Universidade Federal de Santa Catarina
Campus Florianópolis
CURSO DE GRADUAÇÃO EM CIÊNCIAS DA COMPUTAÇÃO

João Janini Ribeiro

Classificação de Litofácies Utilizando Transformers

Florianópolis
2022

João Janini Ribeiro

Classificação de Litofácies Utilizando Transformers

Trabalho de Conclusão de Curso do Curso de Graduação em Ciências da Computação do Campus Florianópolis da Universidade Federal de Santa Catarina para a obtenção do título de bacharel em Ciências da Computação.
Orientador: Prof. Dr. Mauro Roisenberg

Universidade Federal de Santa Catarina
Campus de Florianópolis
CURSO DE GRADUAÇÃO EM CIÊNCIAS DA COMPUTAÇÃO

Florianópolis
2022

João Janini Ribeiro

Classificação de Litofácies Utilizando Transformers

Trabalho de Conclusão de Curso do Curso de Graduação em Ciências da Computação do Campus Florianópolis da Universidade Federal de Santa Catarina para a obtenção do título de bacharel em Ciências da Computação.

Comissão Examinadora

Prof. Dr. Mauro Roisenberg
Universidade Federal de Santa Catarina
Orientador

Prof. Dr. Mateus Grellert da Silva
Universidade Federal de Santa Catarina

Prof. Dr. Rafael de Santiago
Universidade Federal de Santa Catarina

Florianópolis, 24 de dezembro de 2022

Agradecimentos

Agradeço primeiramente ao meu orientador Prof. Dr. Mauro Roisenberg por ter me acolhido e me orientado no aprendizado de machine learning desde o início. Serie eternamente grato pelas lições e pela paciência. Também quero agradecer à minha namorada e parceira de curso, Luana Dalmarco Fronza, que desde o começo até o último momento esteve do meu lado dando suporte nos desafios que enfretei durante a realização desse trabalho. Agradeço ao meu colega Lucas Barzan pelas revisões detalhistas que me salvaram em diversos momentos. Também agradeço meu pai e minha mãe que sempre me ofereceram grandes ensinamentos, além da melhor educação possível, o que me permitiu chegar até aqui. Finalmente agradeço meu avô Dr. Janini, que além de ser uma fonte inacabável de conhecimento e curiosidade, sempre buscou me ensinar a pensar sobre o mundo.

Resumo

O problema de classificação de litofácies em poços envolve a identificação das características das rochas contidas em poços de perfuração. Essa classificação pode ser utilizada na identificação de recursos como gás e petróleo, e por isso é uma atividade de valor econômico significativo. Uma das maneiras mais comuns de se fazer essa classificação é através do uso de *Wireline Logs*. Essas medidas feitas por sondas contém diversos atributos das rochas que ajudam em sua identificação. A partir desses atributos coletados, é possível usá-los como entradas de algoritmos de aprendizado de máquina, que resultam em modelos preditivos que podem ser utilizados para auxiliar geofísicos em suas análises. Os tipos de algoritmos utilizados na área variam bastante, com os métodos envolvendo árvores de decisão sendo os mais comumente utilizados, tanto em artigos quanto em competições. Esses métodos, especialmente o *XGBoost*, são bastante eficazes e conseguem níveis de acurácia significativos quando utilizados. No entanto, se sabe que esses métodos não capturam uma dimensão essencial dos dados dos poços: a relação sequencial entre as amostras. Seguindo essa ideia, este trabalho explora o uso de *Transformers* para a classificação de litofácies a partir de *wirelines logs*. Os *Transformers* são uma arquitetura mais recente, e vêm conquistando o estado da arte em diferentes campos envolvendo dados sequenciais. Portanto, este trabalho propôs duas implementações de *transformers* que servissem para o problema de classificação de litofácies: uma de "sequência-para-sequência"e a outra de "sequência-para-um". Ambas as implementações foram testadas utilizando os dados da competição XEEK de 2020 e tiveram suas métricas comparadas tanto com um modelo base envolvendo *XGBoost*, quanto com os resultados obtidos pelo ganhador da competição. Os resultados obtidos comprovam a efetividade da arquitetura na resolução do problema, e as implementações apresentadas poderão servir como base para trabalhos futuros.

Palavras-Chave: 1. Transformers. 2. Litofácies. 3. Wireline Logs.

Abstract

The lithofacies classification problem in wells involves identifying the characteristics of the rocks contained in the drilling wells. This classification can be used to identify resources such as gas and oil, which is why it is an activity of significant economic value. One of the most common ways to do this classification is through the use of *Wireline Logs*. These measurements made by probes contain several attributes of the rocks that help in their identification. From these collected attributes, it is possible to use them as inputs for machine learning algorithms, which result in predictive models that can be used to assist geophysicists in their analyses. The types of algorithms used in the area vary greatly, with methods involving decision trees being the most used, both in articles and in contests. These methods, especially *XGBoost*, are very effective and achieve significant levels of accuracy when used. However, these methods are known to fail to capture an essential dimension of well data, the sequential relationship between samples.Following this idea, this work explored the use of Transformers for the classification of wireline profile lithofacies. Transformers are a more recent architecture, and have been conquering the state of the art in several fields involving sequential data. Therefore, this work proposed two implementations of transformers that would serve for the lithofacies classification problem, one "sequence-by-sequence" and another "sequence-by-one". Both implementations were tested with data from the XEEK 2020 competition, and had their metrics compared with a base model involving *XGBoost*, as well as with the results obtained by the winner of the competition. The results obtained prove the effectiveness of the architecture in solving the problem, and the implementations presented can serve as a basis for future work.

Keywords: 1. Transformers. 2. Lithofacies. 3. Wireline Logs .

Lista de figuras

Lista de tabelas

Lista de Siglas e Abreviaturas

UFSC	*Universidade Federal de Santa Catarina*
RNN	*Recurrent Neural Networks*
LSTM	*Long Short-Term Memory*
BiLSTM	*Bidirectional Long Short-Term Memory*
CNN	*Convolutional Neural Network)*

Sumário

1 Introdução

Litofácies são subdivisões de sequências sedimentares que possuem uma relação direta com os processos sedimentares que as formaram. A classificação correta dessas subdivisões é de elevado valor econômico. Uma das práticas que auxiliam nessa classificação é o *well logging*. A prática do *well logging* envolve a obtenção de descrições detalhadas sobre as formações de rocha em diferentes níveis e profundidades de um poço, com o objetivo principal de classificação das litofácies em cada profundidade. *Well logging* é essencial para a indústria de hidrocarbonetos, que usa a classificação de litofácies durante todo o processo de extração de recursos, como gás natural e petróleo (BRITTANICA, 2018).

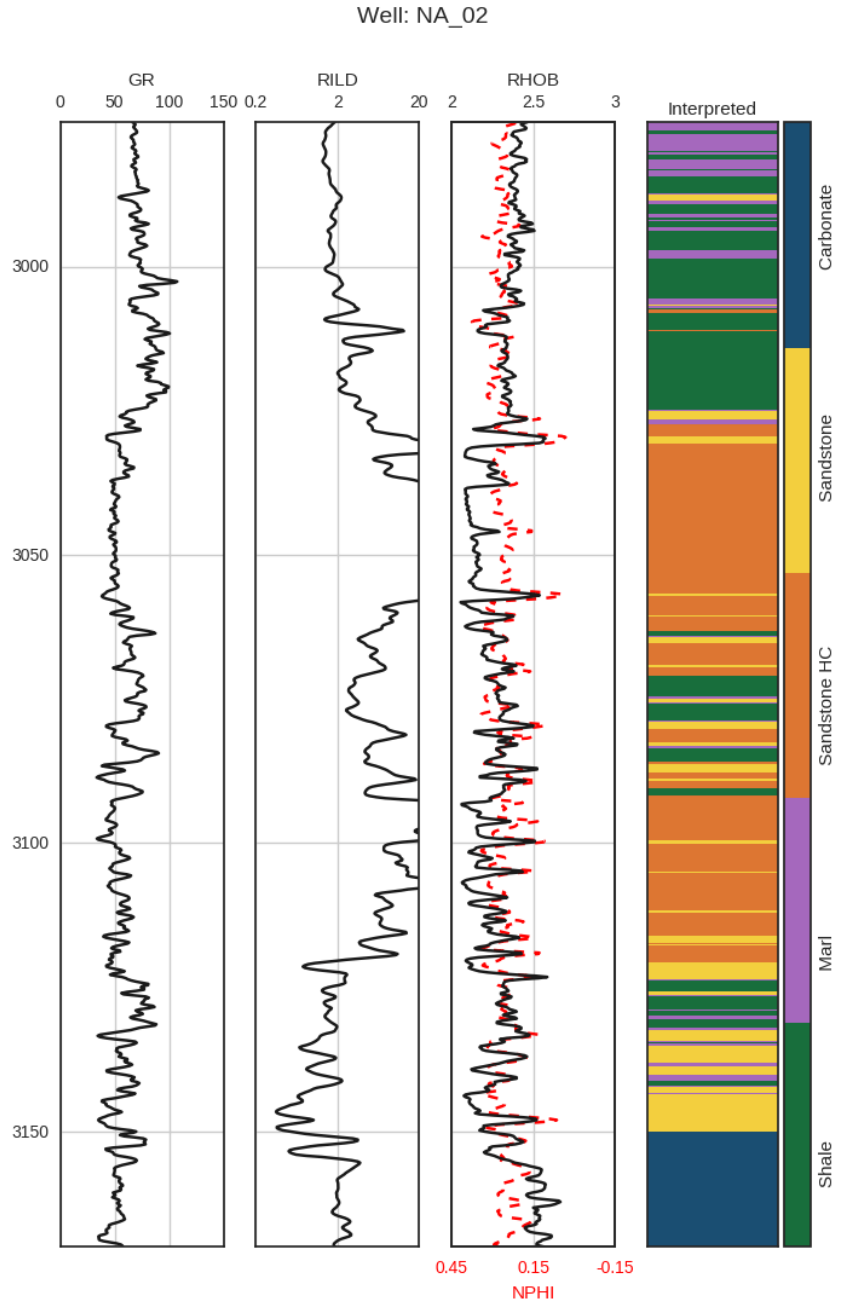
A Figura 1 mostra como diferentes curvas de *logs* se relacionam a diferentes litofácies, ao longo da profundidade de um poço. A prática de *well logging* gera *well logs*, que vão desde descrições físicas e químicas até radioativas sobre as rochas de um poço. Seu processo de obtenção envolve o uso de tecnologias sofisticadas e robustas por causa das condições não favoráveis presentes em poços, juntamente com a complexidade das propriedades de reservatórios. Hidrocarbonetos, por exemplo, são normalmente descobertos em ambientes remotos e criam extrema dificuldade de exploração, mais uma vez justificando o contínuo desenvolvimento de técnicas nessa indústria (Boggs, 2001).

As medições de perfil de poço podem ser comparadas entre si e com medições diretas a partir de amostras de testemunhos (amostras físicas retiradas diretamente dos poços) para desenvolver relações petrofísicas específicas do local. Os dados de perfil também são úteis para vincular dados hidrológicos e geológicos coletados no local do furo de poço com assinaturas geofísicas de variações de propriedade coletadas usando dados.

O problema de classificação de litofácies pode ser formulado como a detecção de uma litofácie em uma determinada profundidade, a partir da análise de conjuntos de *well logs* obtidos naquela profundidade (Bestagini et al., 2017). Essa formulação permite o uso de métodos de aprendizado de máquina, que têm cada vez mais se popularizado na área.

Os métodos utilizados no problema são bem diversos, incluindo *deep learning*, SVG (Support Vector Machines), regressão logística e, principalmente, métodos baseados em árvores de decisão, como o XGBoost (Zhang and Zhan, 2017) e *random forest*. Os resultados de competições em torno do problema ilustram bem a popularidade de métodos como XGBoost na área. Na competição organizada pela SEG *(Society of Exploration Geophisycist)* de 2016 (Hall and Hall, 2017), todos os 5 melhores classificados utilizaram **Boosted Trees** como é mostrado na Tabela 1. Em particular, foi utilizado o *XGBoost* (Chen and Guestrin, 2016), que é uma implementação bastante popular de *Boosted Trees* que usa de métodos de descida de gradiente para encontrar as melhores árvores de decisão para o problema.

Figura 1 – Exemplo de gráfico de um *well log* para um trecho de profundidade específico. A coluna *interpreted* representa a litofácie naquela profundidade, enquanto as outras colunas são exemplos de wireline logs.



Fonte: (Boggs, 2001)

Tabela 1 – Resultados obtidos pelas equipes vencedoras da competição de 2016 e os algoritmos utilizados.

Position	Team	F1	Algorithm	Language
1	LA_Team	**0.6388**	Boosted trees	Python
2	PA Team	**0.6250**	Boosted trees	Python
3	ispl (Bestagini, Tuparo, Lipari)	**0.6231**	Boosted trees	Python
4	esaTeam	**0.6225**	Boosted trees	Python

Fonte: O autor (2022).

Essa mesma tendência se manteve na competição de 2020 organizada pela XEEK (Bormann et al., 2020a), com os 5 mais bem colocados utilizando métodos baseados em árvore, dos quais 4 eram o XGBoost e 1 utilizou *random forest*. A principal diferença entre as técnicas utilizadas pelos times ganhadores desde a competição do SEG girou em torno de técnicas de pré-processamento, que foram significativamente mais elaboradas. Porém, o uso de algoritmos de árvore de decisão se mantém como uma das principais opções.

Entretanto, ainda que essas abordagens consigam atingir níveis de acurácia significativos, o problema continua sendo bastante complexo por vários motivos, como por exemplo: as diferenças de resolução entre poços e a sobreposição dos valores de fácies sobre si mesmas. Além disso, ainda existe a questão de tipos de rocha com funções diferentes dependendo do ambiente de exploração ou produção. Dentre as possíveis explicações para esses problemas, podemos citar a falta de informação geológica e a ausência de sequências de fácies. Essas técnicas ignoram que existe uma correlação entre camadas vizinhas e que padrões de fácies empilhados são significativos para interpretações geológicas (Jaikla et al., 2019).

Para essa questão, (Bestagini et al., 2017) propõem o uso de gradientes de algumas das curvas como atributos adicionais ao modelo. Os melhores colocados das competições XEEK e SEG citadas utilizaram esses atributos adicionais como entrada dos modelos, indicando de fato vantagens claras na abordagem temporal do problema.

A fim de trazer essa abordagem para os algoritmos de aprendizado em si, alguns trabalhos utilizaram CNNs (*Convolutional Neural Network*) para capturar as características espaciais entre as amostras, atingindo resultados satisfatórios. (Theisges et al., 2021) trouxeram resultados positivos utilizando BiLSTM (*Bidirectional Long Short-Term Memory*).

Finalmente, (Liu and Liu, 2022) reconheceram que enquanto CNNs são boas para extrair atributos espaciais, LSTMs são eficientes com problemas envolvendo séries temporais e propuseram uma solução que combina as duas, conseguindo resultados significativos. Esses e outros trabalhos vêm trazendo resultados promissores para algoritmos envolvendo sequências. (Jaikla et al., 2019) também segue uma abordagem parecida e propõem uma arquitetura que também une BRNNs (*Bidirectional Recurrent Neural Networks*) e CNNs.

Algoritmos como CNN, LSTM e também GRU (*Gated Recurrent Units*) compõem o grupo de algoritmos de *deep learning* que incorporam a relação temporal ou espacial entre os dados de entrada de maneira nativa. Além deles, atualmente os *transformers* (Vaswani et al., 2017) vêm ganhando notoriedade por atingirem resultados considerados estados da arte em problemas envolvendo dados sequenciais, principalmente no ramo de NLP (*Natural Language Processing*). (Lin et al., 2021) Diferente das RNNs (*Recurrent Neural Network*), que usam recursão para capturar relações globais entre *input* e *output*, transformadores usam mecanismos de *attention*, em especial, o *self-attention*. Esse

mecanismo traz diversas vantagens aos transformadores em relação à RNN e CNN nos seguintes atributos:

- O total de complexidade computacional por camada;
- A quantidade de computação que pode ser paralelizada (número de operações sequenciais necessárias);
- O tamanho dos caminhos entre qualquer combinação de sequências de saída e de entrada. Ligado à capacidade do modelo de aprender dependências distantes.

Essa capacidade de capturar sequências longas contribuiu para a formação de uma literatura extensa sobre o uso de transformadores nos problemas de classificação e predição de séries temporais. Seguindo essa linha, a fim de contribuir para a literatura de classificação de litofácies utilizando métodos sequenciais, este trabalho busca adaptar diferentes arquiteturas de transformadores ao problema de predição de fácies litológicas.

Na sequência, foi testada a eficiência da abordagem escolhida utilizando os dados da competição XEEK (Bormann et al., 2020a). Os resultados são comparados contra os resultados obtidos pelos primeiros colocados a partir da métrica proposta pela competição. No conhecimento do autor até o momento da escrita, este é o primeiro trabalho que estuda a eficácia dos transformadores no problema de classificação de litofácies.

1.1 Objetivos

O desenvolvimento de um novo método de classificação para fácies litológicas exige a elaboração de algumas tarefas. Estas tarefas refletem o objetivo geral e se desdobram em objetivos específicos, os quais são descritos a seguir.

1.1.1 Objetivos Gerais

Propor um método de classificação de fácies litológicas utilizando transformadores.

1.1.2 Objetivos Específicos

- Estudar as técnicas usadas atualmente para predição de séries temporais utilizando transformadores;
- Explorar maneiras de pré-processar os dados de entrada para séries temporais;
- Trazer diferentes abordagens de classificação de séries temporais usando transformadores, com base na literatura;

- Propor diferentes modelos de transformadores adaptados À classificação de litofácies;
- Treinar os modelos em cima dos dados da competição XEEK;
- Avaliar os modelos utilizando as métricas propostas pela competição, compará-los entre si e com os resultados dos outros participantes.

1.2 Método de Pesquisa

O método de pesquisa é composto por três etapas:

- A primeira etapa foi a etapa de pesquisa. Nessa etapa, foi estudada tanto a literatura de transformadores, quanto a de classificação de litofácies, tendo se compreendido as abordagens de diversos trabalhos relevantes para as áreas. O principal objetivo foi obter material suficiente para construir uma possível abordagem para a classificação de litofácies utilizando transformadores. A partir da pesquisa, foram levantados trabalhos principalmente nas áreas de séries temporais e dados tabulares. Ambas propõem abordagens utilizando transformadores para dados de entrada contínuos e categóricos, de maneira paralela ao problema sendo estudado. Foram estudados também outros trabalhos de classificação de litofácies que, embora não usassem da arquitetura estudada, poderiam servir como parâmetro. Além disso, também foi feita uma análise exploratória acerca dos dados da competição.
- Na segunda etapa, diversas soluções foram construídas a partir dos trabalhos estudados. O principal objetivo foi propor soluções viáveis a partir das pesquisas. Essas soluções foram avaliadas principalmente de acordo com a complexidade da arquitetura e de seus componentes, pelo custo e tempo de acordo com o escopo do trabalho e finalmente pela disponibilidade de materiais. Alguns trabalhos foram desconsiderados pela falta de material disponibilizado pelos autores. As soluções também deveriam ser capazes de atingir resultados minimamente aceitáveis para o problema.
- Na terceira etapa, cada modelo passou pelo método padrão de projetos de aprendizado de máquina: desde o pré-processamento dos dados e a otimização de hiperparâmetros, até o treino do modelo final contendo os parâmetros propostos. Uma vez treinados, os modelos foram comparados entre si e com os outros participantes da competição. Essa avaliação foi feita utilizando a matriz de erro, acurácia e o *F-1-score*, propostos pela competição. Para o caso dos transformadores propostos por esse trabalho, parâmetros como tamanho de sequência e número de épocas foram controlados, buscando trazer clareza aos resultados.

1.3 Estrutura do Trabalho

O capítulo seguinte aborda conceitos importantes para suprir o embasamento teórico deste trabalho. O terceiro capítulo traz alguns trabalhos com soluções relevantes ao tema. O capítulo 4 apresenta como foi o desenvolvimento e o quinto capítulo explora os resultados obtidos. Por fim, o capítulo 6 conclui o trabalho.

2 Fundamentação Teórica

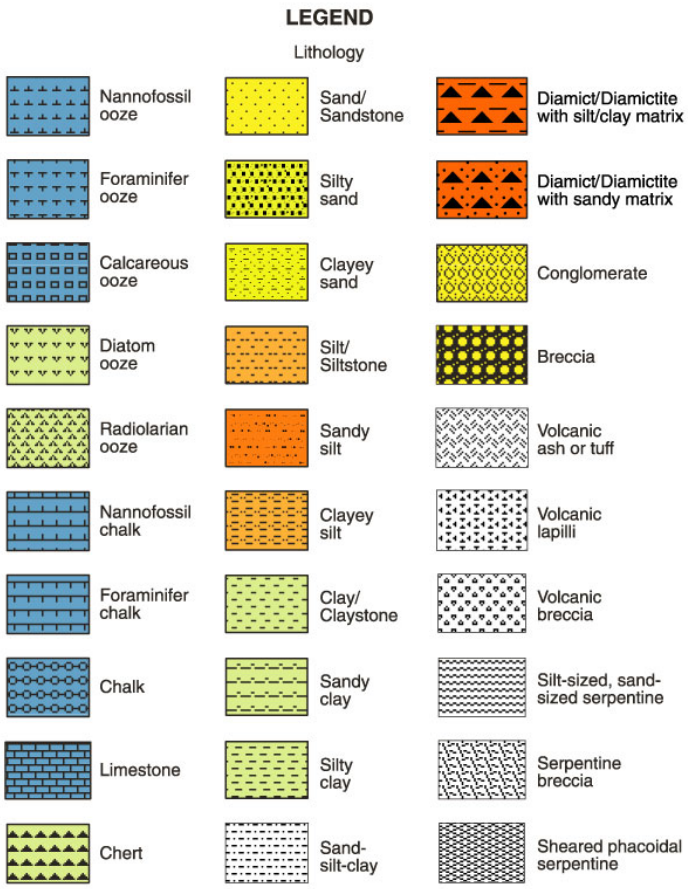
Esse capítulo traz conceitos relevantes para o embasamento teórico que leva ao entendimento deste trabalho. A seguir, serão apresentados alguns pontos que envolvem litofácies, transformadores, RNNs, *decision trees*, entre outros.

2.1 Litofácies

Rochas sedimentares são formadas onde sedimentos são depositados durante tempo suficiente para ficarem compactadas e sedimentadas na forma estrata (BRITTANICA, 2018).

Normalmente, isso se dá quando os sedimentos permanecem intocados por vários anos nas bacias sedimentares. Essas bacias podem ocupar um pequeno espaço ou até milhares de quilômetros e possuir diferentes ambientes deposicionais locais. A justaposição dessas diferentes rochas contemporâneas sedimentares define as litofácies (BRITTANICA, 2018). A Figura 2 ilustra alguns exemplos de litofácies com representações visuais.

Figura 2 – Exemplos de litofácies acompanhadas de legendas.



Fonte: (Program, 2022)

2.2 Well Logging

Well Logging permite uma descrição detalhada das formações de rocha nas diferentes profundidades. Para a obtenção dos *well logs*, existem diversas técnicas, como *testemunho*, *mudlogging*,*wireline logging* (Hubbard and Linde, 2011).

2.2.1 Testemunho

Testemunho envolve a análise de amostras retiradas diretamente do poço. Uma das maneiras de fazer isso é através de um cilindro de rocha contendo todo o *core*.

Alguns problemas relacionados ao testemunho (Hubbard and Linde, 2011):

- Envolvem amostras pequenas que podem não representar reservatórios ou largas porções. Depende bastante da homogeneidade do reservatório sendo analisado;
- O testemunho e a recuperação alteram o estresse e a temperatura e podem alterar a estrutura da rocha;
- O entupimento, a limpeza e a secagem podem alterar a molhabilidade dos tampões.

2.2.2 Mudding

Mudding é uma outra técnica no qual a lama e os cortes circulares durante a perfuração são continuamente monitorados e avaliados, disponibilizando continuamente a taxa de penetração e a litologia presente (Hubbard and Linde, 2011).

2.2.3 Wireline Logging

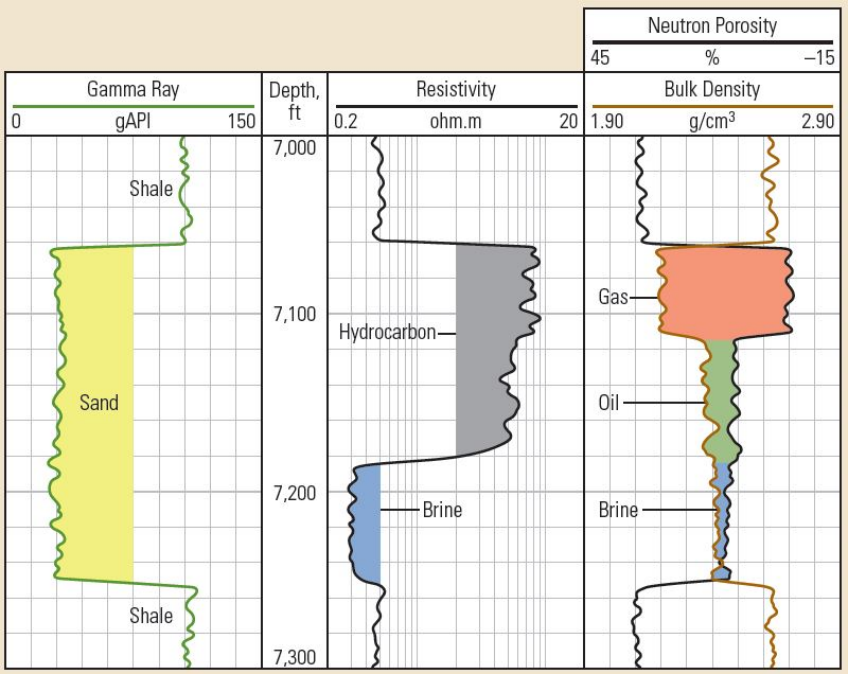
Wireline Logging refere-se ao processo de registro e análise de medições coletadas discretamente ou continuamente dentro de furos de poços. As medições de furos são feitas baixando uma sonda no furo na extremidade de um cabo elétrico. Entre a variedade de sondas de furo de poço disponíveis, pode-se citar: SP, elétrico, EM, gama-gama, gama natural, acústico, temperatura, medidor de vazão, nêutron-nêutron, televisores e registros de calibre (Hubbard and Linde, 2011).

É um método de medição indireto e traz diversas possibilidades de medição comparadas aos outros, além de benefícios econômicos. Seus instrumentos conseguem medir diferentes propriedades, que formam a base da análise petrofísica e geologia.

2.3 *Transformer*

O *transformer* é uma arquitetura de modelo de rede neural utilizada para sequências, que ao invés de usar recorrência, usa apenas mecanismos de *attention* para capturar

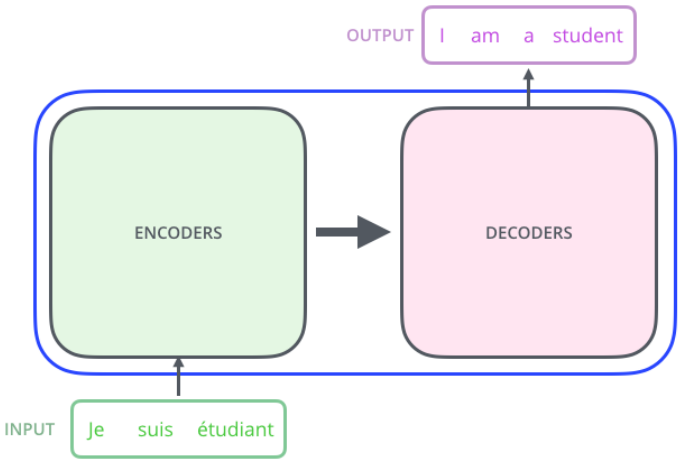
Figura 3 – Exemplo de *well log* que mostra ao longo da profundidade, as informações sobre a litofácies e as *wireline logs* dos hidrocarbonetos.



Fonte: (Program, 2022)

dependências globais entre *input* e *output*. A arquitetura do *transformer* foi originalmente proposta por (Vaswani et al., 2017) no contexto de tradução de sequências de *tokens*. *Tokens* aqui se refere a qualquer elemento individual de uma sequência, como por exemplo palavras em uma frase.

Figura 4 – Visão simplificada da arquitetura *Transformers*.



Fonte: (Alammar, 2022)

No contexto da tradução, o modelo recebe duas entradas durante a predição: uma sequência de *tokens* a ser traduzida e uma sequência contendo os *tokens* já traduzidos até

a posição atual. Como saída, o modelo fornece uma estimativa do *token* para a posição atual. A Figura 4 mostra as duas partes principais da arquitetura: o *encoder* e o *decoder*. No exemplo citado, o *encoder* recebe a sequência de *tokens* a ser traduzida, e o *decoder*, a sequência de *tokens* que já foi traduzida até então.

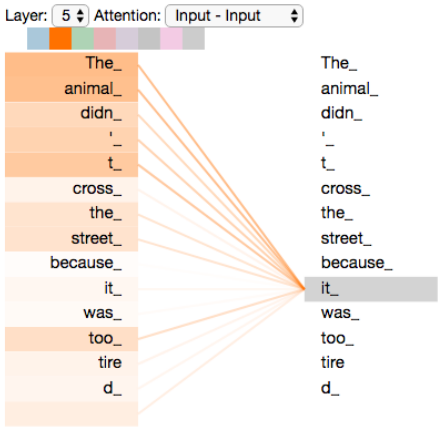
De maneira formal, pode-se afirmar que o *encoder* é utilizado para mapear uma sequência de entrada de símbolos x = (x1, ..., xn) à sequência contínua representada por z = (z1, ..., zn). A partir de z, o *decoder* gera uma sequência de saída y = (y1, ..., ym), um elemento por vez. O modelo é auto-regressivo em cada passo, consumindo os símbolos gerados anteriormente como entradas adicionais quando gera o próximo.

A seguir serão apresentados conceitos importantes para entender os blocos *encoder* e *decoder*.

2.3.1 Attention

Como foi dito, o *Transformer* utiliza mecanismos de *attention*. O mecanismo de *attention* tem o objetivo de relacionar os *tokens* de uma sequência com outros *tokens* da sequência de maneira individual. No contexto de tradução, isso significa que o mecanismo calcula a relação de cada palavra da frase com as outras. A Figura 5 ilustra o cálculo da *attention* para a palavra para o *token it* com as outras palavras de uma frase.

Figura 5 – Visualização simplificada da do mecanismo de *self-attention* entre uma palavra e o resto da frase.

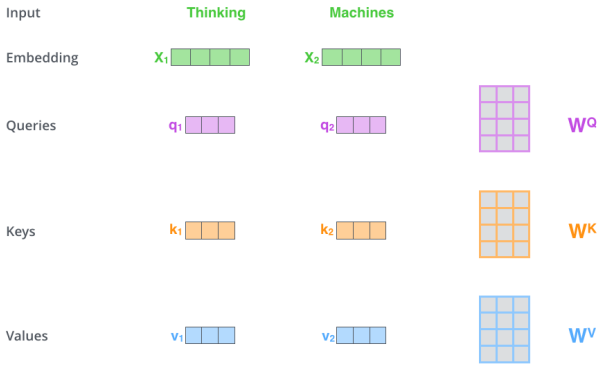


Fonte: (Alammar, 2022)

Matematicamente, a função de *attention* mapeia uma *query* com um conjunto de pares de chaves-valores, no qual a pesquisa (Q), os valores (V), as chaves (K) e a saída são vetores.

A Figura 6 ilustra esses vetores sendo inicializados para as palavras "Thinking"e "Machines", elementos da sequência. Além dos vetores, também são inicializados os pesos que serão calculados durante o treinamento, representados por *W*.

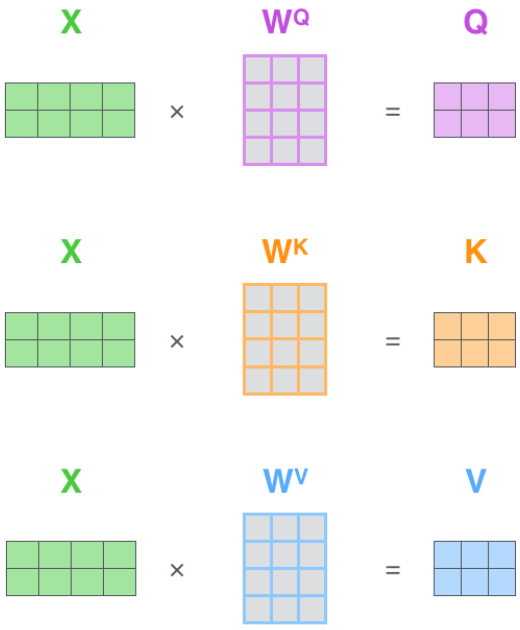
Figura 6 – Visualização dos vetores utilizados pelo mecanismo de *self-attention* sendo inicializados para as palavras *Thinking* e *Machines*.



Fonte: (Alammar, 2022)

A Figura 7 mostra os pesos sendo multiplicados.

Figura 7 – Multiplicação dos vetores de *self-attention* pelos pesos.

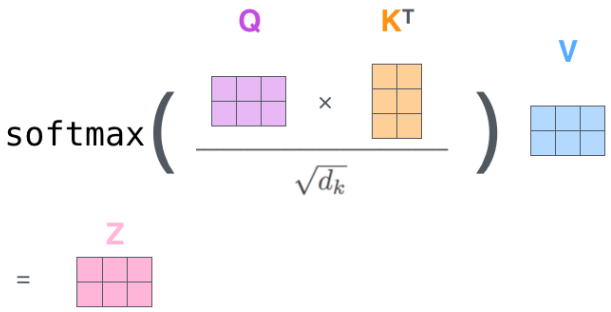


Fonte: (Alammar, 2022)

A saída é computada como uma soma dos pesos dos valores, de maneira que os pesos associados a cada valor é computado por uma função de compatibilidade da pesquisa com a chave correspondente (Vaswani et al., 2017).

O produto escalar nos dá um valor entre negativo e infinito positivo, então aplicamos um softmax para mapear os valores para $[0, 1]$ e para garantir que eles somam 1 em toda a sequência:

Figura 8 – Calculo da *self-attention* usando matriz.



Fonte: (Alammar, 2022)

$$w_{ij} = \frac{\exp w'_{ij}}{\sum_j \exp\ w'_{ij}}$$

De maneira específica, esse mecanismo é denominado *self-attention* quando é aplicado entre os elementos da própria sequência, como mostrado na Figura 5

2.3.1.1 Multi-headed Attention

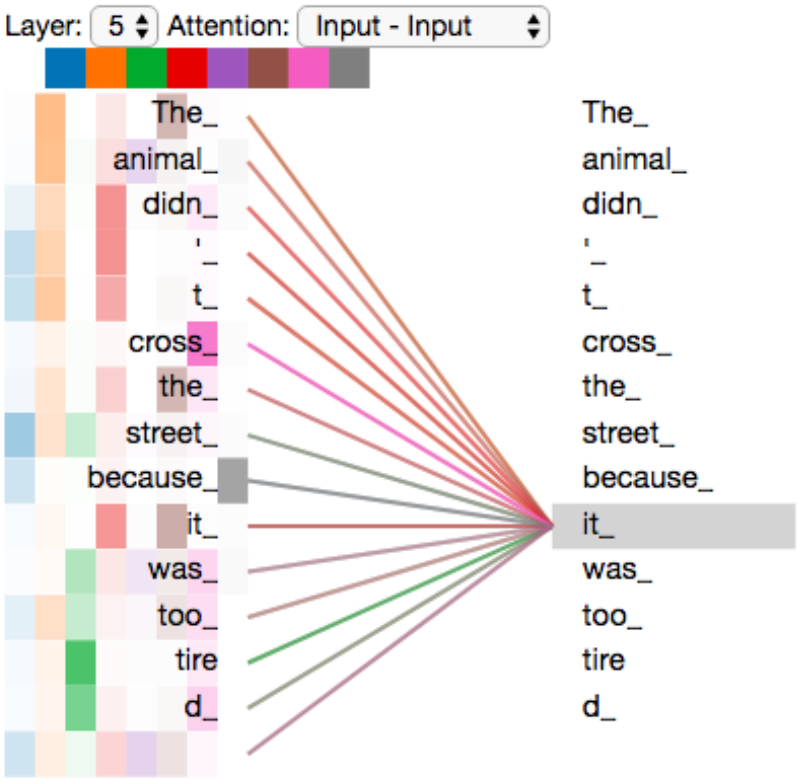
Ao invés de usar o mecanismo de *attention* individual, a arquitetura de *transformer* usa o mecanismo "Multi-Head Attention". Esse mecanismo se resume em calcular a *attention* com as chaves projetadas linearmente. Dessa maneira, o modelo pode aprender diferentes tipos de *attention* entre os *tokens* sequência. Na Figura 9, assim como na Figura, 5, a *attention* é calculada entre uma palavra e as outras da frase. Entretanto, no caso do "Multi-Head Attention"serão aprendidas múltiplas projeções da *attention*, de acordo com o número de *attention heads*. Essas projeções são ilustradas pelas diferentes cores saindo das palavras na Figura 9. Elas permitem que o modelo aprenda relações mais elaboradas entre os *tokens* de uma determinada sequência.

2.3.1.2 Attention Mask

O *transformer* exige que as sequências possuam o mesmo comprimento de entrada. Assim, deve ser definido um tamanho máximo para a sequência. Todas as sequências menores que o tamanho definido devem preencher os espaços faltantes com *tokens* de *padding*.

Esses *tokens* não possuem significados e por isso não interessa ao modelo o resultado do cálculo de sua *attention*. A máscara de *attention* é utilizada para anular o resultado da *attention* para esses *tokens* de *padding*, ou de outros *tokens* em posições indesejadas, como é o caso de de palavras em posições posteriores.

Figura 9 – Ilustração do mecanismo de *multi-head attention*. Cada cor simboliza uma diferente projeção da *attention* sobre uma determinada palavra.



Fonte: (Alammar, 2022)

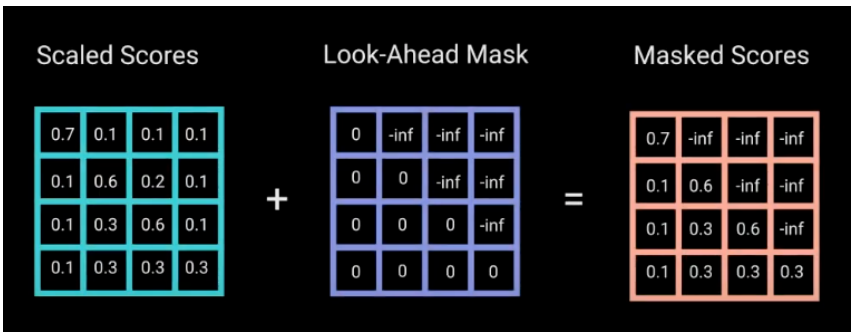
Figura 10 – Matriz do cálculo da *self-attention*, com *tokens* de *padding* preenchendo a sequência.

Fonte: (Alammar, 2022)

Uma implementação bastante comum dessa máscara é através de uma matriz contendo *-infinito* nas posições consideradas ilegais. Ao somá-la a uma matriz de *attention* com as

mesmas dimensões, os valores dessas posições são anulados, como mostrado na Figura 11.

Figura 11 – Adição da matriz de máscara a matriz de *attention*.

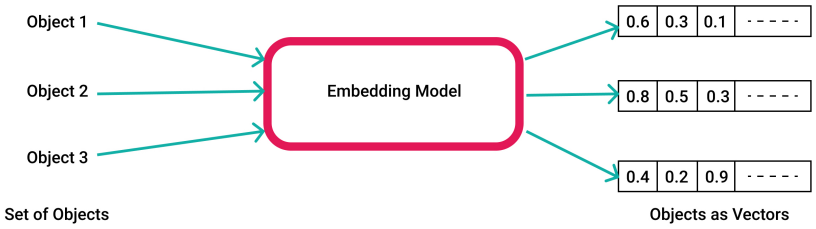


Fonte: (Alammar, 2022)

2.3.2 Embedding

O *embedding* é uma camada que serve como uma *lookup table* para o vetor de entrada. Diferente do conceito *one-hot-encoding* que cria o *lookup table* adicionando 1 na posição do vetor e 0 nas outras partes, o *embedding* faz essa codificação de uma maneira mais eficiente e densa durante o treino, de maneira que palavras parecidas terão vetores parecidos. Uma vez treinados, essas camadas de *embedding* podem ser reutilizadas com seus pesos congelados. A Figura 12 ilustra esses vetores para determinados objetos. Esta abordagem é bastante utilizada no contexto de NLP, uma vez que os significados de palavras podem ser reutilizados para diferentes tarefas, como por exemplo tradução e classificação de sentimentos.

Figura 12 – *Look up table* sendo produzida a partir de um set de objetos



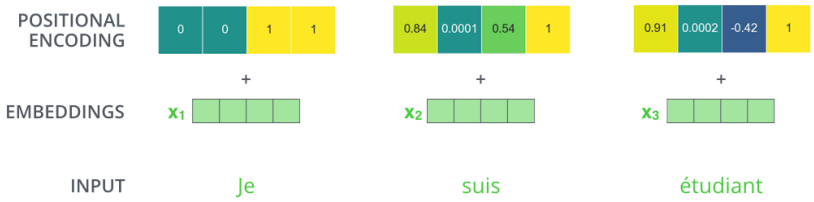
Fonte: (Tripathi, 2022)

No caso da arquitetura do *transformer* proposta por (Vaswani et al., 2017), o *embedding* utilizado já foi pré-treinado e por isso tem seus pesos fixos durante o treino.

2.3.3 Positional Encoder

O mecanismo de *attention*, diferente das recorrências e das convoluções, não captura a posição relativa dos *tokens* na sequência. Por isso, é necessário a adição de um *position*

Figura 13 – Vetor de *positional encoding* sendo somado ao vetor de *embedding* para a palavra *je*.



Fonte: (Alammar, 2022)

encoders ao *embeddings* de cada palavra antes de adentrarem os blocos *encoder* e *decoder*, como mostrado na Figura 13.

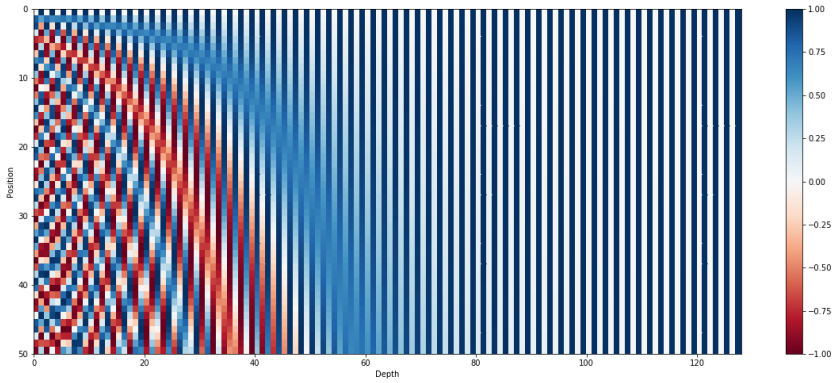
Uma das maneiras de gerar esses vetores de posição é utilizando as frequências do seno e do cosseno na equação da Figura **??** (Vaswani et al., 2017). Na qual *pos* é a posição e i é a dimensão. Ou seja, cada dimensão da codificação posicional corresponde a uma sinusóide. Os comprimentos de onda formam uma progressão geométrica de 2π a $10000 \cdot 2\pi$ (Vaswani et al., 2017).

$$PE(pos, 2i) = sin\left(\frac{pos}{10000^{2i/d_{model}}}\right), PE(pos, 2i+1) = cos\left(\frac{pos}{10000^{2i/d_{model}}}\right)$$

Na qual *pos* é a posição e i é a dimensão. Ou seja, cada dimensão da codificação posicional corresponde a uma sinusóide. Os comprimentos de onda formam uma progressão geométrica de 2π a $10000 \cdot 2\pi$ (Vaswani et al., 2017).

Os valores gerados para diferentes posições podem ser visualizados no gráfico da Figura 14.

Figura 14 – Valores da *positional encoding* que serão adicionados à sequência dependendo da posição de cada elemento.

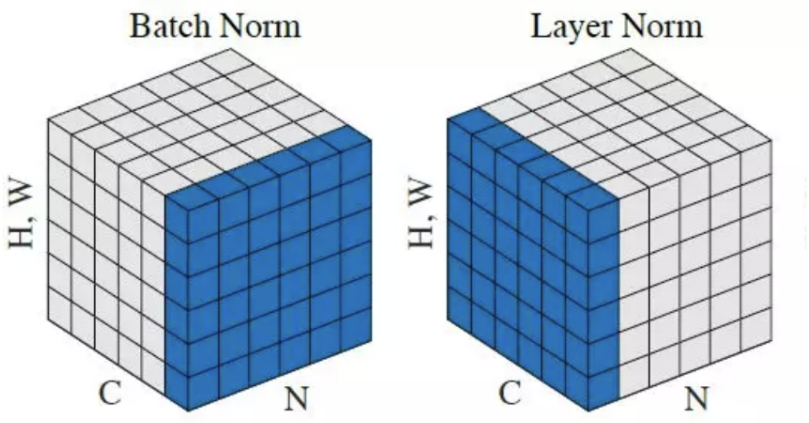


Fonte: (Alammar, 2022)

2.3.4 Layer Normalization

A camada de *layer normalization* normaliza os dados de entrada utilizando estritamente as entradas do neurônio específico. Essa camada é diferente do *batch normalization*, que faz um processo parecido, porém sobre o *batch* de entrada. Os autores de (Vaswani et al., 2017) optaram pela primeira abordagem. A Figura 15 compara as duas para uma entrada de 3 dimensões.

Figura 15 – *Layer normalization* comparada ao *Batch Normalization*.



Fonte: (Ba et al., 2016)

2.3.5 Encoder

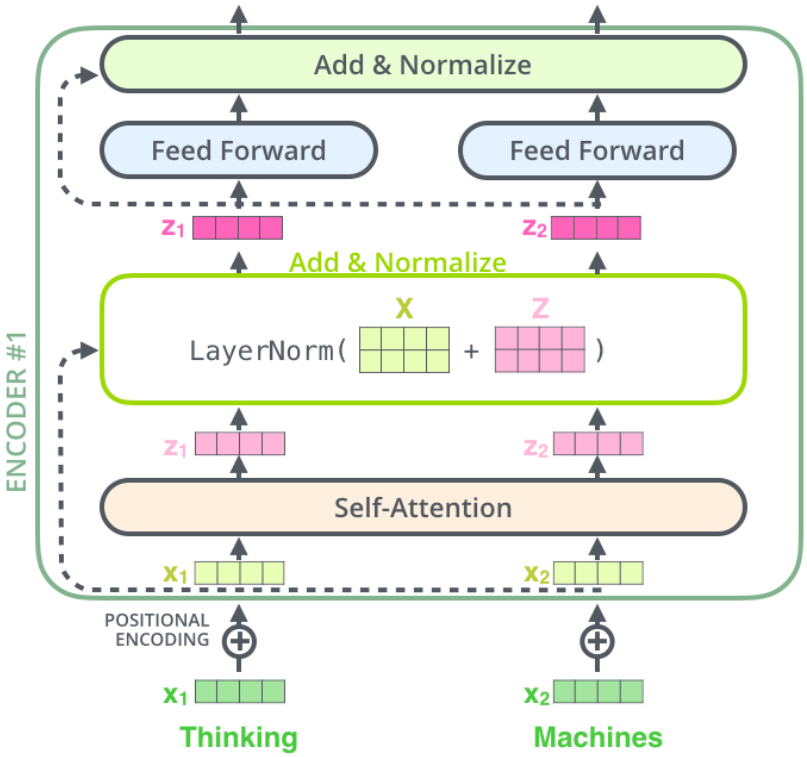
Na implementação original dos *transformers*, o bloco *encoder* é responsável por codificar os dados de entrada.

A Figura 16 dá uma visão simplificada das camadas contidas no bloco. A parte mais importante é feita pelo *self-attention*. Inicialmente, sequências de *tokens* de palavras passam por uma camada de *embedding*. Em seguida, são adicionados os *positional encodings* ao *tokens* da sequência. Isso garante que o mecanismo de *self-attention* use as posições dos *tokens* no cálculo dos pesos.

Em seguida, as sequências são passadas para o bloco de *self-attention*, que irá calcular os pesos de *attention* entre cada *token* da sequência. A saída do bloco de *attention* é normalizada através de uma camada *Layer Norm* juntamente com os resíduos de entrada. A adição dos resíduos serve como forma de regularização durante o treino.

A última parte do bloco codificador da arquitetura original é uma *feedforward* padrão, também seguida de uma camada *Layer Norm* e de adição de resíduos. Na arquitetura,

Figura 16 – Visão dos vetores passando pelas camadas de um bloco *encoder*.



Fonte: (Alammar, 2022)

mais de um bloco codificador é utilizado, de maneira que a saída de cada bloco sirva de entrada para o próxima.

2.3.6 Decoder

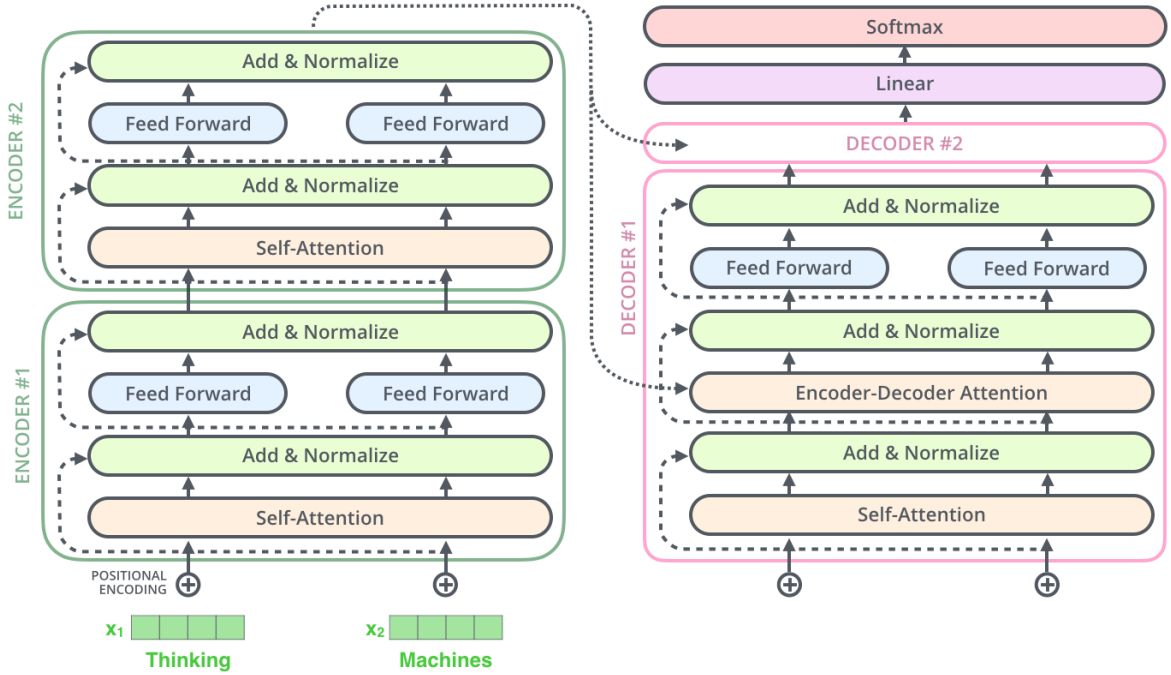
O bloco *decoder* tem duas partes principais, uma camada de *self-attention*, como no bloco *encoder*, e uma camada de*encoder-decoder attention*, que utiliza as saídas do *encoder*. A Figura 17 apresenta ambos os blocos lado a lado.

O *decoder* recebe como entrada *embeddings* de sequências de *tokens* somados com *positional encoding*, com a diferença para o *encoder* de que cada sequência de entrada são os tokens de saída deslocados para a direita. Isso porque um dos objetivos do *decoder* é capturar a *self-attention* entre elementos da sequência anterior. O deslocamento é feito pois o último elemento da sequência não é utilizado.

Outra diferença é no cálculo da *self-attention*, que utiliza de uma máscara de tamanho (seq-len, seq-len), que quando usada no cálculo da *attention*, anula os pesos atribuídos aos *tokens* posteriores àqueles que estão sendo previstos.

Em seguida, uma camada de *encoder-decoder-attention* recebe as saídas *masked self-attention* como V e as saídas dos blocos de *encoder* como K, Q. Após calculada a *encoder-decoder-attention*, as saídas são passadas para um bloco *feedforward*, finalizando o bloco

Figura 17 – Visão do bloco *encoder* e *decoder*.



Fonte: (Alammar, 2022)

decoder.

2.3.7 Layer de Classificação

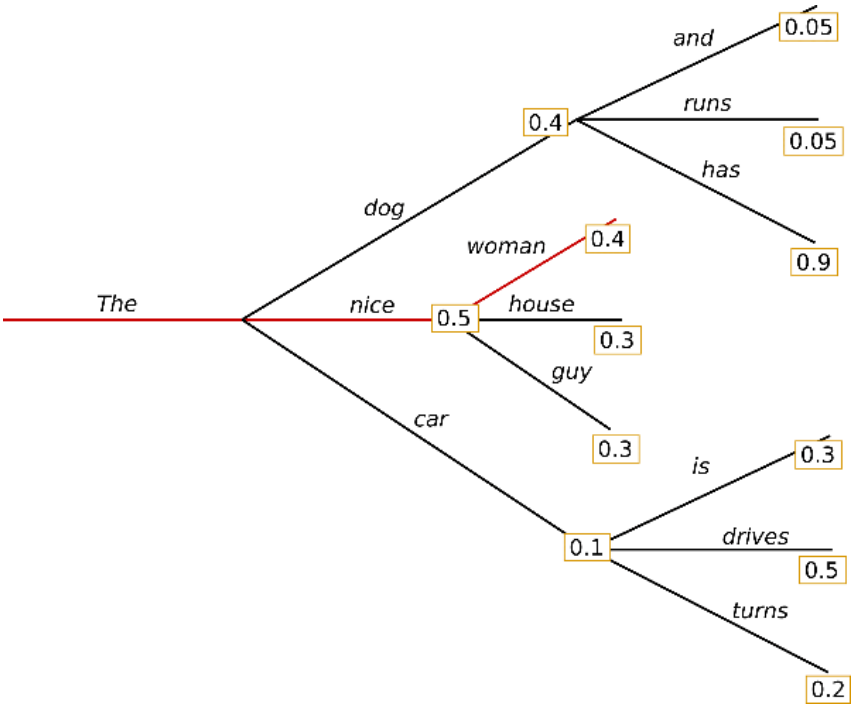
A saída do bloco *decoder* é então utilizada para calcular as probabilidades do *token* na posição atual. É importante reforçar que por conta do *encoder-decoder attention*, essa classificação não dependerá apenas dos *tokens* anteriores da sequência, mas também da entrada do *encoder* (Vaswani et al., 2017).

2.3.8 Estratégias de Decoding

Como foi mostrado, no caso da arquitetura *encoder-decoder* de (Vaswani et al., 2017), o processo de geração de uma sequência de saída acontece de maneira auto-regressiva, com cada classificação para uma determinada posição da sequência consumindo *tokens* de posições anteriores. Esse processo de *decoding* pode ser feito usando atribuindo a classificação com maior probabilidade para a posição atual. Essa estratégia é conhecida como *greedy search*.

A Figura 18 mostra uma *greedy search* para um determinado algoritmo, começando pela palavra *This*. A busca sempre escolhe a próxima palavra com maior probabilidade, resultado na frase ("The","nice","woman"), que possui uma probabilidade de 0.5 x 0.4 = 0.20. Essa técnica não considera as probabilidades da sequência completa, e por isso não

Figura 18 – Árvore de probabilidade mostrando as escolhas de palavras durante uma *greedy search*. Note que para cada bifurcação, a palavra com maior probabilidade é escolhida.



Fonte: (Maxwell, 2022)

necessariamente devolve a sequência mais provável. Uma alternativa bastante utilizada é o *beam search*, ilustrado na Figura 19.

A *beam search* reduz o risco de perder sequências de palavras de alta probabilidade ocultas, mantendo um número de hipóteses (número de *beams*) mais prováveis em cada passo de tempo e, eventualmente, escolhendo a hipótese que tem a probabilidade geral mais alta.

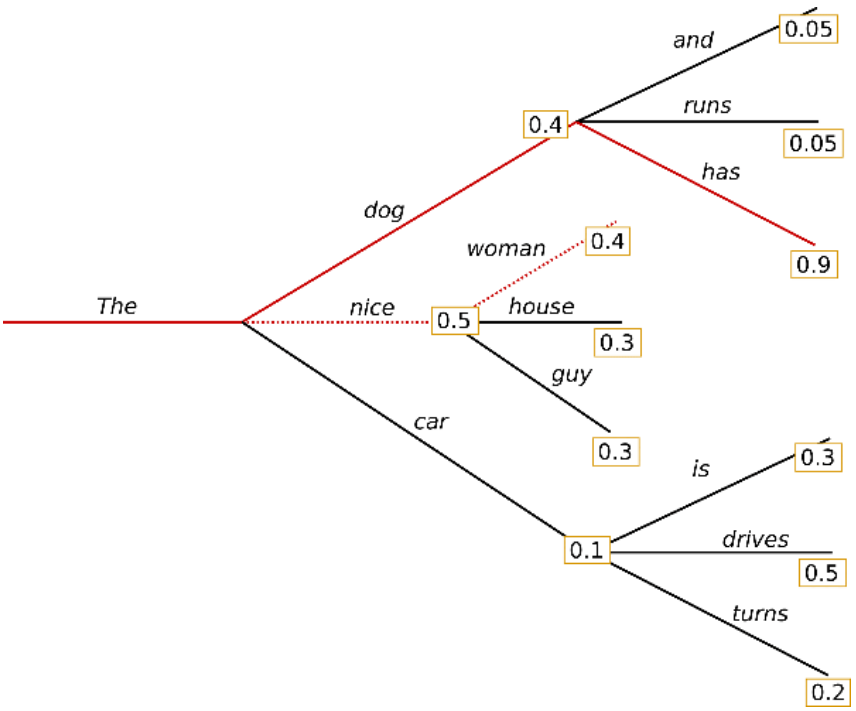
2.3.9 Variações

A Figura 20 mostra a arquitetura do *transformer* na sua forma completa proposta por (Vaswani et al., 2017).

Essa arquitetura é referenciada como *"vanilla"* e gerou diversas variações. A maioria dos trabalhos focados em *transformers* trabalha propondo essas variações. (Lin et al., 2021) propõe algumas categorias para tais:

- Eficiência do modelo;
- Generalização do modelo;
- Adaptação do modelo.

Figura 19 – Árvore de probabilidade mostrando as escolhas de palavras durante uma *beam search* com número de *beams* igual a 2.



Fonte: (Maxwell, 2022)

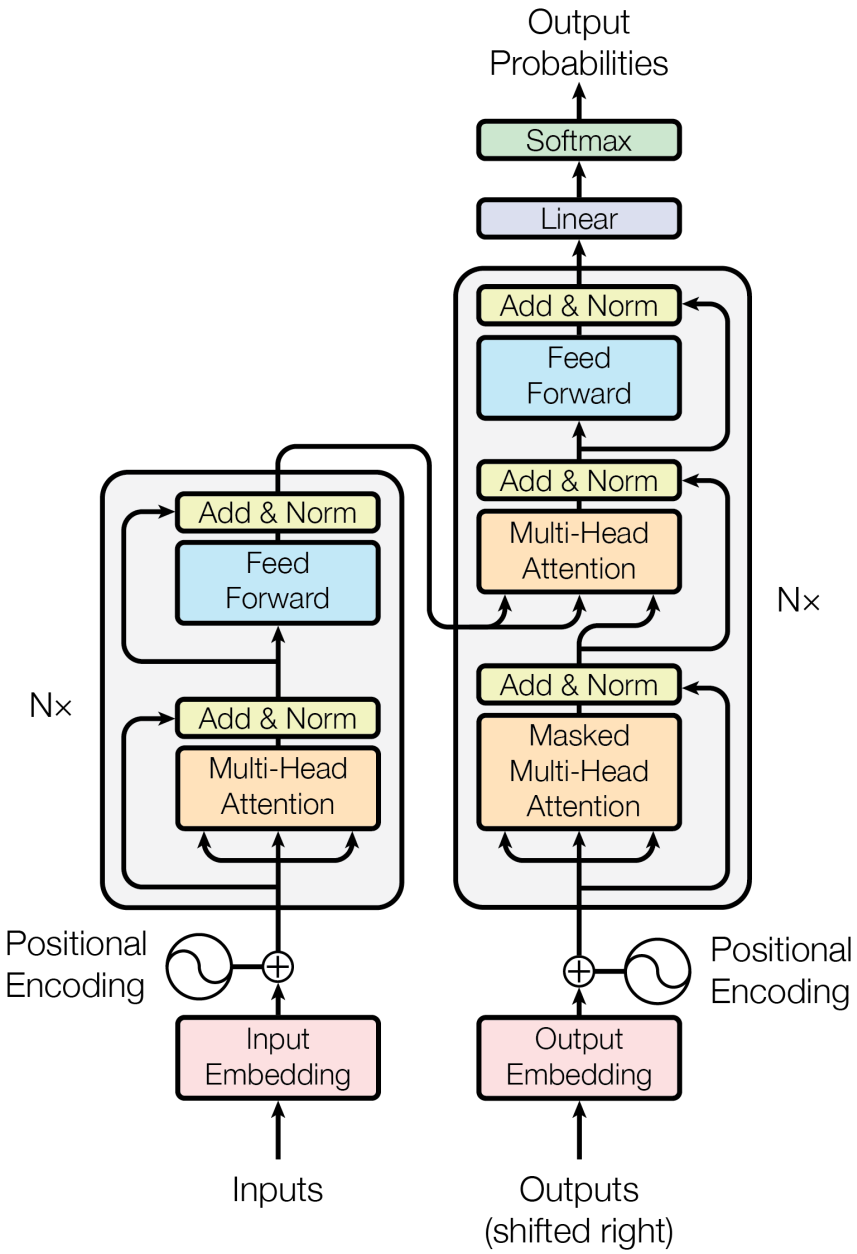
A adaptação do modelo envolve adaptar a arquitetura *transformer* para diferentes tarefas fora do domínio de NLP. De maneira geral, é possível encaixar as arquiteturas de *transformers* nas seguintes categorias:

- Arquitetura Encoder-Decoder: segue a proposta original de (Vaswani et al., 2017) e é ideal para problemas nos quais uma sequência deve ser gerada para cada sequência. No campo de NLP, isso inclui tarefas como tradução;
- Arquitetura somente *encoder*: nesse caso, apenas o componente *encoder* é utilizado, usualmente associado a uma camada de saída *softmax* para associar a sequência com um valor, como é o caso de problemas de classificação;
- Arquitetura somente *decoder*: nesse caso, apenas o componente *decoder* é utilizado, bastante aplicado no contexto de geração de texto.

2.4 RNN - Redes Recorrentes

RNNs são arquiteturas de redes neurais nas quais os neurônios da rede utilizam os dados de saída de neurônios anteriores, além dos dados de entrada. Esse processo de recorrência permite que os neurônios aprendam sobre o ordenamento dos dados de entrada.

Figura 20 – Arquitetura original completa do *transformers*.



Fonte: (Vaswani et al., 2017)

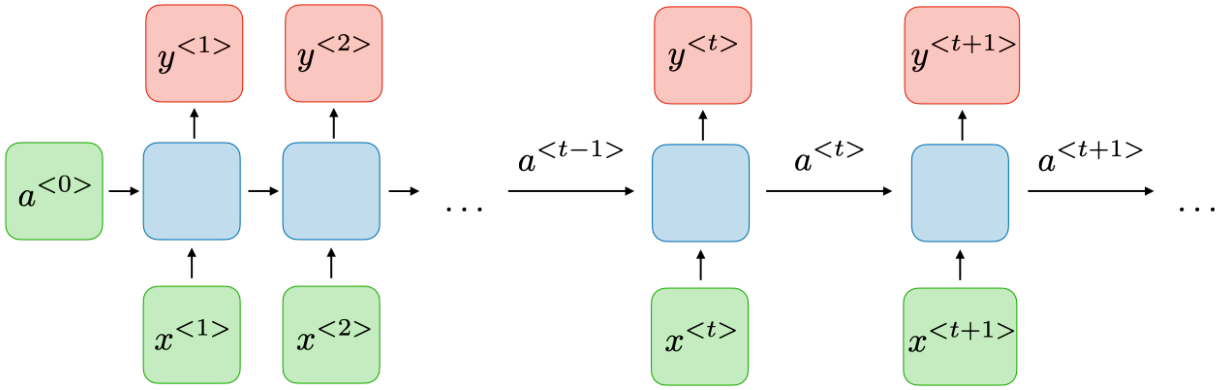
Isso pode ser ilustrado pela Figura 21, que demonstra a passagem das saídas de neurônios para os seguintes ao longo de passos de tempo.

RNNs e suas variações dominaram o estado da arte para problemas envolvendo sequências de dados de entrada, como NLPs e séries temporais antes da chegada dos *transformers*.

2.4.1 LSTM - Long Short-Term Memory

As LSTMs são arquiteturas propostas com o objetivo de melhorar a capacidade das RNNs aprenderem pesos de elementos distantes, possibilitando o uso de sequências maiores. Para isso, cada neurônio da LSTM possui uma célula com diferentes redes (*gates*)

Figura 21 – Exemplo de uma RNN se estendendo por vários "passos"de tempo.



Fonte: (Cho et al., 2014)

que servem para aprender sobre os pesos de maneira selecionada, escolhendo o que ficará na memória. (Sherstinsky, 2020)

A informação é retida pelas células e as manipulações de memória são feitas pelos (*gates*). Existem três portas:

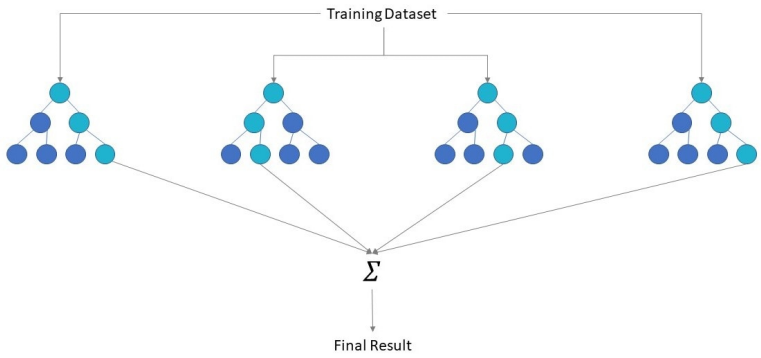
- Forget Gate: as informações que não são mais úteis no estado da célula são removidas com o *forget gate*;
- Input Gate: a adição de informações úteis ao estado da célula é feita pelo *input gate*.
- Output Gate: a tarefa de extrair informações úteis do estado da célula atual para ser apresentada como uma saída é feita pelo *output gate*.

2.4.2 Decision Trees

Decision Trees são métodos de aprendizado supervisionados usados em problemas de classificação e regressão. Os modelos criados predizem valores de uma variável alvo através do aprendizado de regras simples de decisão aprendidas durante o treino. A Figura 22 demonstra uma possível árvore de decisão para a classificação de flores a partir de características das pétalas (IBM, 2022).

Baseado neste algoritmo, foi desenvolvido o algoritmo de *Random Trees*. Durante o treino, este algoritmo constrói um modelo que consiste em várias *decision trees*. Ele usa uma técnica chamada *bagging* para construir as árvores completas em paralelo a partir de amostras de *bootstrap* aleatórias do conjunto de dados. A previsão final é uma média de todas as previsões da árvore de decisão.

Figura 22 – Ilustração do algoritmo *random forest* utilizando um número arbitrário de *decision trees* para realizar uma classificação.



Fonte: (IBM, 2022)

2.4.3 Gradient Boosting Decision Trees

O *Gradient Boosting Decision Trees* é um algoritmo de aprendizado de conjunto de *decision trees* semelhante a "floresta aleatória", para classificação e regressão.

Gradient Boosting é uma extensão de *boosting* onde o processo de geração aditiva de modelos fracos é formalizado como um algoritmo de descida de gradiente sobre uma função objetivo. (Chen and Guestrin, 2016) O GBDT treina iterativamente um conjunto de *decision* rasas, com cada iteração usando os resíduos de erro do modelo anterior para ajustar o próximo modelo. A previsão final é uma soma ponderada de todas as previsões da árvore. O *bagging* aleatório da floresta minimiza a variância e o *overfitting*, enquanto o *boosting* GBDT minimiza o viés e o *underfitting*.

A partir disso, o XGBoost, que significa *Extreme Gradient Boosting*, é uma biblioteca de aprendizado de máquina (GBDT) de árvore de decisão com aumento de gradiente distribuída e escalável. Ele fornece reforço de árvore paralela e é a principal biblioteca de aprendizado de máquina para problemas de regressão, classificação e classificação. Alguns dos termos usados são explicados nas seções a seguir.

3 Trabalhos Relacionados

Neste capítulo, serão explicadas as abordagens de 2 trabalhos que usam transformadores para classificação de séries temporais, um trabalho que traz uma revisão de abordagens de aprendizado de máquina para séries temporais é um trabalho que traz um exemplo de um problema de sequência para séries temporais multivariáveis.

3.1 Método de Pesquisa

A principal hipótese motivadora por trás do uso de transformadores no problema de classificação de facies, é de que a classificação de uma litofácies em uma determinada profundidade está altamente correlacionada aos valores de curvas longe métricas de profundidades próximas. Sendo assim, as buscas no geral giraram em torno de encontrar trabalhos com arquiteturas do tipo sequência-para-sequência. Além disso, a pesquisa foi filtrada para trabalhos nos quais as entradas continham dados de natureza numérica e contínua.

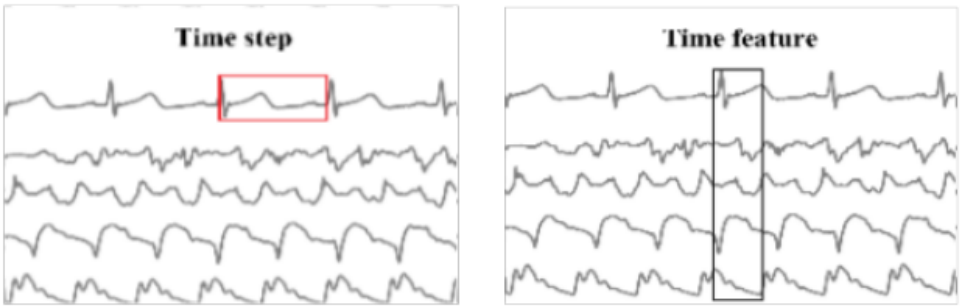
Os seguinte itens foram buscados:

- Classificação de séries temporais envolvendo transformadores.
- *Timeséries classification using transformers.*
- *Sequence to sequence classification using transformers.*
- *Transformer for numerical data.*
- *Transformer for tabular data.*
- *Lithofacies classification using transformers.*
- *Lithofacies classification using lstm.*

3.2 Gated-Transformer-on-MTS

O trabalho proposto por (Liu et al., 2021) propõem uma abordagem de classificação de séries temporais baseada em transformadores, com algumas modificações quando comparada com a original. De acordo com os autores, uma série temporal com múltiplos atributos pode ser decomposta em "canais", no qual cada canal é uma série univariável. Nesse caso, se espera que exista uma correlação entre diferentes canais em passos diferentes.

Figura 23 – Na esquerda, relação temporal entre atributos de um mesmo canal. Na direita, relação entre canais para um mesmo intervalo.



Fonte: (Liu et al., 2021)

Assim, para se obter essa relação, é necessário capturar tanto a informação temporal e relação entre os valores de um canal, quanto às informações espaciais e a relação entre os canais em um determinado espaço de tempo. Na Figura 23, cada curva representa uma serie invariável (um atributo). A Figura da esquerda mostra como seria uma sequencia de dados que contenham apenas informações temporais de um mesmo atributo, enquanto a da direita mostra uma sequencia de dados contendo atributos para um mesmo passo de tempo. Cada uma dessas sequencias sera capturada por um bloco *encoder* diferente.

A Figura 24 mostra a arquitetura proposto pelos autores com os dois blocos *encoder* e a última camada de predição.

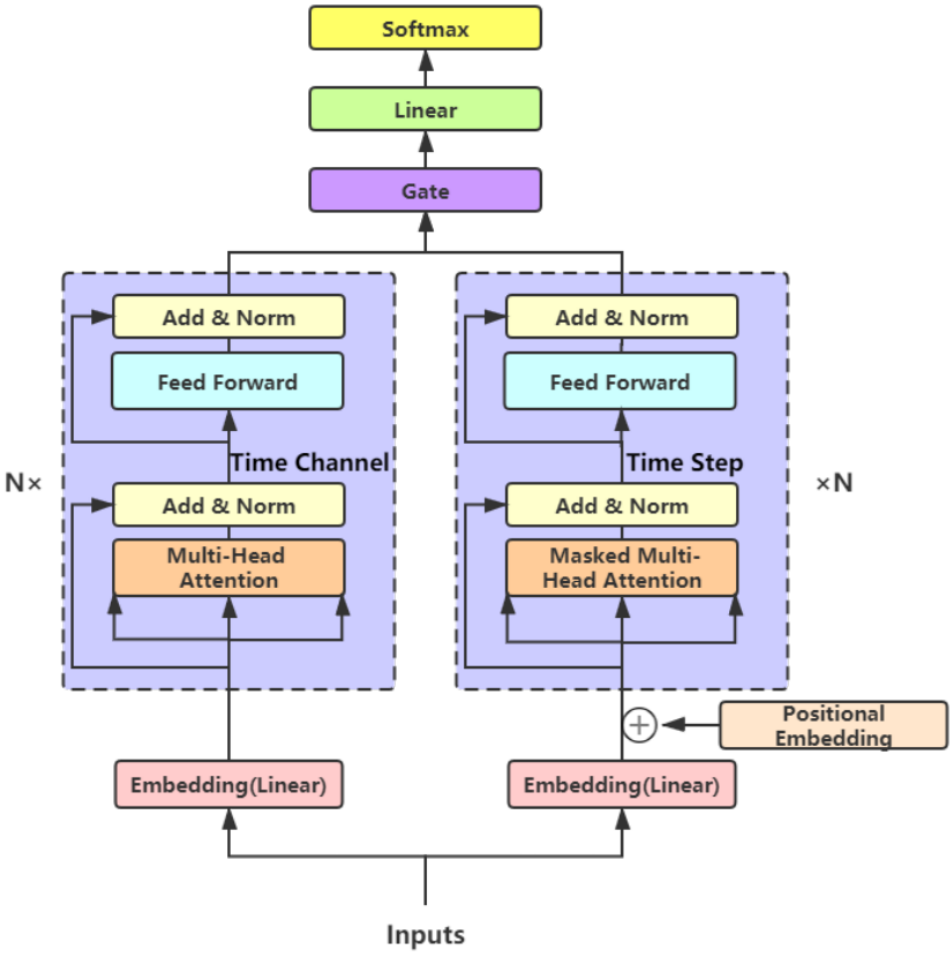
Mais especificamente, os *encoders* são descritos da seguinte forma:

- *Step-wise Encoder*. Calcula a *self-attention* entre os elementos de um mesmo canal. É o bloco esquerdo na Figura 24.

- *Channel-wise Encoder*. Calcula a *self-attention* entre os elementos de canais diferentes. É o bloco direito na Figura 24. Como o objetivo é capturar a relação temporal entre os passos, é adicionado ainda um *positional encoding*. A codificação posicional é feita seguindo a linha de (Vaswani et al., 2017) sobre os dados projetados não-linearmente.

Antes de passar por esses blocos, a sequencias passam por um *fully connected layer* com uma função de ativação não-linear. Essa camada serve função parecida ao do *Embedding Layer* de (Vaswani et al., 2017), e é utilizada por se tratarem de dados de natureza contínua. Outra parte importante da arquitetura é o bloco *gate* mostrado na Figura 24. Esse bloco é explicado pela 25. As saídas dos blocos *encoders Channel-wise Encoder* (C) e *Step-wise Encoder* (S) são concatenadas e passadas por uma camada linear. A saída h

Figura 24 – Arquitetura do GTM.



Fonte: (Liu et al., 2021)

dessa camada é então passada por uma *Softmax* que ira atribuir pesos *g1* e *g2* para cada torre.

Figura 25 – Mecanismo de *gating* que concatena o resultado das duas torres.

$$
\begin{aligned}
h &= \mathbf{W} \cdot Concat(C, S) + b \\
g_1, g_2 &= Softmax(h) \\
y &= Concat(C \cdot g_1, S \cdot g_2)
\end{aligned}
$$

Fonte: (Liu et al., 2021)

Finalmente, os pesos de cada torre são multiplicados por suas respectivas saídas e concatenados entre si. A ideia é que os pesos sejam aprendidos durante o treino, e

capturem o quanto cada torre contribui para uma determinada predição. De acordo com os autores, essa abordagem superou os resultados tanto de uma concatenação simples, quanto do uso de cada torre individual.

A saída do *gete* é passada mais uma vez por uma camada linear e um *Softmax*, e retorna as probabilidades das classes previstas.

Para medir a performance do modelo, os autores utilizaram 13 conjuntos diferentes de dados envolvendo séries temporais. Eles comparam esses resultados com diversos modelos de *deep learning* normalmente utilizados nesse tipo de problema, entre eles *Fully Convolutional Networks* (FCN) e *Residual Networks* (ResNet).

Figura 26 – Comparação entre a performance dos modelos utilizados para cada conjunto de dados utilizando acurácia

	MLP	FCN	ResNet	Encoder	MCNN	t-LeNet	MCDCNN	Time-CNN	TWIESN	GTN
AUSLAN	93.3	**97.5**	97.4	93.8	1.1	1.1	85.4	72.6	72.4	**97.5**
ArabicDigits	96.9	99.4	**99.6**	98.1	10.0	10.0	95.9	95.8	85.3	98.8
CMUsubject1	60.0	**100.0**	99.7	98.3	53.1	51.0	51.4	97.6	89.3	**100.0**
CharacterTrajectories	96.9	**99.0**	**99.0**	97.1	5.4	6.7	93.8	96.0	92.0	97.0
ECG	74.8	87.2	86.7	87.2	67.0	67.0	50.0	84.1	73.7	**91.0**
JapaneseVowels	97.6	**99.3**	99.2	97.6	9.2	23.8	94.4	95.6	96.5	98.7
KickvsPunch	61.0	54.0	51.0	61.0	54.0	50.0	56.0	62.0	67.0	**90.0**
Libras	78.0	96.4	**95.4**	78.3	6.7	6.7	65.1	63.7	79.4	88.9
NetFlow	55.0	89.1	62.7	77.7	77.9	72.3	63.0	89.0	94.5	**100.0**
UWave	90.1	**93.4**	92.6	90.8	12.5	12.5	84.5	85.9	75.4	91.0
Wafer	89.4	98.2	98.9	98.6	89.4	89.4	65.8	94.8	94.9	**99.1**
WalkvsRun	70.0	**100.0**	**100.0**	**100.0**	75.0	60.0	45.0	**100.0**	94.4	**100.0**
PEMS	-	-	-	-	-	-	-	-	-	93.6

Fonte: (Liu et al., 2021)

Os resultados na Figura 26 mostram que a GTN conseguiu resultados comparáveis à FCN e a ResNet. Em especial, nos *datasets* NetFlow e KickvsPunch, a GTN conseguiu resultados superiores.

3.3 A Transformer-based Framework for Multivariate Time séries Representation

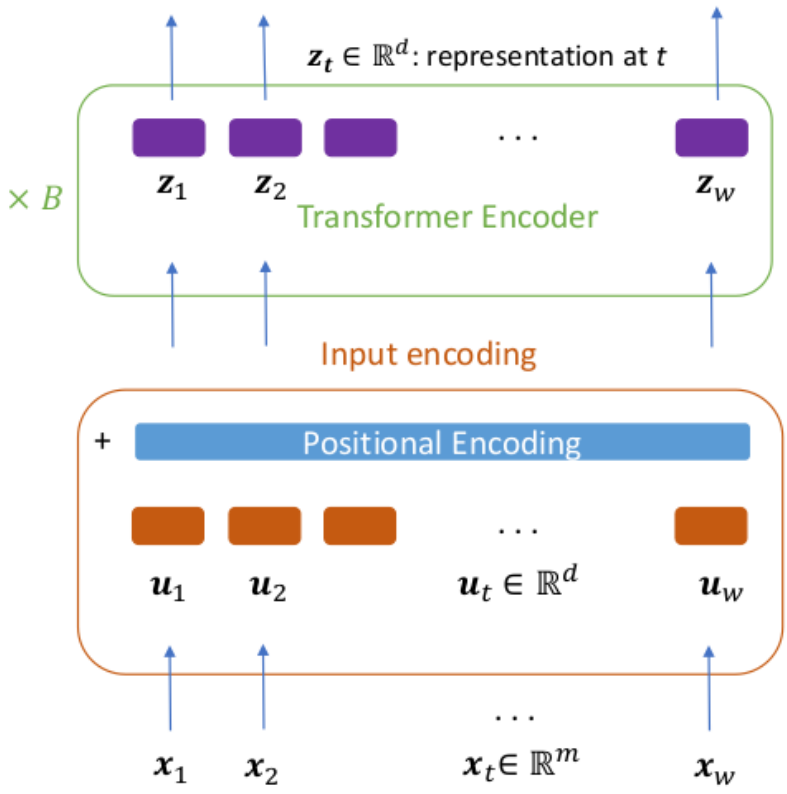
O trabalho de (Zerveas et al., 2020) busca propor uma metodologia para lidar com problemas de séries temporais de classificação ou de regressão, composta pelas seguintes etapas:

- Pré-treino não supervisionado de modelo *transformer encoder-only* através de *denoising* de de dados de entrada. O objetivo é extrair vetores densos de séries temporais multivariáveis.
- Retreinar o modelo pré-treinado de maneira supervisionada, adaptando-o para classificação ou regressão.

Como foi dito, arquitetura possui apenas bloco um *encoder* se semelhante ao da arquitetura *vanilla* de (Vaswani et al., 2017). O motivo dessa decisão foi tornar a arquitetura o mais genérica possível, facilitando a adaptação para diferentes tarefas. Outras vantagens citadas pelos autores são de que bloco *decoder*, exige o uso de *masked output sequences* como *inputs*, o que o tornaria impróprio para classificação. Além disso, utilizar apenas o *encoder* permite o uso de apenas metade dos parâmetros do modelo.

Somado a essa arquitetura *Encoder-Only* foram feitas mudanças para que a arquitetura fosse compatível com séries temporais multivariáveis.

Figura 27 – Arquitetura base.



Fonte: (Zerveas et al., 2020)

A Figura 27 mostra o bloco *encoder* base da arquitetura. Inicialmente, assim como em (Vaswani et al., 2017), os vetores são projetados para uma dimensão D, porém através de uma camada *feed-forward*. Note que assim, como em (Liu et al., 2021), essa camada é utilizada ao invés de um *Embedding Layer* por se tratarem de dados contínuos.

Os valores de entrada projetados são em seguidas somado a uma *sinusiodal encodings* assim como em (Vaswani et al., 2017), porem com pesos não-fixos, que serão aprendidos durante o treino. Os autores afirmam essa escolha obteve melhores resultados.

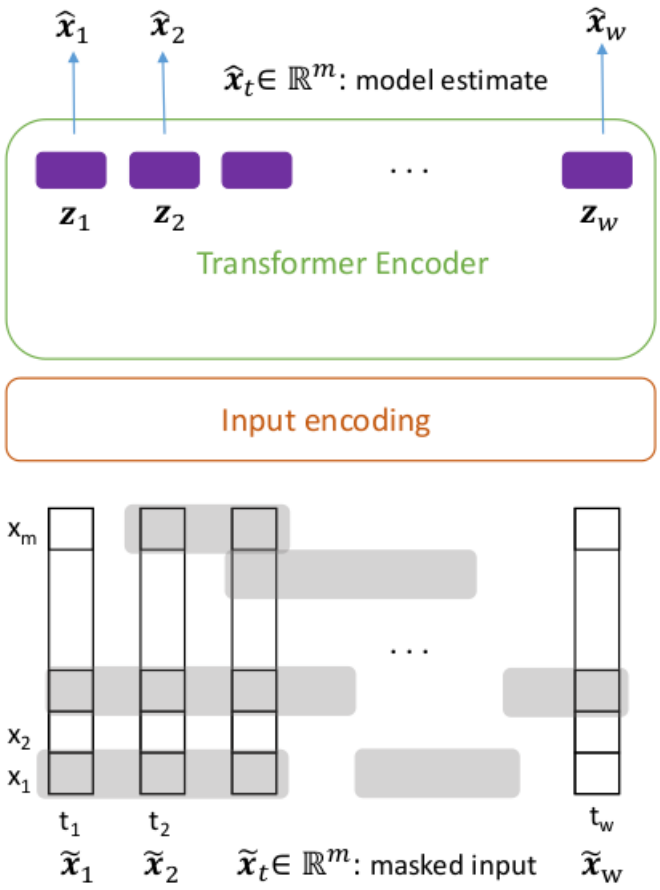
Para lidar com o diferentes tamanhos de sequência, os autores definem um tamanho máximo de sequência para conjunto de dados inteiro e fazem o *padding* das sequências me-

nores utilizando dados arbitrários. Uma máscara de *padding* é gerada para adicionar grandes valores negativos aos *attention scores* antes de computar a distribuição *self-attention* com a função softmax.

Com o modelo base definido, existe a possibilidade de treina-lo diretamente de maneira supervisionada para classificação ou para regressão, ou, de primeiramente realizar um pré-treino não-supervisionado (ausência de *groud truth*).

No caso da tarefa de pré-treino não-supervisionado, os autores consideram a *task autoregressiva* de remoção de ruído da entrada. Para isso, uma parte da entrada é definida com 0 e o modelo deve aprender a predizer a parte mascarada. Na parte de baixo da Figura 28, os retângulos cinza representam as parte das matrizes de entradas mascaradas.

Figura 28 – Arquitetura para o treino não-supervisionado.



Fonte: (Zerveas et al., 2020)

Note que essa mascara é adicionado tanto entre diferentes atributos xm, como também entre diferentes passos t de um mesmo atributo. Isso pois o treino possui o intuito de atender ambos os segmentos sucessores e posteriores das de um atributo, assim como atributos pertencentes a outras séries temporais, possibilitando o aprendizado de inter-dependências entre variáveis. Entretanto, para cada amostra, os autores apenas consideram os valores mascarados são considerados no cálculo do erro médio.

Finalmente Apos passado pelo *encoder*, o modelo retorna de maneira concorrente sua estimativa não corrompida dos vetores de entrada através de uma camada linear.

Para o caso de um treino supervisionado como de classificação, as predições $\hat{y}$ serão também passadas através de uma função softmax para obter uma distribuição sobre as classes, *cross-entropy* com as *categorical groud truth labels* para calcular o *loss* da amostra.

Essa metodologia foi testada no contexto de regressão e de classificação. No caso dos problemas de classificação, foram selecionados 11 *multivariate datasets* de *UEA Time series Classification Archive* (Bagnall et al., 2018). Para cada um dos conjuntos de dados, os autores treinaram dois modelos em cima do modelo base, um com pré-treino e o outro sem, além de outros modelos bastante usados nestes tipos de problema, como XGBoost, ROCKET e LSTM.

Em tarefas de classificação, o modelo obteve a melhor performance em sete dos 11 conjuntos, com um ranque médio de 1.7.

Os autores enfatizam que, além dos resultados obtidos acima, a arquitetura conseguiu atingir esse resultados com um tempo de pré-treino e um número de parâmetros bastante aceitável, inclusive para treino em CPU. Com GPU, é possível ter um treino mais rápido do que as abordagens com maior acurácia que não utilizam *deep learning*.

Os autores mostram que a metodologia proposta consegue performar melhor que todos os estados da arte atual em diversos conjuntos de dados públicos diferentes, mesmo com um número limitado de amostras.

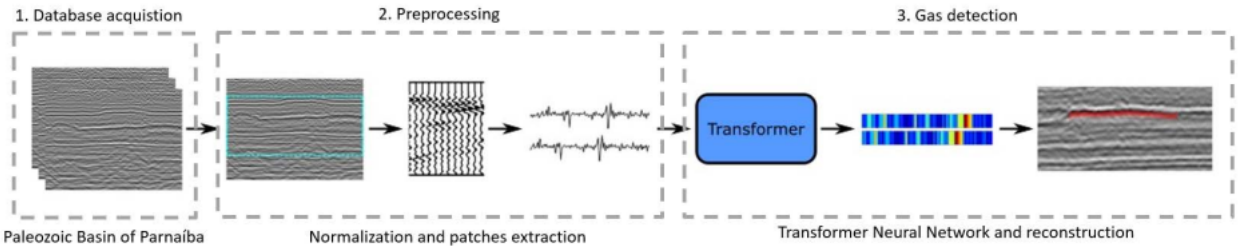
3.4 Time2Vec Transformer for leak detection

O principal objetivo deste trabalho é desenvolver um método de detecção de possível acumulação de gás em imagens de reflexões sísmicas usando processamento de imagem, prosseguido dos transformadores. Os autores separam as principais contribuições do trabalho como sendo a adaptação da rede neural transformador para campo de dados sísmicos, tornando-o robusto na detecção de gás. O método proposto é o Time2Vec-Transformer.

A primeira etapa do processo envolve o pré-processamento das imagens sísmicas para que possam ser entendidos pela arquitetura. Depois do pré-processamento, o próximo passo é projeta os sinais sísmicos em representações de vetores para que a arquitetura de transformador possa utilizá-los. Para isso, é utilizado o método Time2Vec (Kazemi et al., 2019). Esse método é escolhido como alternativa ao *feed-foward layear*, que normalmente é utilizado como *Embedding Layer* para dados contínuos. Esse método foi criado para ser utilizado com dados de séries temporais na forma de vetores de várias dimensões, e pode ser utilizado junto com diferentes arquiteturas de *deep learning*.

O *Time2Vec* dispensa também o uso de*positional encoding* como na arquitetura original. Os autores esperam que com essa camada, será possível extrair informações temporais

Figura 29 – Processo completo do trabalho. Desde do processamento das imagens sísmicas até a predição com a arquitetura proposta.



Fonte: (Diniz et al., 2022)

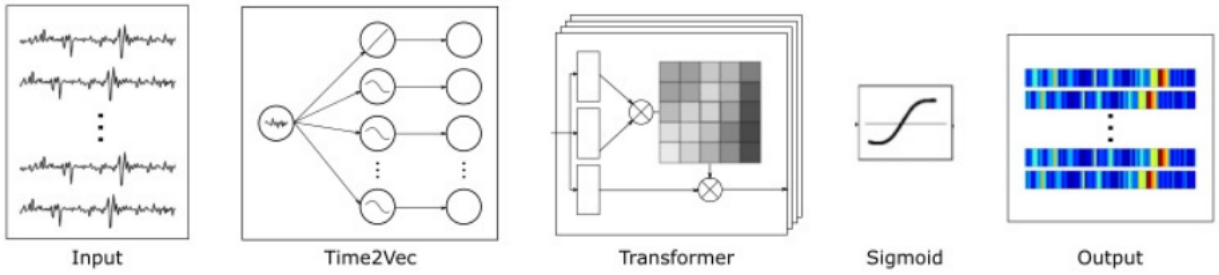
Figura 30 – Método Time2Vec.

$$\mathbf{t2v}(\tau)[i] = \begin{cases} \omega_i \tau + \varphi_i, & \text{if } i = 0. \\ \mathcal{F}(\omega_i \tau + \varphi_i), & \text{if } 1 \leq i \leq k. \end{cases}$$

Fonte: (Kazemi et al., 2019)

de maneira mais eficiente. O resto da arquitetura segue a estrutura proposta originalmente por, contendo os dois componentes *encoder* e *decoder*.

Figura 31 – Componentes da arquitetura proposta.



Fonte: (Diniz et al., 2022)

Figura 32 – Tabela de comparação da arquitetura time2vec para as outras abordagens.

Neural Network	SEN	SPE	ACC	AUC
1-dimentional CNN	57.21%	95,75%	95.13%	76.48%
LSTM	52,99%	96,69%	95,97%	74,84%
Standard Transformer	69.45%	88,43%	88,29%	84,73%
Time2Vec Transformer	**79.61%**	**97.47%**	**97.15%**	**88.54%**

Fonte: (Diniz et al., 2022)

O método Time2Vec proposto obteve resultados superiores a todas as outras abordagens testadas.

3.5 Comparação entre os trabalhos

Nesta seção serão ressaltados os principais pontos sobres os trabalhos que são relevantes para o desenvolvimento.

Primeiramente, sobre a arquitetura de transformadores desses trabalhos, apenas o trabalho de (Kazemi et al., 2019) propõe uma arquitetura que utiliza tanto o *encoder* e o *decoder*, o que o tornaria uma referência interessante. Entretanto, pouco se foi divulgado da implementação, o que dificulta o desenvolvimento de algo equivalente. Já os trabalhos propostos por (Liu et al., 2021) e (Diniz et al., 2022), embora sejam problemas de *sequence-to-label*, trazem abordagens completas, desde o *embedding* do dado até o tratamento das *features*. Ambas também disponibilizam implementações bem documentadas e de fácil reprodução.

As arquiteturas *sequence-to-sequence* permitem o uso de *decoders* arbitrários, o que significa que ambas as soluções dos artigos poderiam ser possivelmente aproveitadas. Nenhuma das soluções apresentadas menciona tratamento para o caso de atributos categóricos, o que não permitiria o uso de atributos categóricos durante o treino.

Ambas as arquiteturas *encoder-only* permitem o uso de pré-treino dos modelos, técnicas na qual o peso de um modelo já treinado é reaproveitado. Esta também é uma abordagem interessante sobre o tempo de desenvolvimento.

As soluções *encoder-only* são utilizadas em cima de diversos *datasets*, com sequências de diversos tamanhos. Nesse caso, é possível testar as soluções com diferentes tamanhos de sequências de fácies.

No geral, podemos resumir a diferença entre as abordagens nos seguintes pontos:

- Tipo de arquitetura (*encoder-decoder* X *decoder-only*)
- Formato da entrada (comprimento da série temporal)
- Formato da saída (valor único X sequência)
- *Positional encoding* (Time2Vec, etc.)
- Possibilidade de pré-treino

4 Desenvolvimento

Neste capítulo é explicado o processo de desenvolvimento planejado para esse trabalho. Serão mostradas as ferramentas que foram utilizadas, tanto para a construção do módulo, quanto para o pré-processamento dos dados e monitoramento dos experimentos. Em seguida, serão apresentados os conjuntos de dados utilizados por esse trabalho. Na sequência, cada modelo é mostrado de uma maneira mais aprofundada. Além dos modelos implementados, também é discutido o modelo produzido pelo ganhador da (Bormann et al., 2020a), Olawale. Finalmente, o procedimento de treino é apresentado. O objetivo é propor diferentes modelos para a classificação de litofácies, fazer uma análise sobre os dados dos poços, propor técnicas de pré-processamento para os dados e por fim, definir as métricas para avaliação final.

4.1 Ferramentas

Essa seção apresenta as ferramentas utilizadas no desenvolvimento de cada etapa das arquiteturas propostas.

4.1.1 Criação de Modelos

O principal *framework* utilizado para construção de modelos foi o Pytorch. O Pytorch é uma biblioteca de *machine learning* escrita em Python, utilizada para a construção de redes neurais. Ela possui uma API simples que permite construir os modelos de maneira personalizada com facilidade. Foi escolhida principalmente pelo fato da maioria dos trabalhos levantados utilizá-la.

Além dela, outro *framework* utilizado foi a biblioteca *transformers* da HuggingFace. Essa biblioteca fornece implementações de alto nível para modelos transformadores. A escolha dessa biblioteca foi para o uso dessas APIs para facilitar o treino e predição dos modelos transformadores, principalmente, do modelo *encoder-decoder*. Um exemplo de abstração utilizada é o método *generate*, que já implementa diversos métodos de decodificação para transformadores auto-regressivos, como o *Beam Search*. A biblioteca *transformers* integra diretamente com o Pytorch e Tensorflow, e exigiu apenas a adição de algumas interfaces nos modelos implementados.

Finalmente, também foi utilizado a biblioteca XGBoost para a implementação do modelo XGBoost.

4.1.2 Experimentos

Foi utilizado o TensorBoard para monitorar a perda e o progresso do treinamento do modelo de aprendizado de máquina. O TensorBoard é uma ferramenta fornecida pelo *framework* TensorFlow que permite visualizar várias métricas e características do modelo.

4.1.3 Tratamento de Dados

A manipulação dos dados foi feita com o Pytorch e, além dele, também foram utilizadas as bibliotecas Pandas e Numpy. Pandas foi bastante utilizado na etapa de carregamento dos dados, para agilizar o processo.

Na etapa de reprocessamento foi utilizado o Sklearn e o Pycaret. Pycaret é uma biblioteca de AutoML escrita em Python. Essa biblioteca foi utilizada apenas para pré-processamento, já que possui uma API simples para o tratamento dos dados da maneira desejada.

4.2 Modelos

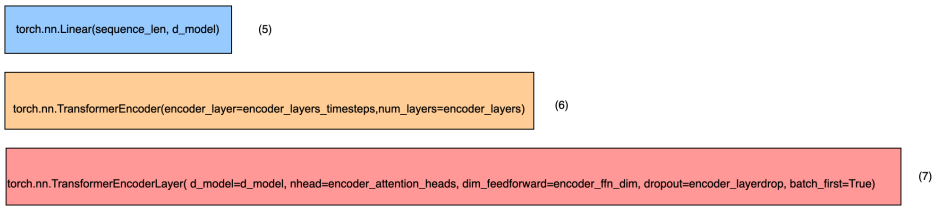
Foram propostas duas soluções utilizando transformadores. A primeira solução é idêntica à solução proposta por (Liu et al., 2021), que possui dois *encoders*. A segunda solução foi utilizar os mesmos *decoders* da primeira, porém traz também um *decoder* que recebe as facies previstas em passos anteriores. Por fim, também foi treinado um modelo XGBoost para ser utilizado como base. Esse modelo utiliza os mesmos parâmetros utilizados pelo ganhador da competição. Nesta seção, serão apresentadas as implementações desses modelos em mais detalhes.

4.2.1 Transformer com um Encoder

Essa arquitetura segue a implementação proposta no trabalho de (Liu et al., 2021). A ideia principal é conseguir capturar os pesos entre atributos diferentes em uma mesma profundidade e os pesos entre passos de tempo diferentes para um mesmo atributo. Para tal, esse modelo é composto por um codificador temporal e um codificador de atributos. Os codificadores se diferenciam no formato de entrada do dado, como será ilustrado adiante.

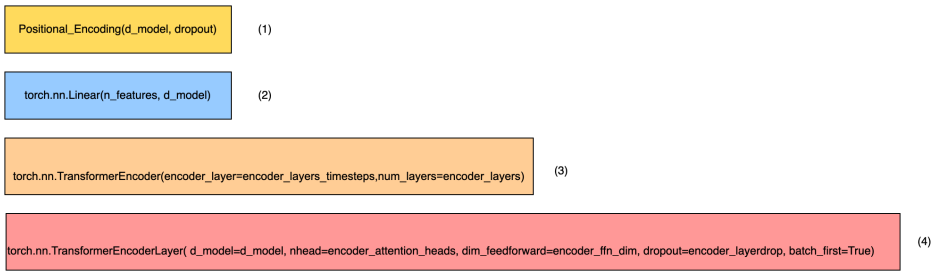
Além desses dois codificadores descritos pelo autor como “torres”, o modelo ainda conta com um mecanismo de *gating*. Esse mecanismo é responsável por capturar e comparar os pesos de cada torre e aprender a contribuição de ambos para a predição da facie. Finalmente, esse peso é utilizado para fazer a predição através de uma softmax sobre um *Linear Layer*.

Figura 33 – Blocos de transformers para *step-wise-encoder* e *channel-wise-encoder* do pacote torch



Fonte: (Tripathi, 2022)

Figura 34 – Blocos de transformers para *channel-wise-encoder* com camadas do pacote *torch*



Fonte: (Tripathi, 2022)

4.2.1.1 Camadas

LinearLayers: São utilizadas duas camadas lineares para os dados de entrada, uma para *step-wise-encoder* e a outra para o *channel-wise*. Nota-se que é utilizada uma camada linear por se tratarem de dados contínuos, diferente dos *Embeddings* utilizados por (Vaswani et al., 2017) na implementação original.

PositionalEncoding: Também é criado um *Positional Encoding* ((1) da figura para ser adicionado ao *input* do *step-wise-encoder*, como forma de codificar a posição de cada atributo na sequência durante o cálculo da atenção. O mesmo não é feito para *channel-wise-encoder* uma vez que essa relação não existe entre atributos de uma mesma profundidade. O camada PoisitionalEncoding é implementada utilizando o módulo nn.Module do torch

```
class PositionalEncoding(nn.Module):
    "Implement the PE function."

    def __init__(self, d_model, dropout, max_len=5000):
        super(PositionalEncoding, self).__init__()
        self.dropout = nn.Dropout(p=dropout)

        # Compute the positional encodings once in log space.
        pe = torch.zeros(max_len, d_model)
```

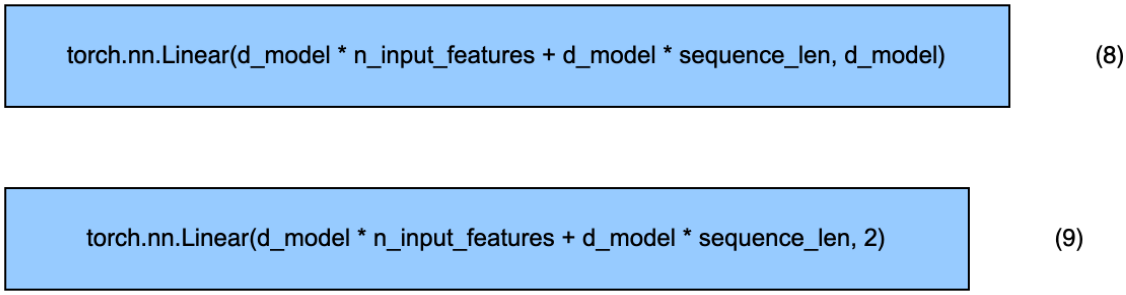
```
        position = torch.arange(0, max_len).unsqueeze(1)
        div_term = torch.exp(
            torch.arange(0, d_model, 2) * -(math.log(10000.0) / d_model)
        )
        pe[:, 0::2] = torch.sin(position * div_term)
        pe[:, 1::2] = torch.cos(position * div_term)
        pe = pe.unsqueeze(0)
        self.register_buffer("pe", pe)

    def forward(self, x):
        x = x + Variable(self.pe[:, : x.size(1)], requires_grad=False)
        return self.dropout(x)
```

Listing 4.1 – Implementacao do Poisitional Encoding em torch*torch*

Encoders: Para umas das torres codificadoras, são criados *layers* de *encoders* utilizando *TransformersEncoders* do Pytorch. Cada torre utiliza o *TransformerEncoder*, que por sua vez possui números arbitrários de *TransformerEncoderLayers*.

Gating e Output: A figura 35 mostra camada linear é utilizada para o modelo de *gating* (8) e outra para a camada de saída (9).

Figura 35 – Camadas Pytorch para o *Gate* e *Linear*

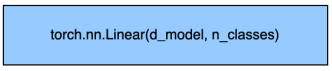


Fonte: (Tripathi, 2022)

Classification Layer: A figura 36 mostra camada linear utilizada como saída para a para classificação.

A tabela 2 descreve os parametros utilizados pelas camadas da arquitetura encoder-only.

Figura 36 – Camadas Pytorch para o *Gate* e *Linear*



(7)

Fonte: (Tripathi, 2022)

Tabela 2 – Parâmetros usados pelas camadas dos modelos transformers.

Parâmetro	Descrição
`encoder_layers`	Número camadas encoder
`encoder_ffn_dim`	Dimensão da camada feedforward do encoder
`encoder_attention_heads`	Número de attention heads no encoder
`encoder_layerdrop`	Taxa de dropout no bloco encoder
`activation_function` elu	Função de ativação usada
`d_model`	Dimensão dos embeddings
`n_input_features`	número de atributos
`n_output_features`	número de classes no atributo alvo
`sequence_len`	comprimento das sequências
`dropout`	Taxa de dropout
`attention_dropout`	Dropout na cross attention
`activation_dropout`	Dropout na camada de ativação
`classification_dropout`	Dropout na camada de classificação

4.2.1.2 Feedforward

Nessa etapa será mostrado o fluxo do modelo.

Step-Wise-Encoder: os dados de entrada no formato de

$$[batch\_size, sequence\_len, input\_len]$$

são (1) projetados linearmente pelo stepwiselinearlayer; (2) concatenados pelo *positional encoder* e (3) utilizados como entrada na *step-wise tower*.

```
encoding_1 = self.embed_tokens_steps(input_ids)

encoding_1 = self.positional_encoding(encoding_1)

output_encoder_1 = self.model_timestep(
    encoding_1, mask = None, src_key_padding_mask=None
)
```

Listing 4.2 – Inputs sendo passados pelas camadas do step-wise-encoder

Nota-se que os dados de entrada projetada para essa torre possuem dimensões

$$[batch\_size, sequence\_len, input\_len]$$

. O que condiz com a atenção ao longo dos elementos da sequência.

Channel-Wise-Encoder: nesta torre, os dados são primeiramente transpostos de maneira que a projeção seja feita ao longo dos atributos da mesma sequência e depois, passados como *input* aos codificadores de atributo.

```
    channel_encoding = self.embed_tokens_channel(input_ids.transpose(-1,
-2))
    output_encoder_2 = self.model_channel(
        channel_encoding, mask=None, src_key_padding_mask=None
    )
     )
```

Listing 4.3 – Inputs sendo passados pelo channel encoding

Gating: As saídas de ambas as torres codificadores é concatenada e em seguida passada pelo *layer gate*. O *gate* então é usado de entrada para o softmax gerando dois pesos, g1 e g2. Ambos são concatenados e multiplicados pelas respectivas saídas dos codificadores. Por fim, o resultado da concatenação é passado pela camada de saída.

```
    output_encoder_1 = output_encoder_1.reshape(output_encoder_1.shape[0],
-1)
    output_encoder_2 = output_encoder_2.reshape(output_encoder_2.shape[0],
-1)
    gate = F.softmax(
        self.gate(torch.cat([output_encoder_1, output_encoder_2],
dim=-1)), dim=-1
    )
    encoding = torch.cat(
        [output_encoder_1 * gate[:, 0:1], output_encoder_2 * gate[:,
1:2]], dim=-1
    )
```

Listing 4.4 – Saída dos blocos encoder sendo passadas para o mecanismo de *gate*

Output: Finalmente, a saída do *gating* é projetada linearmente para que possa ser utilizada para classificação.

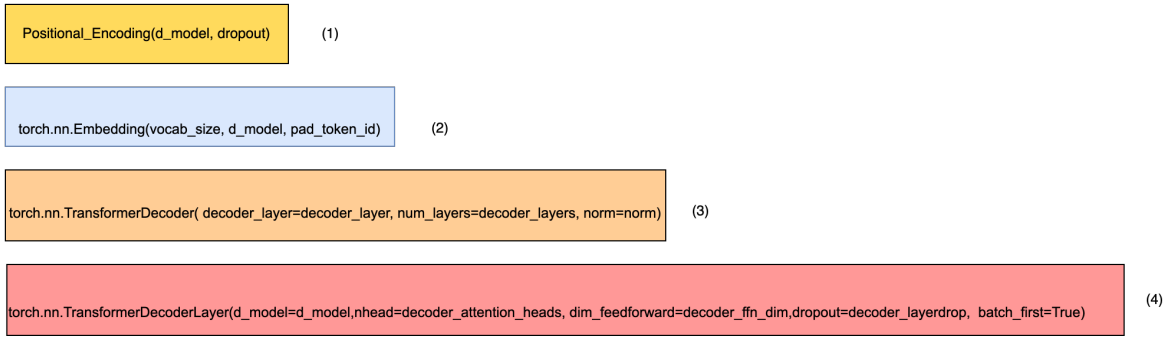
4.2.2 Transformador Encoder-Decoder

Esta segunda arquitetura de transformadores implementada, conta com uma etapa de codificação e uma etapa de decodificação. A etapa de codificação usa blocos codificadores idênticos aos de (Liu et al., 2021) e por isso usa a mesma implementação da primeira arquitetura proposta. O decodificador, assim como na implementação original de (Vaswani et al., 2017), funciona de maneira auto-regressiva, consumindo os *tokens* (litofácies) de posições anteriores durante o treino e a previsão. Durante o treino, isso é feito através de batches de sequência deslocadas uma posição para a direita, uma máscara para as posições futuras.

O bloco conta com duas etapas principais. A primeira etapa é, a *self-attention*, e recebe como entrada um *batch* contendo sequências de previsões deslocadas de uma posição para a direita. A segunda etapa calcula a *cross-attention* utilizando as saídas da sua primeira etapa, e dos blocos *encoder*.

4.2.2.1 Camadas

Figura 37 – Camadas Pytorch para o bloco decoder



Fonte: (Tripathi, 2022)

Embedding:Camada embedding do *torch*, com os parametros mostradas na figura 37.

Positional Encoding: Mesma implementação da primeira arquitetura proposta.

Encoder: Mesma implementação da arquitetura encoder-only.

Decoder: Modelos TranasformerDecoder da biblioteca do Pytorch como mostrado na figura 37.

Output: Uma camada linear do Pytorch identica a mostra na figura 36.

A tabela 3 descreve os parametros utilizados pelas camadas na arquitetura encoder-decoder.

Tabela 3 – Parâmetros usados pelas camadas dos modelos transformers encoder-decoder.

Parâmetro	Descrição
`encoder_layers`	Número camadas encoder
`encoder_ffn_dim`	Dimensão da camada feedforward do encoder
`encoder_attention_heads`	Número de attention heads no encoder
`decoder_layers`	Número de camadas decoder no bloco decoder
`decoder_ffn_dim`	Dimensão da camada feedfoward do bloco decoder
`decoder_attention_heads`	Número de attention heads no decoder
`encoder_layerdrop`	Taxa de dropout no bloco encoder
`decoder_layerdrop`	Taxa de dropout no bloco decoder
`activation_function` elu	Função de ativação usada
`d_model`	Dimensão dos embeddings
`n_input_features`	número de atributos
`n_output_features`	número de classes no atributo alvo
`sequence_len`	comprimento das sequências
`dropout`	Taxa de dropout
`attention_dropout`	Dropout na cross attention
`activation_dropout`	Dropout na camada de ativação
`Dropout no classificador`	Dropout na camada de classificação

4.2.2.2 Feedforward

Encoder: Assim como na primeira implementação, o *encoder* recebe as curvas de *logs* de poços. A diferença é que ao invés de usar essa saída para a classificação em um bloco linear, ela será passada para o bloco *decoder*.

Deslocamento + BOS Token: Como o objetivo é calcular a atenção utilizando apenas facies anteriores, não é necessário passar a última fase de cada sequência. Por isso, cada sequência de facies é deslocada um elemento para a direita, com a adição de um índice de *padding* no início.

```
def shift_tokens_right(
    input_ids: torch.Tensor, pad_token_id: int, decoder_start_token_id: int
):
    """
    Shift input ids one token to the right.
    """
    shifted_input_ids = input_ids.new_zeros(input_ids.shape)
    shifted_input_ids[:, 1:] = input_ids[:, :-1].clone()
    shifted_input_ids[:, 0] = decoder_start_token_id

    if pad_token_id is None:
        raise ValueError("self.model.config.pad_token_id has to be defined.")
```

```
    shifted_input_ids.masked_fill_(shifted_input_ids == decoder_start_token_id,
    pad_token_id)

    return shifted_input_ids
```

Esse índice de *padding* também serve uma função similar ao "BOS" (Vaswani et al., 2017) utilizado em problemas de NLP. Entretanto, como prever o início de uma sequência não interessa o problema, esse índice é apenas ignorado durante o cálculo da atenção.

Embedding + Positional Encoding: Os dados de entrada do decodificador já deslocados passam pela camada *embedding* dos dados de entrada. Esse *embedding* projeta os dados de entrada de maneira que a posição entre cada facie seja considerada. Nele, é adicionado um *positional encoding*.

Masking: Uma máscara é criada no formato de uma matriz triangular, com -inf nos elementos da parte superior da matriz. Essa máscara garante que apenas posições de facies anteriores sejam consideradas durante a previsão para uma determinada posição.

```
def _make_causal_mask(
    input_ids_shape: torch.Size, dtype: torch.dtype, past_key_values_length:
    int = 0
):

    bsz, tgt_len = input_ids_shape
    mask = torch.full((tgt_len, tgt_len), torch.tensor(torch.finfo(dtype).min))
    mask_cond = torch.arange(mask.size(-1))
    mask.masked_fill_(mask_cond < (mask_cond + 1).view(mask.size(-1), 1), 0)
    mask = mask.to(dtype)

    if past_key_values_length > 0:
        mask = torch.cat(
            [torch.zeros(tgt_len, past_key_values_length, dtype=dtype), mask],
    dim=-1
        )
    return mask
```

Decoder: Os blocos do *decoder* recebem como entrada a saída do bloco *encoder*, as sequências de facies já deslocadas e máscara para posições anteriores.

```
        decoder_outputs = self.model(
            hidden_state,
            encoder_hidden_states,
```

```
            tgt_mask=attention_mask,
        )
```

Classification Finalmente, a saída do *decoder* é passado para uma cada linear, que, depois de passada por uma softmax, irá projetar as probabilidades para cada facie.

```
lm_logits = self.lm_head(outputs[0])
```

4.2.3 XGBoost

Um modelo XGBoost foi utilizado como base de comparação para avaliar a eficácia dos métodos propostos. O modelo utiliza os mesmos parâmetros do modelo proposto pelo vencedor da competição, Olawell. O objetivo é criar um modelo base que já consiga resultados desafiadores sem gerar um trabalho significativo. Por conta disso, não foram utilizadas as mesmas técnicas de engenharia de atributos, como os atributos delta propostos por (Bestagini et al., 2017).

```
model = XGBClassifier(n_estimators=100, max_depth=10, booster='gbtree',
                              objective='multi:softprob', learning_rate=0.1,
    random_state=0,
                              subsample=0.9, colsample_bytree=0.9,
    tree_method='gpu_hist',
                              eval_metric='mlogloss', verbose=2020,
    reg_lambda=1500)
```

4.3 Dados

O objetivo deste trabalho é buscar soluções de transformadores que obtenham resultados relevantes no conjunto de dados fornecidos pela organização XEEK FORCE (Bormann et al., 2020b) durante a competição de 2020. Nessa seção, esse conjunto de dados será explorado mais a fundo

Para cada profundidade do poço, existem diversos atributos de treino relacionados, além da litofácies que é o atributo alvo.

4.3.1 Atributos

Os dados do conjunto estão separados em poços, com 98 poços separados para treino, e 10 poços para teste. Esses 10 poços de teste são os mesmo utilizados para avaliar a competição. A Tabela 4 apresenta esses atributos

Tabela 4 – Atributos disponíveis para o conjunto de dados de poços.

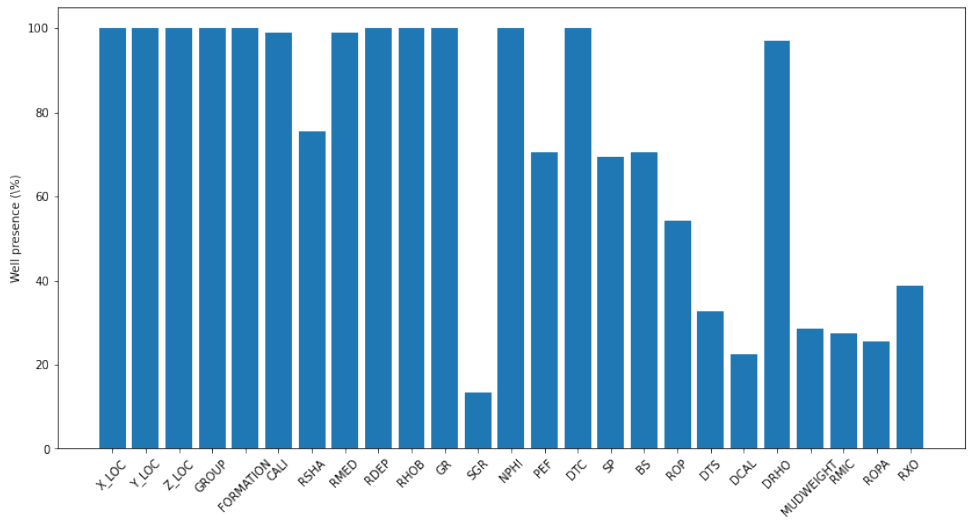
Atributo	Significado
DEPTH_MD	measured depth
X_LOC	UTM X coordinate
Y_LOC	UTM Y coordinate
Z_LOC	DEPTH
GROUP	NPD lithostratigraphy group
FORMATION	NPD lithostratgraphy formation
BS	Bit Size
CALI	Caliper
RDEP	Deep Resistivity
RHOB	Bulk Density
GR	Raw gamma data
SGR	Spectral Gamma Ray
RMED	Medium Resistivity
ROP	Rate of Penetration
NPHI	Neutron Porosity
PEF	Photoelectric Absorption Factor
RSHA	Shallow Resistivity
DTS	Sonic (Sheer Slowness
DTC	Sonic (Compressional Slowness

Fonte: (Bormann et al., 2020a)

4.3.2 Dados faltantes

Um número significativo de amostras possui dados faltantes. a figura 38 mostra essa relação. O método para lidar com essas dados faltante é descrito de acordo com cada modelo junto com as técnicas de pré-processamento.

Figura 38 – Relação de atributos faltando nos poços.



Fonte: (Bormann et al., 2020a)

4.3.3 Atributo Alvo

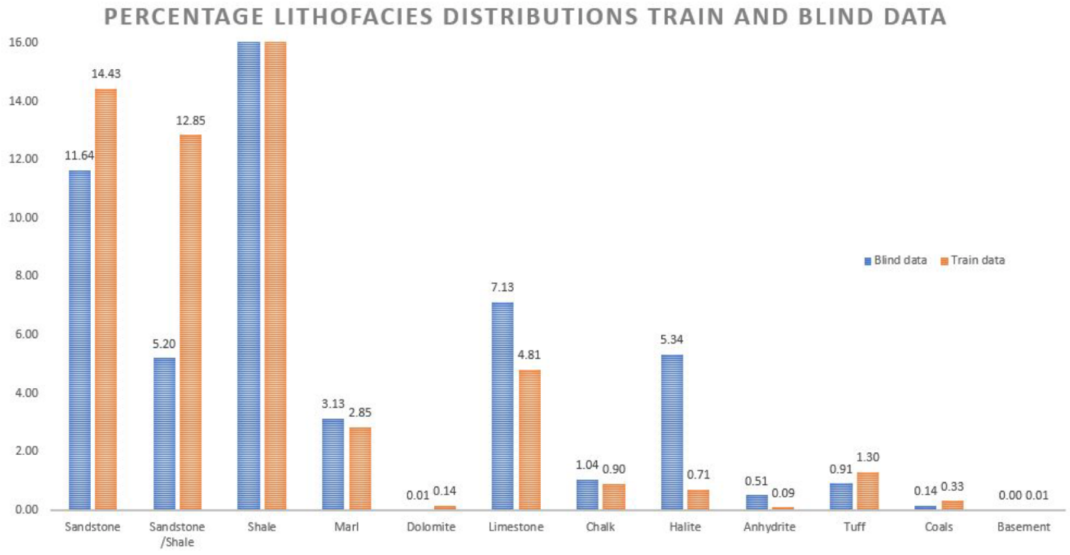
A tabela 5 mostra as litofácies contidas no conjunto de dados que foram utilizadas como atributos alvos. O gráfico na 39 mostra como as diferentes litofácies estavam distribuidas nos *datasets* de treino e teste. Existe um desbalanceamento claro, com mais de 16% dos das amostras contendo *Shale*. Entretanto, nenhuma técnica específica foi utilizada para lidar com o desbalanceamento.

Tabela 5 – Litofácies presentes nos conjuntos de dados da competição e seus respectivos códigos.

Lithology Key	Lithology Name
30000	Sandstone
65030	Sandstone/Shale
65000	Shale
80000	Marl
74000	Dolomite
70000	Limestone
70032	Chalk
88000	Halite
86000	Anhydrite
99000	Tuff
90000	Coal
93000	Basement

Fonte: (Bormann et al., 2020a)

Figura 39 – Distribuição das litofácies entre os conjuntos de teste e de treino.



Fonte: (Bormann et al., 2020a)

4.4 Pré-processamento dos dados

Esta seção explica o pré-processamento feito nos dados para prepará-los para o treinamento. O processamento inclui a normalização dos dados, a imputação dos dados faltantes e a manipulação do formato dos dados de acordo com o algoritmo utilizado. Também serão mostrados os atributos escolhidos. A Tabela 6 da uma visão geral do que foi feito para cada modelo.

Tabela 6 – Tabela resumindo o pré processamento utilizado nos dados de entrada de cada um dos modelos.

Modelo	Normalização	Imputação	Feature Engineering	Codificação Categoria	Formato atributos de entrada	Formato Facies
Encoder-Only	StandardScaler	ffill + bbfill	-	LabelEncoder	(Batch_size, Sequence_len, Atributos)	(Batch_size, 1)
Encode-Decoder	StandardScaler	ffill + bbfill	-	LabelEncoder	(Batch_size, Sequence_len, Atributos)	(Batch_size, Sequence_len)
Simple XGBoost	StandardScaler	ffill + bbfill	-	LabelEncoder	(Número de Amostras, número de Atributos)	(Número de atributos, 1)
Orwale's XGBoost	StandardScaler	-999	Paolo Bestagini's Gradient	LabelEncoder	(Número de Amostras, número de Atributos)	(Número de atributos, 1)

Fonte: O autor (2022).

4.4.1 Seleção dos Atributos

Para facilitar a comparação direta, a seleção de atributos foi baseada na solução proposta pelo participante Olawale. O participante escolheu os atributos baseando-se principalmente na quantidade de dados faltantes.

- Atributos Utilizados: `WELL, DEPTH_MD, X_LOC, Y_LOC, Z_LOC, GROUP, FORMATION, CALI, RSHA, RMED, RDEP, RHOB, GR, SGR, NPHI, PEF, DTC, SP, BS, ROP, DTS, DCAL, DRHO, MUDWEIGHT, RMIC`
- Atributos não selecionados: ROPA, RXO, DTS, SGR

4.4.2 Imputação

Para simplificar o processo, todos os dados foram imputados utilizando *forward fill* da biblioteca Pandas e em seguida *backfill*. O *forward fill* preenche um dado faltante com o valor da primeira amostra daquele produto igual da primeira amostra não faltante. O *backfill* faz o mesmo processo utilizando dados anteriores. Essa abordagem foi escolhida por ser simples e por amostras próximas serem também fisicamente próximas.

4.4.3 Normalização

Para a normalização dos dados, foi utilizado o *Standard Scaler* do SKLearn.

4.4.4 Separação dos Dados

Encoder-only: os dados tiveram que ser separados em sequências de mesmo tamanho. Esse processo foi feito em 3 etapas:

- Divisão dos dados em poços: para garantir que nenhuma sequência contendo dados de poços diferentes;
- Divisão dos dados em sequências: divisão de cada poço em sequência utilizando uma *sliding window* para cada profundidade. Cada sequência foi formada por uma amostra no início da sequência e mais amostras de acordo com a profundidade desejada. A face correspondente a cada sequência está localizada na profundidade da primeira amostra da sequência;
- Preenchimento das sequências Incompletas: Para que todas as sequências tivessem o mesmo tamanho, foi feita uma imputação utilizando *fill forward*.

Encoder-Decoder: o formato do dado de entrada foi dado por: (sequência-atributos) -> (sequência-y)

- Divisão dos dados em poços: Para garantir que nenhuma sequência contendo dados de poços diferentes.
- Divisão dos dados em sequências: Divisão de cada poço em sequências, de maneira que para cada sequência de amostras, exista uma sequência de facies de mesmo tamanho para a mesma profundidade.
- Preenchimento das sequências Incompletas: Para que todas as sequências tivessem o mesmo tamanho, foi feita uma imputação utilizando *fill forward* nas sequências faltantes.

XGBoost: as amostras de treino não foram separadas em sequência, apenas embaralhadas.

4.4.5 Engenharia de atributos

Transformers and XGBoost: Para os dados de entrada dos modelos transformadores e para o XGBoost do exemplo, a única transformação significativa dos dados além da normalização foi a codificação dos dados categóricos. Para as colunas "Group", "Formation"e "Well", foi feito uma codificação ordinal, atribuindo um inteiro começando em zero para cada classe do atributo com o id máximo sendo o número de classes.

Solução de Olawale Ibrahim Na solução proposta pelo competidor, além da codificação das classes categóricas, também foi utilizada a técnica proposta por Paolo Bestagini, para gerar novos atributos utilizando gradientes de curvas já presentes. Nota-se que tais técnicas foram utilizadas pela maior parte dos competidores que lideraram a competição.

4.5 Treino

Esta seção pretende explicar como foi feito o treino de cada um dos modelos propostos.

4.5.1 Loss Function

A *loss function* utilizada foi a *Categorical cross-entropy*.

4.5.2 Otimização de hiperparâmetros

Em um primeiro momento, foi feita uma busca de hiperparâmetros utilizando com a ajuda da API de *Transformers* utilizando o método *hyper parameter search* do *trainer*. A biblioteca oferece diversas alternativas para *backend*. A utilizada foi da biblioteca RAY, pela facilidade de integração com a biblioteca, e facilidade de integração de *logs* junto ao *TensorBoard*.

4.5.3 Hiperparâmetros

Olawale e XGBoost: Pelo o código divulgado pelo (Bormann et al., 2020a), os hiperparâmetros foram escolhidos com a ajuda do GridSearch e também por escolhas manuais. Também foi utilizado um método de validação estratificado. Os parâmetros escolhidos são mostrados na Tabela 7. Os mesmos parâmetros foram utilizados para o XGboost base.

Tabela 7 – Valores do Hiperparametros para os modelos transformadores.

Hiperparâmetro	Valor
`n_estimators`	100
`objective`	multi:softprob'
`subsample`	0.9
`eval_metric`	mlogloss'
`max_depth`	10
`learning_rate`	0.1
`colsample_bytree`	0.9
`verbose`	2020
`booster`	gbtree'
`random_state`	0
`tree_method`	gpu_hist'
`reg_lambda`	1500

Fonte: (Bormann et al., 2020a)

Transformers: No caso dos modelos transformadores, os parâmetros mostrados na Tabela 8 foram obtidos apos uma *RandomSearch* utilizando 20% dos dados como validação.

Tabela 8 – Valores do Hiperparâmetros para os modelos XG Boost.

Hiperparâmetro	Valor	Descrição
`encoder_layers`	6	Número camadas encoder
`encoder_ffn_dim`	512	Dimensão da camada feedforward do encoder
`encoder_attention_heads`	8	Número de attention heads no encoder
`decoder_layers`	6	Número de camadas decoder no bloco decoder
`decoder_ffn_dim`	512	Dimensão da camada feedfoward do bloco decoder
`decoder_attention_heads`	8	Número de attention heads no decoder
`encoder_layerdrop`	0.1	Taxa de dropout no bloco encoder
`decoder_layerdrop`	0.1	Taxa de dropout no bloco decoder
`activation_function`	relu	Função de ativação usada
`d_model`	512	Dimensão dos embeddings
`n _ features`	8	número de atributos
`n_output`	12	número de classes no atributo alvo
`sequence_ len`	10	comprimento das sequências
`dropout`	0.1	Taxa de dropout
`attention_dropout`	0.1	Dropout na cross attention
`activation_dropout`	0.1	Dropout na camada de ativação
`Dropout no classificador`	0.1	Dropout na camada de classificação

Fonte: (Bormann et al., 2020a)

4.6 Avaliação

Para medir a eficácia dos métodos propostos, foi utilizada a mesma matriz de pontuação utilizada na competição XEEK (Bormann et al., 2020a). A pontuação é dada pela soma dos erros de predições de litofácies para uma determinada amostra, com a penalidade variando pelo tipo do erro, de acordo com a matriz mostrada na figura 40 . Como exemplo, uma amostra de Sandstone prevista como Shale contabiliza 3.5 de penalidade. Já a mesma amostra classificada como Limestone sofre 4 pontos de penalidade, indicando que tal erro é menos aceitável. Note também que as diagonais, possuem valor 0, o que significa que a pontuação máxima é zero.

Figura 40 – Matriz usada para calcular o resultado baseado no tipo de erro da predição.

label \ prediction	Sandstone	Sandstone/shale	Shale	Marl	Dolomite	Limestone	Chalk	Halite	Anhydrite	Tuff	Coal	Crystalline Basement
Sandstone	0	2	3.5	3	3.75	3.5	3.5	4	4	2.5	3.875	3.25
Sandstone/Shale	2	0	2.375	2.75	4	3.75	3.75	3.875	4	3	3.75	3
Shale	3.5	2.375	0	2	3.5	3.5	3.75	4	4	2.75	3.25	3
Marl	3	2.75	2	0	2.5	2	2.25	4	4	3.375	3.75	3.25
Dolomite	3.75	4	3.5	2.5	0	2.625	2.875	3.75	3.25	3	4	3.625
Limestone	3.5	3.75	3.5	2	2.625	0	1.375	4	3.75	3.5	4	3.625
Chalk	3.5	3.75	3.75	2.25	2.875	1.375	0	4	3.75	3.125	4	3.75
Halite	4	3.875	4	4	3.75	4	4	0	2.75	3.75	3.75	4
Anhydrite	4	4	4	4	3.25	3.75	3.75	2.75	0	4	4	3.875
Tuff	2.5	3	2.75	3.375	3	3.5	3.125	3.75	4	0	2.5	3.25
Coal	3.875	3.75	3.25	3.75	4	4	4	3.75	4	2.5	0	4
Crystalline Basement	3.25	3	3	3.25	3.625	3.625	3.75	4	3.875	3.25	4	0

Fonte: (Bormann et al., 2020a)

Outras métricas de classificação também foram utilizadas.

- Média
- F1-Score
- Recall
- Precisão
- Pontuação da competição

5 Experimentos e Resultados

Essa seção mostra os resultados obtidos a partir do conjunto de dados de teste disponibilizados para a competição. A performance foi medida através de métricas normalmente utilizadas, como acurácia, *F-1-score*, além da própria métrica proposta pela competição. Serão apresentadas algumas visualizações importantes como a matriz de confusão das facies litológicas e curvas de facies através de gráficos. Além disso, detalhes de predições individuais para os diferentes modelos serão apresentadas. Os modelos foram treinados utilizando uma GPU RTX NVIDEA 2060. Por conta de limitações de tempo e de capacidade de processamento, os modelos foram treinados por apenas 3 *epochs*

5.1 Métricas para todos os modelos

A Tabela 9 mostra os resultados que cada um dos modelos testados obtiveram em todos os poços de teste. De maneira geral, o modelo do ganhador da competição, de Olawale, obteve os melhores resultados em todas as métricas propostas.

Tabela 9 – Métricas de classificação para em relação a todos os poços, para cada modelo

	accuracy	precision	recall	f1	competition_score
Encoder-Decoder Model	0.728964	0.477379	0.409583	0.423422	-0.679454
Olawale's Model	**0.805273**	**0.593302**	**0.532208**	**0.547874**	**-0.46978**
Encoder-Only Model	0.761979	0.524509	0.36536	0.347022	-0.620015
XGBoost Model	0.744201	0.512636	0.455306	0.469092	-0.630823

Fonte: O autor (2022).

O modelo *Encoder-only* obteve resultados significativos também, atingindo o segundo melhor resultado na métrica da competição, a segunda melhor acurácia, e o segundo melhor *recall*. Ele perde para o XGBoost no *recall* e no *F-1-score*.

O modelo *Encoder-decoder* ficou em último para todas as métricas apresentadas.

Os resultados dessa tabela mostram que, a princípio, adicionar o *decoder* no modelo de previsão de fácies não mostrou vantagens em relação ao modelo de apenas um *encoder*; na verdade, isso piorou os resultados.

Tabela 10 – Acurácia total para tipos de facie, por para cada um dos modelos.

	Sandstone	Sandstone/Shale	Shale	Marl	Dolomite	Limestone	Chalk	Halite	Anhydrite	Tuff	Coal	Basement
Encoder-Decoder Model	0.588822	0.513392	0.909978	0.430619	0.463415	0.509434	0.661962	0.976916	0.675042	0.682253	0.491803	nan
Olawale Model	**0.796084**	0.537247	0.955128	0.370564	**0.533101**	**0.634703**	0.634423	**0.993998**	**0.782245**	**0.755579**	0.573770	nan
Encoder-Only Model	0.677536	0.500692	**0.962284**	**0.543904**	0.484321	0.482207	**0.890878**	0.964451	0.659966	0.741764	**0.631148**	nan
XGBoost Model	0.682236	**0.558495**	0.922494	0.518198	0.487805	0.403989	0.745955	0.985226	0.380235	0.555792	0.627049	nan

Fonte: O autor (2022).

A Tabela 10 mostra a porcentagem de acertos de cada um dos modelos para cada uma das facies contidas nos poços de teste. Com exceção das fábricas.

O modelo de Olawale obteve os melhores resultados para todas as facies, com exceção de: Sandstone/Shale, Shale, Marl, Chalk e Coal. Em segundo lugar, o modelo classificado foi o *Encoder-only*, que atingiu os melhores resultados em Shale Marl Chalk Coal. Em terceiro, o Xgboost obteve o melhor resultado apenas no Sandstone/shale. Novamente, o modelo Encoder-decoder ficou em último, sem ter obtido o melhor resultado para nenhuma das facies estudadas.

Esses resultados deixam ainda mais evidente a falta de vantagem do *Encoder-decoder*, ainda se olhando tipos de facies específicas. O modelo *Encoder-only* mostra potencial ao conseguir superar o de Olawale para diversas fábricas.

5.2 Métricas por Poços

Nessa seção, são exploradas as métricas para cada poço. O objetivo dessa comparação é identificar possíveis vantagens entre modelos por conta de características específicas de cada poço.

Tabela 11 – Acurácia obtida em cada um dos poços, para cada modelo.

	Encoder-Decoder Model	Olawale's Model	Encoder-Only Model	XGBoost Model
15/9-23	0.779083	**0.811534**	0.798156	0.689777
16/2-7	0.562698	**0.812035**	0.630489	0.649320
16/7-6	0.762962	0.683428	0.694189	**0.798278**
17/4-1	0.618320	**0.770598**	0.752881	0.655955
25/10-9	0.753708	**0.809047**	0.630237	0.789025
31/2-10	0.867154	**0.906011**	0.855198	0.875346
31/2-21 S	0.737628	**0.853827**	0.810969	0.830485
34/3-2 S	0.873936	**0.893828**	0.840946	0.884414
35/11-5	0.716904	0.748889	**0.850074**	0.690057
35/9-7	0.715744	**0.852206**	0.697668	0.724923

Fonte: O autor (2022).

A tabela 11 mostra a porcentagem de acertos de cada um dos modelos para cada uma das fácies contidas nos poços de teste. Com exceção dos poços 35/11-5 e 16/7-6, o modelo de Olawale obteve os melhores resultados. O *Encoder-only* obteve o melhor resultado no poço 35/11-5.

Tabela 12 – Métrica da competição obtida em cada um dos poços, para cada modelo.

	Encoder-Decoder Model	Olawale's Model	Encoder-Only Model	XGBoost Model
15/9-23	-0.469674	**-0.433574**	-0.462081	-0.589939
16/2-7	-1.124315	**-0.479308**	-0.955737	-0.986209
16/7-6	-0.475152	-0.640934	-0.587373	**-0.424195**
17/4-1	-0.967373	**-0.530246**	-0.641263	-0.850761
25/10-9	-0.710002	**-0.509223**	-1.218877	-0.570877
31/2-10	-0.370378	**-0.259106**	-0.414508	-0.347739
31/2-21 S	-0.667044	**-0.374091**	-0.520982	-0.424075
34/3-2 S	-0.309921	**-0.277935**	-0.384926	-0.282529
35/11-5	-0.709238	-0.595810	**-0.393139**	-0.755069
35/9-7	-0.742168	**-0.408331**	-0.782722	-0.725344

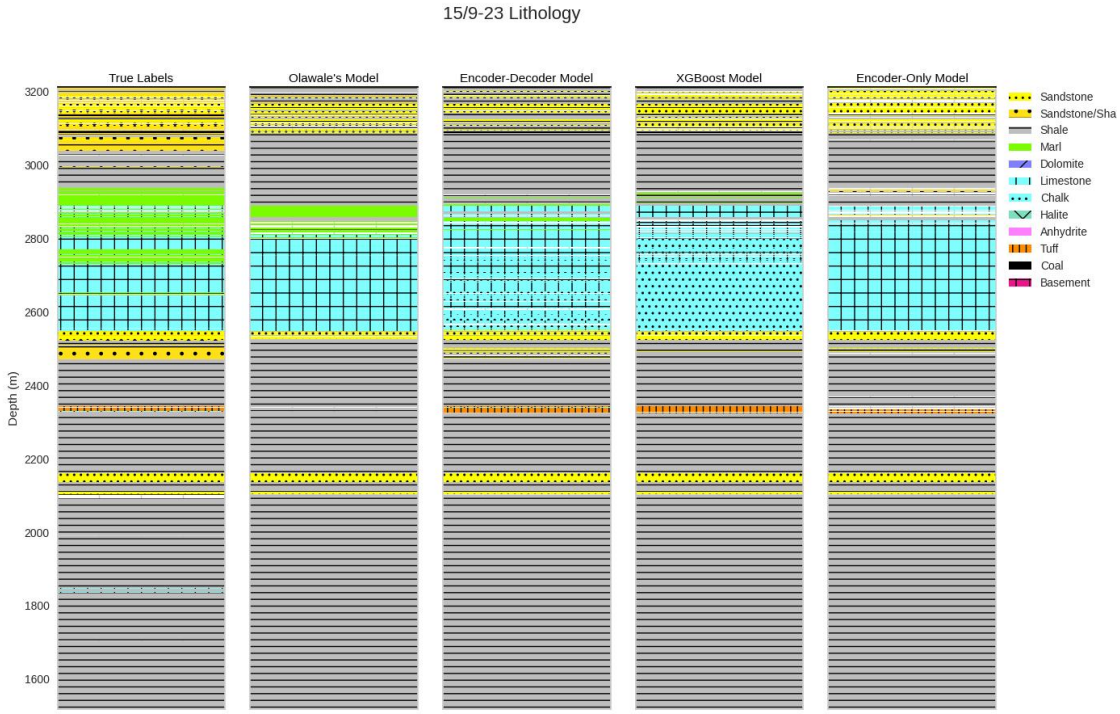
Fonte: O autor (2022).

A tabela 12 mostra a porcentagem de acertos de cada um dos modelos para cada uma das facies contidas nos poços de teste. Com exceção dos poços 35/11-5 e 16/7-6, o modelo de Olawale obteve os melhores resultados. O *Encoder-only* obteve o melhor resultado no poço 35/11-5.

5.3 Comparação Imagens de Poços

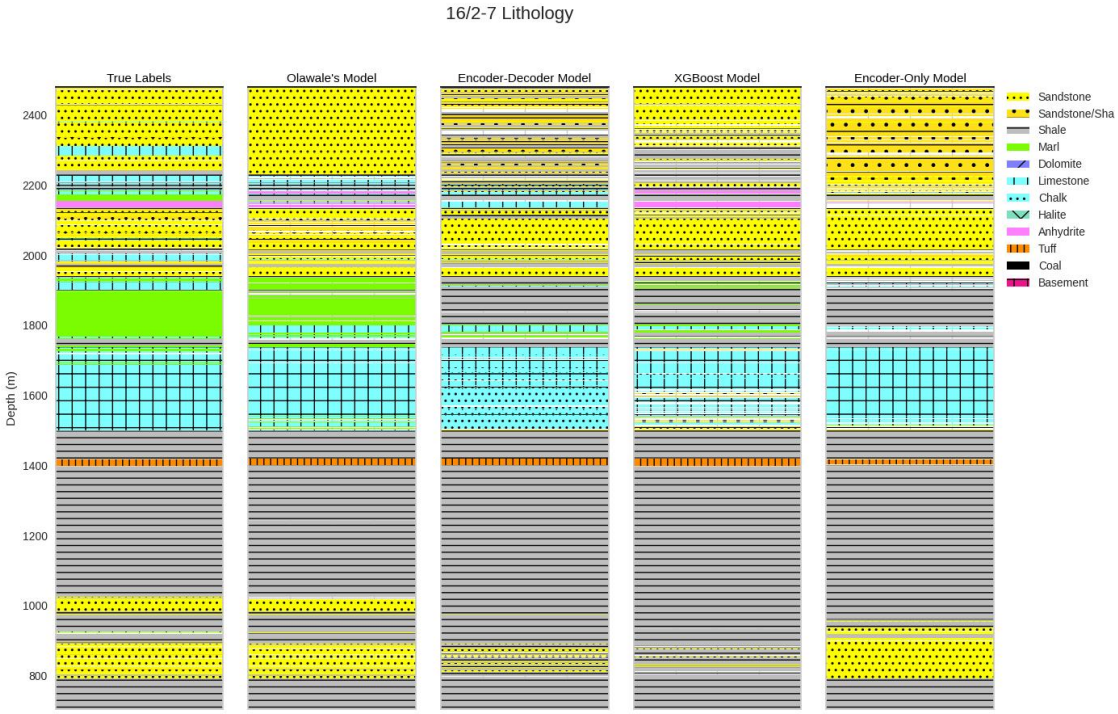
Nesta seção, são mostrados os resultados das predições de poços para cada um dos modelos lado a lado.

Figura 41 – Litofácies ao longo da profundidade do poco 15/_9-23



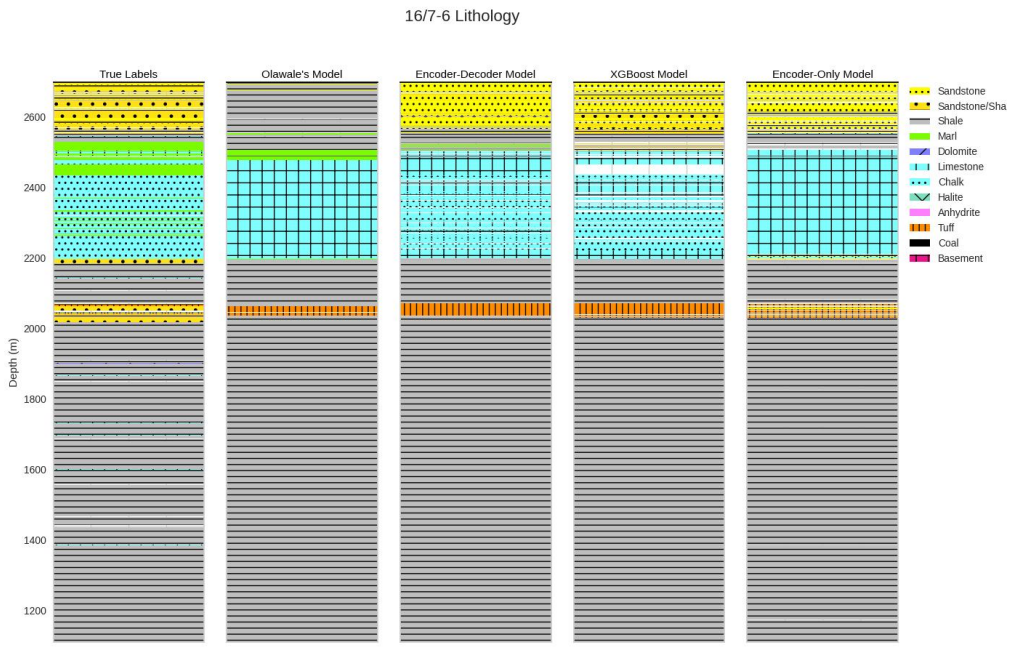
Fonte: O autor (2022).

Figura 42 – Litofácies ao longo da profundidade do poco 16/_2-7



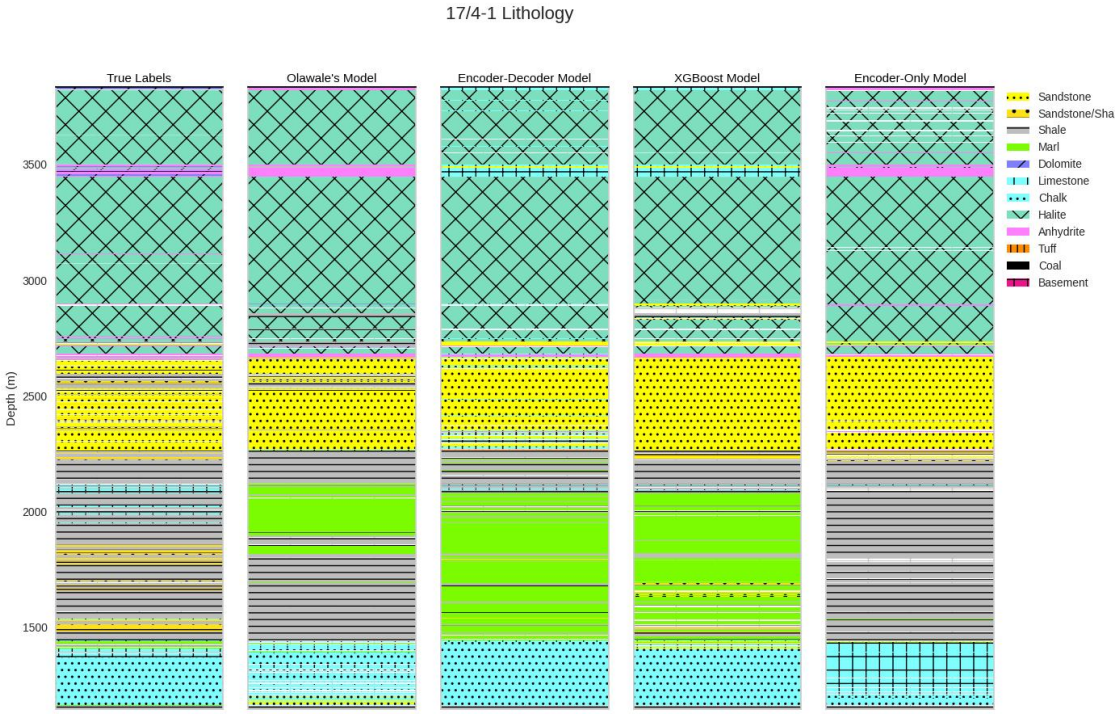
Fonte: O autor (2022).

Figura 43 – Litofácies ao longo da profundidade do poco 16/_7-6



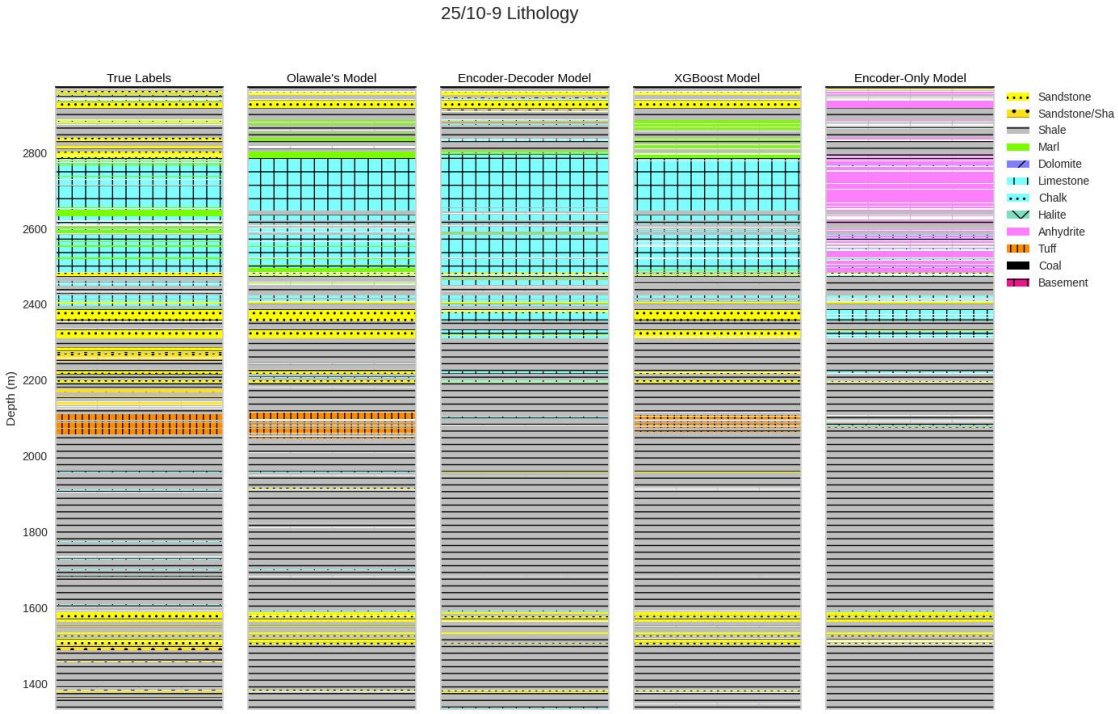
Fonte: O autor (2022).

Figura 44 – Litofácies ao longo da profundidade do poco 17/_4-1



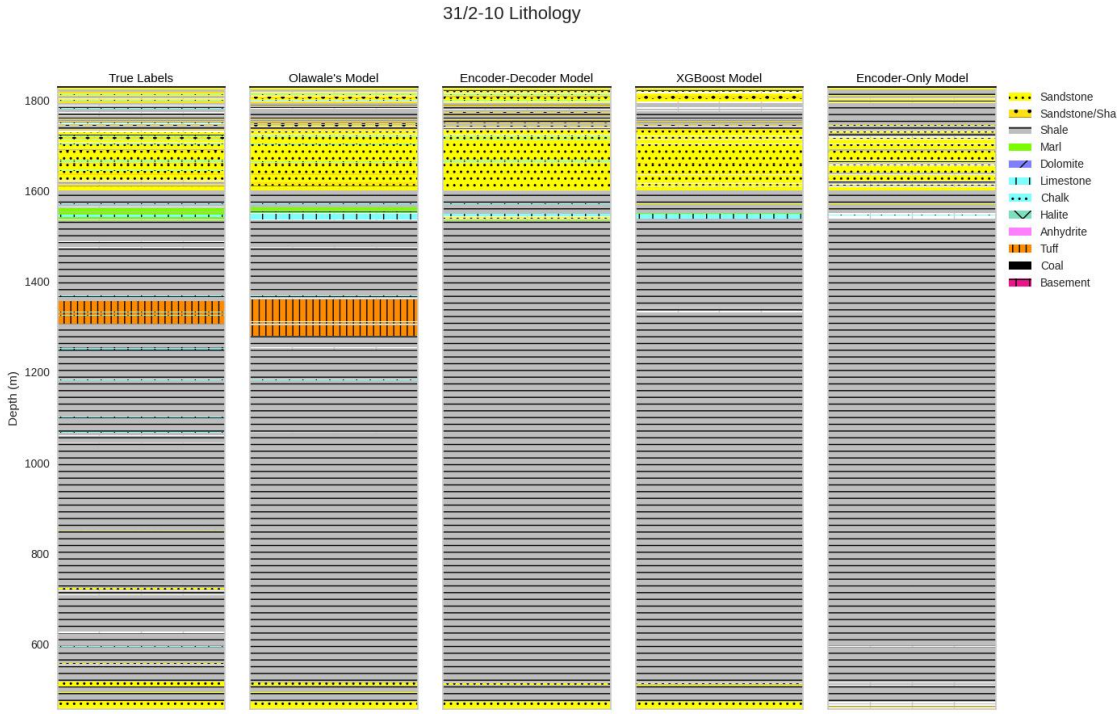
Fonte: O autor (2022).

Figura 45 – Litofácies ao longo da profundidade do poco 25/_10-9



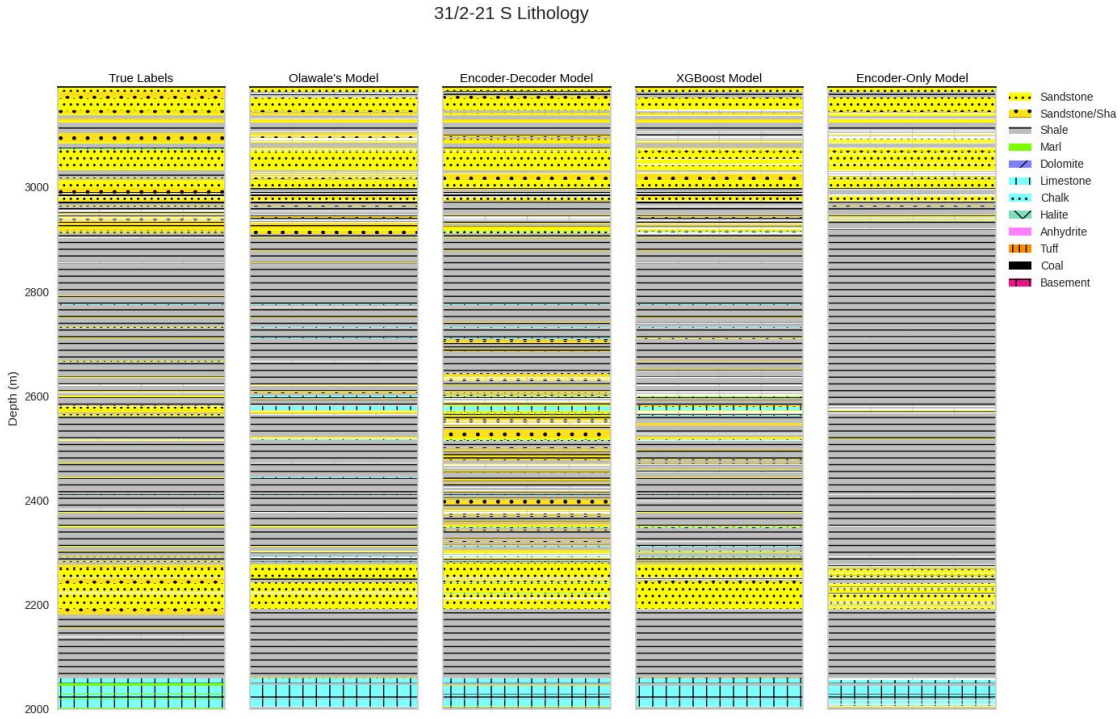
Fonte: O autor (2022).

Figura 46 – Litofácies ao longo da profundidade do poco 31/_2-10



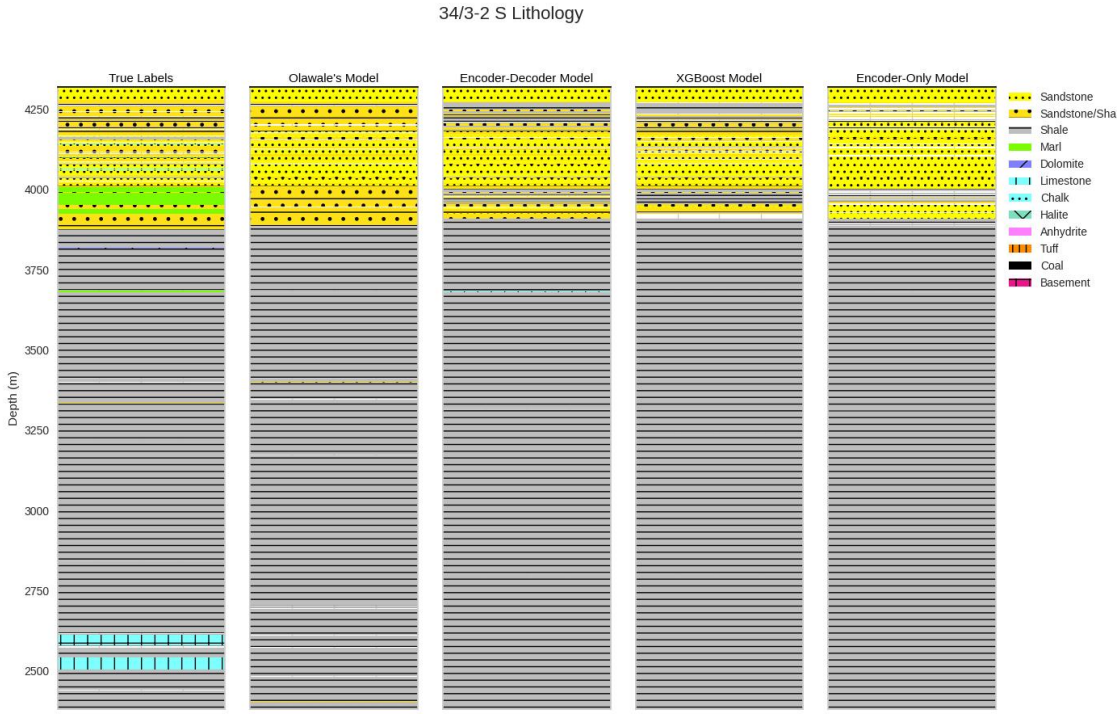
Fonte: O autor (2022).

Figura 47 – Litofácies ao longo da profundidade do poco 34/_3-2 S



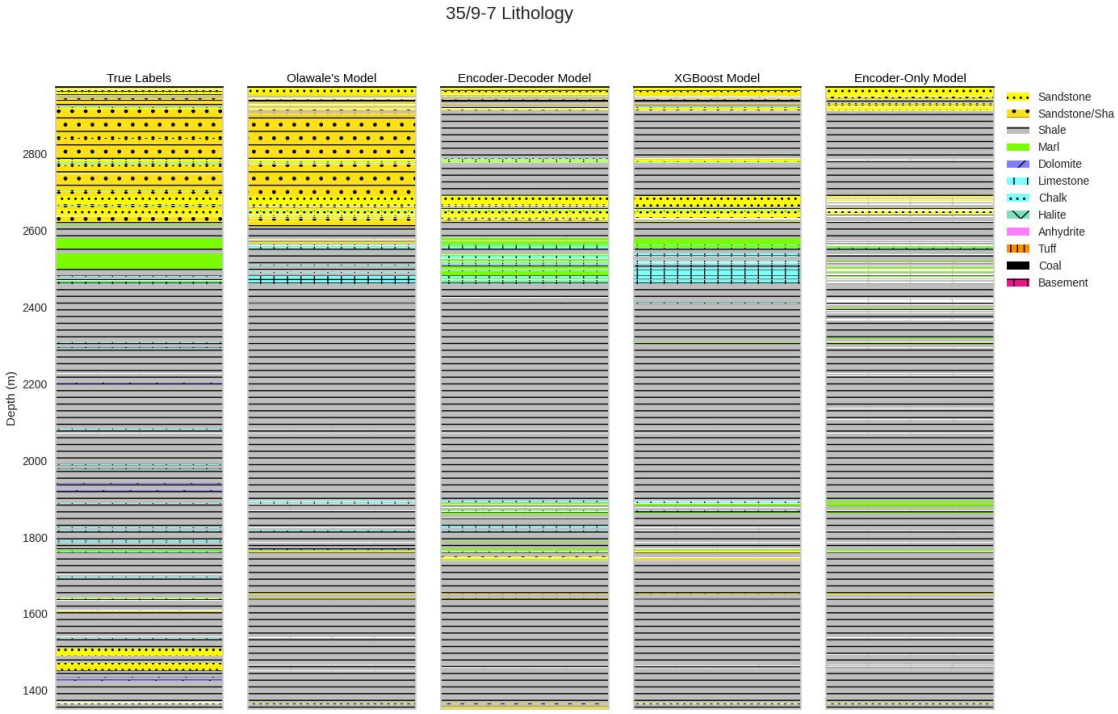
Fonte: O autor (2022).

Figura 48 – Litofácies ao longo da profundidade do poco 34/_3-2 S



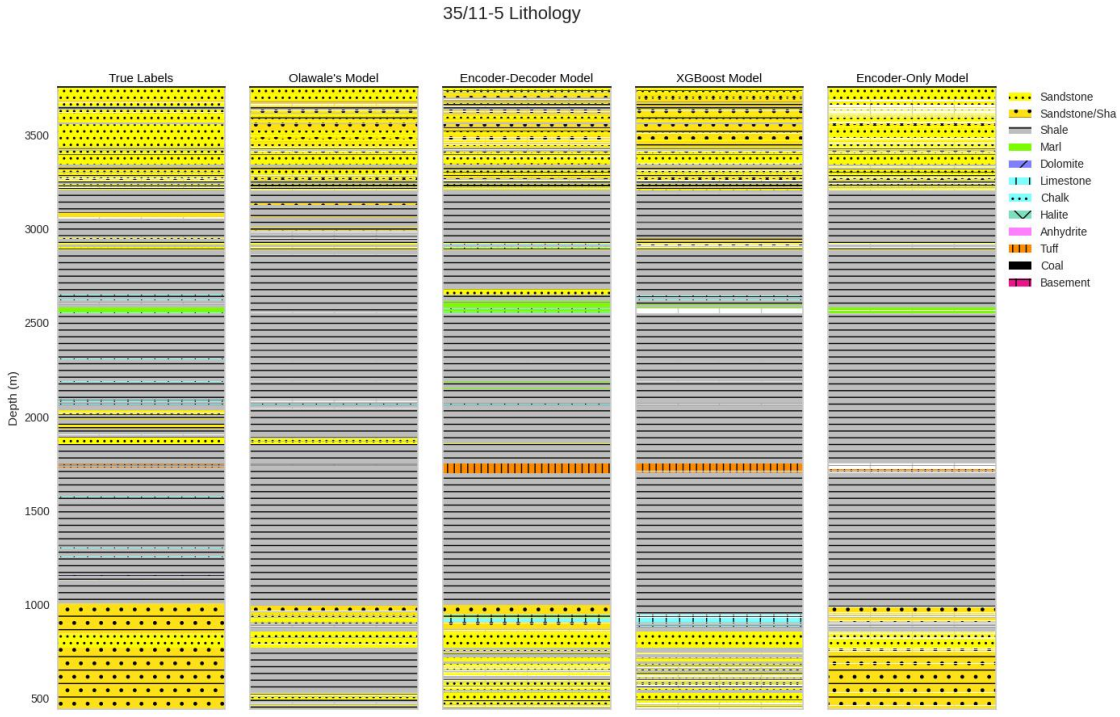
Fonte: O autor (2022).

Figura 49 – Litofácies ao longo da profundidade do poco 35/_9-7



Fonte: O autor (2022).

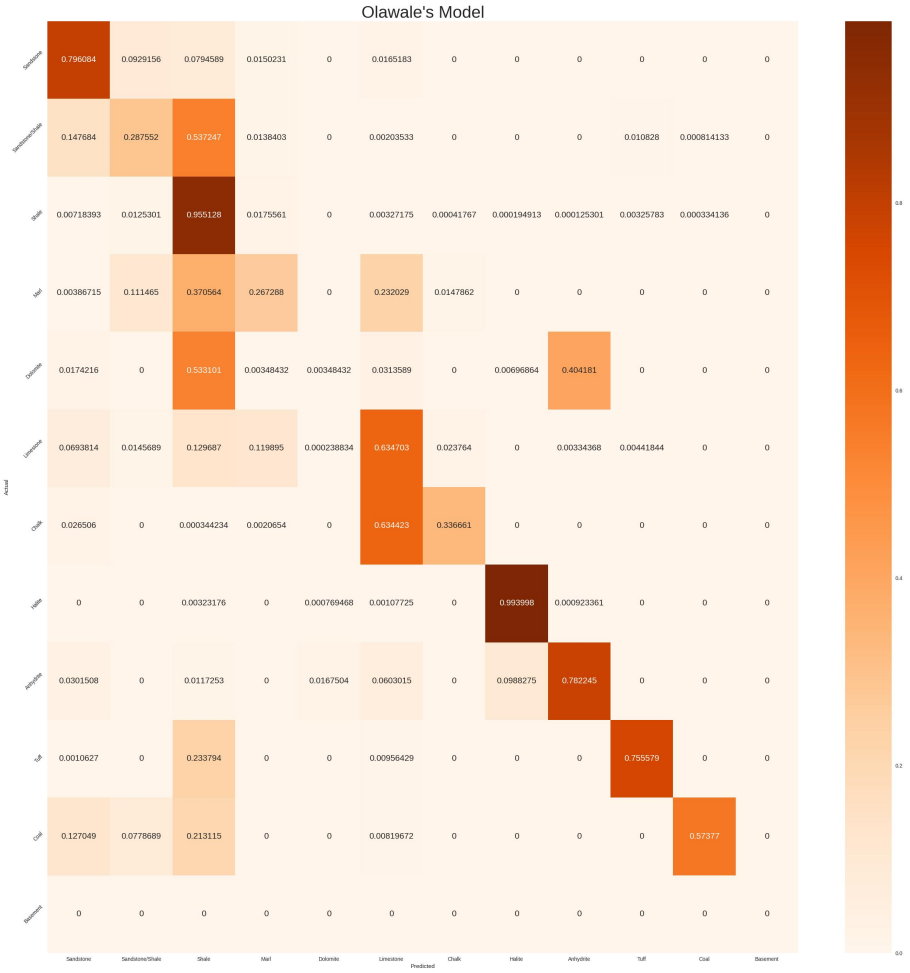
Figura 50 – Litofácies ao longo da profundidade do poco 35/_11-5



Fonte: O autor (2022).

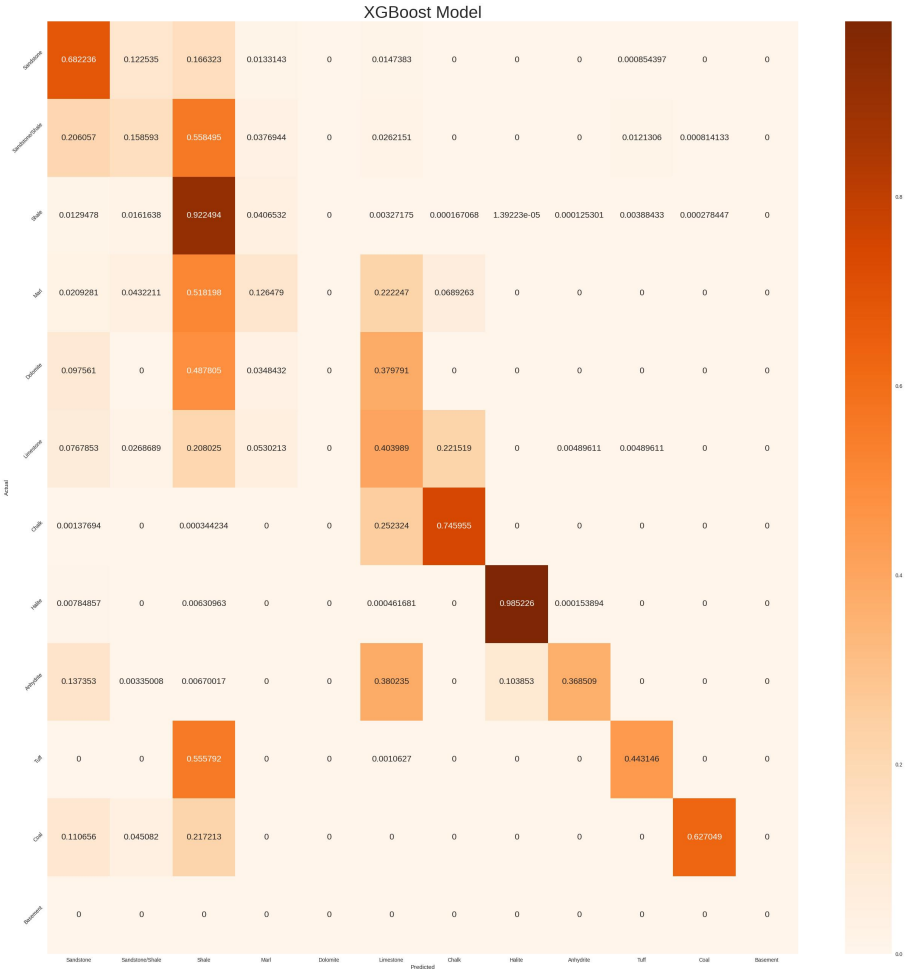
5.4 Matrizes de Confusão

Figura 51 – Matriz de Confusão para o modelo de Olawale



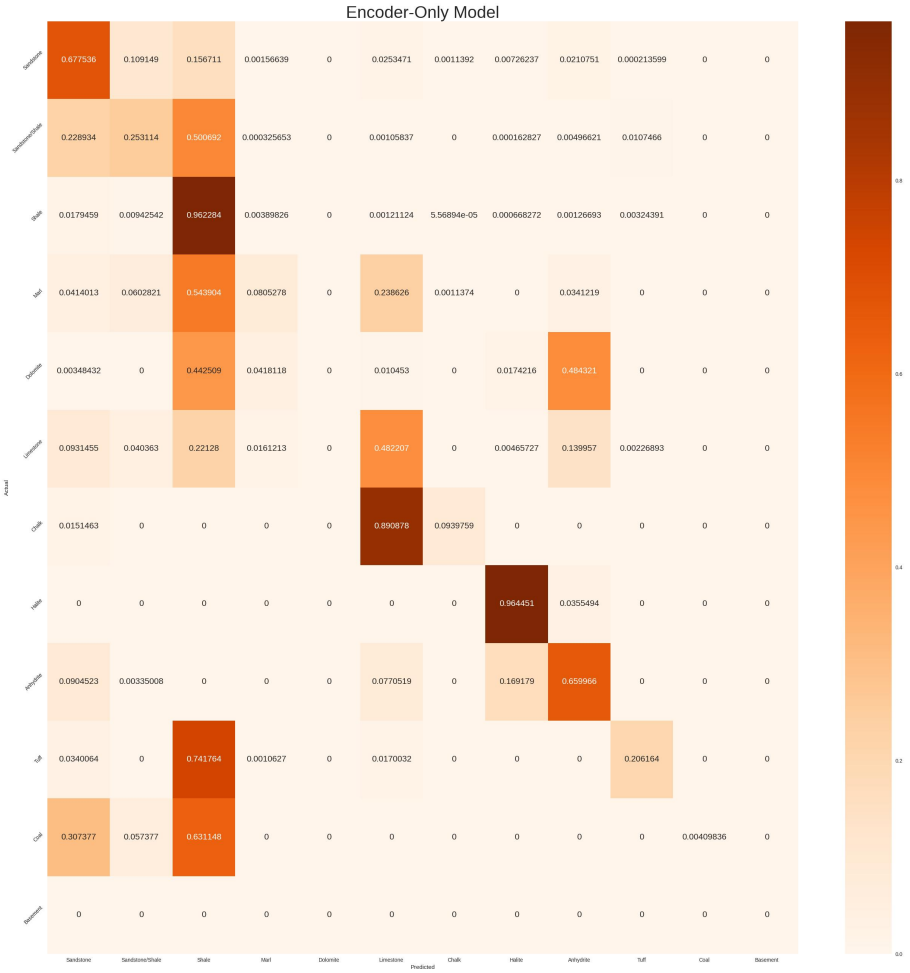
Fonte: O autor (2022).

Figura 52 – Matriz de Confusão para o XGBoost baseline



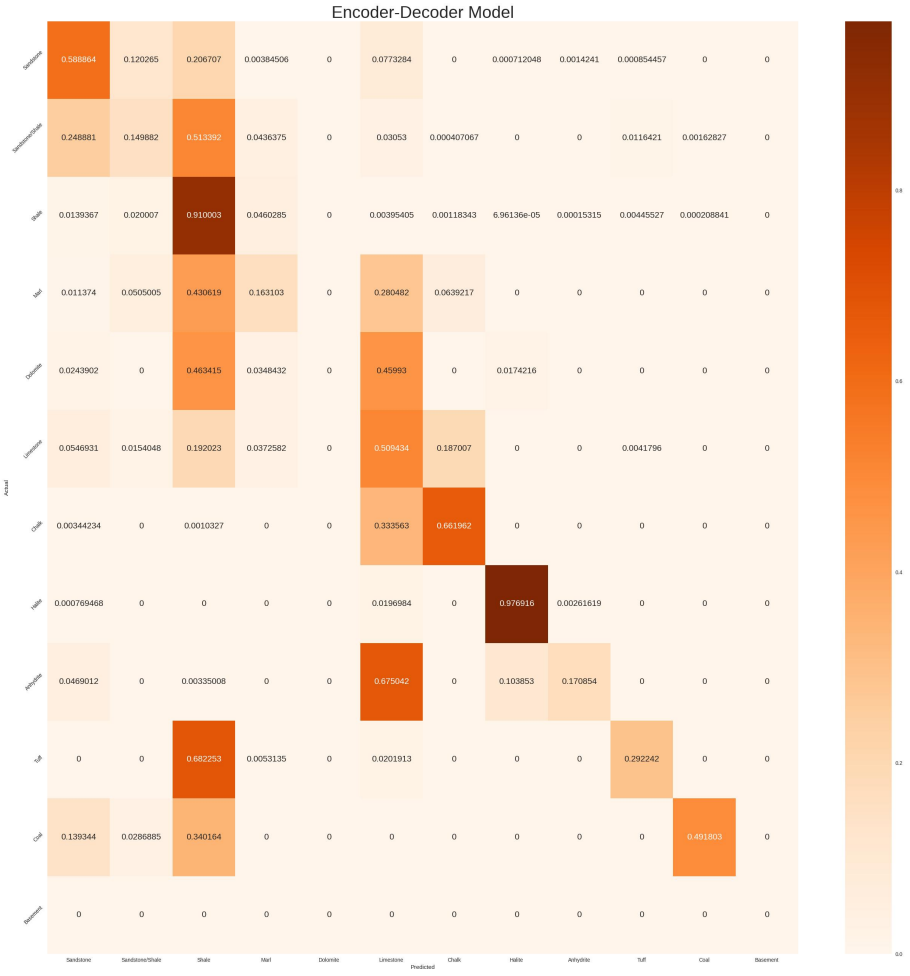
Fonte: O autor (2022).

Figura 53 – Matriz de Confusão para o modelo de Encoder-only



Fonte: O autor (2022).

Figura 54 – Matriz de Confusão para o modelo Encoder-Decoder



Fonte: O autor (2022).

5.5 Discussão

5.5.1 Geral

De maneira geral, o modelo proposto pelo ganhador da competição, Olawale, obteve os melhores resultados. O modelo *Encoder-only* obteve o segundo melhor desempenho para os resultados analisados. O modelo XGBoost obteve o terceiro melhor desempenho. Por fim, o modelo *Encoder-decoder* obteve o pior desempenho.

5.5.2 Modelo de Olawale vs Encoder-only

Nas métricas gerais, incluindo todos os poços, o modelo de Olawale se saiu melhor. Porém, ao comparar os acertos para cada facie, o modelo *Encoder-only* conseguiu acertos superiores para Shale, Marl, Chalk, Coal.

5.5.3 Encoder-only vs Encoder-Decoder

O modelo *Encoder-only* obteve melhor resultado em todos os experimentos. Um dos motivos pode estar ligado ao fato do *Encoder-Decoder* não usar o mecanismo de *gating* da mesma maneira que o modelo *Encoder-only* e o resultado da camada para alimentar o *decoder* do modelo.

5.5.4 Encoder-only vs XGBoost

O modelo *Encoder-only* obteve resultados melhores do que o modelo XGBoost com os parâmetros propostos por Olawale sem o treino rigoroso, indicando que este modelo possa ter potencial futuro.

6 Conclusão

O objetivo do trabalho foi elaborar uma abordagem para a classificação de litofácies utilizando *Wireline Logs* que fizesse uso da arquitetura de Transformadores. Esse objetivo se encaixa com a hipótese de que profundidades próximas estão altamente correlacionadas. Ao conhecimento do autor, este é o primeiro trabalho a utilizar essa arquitetura para a classificação de facies utilizando *Wireline Logs* e, por isso, este projeto também visa servir como um ponto de partida para futuros trabalhos na área.

Na primeira etapa do trabalho, foi realizado um estudo da literatura em busca de modelos de transformadores que poderiam ser adaptados para a classificação de litofácies. Dois tipos de trabalhos se mostraram adequados para o problema: Transformadores Tabulares e Transformadores para séries temporais. Além disso, os trabalhos se diferenciavam quanto a aspectos como: *embedding*, uso de pré-treino, uso de *encoder* e *decoder* ou apenas *encoder*. Neste trabalho, optou-se por explorar a literatura de séries temporais.

A partir dos trabalhos relacionados, foram propostas duas soluções: uma arquitetura que utiliza um *encoder* idêntico ao proposto por (Liu et al., 2021), e uma outra que, além disso, conta com a adição de um bloco *decoder* para capturar a *self-attention* das predições de entradas anteriores. Ambos os modelos foram desenvolvidos em Pytorch, com classes implementadas pela biblioteca *Transformer* do Huggingface. Este trabalho também buscou comparar os resultados com um modelo XGBoost baseado na solução do ganhador da competição XEEK.

Para facilitar a comparação entre os resultados das diferentes soluções, o pré-processamento dos dados e a seleção de atributos foi feita de maneira similar, com a principal diferença sendo a dimensão dos dados de entrada, que varia pelo tipo do modelo utilizado: (label, label), (sequência, label) e (sequência, sequência). Em seguida, foi aplicado um treino supervisionado utilizando os modelos escolhidos.

Os modelos treinados foram, então, utilizados para classificar as fácies dos poços de teste. A partir dessas facies previstas, os resultados do treino mostraram que os modelos transformadores conseguem atingir resultados minimamente razoáveis e podem servir como ferramentas alternativas aos demais métodos já presentes na literatura. Entretanto, para atingir resultados verdadeiramente competitivos, ainda é necessário um estudo mais aprofundado de *Wireline Logs* e litofácies que se encaixam no problema.

6.1 Trabalhos Futuros

Tendo em vista possíveis trabalhos futuros para dar continuidade ao estudo iniciado por este projeto, a seguir serão apresentados alguns temas interessantes a serem levanta-

dos.

Tratamento dos dados categóricos. Na literatura, diversos trabalhos, como o *Transformer* e o *FNTTaa Former* trazem abordagens de *embedding* para os dados de categóricos. No caso do *Transformer*, além do *embedding*, os dados categóricos passam por um bloco de autoatenção, chamado pelos autores de "bloco de contexto". É possível que técnicas como essa possam ser incorporadas pelo bloco *encoder* para o caso de colunas como 'GRUPO' e 'FORMAÇÃO'.

Pré-treino não-supervisionado. O pré-treino do não-supervisionado utilizando apenas os dados de entrada antes do treino supervisionado é uma técnica bastante utilizada para melhorar a performance de modelos transformadores. Uma das abordagens seria mascarar algumas amostras já conhecidas das sequências de entrada, e treinar utilizando os outros dados presentes na sequência. Uma vez pré-treinados, os pesos do *encoder* podem servir como ponto de partida para um treino não-supervisionado. Tanto (Liu et al., 2021) e (Zerveas et al., 2020) *transformer* mostraram resultados significativos com a técnica. Esse pré treino não-supervisionado também poderia ser feito sobre outros *datasets*, abordagem bastante comum na literatura. Esse pré-treino pode ser realizado também sobre outras bases de dados, possibilitando um enriquecimento dos padrões capturados.

Imputação dos dados. A imputação dos dados foi feita de maneira ingênua nesse trabalho. A imputação poderia ser feita mascarando-se os dados faltantes na sequência de entrada, e usando os outros elementos, juntamente com as *self-attention* do *encoder* para estimá-los.

Balanceamento de classes. Os dados desse *dataset* e de outros de litologia são bastante desbalanceados, 70% dos observadores continham a classes 'SHALE' nesse set de treino. Um cuidado maior com isso no futuro poderia melhorar a performance.

Diferentes *encoders* para diferentes partes da arquitetura. Assim como com (Liu et al., 2021), nos quais os autores utilizam de uma torre para diferentes atributos da entrada, seria válido a tentativa de desenvolver "torres" que capturem o *self-attention* de maneira independente para diferentes combinações de *wireline logs*.

Referências Bibliográficas

Alammar, 2022 Alammar, J. (2022). Visualizing machine learning one concept at a time. Disponível em: <https://jalammar.github.io>. Acesso: Outubro, 2022. 22, 23, 24, 25, 26, 27, 28, 30, 31

Ba et al., 2016 Ba, J., Kiros, J., and Hinton, G. (2016). Layer normalization. 29

Bestagini et al., 2017 Bestagini, P., Lipari, V., and Tubaro, S. (2017). A machine learning approach to facies classification using well logs. pages 2137–2142. 14, 16, 55

Boggs, 2001 Boggs, S. (2001). Principles of sedimentology and stratigraphy. 14, 15

Bormann et al., 2020a Bormann, P., Aursand, P., and Dilib, F. (2020a). FORCE Machine Learning Competition. 16, 17, 46, 56, 57, 60, 61, 62

Bormann et al., 2020b Bormann, P., Aursand, P., Dilib, F., Manral, S., and Dischington, P. (2020b). FORCE 2020 Well well log and lithofacies dataset for machine learning competition. 55

BRITTANICA, 2018 BRITTANICA (2018). sedimentary facies. 14, 20

Chen and Guestrin, 2016 Chen, T. and Guestrin, C. (2016). XGBoost: A scalable tree boosting system. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, pages 785–794, New York, NY, USA. ACM. 14, 36

Cho et al., 2014 Cho, K., van Merrienboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., and Bengio, Y. (2014). Learning phrase representations using rnn encoder-decoder for statistical machine translation. 35

Diniz et al., 2022 Diniz, P., Junior, D. A. D., Diniz, J. a. O. B., de Paiva, A. C., Silva, A. C. d., Gattass, M., Quevedo, R., Michelon, D., Siedschlag, C., and Ribeiro, R. (2022). Time2vec transformer: A time series approach for gas detection in seismic data. page 66–72. 44, 45

Hall and Hall, 2017 Hall, M. and Hall, B. (2017). Distributed collaborative prediction: Results of the machine learning contest. *The Leading Edge*, 36(3):267–269. 14

Hubbard and Linde, 2011 Hubbard, S. and Linde, N. (2011). 2.15 - hydrogeophysics. pages 401–434. 21

IBM, 2022 IBM (2022). What is random forest? Disponível em: <https://www.ibm.com/topics/random-forest>. Acesso: Agosto, 2022. 35, 36

Jaikla et al., 2019 Jaikla, C., Devarakota, P., Auchter, N., Sidahmed, M., and Espejo, I. (2019). Faciesnet: Machine learning applications for facies classification in well logs. 16

Kazemi et al., 2019 Kazemi, S. M., Goel, R., Eghbali, S., Ramanan, J., Sahota, J., Thakur, S., Wu, S., Smyth, C., Poupart, P., and Brubaker, M. (2019). Time2vec: Learning a vector representation of time. 43, 44, 45

Lin et al., 2021 Lin, T., Wang, Y., Liu, X., and Qiu, X. (2021). A survey of transformers. *CoRR*, abs/2106.04554. 16, 32

Liu and Liu, 2022 Liu, J.-J. and Liu, J.-C. (2022). Integrating deep learning and logging data analytics for lithofacies classification and 3d modeling of tight sandstone reservoirs. *Geoscience Frontiers*, 13(1):101311. 16

Liu et al., 2021 Liu, M., Ren, S., Ma, S., Jiao, J., Chen, Y., Wang, Z., and Song, W. (2021). *Gated Transformer Networks for Multivariate Time Series Classification*. 37, 38, 39, 40, 41, 45, 47, 52, 80, 81

Maxwell, 2022 Maxwell, J. (2022). Why gpt wants to mesa-optimize how we might change this. Disponível em: <https://www.alignmentforum.org/posts/BGD5J2KAoNmpPMzMQ/why-gpt-wants-to-mesa-optimize-and-how-we-might-change-this>. Acesso: Setembro, 2022. 32, 33

Program, 2022 Program, O. D. (2022). Proceeding of the ocean drilling program. Disponível em: <http://www.odp.tamu.edu/publications/195_IR/chap_02/c2_f4.htm>. Acesso: Agosto, 2022. 20, 22

Sherstinsky, 2020 Sherstinsky, A. (2020). Fundamentals of recurrent neural network (rnn) and long short-term memory (lstm) network. *Physica D: Nonlinear Phenomena*, 404:132306. 35

Theisges et al., 2021 Theisges, D., Roisenberg, M., and Nascimento, M. D. S. (2021). Deep recurrent neural networks approach to sedimentary facies classification using well logs. *IEEE Geoscience and Remote Sensing Letters*, PP:1–5. 16

Tripathi, 2022 Tripathi, R. (2022). What are vector embeddings? Disponível em: <https://www.pinecone.io/learn/vector-embeddings/>. Acesso: Setembro, 2022. 27, 48, 49, 50, 52

Vaswani et al., 2017 Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. (2017). *Attention Is All You Need*. 16, 22, 24, 27, 28, 29, 31, 32, 33, 34, 38, 41, 48, 52, 54

Zerveas et al., 2020 Zerveas, G., Jayaraman, S., Patel, D., Bhamidipaty, A., and Eickhoff, C. (2020). A transformer-based framework for multivariate time series representation learning. 40, 41, 42, 81

Zhang and Zhan, 2017 Zhang, L. and Zhan, C. (2017). Machine learning in rock facies classification: An application of xgboost. pages 1371–1374. 14

A Código

O código fonte de todas as implementações feitas para o desenvolvimento deste trabalho pode ser acessado no repositório: <https://github.com/JoaoJanini/seq2seq>, para que possíveis continuações e modificações de componentes possam ser feitas no futuro.

A Artigo

Classificação de Litofácies Utilizando Transformers

João Janini Ribeiro[1]

[1]Universidade Federal de Santa Catarina (UFSC)

[2]Departamento de Informática e Estatística

[3]Campus Universitário – Florianópolis, SC - Brasil

jjaninir@gmail.com

Abstract. *The lithofacies classification problem in wells involves identifying the characteristics of the rocks contained in the drilling wells. This classification can be used to identify resources such as gas and oil, which is why it is an activity of significant economic value. One of the most common ways to do this classification is through the use of Wireline Logs. These measurements made by probes contain several attributes of the rocks that help in their identification. From these collected attributes, it is possible to use them as inputs for machine learning algorithms, which result in predictive models that can be used to assist geophysicists in their analyses. The types of algorithms used in the area vary greatly, with methods involving decision trees being the most used, both in articles and in contests. These methods, especially XGBoost, are very effective and achieve significant levels of accuracy when used. However, these methods are known to fail to capture an essential dimension of well data, the sequential relationship between samples.Following this idea, this work explored the use of Transformers for the classification of wireline profile lithofacies. Transformers are a more recent architecture, and have been conquering the state of the art in several fields involving sequential data. Therefore, this work proposed two implementations of transformers that would serve for the lithofacies classification problem, one "sequence-by-sequence"and another "sequence-by-one". Both implementations were tested with data from the XEEK 2020 competition, and had their metrics compared with a base model involving XGBoost, as well as with the results obtained by the winner of the competition. The results obtained prove the effectiveness of the architecture in solving the problem, and the implementations presented can serve as a basis for future work.*

Resumo. *O problema de classificação de litofácies em poços envolve a identificação das características das rochas contidas em poços de perfuração. Essa classificação pode ser utilizada na identificação de recursos como gás e petróleo, e por isso é uma atividade de valor econômico significativo. Uma das maneiras mais comuns de se fazer essa classificação é através do uso de Wireline Logs. Essas medidas feitas por sondas contém diversos atributos das rochas que ajudam em sua identificação. A partir desses atributos coletados, é possível usá-los como entradas de algoritmos de aprendizado de máquina, que resultam em modelos preditivos que podem ser utilizados para auxiliar geofísicos em suas análises. Os tipos de algoritmos utilizados na área variam bastante, com os métodos envolvendo árvores de decisão sendo os mais comu-*

mente utilizados, tanto em artigos quanto em competições. Esses métodos, especialmente o XGBoost, são bastante eficazes e conseguem níveis de acurácia significativos quando utilizados. No entanto, se sabe que esses métodos não capturam uma dimensão essencial dos dados dos poços: a relação sequencial entre as amostras. Seguindo essa ideia, este trabalho explora o uso de Transformers para a classificação de litofácies a partir de wirelines logs. Os Transformers são uma arquitetura mais recente, e vêm conquistando o estado da arte em diferentes campos envolvendo dados sequenciais. Portanto, este trabalho propôs duas implementações de transformers que servissem para o problema de classificação de litofácies: uma de "sequência-para-sequência"e a outra de "sequência-para-um". Ambas as implementações foram testadas utilizando os dados da competição XEEK de 2020 e tiveram suas métricas comparadas tanto com um modelo base envolvendo XGBoost, quanto com os resultados obtidos pelo ganhador da competição. Os resultados obtidos comprovam a efetividade da arquitetura na resolução do problema, e as implementações apresentadas poderão servir como base para trabalhos futuros.

1. Introdução

Litofácies são subdivisões de sequências sedimentares que possuem uma relação direta com os processos sedimentares que as formaram. A classificação correta dessas subdivisões é de elevado valor econômico. Uma das práticas que auxiliam nessa classificação é o *well logging*. A prática do *well logging* envolve a obtenção de descrições detalhadas sobre as formações de rocha em diferentes níveis e profundidades de um poço, com o objetivo principal de classificação das litofácies em cada profundidade. *Well logging* é essencial para a indústria de hidrocarbonetos, que usa a classificação de litofácies durante todo o processo de extração de recursos, como gás natural e petróleo [BRITTANICA 2018].

O problema de classificação de litofácies pode ser formulado como a detecção de uma litofácie em uma determinada profundidade, a partir da análise de conjuntos de *well logs* obtidos naquela profundidade [Bestagini et al. 2017]. Essa formulação permite o uso de métodos de aprendizado de máquina, que têm cada vez mais se popularizado na área.

Os métodos utilizados no problema são bem diversos, incluindo *deep learning*, SVG (Support Vector Machines), regressão logística e, principalmente, métodos baseados em árvores de decisão, como o XGBoost [Zhang and Zhan 2017] e *random forest*. Os resultados de competições em torno do problema ilustram bem a popularidade de métodos como XGBoost na área. Na competição organizada pela SEG *(Society of Exploration Geophisycist)* de 2016 [Hall and Hall 2017], todos os 5 melhores classificados utilizaram **Boosted Trees**. Em particular, foi utilizado o *XGBoost* [Chen and Guestrin 2016], que é uma implementação bastante popular de *Boosted Trees* que usa de métodos de descida de gradiente para encontrar as melhores árvores de decisão para o problema.

Entretanto, ainda que essas abordagens consigam atingir níveis de acurácia significativos, o problema continua sendo bastante complexo por vários motivos, como por exemplo: as diferenças de resolução entre poços e a sobreposição dos valores de fácies sobre si mesmas. Além disso, ainda existe a questão de tipos de rocha com funções diferentes dependendo do ambiente de exploração ou produção. Dentre as possíveis explicações para esses problemas, podemos citar a falta de informação geológica e a ausência de

sequências de fácies. Essas técnicas ignoram que existe uma correlação entre camadas vizinhas e que padrões de fácies empilhados são significativos para interpretações geológicas [Jaikla et al. 2019].

Para essa questão, [Bestagini et al. 2017] propõem o uso de gradientes de algumas das curvas como atributos adicionais ao modelo. Os melhores colocados das competições XEEK e SEG citadas utilizaram esses atributos adicionais como entrada dos modelos, indicando de fato vantagens claras na abordagem temporal do problema.

A fim de trazer essa abordagem para os algoritmos de aprendizado em si, alguns trabalhos utilizaram CNNs (*Convolutional Neural Network*) para capturar as características espaciais entre as amostras, atingindo resultados satisfatórios. [Theisges et al. 2021] trouxeram resultados positivos utilizando BiLSTM (*Bidirectional Long Short-Term Memory*).

Algoritmos como CNN, LSTM e também GRU (*Gated Recurrent Units*) compõem o grupo de algoritmos de *deep learning* que incorporam a relação temporal ou espacial entre os dados de entrada de maneira nativa. Além deles, atualmente os *transformers* [Vaswani et al. 2017] vêm ganhando notoriedade por atingirem resultados considerados estados da arte em problemas envolvendo dados sequenciais, principalmente no ramo de NLP (*Natural Language Processing*). [Lin et al. 2021] Diferente das RNNs (*Recurrent Neural Network*), que usam recursão para capturar relações globais entre *input* e *output*, transformadores usam mecanismos de *attention*, em especial, o *self-attention*.

Essa capacidade de capturar sequências longas contribuiu para a formação de uma literatura extensa sobre o uso de transformadores nos problemas de classificação e predição de séries temporais. Seguindo essa linha, a fim de contribuir para a literatura de classificação de litofácies utilizando métodos sequenciais, este trabalho busca adaptar diferentes arquiteturas de transformadores ao problema de predição de fácies litológicas.

Na sequência, foi testada a eficiência da abordagem escolhida utilizando os dados da competição XEEK [Bormann et al. 2020a]. Os resultados são comparados contra os resultados obtidos pelos primeiros colocados a partir da métrica proposta pela competição. No conhecimento do autor até o momento da escrita, este é o primeiro trabalho que estuda a eficácia dos transformadores no problema de classificação de litofácies.

2. Fundamentação Teórica

Esse capítulo traz conceitos relevantes para o embasamento teórico que leva ao entendimento deste trabalho. A seguir, serão apresentados alguns pontos que envolvem litofácies e transformadores

2.1. Litofácies

Rochas sedimentares são formadas onde sedimentos são depositados durante tempo suficiente para ficarem compactadas e sedimentadas na forma estrata [BRITTANICA 2018].

Normalmente, isso se dá quando os sedimentos permanecem intocados por vários anos nas bacias sedimentares. Essas bacias podem ocupar um pequeno espaço ou até milhares de quilômetros e possuir diferentes ambientes deposicionais locais. A justaposição dessas diferentes rochas contemporâneas sedimentares define as litofácies [BRITTANICA 2018].

2.1.1. Wireline Logging

Wireline Logging refere-se ao processo de registro e análise de medições coletadas discretamente ou continuamente dentro de furos de poços. As medições de furos são feitas baixando uma sonda no furo na extremidade de um cabo elétrico. Entre a variedade de sondas de furo de poço disponíveis, pode-se citar: SP, elétrico, EM, gama-gama, gama natural, acústico, temperatura, medidor de vazão, nêutron-nêutron, televisores e registros de calibre [Hubbard and Linde 2011].

É um método de medição indireto e traz diversas possibilidades de medição comparadas aos outros, além de benefícios econômicos. Seus instrumentos conseguem medir diferentes propriedades, que formam a base da análise petrofísica e geologia.

2.2. *Transformer*

O *transformer* é uma arquitetura de modelo de rede neural utilizada para sequências, que ao invés de usar recorrência, usa apenas mecanismos de *attention* para capturar dependências globais entre *input* e *output*. A arquitetura do *transformer* foi originalmente proposta por [Vaswani et al. 2017] no contexto de tradução de sequências de *tokens*. *Tokens* aqui se refere a qualquer elemento individual de uma sequência, como por exemplo palavras em uma frase.

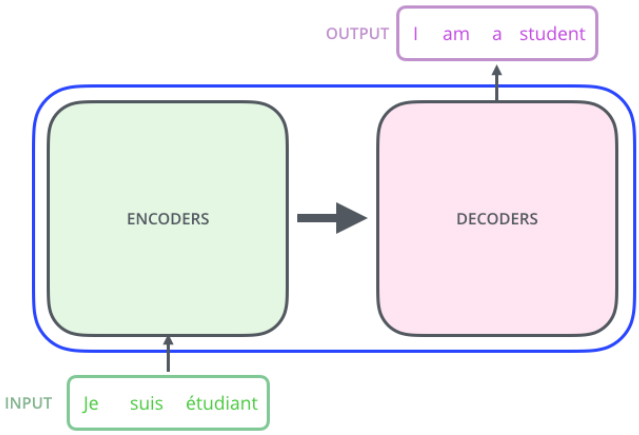


Figura 1. Visão simplificada da arquitetura *Transformers* [Alammar 2022].

No contexto da tradução, o modelo recebe duas entradas durante a predição: uma sequência de *tokens* a ser traduzida e uma sequência contendo os *tokens* já traduzidos até a posição atual. Como saída, o modelo fornece uma estimativa do *token* para a posição atual.

A Figura 1 mostra de maneira simplificada as duas partes principais da arquitetura: o *encoder* e o *decoder*. No exemplo citado, o *encoder* recebe a sequência de *tokens* a ser traduzida, e o *decoder*, a sequência de *tokens* que já foi traduzida até então.

De maneira formal, pode-se afirmar que o *encoder* é utilizado para mapear uma sequência de entrada de símbolos x = (x1, ..., xn) à sequência contínua representada por z = (z1, ..., zn). A partir de z, o *decoder* gera uma sequência de saída y = (y1, ..., ym), um

elemento por vez. O modelo é auto-regressivo em cada passo, consumindo os símbolos gerados anteriormente como entradas adicionais quando gera o próximo.

3. Trabalhos Relacionados

Neste capítulo, serão explicadas as abordagens de 2 trabalhos que usam transformadores para classificação de séries temporais, um trabalho que traz uma revisão de abordagens de aprendizado de máquina para séries temporais é um trabalho que traz um exemplo de um problema de sequência para séries temporais multivariáveis.

3.1. Gated-Transformer-on-MTS

O trabalho proposto por [Liu et al. 2021] propõem uma abordagem de classificação de séries temporais baseada em transformadores, com algumas modificações quando comparada com a original. De acordo com os autores, uma série temporal com múltiplos atributos pode ser decomposta em ”canais”, no qual cada canal é uma série univariável. Nesse caso, se espera que exista uma correlação entre diferentes canais em passos diferentes.

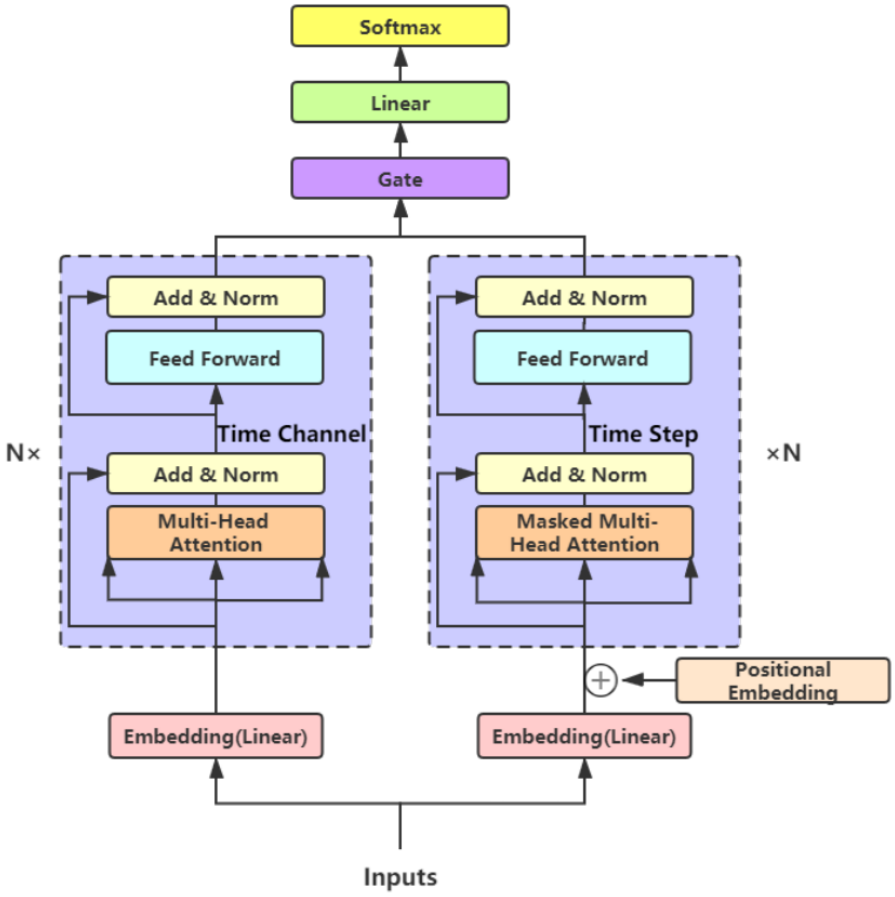


Figura 2. Arquitetura do GTM [Liu et al. 2021].

A Figura 2 mostra a arquitetura proposto pelos autores com os dois blocos *encoder* e a última camada de predição.

Mais especificamente, os *encoders* são descritos da seguinte forma:

- *Step-wise Encoder*. Calcula a *self-attention* entre os elementos de um mesmo canal. É o bloco esquerdo na Figura 2.
- *Channel-wise Encoder*. Calcula a *self-attention* entre os elementos de canais diferentes. É o bloco direito na Figura 2. Como o objetivo é capturar a relação temporal entre os passos, é adicionado ainda um *positional encoding*. A codificação posicional é feita seguindo a linha de [Vaswani et al. 2017] sobre os dados projetados não-linearmente.

Antes de passar por esses blocos, a sequencias passam por um *fully connected layer* com uma função de ativação não-linear. Essa camada serve função parecida ao do *Embedding Layer* de [Vaswani et al. 2017], e é utilizada por se tratarem de dados de natureza contínua.

$$
\begin{aligned}
h &= \mathbf{W} \cdot Concat(C, S) + b \\
g_1, g_2 &= Softmax(h) \\
y &= Concat(C \cdot g_1, S \cdot g_2)
\end{aligned}
$$

Figura 3. Mecanismo de *gating* que concatena o resultado das duas torres [Liu et al. 2021].

Outra parte importante da arquitetura é o bloco *gate* mostrado na Figura 2. Esse bloco é explicado pela Figura 3. As saídas dos blocos *encoders Channel-wise Encoder* (*C*) e *Step-wise Encoder* (*S*) são concatenadas e passadas por uma camada linear. A saída *h* dessa camada é então passada por uma *Softmax* que ira atribuir pesos *g1* e *g2* para cada torre.

Finalmente, os pesos de cada torre são multiplicados por suas respectivas saídas e concatenados entre si. A ideia é que os pesos sejam aprendidos durante o treino, e capturem o quanto cada torre contribui para uma determinada predição. De acordo com os autores, essa abordagem superou os resultados tanto de uma concatenação simples, quanto do uso de cada torre individual.

A saída do *gate* é passada mais uma vez por uma camada linear e um *Softmax*, e retorna as probabilidades das classes previstas.

3.2. Gated-Transformer-on-MTS

4. Desenvolvimento

Nessa seção é explicado o processo de desenvolvimento planejado para esse trabalho. Cada modelo implementado é mostrado de uma maneira mais aprofundada, incluindo o modelo base. Além dos modelos implementados, também é discutido o modelo produzido pelo ganhador da [Bormann et al. 2020a], Olawale. Também são apresentadas os dados utilizados, as técnicas de de pré-processamento aplicadas, e informações sobre o treino.

4.1. Modelos

Foram propostas duas soluções utilizando transformadores. A primeira solução é idêntica à solução proposta por [Liu et al. 2021], que possui dois *encoders*. A segunda solução foi utilizar os mesmos *decoders* da primeira, porém traz também um *decoder* que recebe as facies previstas em passos anteriores. Por fim, também foi treinado um modelo XGBoost para ser utilizado como base. Esse modelo utiliza os mesmos parâmetros utilizados pelo ganhador da competição. Nesta seção, serão apresentadas as implementações desses modelos em mais detalhes.

Hiperparâmetro	Valor	Descrição
encoder_layers	6	Número camadas encoder
encoder_ffn_dim	512	Dimensão da camada feedforward do encoder
encoder_attention_heads	8	Número de attention heads no encoder
decoder_layers	6	Número de camadas decoder no bloco decoder
decoder_ffn_dim	512	Dimensão da camada feedfoward do bloco decoder
decoder_attention_heads	8	Número de attention heads no decoder
encoder_layerdrop	0.1	Taxa de dropout no bloco encoder
decoder_layerdrop	0.1	Taxa de dropout no bloco decoder
activation_function	relu	Função de ativação usada
d_model	512	Dimensão dos embeddings
n _ features	8	número de atributos
n_output	12	número de classes no atributo alvo
sequence_ len	10	comprimento das sequências
dropout	0.1	Taxa de dropout
attention_dropout	0.1	Dropout na cross attention
activation_dropout	0.1	Dropout na camada de ativação
Dropout no classificador	0.1	Dropout na camada de classificação

Tabela 1. Valores do hiperparâmetros para os modelos *transformers*.

4.1.1. Transformador Encoder-Only

Essa arquitetura segue a implementação proposta no trabalho de [Liu et al. 2021]. A ideia principal é conseguir capturar os pesos entre atributos diferentes em uma mesma profundidade e os pesos entre passos de tempo diferentes para um mesmo atributo. Para tal, esse modelo é composto por um *encoder* temporal e um *encoder* de atributos.

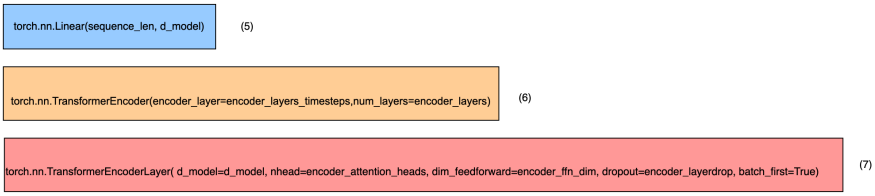


Figura 4. Blocos de transformers para *step-wise-encoder* e *channel-wise-encoder* do pacote torch [Tripathi 2022].

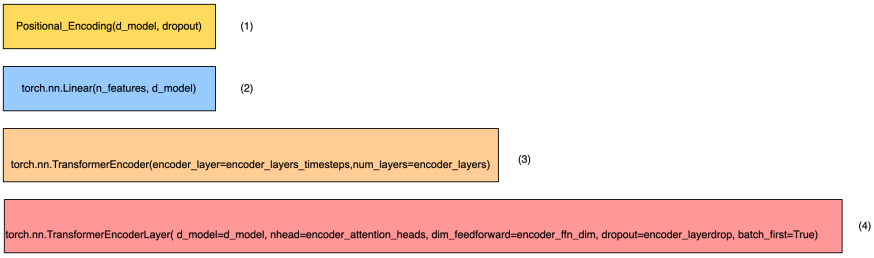


Figura 5. Blocos de *transformers* para *channel-wise-encoder* com camadas do pacote *torch* [Tripathi 2022].

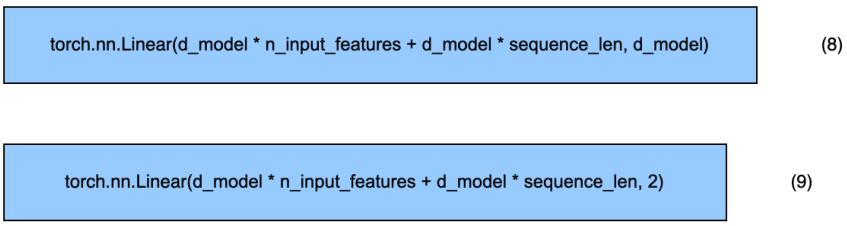


Figura 6. Camadas Pytorch para o *Gate* e *Linear* [Tripathi 2022].

Figura 7. Camadas Pytorch para o *Gate* e *Linear* [Tripathi 2022]

- LinearLayers: São utilizadas duas camadas lineares para os dados de entrada, uma para *step-wise-encoder* e a outra para o *channel-wise*. Nota-se que é utilizada uma camada linear por se tratarem de dados contínuos, diferente dos *Embeddings* utilizados por [Vaswani et al. 2017] na implementação original.
- PositionalEncoding: Também é criado um *Positional Encoding* ((1) da Figura 4 para ser adicionado ao *input* do *step-wise-encoder*, como forma de codificar a posição de cada atributo na sequência durante o cálculo da atenção. O mesmo não é feito para *channel-wise-encoder* uma vez que essa relação não existe entre atributos de uma mesma profundidade. O camada PoisitionalEncoding é implementada utilizando o módulo nn.Module do torch
- Encoders: Para umas das torres *encoder*as, são criados *layers* de *encoders* utilizando *TransformersEncoders* do Pytorch. Cada torre utiliza o *TransformerEncoder*, que por sua vez possui números arbitrários de *TransformerEncoderLayers*.
- Gating e Output: A Figura 6 mostra camada linear é utilizada para o modelo de *gating* (8) e outra para a camada de saída (9).
- Classification Layer: A Figura 7 mostra camada linear utilizada como saída para a para classificação.

4.1.2. Transformador Encoder-Decoder

Esta segunda arquitetura de transformadores implementada, conta com uma etapa de codificação e uma etapa de decodificação. A etapa de codificação usa blocos *encoder*es idênticos aos de [Liu et al. 2021] e por isso usa a mesma implementação da primeira arquitetura proposta. O de*encoder*, assim como na implementação original de [Vaswani et al. 2017], funciona de maneira auto-regressiva, consumindo os *tokens* (litofácies) de posições anteriores durante o treino e a previsão. Durante o treino, isso é feito através de batches de sequência deslocadas uma posição para a direita, uma máscara para as posições futuras.

O bloco conta com duas etapas principais. A primeira etapa é, a *self-attention*, e recebe como entrada um *batch* contendo sequências de previsões deslocadas de uma posição para a direita. A segunda etapa calcula a *cross-attention* utilizando as saídas da sua primeira etapa, e dos blocos *encoder*.

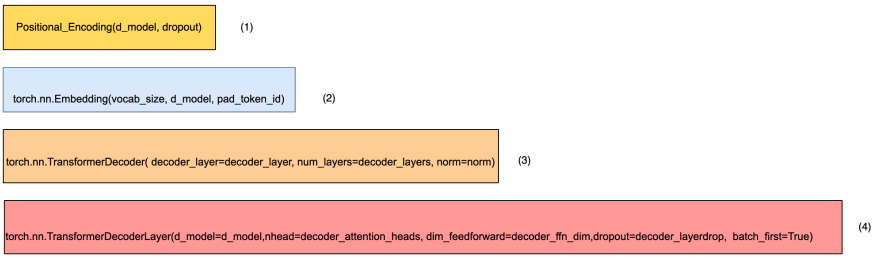


Figura 8. Camadas Pytorch para o bloco *decoder*.

- Embedding:Camada *embedding* do *torch*, com os parâmetros mostradas na Figura 8.
- Positional Encoding: Mesma implementação da primeira arquitetura proposta.
- Encoder: Mesma implementação da arquitetura encoder-only.
- Decoder: Modelos TransformerDecoder da biblioteca do Pytorch como mostrado na Figura 8.
- Output: Uma camada linear do Pytorch idêntica a mostra na Figura 7.

A Tabela 1 descreve os parâmetros utilizados pelas camadas na arquitetura *encoder-decoder*, bem como os valores utilizados.

4.1.3. XGBoost

Um modelo XGBoost foi utilizado como base de comparação para avaliar a eficácia dos métodos propostos. O modelo utiliza os mesmos parâmetros do modelo proposto pelo vencedor da competição, Olawell. O objetivo é criar um modelo base que já consiga resultados desafiadores sem gerar um trabalho significativo. Por conta disso, não foram utilizadas as mesmas técnicas de engenharia de atributos, como os atributos delta propostos por [Bestagini et al. 2017].

4.2. Dados

O objetivo deste trabalho é buscar soluções de transformadores que obtenham resultados relevantes no conjunto de dados fornecidos pela organização XEEK FORCE [Bormann et al. 2020b] durante a competição de 2020.

4.3. Atributos

Os dados do conjunto estão separados em poços, com 98 poços separados para treino, e 10 poços para teste. Esses 10 poços de teste são os mesmo utilizados para avaliar a competição. Para facilitar a comparação direta, a seleção de atributos foi baseada na solução proposta pelo participante Olawale. O participante escolheu os atributos baseando-se principalmente na quantidade de dados faltantes. Os seguintes atributos foram selecionados:

- Atributos Utilizados: `WELL`, `DEPTH_MD`, `X_LOC`, `Y_LOC`, `Z_LOC`, `GROUP`, `FORMATION`, `CALI`, `RSHA`, `RMED`, `RDEP`, `RHOB`, `GR`, `SGR`, `NPHI`, `PEF`, `DTC`, `SP`, `BS`, `ROP`, `DTS`, `DCAL`, `DRHO`, `MUDWEIGHT`, `RMIC`;
- Atributos não selecionados: ROPA, RXO, DTS, SGR.

5. Pré-processamento dos dados

Modelo	Normalização	Imputação	Feature Engineering	Codificação Categoria	Formato atributos de entrada	Formato Facies
Encoder-Only	StandardScaler	ffill + bbfill	-	LabelEncoder	(Batch_size, Sequence_len, Atributos)	(Batch_size, 1)
Encode-Decoder	StandardScaler	ffill + bbfill	-	LabelEncoder	(Batch_size, Sequence_len, Atributos)	(Batch_size, Sequence_len)
Simple XGBoost	StandardScaler	ffill + bbfill	-	LabelEncoder	(Número de Amostras, número de Atributos)	(Número de atributos, 1)
Orwale's XGBoost	StandardScaler	-999	Paolo Bestagini's Gradient	LabelEncoder	(Número de Amostras, número de Atributos)	(Número de atributos, 1)

Tabela 2. Tabela resumindo o pré processamento utilizado nos dados de entrada de cada um dos modelos.

A Tabela 2 da uma visão geral do que foi feito para cada modelo.

6. Treino

Esta seção pretende explicar como foi feito o treino de cada um dos modelos propostos.

6.1. Loss Function

A loss function utilizada foi a cross-entropy.

6.2. Otimização de hiperparâmetros

Transformers: No caso dos modelos transformadores, os parâmetros mostrados na Tabela 1 foram obtidos apos uma *RandomSearch* utilizando 20% dos dados como validação.

6.3. Avaliação

Para medir a eficácia dos métodos propostos, foi utilizada a mesma matriz de pontuação da Figura 9 utilizada na competição XEEK [Bormann et al. 2020a]. Outras métricas de classificação também foram utilizadas como média, f1-score, recall e precisão.

Figura 9. Matriz usada para calcular o resultado baseado no tipo de erro da predição [Bormann et al. 2020a].

7. Experimentos e Resultados

Essa seção mostra os resultados obtidos a partir do conjunto de dados de teste disponibilizados para a competição. A performance foi medida através de métricas normalmente utilizadas, como acurácia, *F-1-score*, além da própria métrica proposta pela competição. Serão apresentadas algumas visualizações importantes como a matriz de confusão das facies litológicas e curvas de facies através de gráficos. Além disso, detalhes de predições individuais para os diferentes modelos serão apresentadas. Os modelos foram treinados utilizando uma GPU RTX NVIDEA 2060.

7.1. Métricas para todos os modelos

	accuracy	precision	recall	f1	competition_score
Encoder-Decoder Model	0.728964	0.477379	0.409583	0.423422	-0.679454
Olawale's Model	**0.805273**	**0.593302**	**0.532208**	**0.547874**	**-0.46978**
Encoder-Only Model	0.761979	0.524509	0.36536	0.347022	-0.620015
XGBoost Model	0.744201	0.512636	0.455306	0.469092	-0.630823

Tabela 3. Métricas de classificação para em relação a todos os poços, para cada modelo

A Tabela 3 mostra os resultados que cada um dos modelos testados obtiveram em todos os poços de teste. De maneira geral, o modelo do ganhador da competição, de Olawale, obteve os melhores resultados em todas as métricas propostas.

O modelo *Encoder-only* obteve resultados significativos também, atingindo o segundo melhor resultado na métrica da competição, a segunda melhor acurácia, e o segundo melhor *recall*. Ele perde para o XGBoost no *recall* e no *F-1-score*. O modelo *Encoder-decoder* ficou em último para todas as métricas apresentadas. Os resultados dessa Tabela mostram que, a princípio, adicionar o *decoder* no modelo de previsão de fácies não mostrou vantagens em relação ao modelo de apenas um *encoder*; na verdade, isso piorou os resultados.

	Sandstone	Sandstone/Shale	Shale	Marl	Dolomite	Limestone	Chalk	Halite	Anhydrite	Tuff	Coal	Basement
Encoder-Decoder Model	0.588822	0.513392	0.909978	0.430619	0.463415	0.509434	0.661962	0.976916	0.675042	0.682253	0.491803	nan
Olawale Model	**0.796084**	0.537247	0.955128	0.370564	**0.533101**	**0.634703**	0.634423	**0.993998**	**0.782245**	**0.755579**	0.573770	nan
Encoder-Only Model	0.677536	0.500692	**0.962284**	**0.543904**	0.484321	0.482207	**0.890878**	0.964451	0.659966	0.741764	**0.631148**	nan
XGBoost Model	0.682236	**0.558495**	0.922494	0.518198	0.487805	0.403989	0.745955	0.985226	0.380235	0.555792	0.627049	nan

Tabela 4. Acurácia total para tipos de facie, por para cada um dos modelos.

A Tabela 4 mostra a porcentagem de acertos de cada um dos modelos para cada uma das facies contidas nos poços de teste. Com exceção das fábricas.

O modelo de Olawale obteve os melhores resultados para todas as facies, com exceção de: Sandstone/Shale, Shale, Marl, Chalk e Coal. Em segundo lugar, o modelo classificado foi o *Encoder-only*, que atingiu os melhores resultados em Shale Marl Chalk Coal. Em terceiro, o Xgboost obteve o melhor resultado apenas no Sandstone/shale. Novamente, o modelo Encoder-decoder ficou em último, sem ter obtido o melhor resultado para nenhuma das facies estudadas.

Esses resultados deixam ainda mais evidente a falta de vantagem do *Encoder-decoder*, ainda se olhando tipos de facies específicas. O modelo *Encoder-only* mostra potencial ao conseguir superar o de Olawale para diversas fábricas.

7.2. Métricas por Poços

Nessa seção, são exploradas as métricas para cada poço. O objetivo dessa comparação é identificar possíveis vantagens entre modelos por conta de características específicas de cada poço.

	Encoder-Decoder Model	Olawale's Model	Encoder-Only Model	XGBoost Model
15/9-23	-0.469674	**-0.433574**	-0.462081	-0.589939
16/2-7	-1.124315	**-0.479308**	-0.955737	-0.986209
16/7-6	-0.475152	-0.640934	-0.587373	**-0.424195**
17/4-1	-0.967373	**-0.530246**	-0.641263	-0.850761
25/10-9	-0.710002	**-0.509223**	-1.218877	-0.570877
31/2-10	-0.370378	**-0.259106**	-0.414508	-0.347739
31/2-21 S	-0.667044	**-0.374091**	-0.520982	-0.424075
34/3-2 S	-0.309921	**-0.277935**	-0.384926	-0.282529
35/11-5	-0.709238	-0.595810	**-0.393139**	-0.755069
35/9-7	-0.742168	**-0.408331**	-0.782722	-0.725344

Tabela 5. Métrica da competição obtida em cada um dos poços, para cada modelo.

A Tabela 5 mostra a porcentagem de acertos de cada um dos modelos para cada uma das litofácies contidas nos poços de teste. Com exceção dos poços 35/11-5 e 16/7-6, o modelo de Olawale obteve os melhores resultados. O *Encoder-only* obteve o melhor resultado no poço 35/11-5.

8. Discussão

8.1. Geral

De maneira geral, o modelo proposto pelo ganhador da competição, Olawale, obteve os melhores resultados. O modelo *Encoder-only* obteve o segundo melhor desempenho para os resultados analisados. O modelo XGBoost obteve o terceiro melhor desempenho. Por fim, o modelo *Encoder-decoder* obteve o pior desempenho.

8.2. Modelo de Olawale vs Encoder-only

Nas métricas gerais, incluindo todos os poços, o modelo de Olawale se saiu melhor. Porém, ao comparar os acertos para cada facie, o modelo *Encoder-only* conseguiu acertos superiores para Shale, Marl, Chalk, Coal.

8.3. Encoder-only vs Encoder-Decoder

O modelo *Encoder-only* obteve melhor resultado em todos os experimentos. Um dos motivos pode estar ligado ao fato do *Encoder-Decoder* não usar o mecanismo de *gating* da mesma maneira que o modelo *Encoder-only* e o resultado da camada para alimentar o *decoder* do modelo.

8.4. Encoder-only vs XGBoost

O modelo *Encoder-only* obteve resultados melhores do que o modelo XGBoost com os parâmetros propostos por Olawale sem o treino rigoroso, indicando que este modelo possa ter potencial futuro.

9. Conclusão

O objetivo do trabalho foi elaborar uma abordagem para a classificação de litofácies utilizando *Wireline Logs* que fizesse uso da arquitetura de Transformadores. Esse objetivo se encaixa com a hipótese de que profundidades próximas estão altamente correlacionadas. Ao conhecimento do autor, este é o primeiro trabalho a utilizar essa arquitetura para a classificação de facies utilizando *Wireline Logs* e, por isso, este projeto também visa servir como um ponto de partida para futuros trabalhos na área.

Na primeira etapa do trabalho, foi realizado um estudo da literatura em busca de modelos de transformadores que poderiam ser adaptados para a classificação de litofácies. Dois tipos de trabalhos se mostraram adequados para o problema: Transformadores Tabulares e Transformadores para séries temporais. Além disso, os trabalhos se diferenciavam quanto a aspectos como: *embedding*, uso de pré-treino, uso de *encoder* e *decoder* ou apenas *encoder*. Neste trabalho, optou-se por explorar a literatura de séries temporais.

A partir dos trabalhos relacionados, foram propostas duas soluções: uma arquitetura que utiliza um *encoder* idêntico ao proposto por [Liu et al. 2021], e uma outra que, além disso, conta com a adição de um bloco *decoder* para capturar a *self-attention* das predições de entradas anteriores. Ambos os modelos foram desenvolvidos em Pytorch, com classes implementadas pela biblioteca *Transformer* do Huggingface. Este trabalho também buscou comparar os resultados com um modelo XGBoost baseado na solução do ganhador da competição XEEK.

Para facilitar a comparação entre os resultados das diferentes soluções, o pré-processamento dos dados e a seleção de atributos foi feita de maneira similar, com a principal diferença sendo a dimensão dos dados de entrada, que varia pelo tipo do modelo utilizado: (label, label), (sequência, label) e (sequência, sequência). Em seguida, foi aplicado um treino supervisionado utilizando os modelos escolhidos.

Os modelos treinados foram, então, utilizados para classificar as fácies dos poços de teste. A partir dessas facies previstas, os resultados do treino mostraram que os modelos transformadores conseguem atingir resultados minimamente razoáveis e podem servir

como ferramentas alternativas aos demais métodos já presentes na literatura. Entretanto, para atingir resultados verdadeiramente competitivos, ainda é necessário um estudo mais aprofundado de *Wireline Logs* e litofácies que se encaixam no problema.

Referências

Alammar, J. (2022). Visualizing machine learning one concept at a time. Disponível em: `https://jalammar.github.io`. Acesso: Outubro, 2022.

Bestagini, P., Lipari, V., and Tubaro, S. (2017). A machine learning approach to facies classification using well logs. pages 2137–2142.

Bormann, P., Aursand, P., and Dilib, F. (2020a). FORCE Machine Learning Competition.

Bormann, P., Aursand, P., Dilib, F., Manral, S., and Dischington, P. (2020b). FORCE 2020 Well well log and lithofacies dataset for machine learning competition.

BRITTANICA (2018). sedimentary facies.

Chen, T. and Guestrin, C. (2016). XGBoost: A scalable tree boosting system. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, pages 785–794, New York, NY, USA. ACM.

Hall, M. and Hall, B. (2017). Distributed collaborative prediction: Results of the machine learning contest. *The Leading Edge*, 36(3):267–269.

Hubbard, S. and Linde, N. (2011). 2.15 - hydrogeophysics. pages 401–434.

Jaikla, C., Devarakota, P., Auchter, N., Sidahmed, M., and Espejo, I. (2019). Faciesnet: Machine learning applications for facies classification in well logs.

Lin, T., Wang, Y., Liu, X., and Qiu, X. (2021). A survey of transformers. *CoRR*, abs/2106.04554.

Liu, M., Ren, S., Ma, S., Jiao, J., Chen, Y., Wang, Z., and Song, W. (2021). *Gated Transformer Networks for Multivariate Time Series Classification*.

Theisges, D., Roisenberg, M., and Nascimento, M. D. S. (2021). Deep recurrent neural networks approach to sedimentary facies classification using well logs. *IEEE Geoscience and Remote Sensing Letters*, PP:1–5.

Tripathi, R. (2022). What are vector embeddings? Disponível em: `https://www.pinecone.io/learn/vector-embeddings/`. Acesso: Setembro, 2022.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. (2017). *Attention Is All You Need*.

Zhang, L. and Zhan, C. (2017). Machine learning in rock facies classification: An application of xgboost. pages 1371–1374.