does not oscillate very much within each day (this plot shows almost 42 days). The temperature normally oscillates within each day, see e.g. the plot above where the daily temperature-oscillation is very clear. It is unclear whether this period had unusually stable temperature, or if perhaps there's a data-error.

In [ ]:
```
plot_comparison(start_idx=200000, length=1000, train=True)
```



As a check, we can plot this signal directly from the resampled data-set, which looks similar.

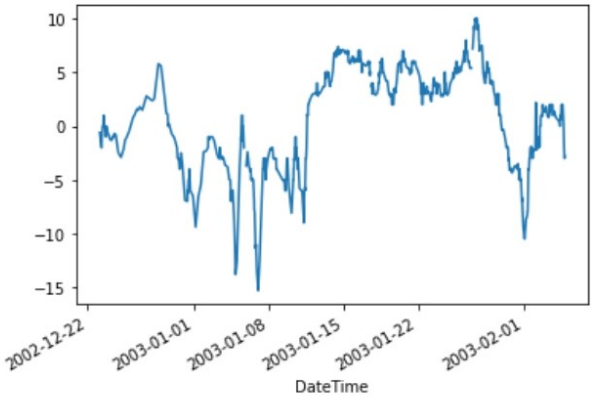
In [ ]:
```
df['Odense']['Temp'][200000:200000+1000].plot();
```



We can plot the same period from the original data that has not been resampled. It also looks similar.

So either the temperature was unusually stable for a part of this period, or there is a data-error in the raw data that was obtained from the internet weather-database.

In [ ]:
```
df_org = weather.load_original_data()
df_org.xs('Odense')['Temp']['2002-12-23':'2003-02-04'].plot();
```



Example from Test-Set

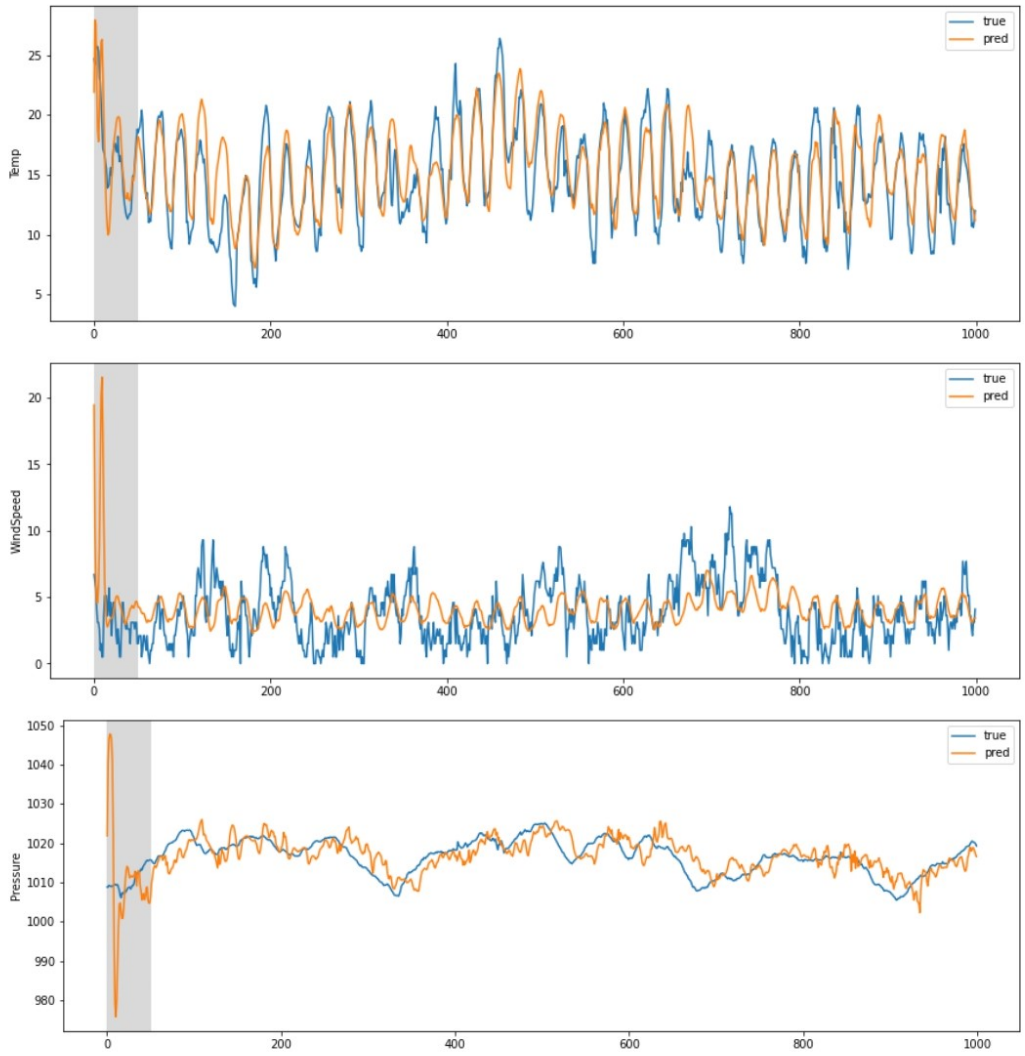
Now consider an example from the test-set. The model has not seen this data during training.

The temperature is predicted reasonably well, although the peaks are sometimes inaccurate.

The wind-speed has not been predicted so well. The daily oscillation-frequency seems to match, but the center-level and the peaks are quite inaccurate. A guess would be that the wind-speed is difficult to predict from the given input data, so the model has merely learnt to output sinusoidal oscillations in the daily frequency and approximately at the right center-level.

The atmospheric pressure is predicted reasonably well, except for a lag and a more noisy signal than the true time-series.

In [ ]:
```
plot_comparison(start_idx=200, length=1000, train=False)
```



Conclusion

This tutorial showed how to use a Recurrent Neural Network to predict several time-series from a number of input-signals. We used weather-data for 5 cities to predict tomorrow's weather for one of the cities.

It worked reasonably well for predicting the temperature where the daily oscillations were predicted well, but the peaks were sometimes not predicted so accurately. The atmospheric pressure was also predicted reasonably well, although the predicted signal was more noisy and had a short lag. The wind-speed could not be predicted very well.

You can use this method with different time-series but you should be careful to distinguish between *causation and correlation* in the data. The neural network may easily discover patterns in the data that are only temporary correlations which do not generalize well to unseen data.

You should select input- and output-data where a *causal* relationship probably exists. You should have a lot of data available for training, and you should try and reduce the risk of over-fitting the model to the training-data, e.g. using early-stopping as we did in this tutorial.

Exercises

These are a few suggestions for exercises that may help improve your skills with TensorFlow. It is important to get hands-on experience with TensorFlow in order to learn how to use it properly.

You may want to backup this Notebook before making any changes.

- Remove the wind-speed from the target-data. Does it improve prediction for the temperature and pressure?
- Train for more epochs, possibly with a lower learning-rate. Does it improve the performance on the test-set?
- Try a different architecture for the neural network, e.g. higher or lower state-size for the GRU layer, more GRU layers, dense layers before and after the GRU layers, etc.
- Use hyper-parameter optimization from Tutorial #19.
- Try using longer and shorter sequences for the batch-generator.
- Try and remove the city "Odense" from the input-signals.
- Try and add last year's weather-data to the input-signals.
- How good is the model at predicting the weather 3 or 7 days into the future?
- Can you train a single model with the output-signals for multiple time-shifts, so that a single model predicts the weather in e.g. 1, 3 and 7 days.
- Explain to a friend how the program works.

License (MIT)

Fonte: https://github.com/Hvass-Labs/TensorFlow-Tutorials/blob/master/23_Time-Series-Prediction.ipynb