Comparação entre os modelos RNNs

1. Modelo RNNs com LSTM

Para a construção e aprimoramento do modelo para a Sprint 4, primeiro foi necessário a junção das bases de dados de consumo dos anos de 2019 até 2024, como segue na imagem abaixo:

5. Juntando as Bases de Dados de Consumo

```
import pandas as pd

print(df_2019_amostra.columns == df_2020_amostra.columns)
print(df_2020_amostra.columns == df_2021_amostra.columns)
print(df_2021_amostra.columns == df_2022_amostra.columns)
print(df_2022_amostra.columns == df_2023_amostra.columns)
print(df_2023_amostra.columns == df_2024_amostra.columns)

df_total = pd.concat([df_2019_amostra, df_2020_amostra, df_2021_amostra, df_2022_amostra, df_2023_amostra, df_2024_amostra], ignore_index=True)

print(df_total.head())
```

Depois disso, tem-se um novo dataframe com todos os dados de todos os anos juntos. E então, para a detecção de fraudes e modelo preditivo que identifica se é fraude, ou não, é necessário juntar nesse 'df_total' a base de dados que possui todos os dados que estão relacionados a fraudes. Logo, isso é feito em seguida no código, da seguinte forma abaixo:

6. Juntando a Base de Dados de Consumo Total + Base de Dados de Fraudes

```
# Fazendo o merge com base na coluna 'MATRICULA'
df = pd.merge(df_total, df_fraudes_amostra, on='MATRICULA', how='left')

df.head(2)
```

	Unnamed: 0	EMP_CODIGO	REFERENCIA	COD_GRUPO	COD_SETOR_COMERCIAL	NUM_QUADRA	COD_ROTA_LEITURA	MATRICULA	SEQ_RESPONSAVEL	ECO_RESIDENCIAL	...	VOLUME_ESTIMADO	VOLUME_ESTIMADO_ACUM	FATURADO_MEDIA	COD_LEITURA_INT	STA_TROCA	EXC
0	995459	2.0	2019-01-01	12.0	83.0	524.0	38.0	17458973.0	426182.0	1.0	...	0.0	0.0	NaN	900.0	N	N
1	3500294	2.0	2019-10-01	9.0	63.0	196.0	36.0	17799736.0	282340.0	1.0	...	0.0	0.0	NaN	900.0	N	N

2 rows × 38 columns

Portanto, agora temos o dataframe com todos os dados que precisamos. No entanto, alguns pré-processamentos e tratamento são necessários, como por exemplo, em uma parte do código, teve-se a criação da coluna "Fraudador" para a base de dados de fraude, em que todos os valores, recebem o valor de 1, ou seja, é fraude.

Definindo Fraudador - Nova Coluna

```
def definirFraudador(df, nome_coluna):

    df[nome_coluna] = 1

    return df
```

```
df_fraudes_amostra = definirFraudador(df_fraudes_amostra, 'FRAUDADOR')
df_fraudes_amostra.head(2)
```

Além disso, é importante ressaltar que por conta da base de dados ser muito extensa, durante o código foi definida uma amostra de 30% dos dados de cada base.

No df atualizado, que é o que recebe todos os dados, tanto de fraude quanto de consumo, para os dados que não vieram da base de fraude, ou seja, vieram da base de dados de consumo, os valores da coluna 'Fraudador' ficam Nan, e portanto, o código abaixo atribui o valor de 0 para esses casos:

Colocando o Valor 0 para as linhas com Nan

```
[ ]  # Substituir NaN por 0 na coluna 'FRAUDADOR' e converter os valores para inteiro
     df['FRAUDADOR'].fillna(0, inplace=True)
     df['FRAUDADOR'] = df['FRAUDADOR'].astype(int)

[ ]  df['DAT_LEITURA'] = pd.to_datetime(df['DAT_LEITURA'])
     df['HORA_LEITURA'] = pd.to_numeric(df['HORA_LEITURA'], errors='coerce').fillna(0)
```

Em seguida, foi necessário realizar o tratamento dos dados, o qual preenche os valores Nan das colunas numéricas com a média:

Tratamento de Dados Nulos

- Decisão de manter ou excluir colunas com muitos valores bulos: Se uma coluna contém uma alta proporção de valores nulos (ex. > 60%), talvez seja necessário removê-la, pois pode não fornecer informação útil ao modelo.

```
threshold = 0.6
df = df.dropna(thresh=int((1-threshold)*len(df)), axis=1)
```

```
[ ]  colunasnumericas = df.select_dtypes(include=['float', 'int']).columns

     # Preencha os NaN das colunas numéricas com a média
     for column in colunasnumericas:
         mean_value = df[column].mean()
         df[column].replace(0.0, mean_value, inplace=True)
         df[column].fillna(mean_value, inplace=True)

     df.head()
```

Um outro tratamento importante, é a normalização dos dados, o qual tem como objetivo normalizar as colunas numéricas do dataframe df usando o MinMaxScaler da biblioteca sklearn. A normalização ajusta os valores das colunas numéricas para que fiquem entre 0 e 1, preservando as relações e proporções dos dados originais, o que pode ser útil para algoritmos de machine learning que são sensíveis à escala das variáveis (como redes neurais e SVMs), ajudando a melhorar a performance do modelo.

Essa normalização pode ser vista na imagem do código abaixo:

```
[ ]  from sklearn.preprocessing import MinMaxScaler
     colunasnumericas = df.select_dtypes(include=['float64', 'int64']).columns

     scaler = MinMaxScaler()
     df[colunasnumericas] = scaler.fit_transform(df[colunasnumericas])
```

Um outro tratamento importante, é a conversão de valores categóricos em valores numéricos, utilizando um mapeamento de strings para inteiros na coluna 'CATEGORIA' do dataframe df. Isso é útil quando precisa-se transformar variáveis categóricas e numéricas, já que muitos algoritmos de machine learning requerem que as variáveis estejam em formato numérico.

```
import pandas as pd

mapping = {
    'RESIDENCIAL': 0,
    'COMERCIAL': 1,
    'PUBLICA': 2,
    'INDUSTRIAL': 3
}

df['CATEGORIA'] = df['CATEGORIA'].map(mapping)
```

- 'RESIDENCIAL' será mapeado para 0.
- 'COMERCIAL' será mapeado para 1.
- 'PUBLICA' será mapeado para 2.
- 'INDUSTRIAL' será mapeado para 3.

Após isso, é feita a seleção de features e a definição de amostra dos dados:

```
[ ]  colunas_desejadas = ['MATRICULA', 'CONS_MEDIDO', 'CATEGORIA', 'COD_LATITUDE', 'COD_LONGITUDE', 'FRAUDADOR']

     df = df[colunas_desejadas]

[ ]  df_amostras = df.sample(frac=0.1, random_state=42)

[ ]  timesteps = 1
     num_features = df_amostras.shape[1] - 1
```

1.1 Explicação da Preparação do Modelo

Essa etapa da documentação tem como objetivo explicar como foi preparar os dados para o treinamento de um modelo de machine learning, especificamente aplicando técnicas de balanceamento, imputação de valores ausentes e divisão dos dados entre treino e teste.

- Primeiro, os dados são organizados. As variáveis independentes, ou seja, aquelas que serão usadas para prever se há fraude ou não, são armazenadas na variável X, enquanto a variável dependente, que contém as informações sobre se o caso é de fraude ('FRAUDADOR'), é armazenada na variável y.
- Em seguida, os dados em `X` são redimensionados para um formato apropriado para modelos que exigem uma estrutura de múltiplos passos no tempo, como redes neurais recorrentes. Isso é feito ao usar a função `reshape`, que reorganiza o array `X` para que cada amostra tenha um número específico de "timesteps" (passos de tempo) e "num_features" (número de variáveis ou características).
- Depois, o código lida com valores ausentes. Para garantir que o modelo possa ser treinado adequadamente, é necessário substituir (ou "imputar") esses valores. No caso, a biblioteca `scikit-learn` (importada usando o comando `!pip install sklearn` e o módulo `SimpleImputer`) é utilizada para substituir os valores ausentes pela média das respectivas colunas, o que é uma estratégia comum de imputação. O `SimpleImputer` é aplicado após os dados terem sido reduzidos de uma forma multidimensional para uma forma bidimensional (ou seja, uma linha por amostra).
- Após a imputação dos valores ausentes, os dados são novamente redimensionados para o formato original com os "timesteps" e "num_features".
- Para lidar com o problema de **desbalanceamento de classes** (onde pode haver muito mais casos de não fraude do que de fraude), o código aplica uma técnica chamada **SMOTE** (Synthetic Minority Over-sampling Technique), que cria novas amostras sintéticas da classe minoritária (neste caso, casos de fraude), para equilibrar o número de exemplos entre as classes. No entanto, como o SMOTE requer que os dados estejam em uma forma bidimensional, os dados são temporariamente reduzidos novamente para aplicar a técnica. Depois, os dados resultantes são reorganizados de volta para o formato adequado.
- Por fim, com os dados balanceados, eles são divididos em conjuntos de treino e teste usando a função train_test_split da biblioteca scikit-learn.
    - Cerca de 70% dos dados serão usados para treinar o modelo (X_train, y_train), e;
    - 30% serão usados para testar sua performance (X_test, y_test).

Essa divisão é feita aleatoriamente, mas com um "random_state" fixo, o que garante que os resultados sejam reproduzíveis.

O código abaixo treina um modelo de **rede neural** chamado **LSTM**, que é usado para lidar com **dados sequenciais**, como séries temporais. A ideia é ensinar o modelo a fazer previsões sobre os dados.

```
[ ]  # Divisão entre treino e teste
     X_train, X_test, y_train, y_test = train_test_split(X_resampled, y_resampled, test_size=0.3, random_state=42)

[ ]  model = Sequential()
     model.add(Input(shape=(timesteps, num_features)))
     model.add(LSTM(128, activation='tanh', return_sequences=True))
     model.add(Dropout(0.2))
     model.add(LSTM(64, activation='tanh', return_sequences=False))
     model.add(Dropout(0.2))
     model.add(Dense(32, activation='relu'))
     model.add(Dense(1, activation='sigmoid'))
     model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])

[ ]  history = model.fit(X_train, y_train, epochs=30, batch_size=64, validation_data=(X_test, y_test))

     y_pred_prob = model.predict(X_test)
     threshold = 0.3
     y_pred = (y_pred_prob >= threshold).astype(int)
```

Primeiro, é feita uma divisão dos dados, em que eles são divididos em duas partes: uma para treinar o modelo (treino) e outra para testar sua performance (test).

Em seguida o modelo é montado com várias camadas, incluindo camadas LSTM que são especializadas para dados sequenciais, camadas de Dropout, as quais ajudam a evitar que o modelo aprenda demais e fique muito específico, e camadas Dense, que são como "nós" que ajudam a tomar decisões.

Sendo assim, o modelo é treinado usando os dados de treino por 30 rodadas (épocas), ajustando-se com o tempo para melhorar sua capacidade de prever corretamente.

Após o treinamento, o modelo faz previsões sobre os dados de teste. Ele retorna uma probabilidade de cada exemplo ser da classe 1 (verdadeiro), e essa probabilidade é comparada a um limite (threshold) de 0.3 para decidir se o resultado final será 0 ou 1.

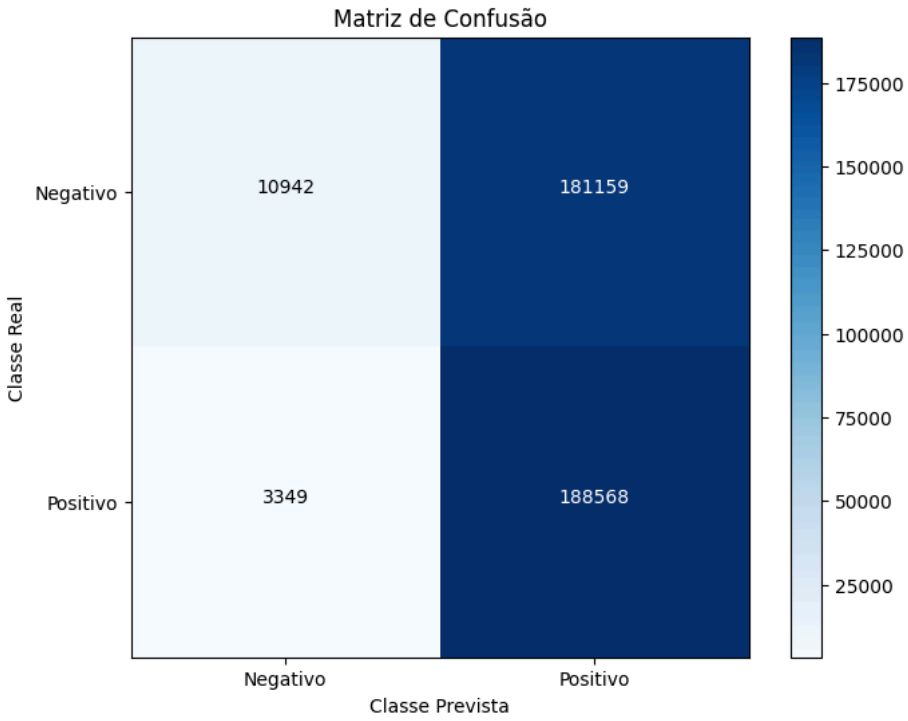
2. Refatoração do Modelo

Com toda a refatoração do modelo para essa Sprint, com ajustes no tratamento dos dados, alteração das amostras de dados, número de épocas, alterações nas camadas do modelo e na arquitetura em si, o modelo em sua última época treinada, atingiu o seguinte resultado:

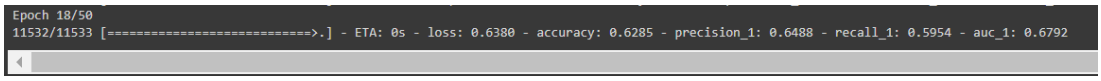
```
14001/14001 [==============================] - 91s 6ms/step - loss: 0.6753 - accuracy: 0.5736 - val_loss: 0.6736 - val_accuracy: 0.5782
12001/12001 [==============================] - 26s 2ms/step
```

- Loss: 0.6753
- Acurácia 0.5736

Segue abaixo a imagem de sua matriz de confusão:



No entanto, um outro modelo também foi treinado e com outras técnicas foi treinado de uma outra maneira. Esse modelo é denominado de modelo de rede neural recorrente com long short term memory, e o resultado obtido desse modelo em si foi o seguinte:



- Loss: 0.6380
- Acurácia: 0.6285

Os dois modelos têm semelhanças, pois ambos utilizam redes neurais recorrentes para lidar com dados sequenciais. No entanto, existem algumas diferenças importantes em termos de arquitetura, regularização e avaliação. Aqui estão as principais distinções entre o primeiro modelo (LSTM) e o segundo (GRU):

1. **Arquitetura da Rede**:

Primeiro modelo (LSTM): usa camadas LSTM (Long Short-Term Memory), que são uma versão mais complexa das redes recorrentes, capazes de capturar dependências de longo prazo. O modelo começa com uma camada LSTM de 128 unidades, seguida por outra LSTM de 64 unidades, ambas com a função de ativação

'tanh'. O LSTM tem uma estrutura interna mais sofisticada, com uma célula de memória separada que gerencia o fluxo de informações.

Segundo modelo (GRU): usa camadas GRU (Gated Recurrent Units), que são uma variação mais simples das LSTM, com menos parâmetros e um mecanismo mais direto de controle de informações. As GRUs tendem a ser mais rápidas para treinar e podem funcionar tão bem quanto as LSTMs em muitos casos, especialmente quando se trabalha com dados temporais que não exigem uma modelagem tão complexa das dependências de longo prazo.

2. **Regularização (Dropout)**: ambos os modelos aplicam dropout para reduzir o overfitting.

No primeiro modelo (LSTM), o dropout é aplicado após cada camada LSTM com uma taxa de 0.2;

Enquanto no segundo modelo (GRU), também há dropout aplicado com a mesma taxa de 0.2 após cada camada GRU.

3. **Métricas e Compilação**:

Primeiro modelo (LSTM): compila o modelo com o otimizador **Adam**, função de perda **binary_crossentropy** e mede a acurácia ('accuracy'). As métricas se concentram principalmente na acurácia durante o treinamento e validação.

Segundo modelo (GRU): compila o modelo também com o otimizador **Adam**, mas com métricas adicionais: **Precision**, **Recall**, e **AUC (Area Under the Curve)**. Essas métricas fornecem uma visão mais completa do desempenho do modelo, especialmente em problemas de classificação binária desequilibrada. Além disso, é calculada a **F1-Score** após o treinamento para combinar a precisão e recall, oferecendo uma métrica robusta em cenários de classes desbalanceadas.

4. **Estratégia de Treinamento**:

Primeiro modelo (LSTM): usa uma simples divisão de treino e teste com o método `train_test_split`, treinando o modelo por 30 épocas e avaliando a performance apenas no conjunto de teste.

Segundo modelo (GRU): implementa uma validação cruzada com **K-Fold**, o que significa que o modelo é treinado em diferentes partes dos dados e validado em outras, repetidamente, para garantir que os resultados não sejam influenciados pela aleatoriedade de uma única divisão de dados. Além disso, utiliza **EarlyStopping**, que para o treinamento caso o modelo pare de melhorar após um certo número de épocas, prevenindo o overfitting.

5. **Predição e Decisão Final**:

Primeiro modelo (LSTM): faz previsões sobre os dados de teste e compara as probabilidades com um limite (threshold) de 0.3 para decidir entre 0 e 1 (binário).

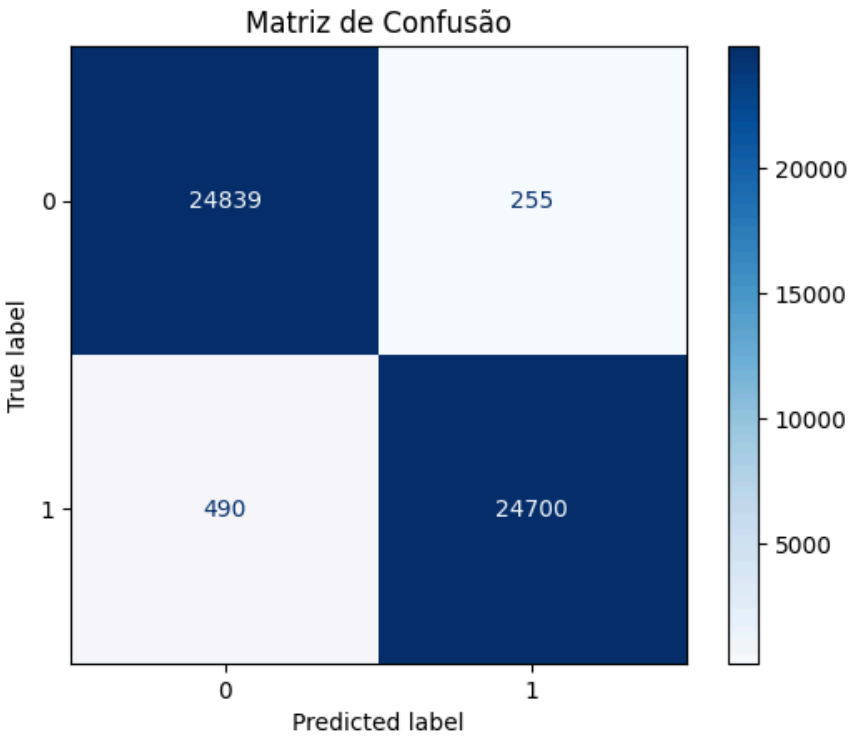
Segundo modelo (GRU): cada fold é avaliado com várias métricas após o treinamento. Não há uma menção específica de threshold no código, o que sugere que as métricas como precisão e recall já consideram essa divisão implicitamente durante o treinamento e avaliação.

No entanto, há um outro modelo que em comparação a esses dois modelos, obteve uma melhor performance:

```
Epoch 29/30
786/786 [==============================] - 20s 25ms/step - loss: 0.0187 - accuracy: 0.9934 - precision_2: 0.9945 - recall_2: 0.9923 - auc_2: 0.9995 -
1572/1572 [==============================] - 6s 4ms/step - loss: 0.0431 - accuracy: 0.9852 - precision_2: 0.9898 - recall_2: 0.9805 - auc_2: 0.9980
Validation Loss: 0.0431
Validation Accuracy: 0.9852
Validation Precision: 0.9898
Validation Recall: 0.9805
Validation AUC: 0.9980
Validation F1 Score: 0.9851
```

- Loss: 0.187
- Acurácia: 0.9852

A imagem abaixo mostra a sua matriz de confusão:



Portanto, abaixo, segue a comparação e diferença desses três modelos apresentados:

No **primeiro** modelo, é utilizada uma arquitetura de redes neurais LSTM com duas camadas, sendo a primeira com 128 unidades e a segunda com 64 unidades, ambas com dropout de 0.2. A divisão dos dados é feita de maneira simples entre treino e teste, com 70% para treino e 30% para teste. A única métrica usada para avaliar o desempenho do modelo é a acurácia, e o treinamento ocorre por 30 épocas sem early stopping. Esse modelo é direto, focando em uma simples avaliação de acurácia e uma estrutura básica de rede neural recorrente.

Já o **segundo** modelo opta pelo uso de GRU em vez de LSTM. As camadas GRU possuem 64 e 32 unidades, respectivamente, com dropout de 0.2 após cada camada. Diferentemente do primeiro modelo, ele usa K-Fold Cross-Validation com dois folds, o que permite avaliar o modelo em diferentes divisões de treino e validação, aumentando a robustez da avaliação. Além disso, várias métricas são utilizadas, como acurácia, precisão, recall, AUC e F1-Score, oferecendo uma análise mais detalhada do desempenho do modelo. A introdução de early stopping com uma paciência de 5 épocas ajuda a evitar overfitting e ajustar automaticamente o número de épocas necessário.

No **terceiro** modelo, que também usa GRU, a arquitetura é bastante semelhante ao segundo modelo, com camadas de 64 e 32 unidades, e dropout de 0.3, o que indica uma regularização mais forte para prevenir overfitting. Ele também adota uma abordagem mais simples em relação à validação, dividindo o conjunto de dados em 80% para treino e 20% para validação, sem o uso de K-Fold. As mesmas métricas detalhadas de precisão, recall, AUC e F1-Score são calculadas, tornando a análise do desempenho igualmente detalhada. No entanto, uma diferença significativa é a inclusão de uma matriz de confusão que é gerada e plotada ao final, permitindo uma visualização clara dos verdadeiros positivos, falsos positivos, verdadeiros negativos e falsos negativos, o que facilita a análise qualitativa do modelo. Além disso, o modelo também utiliza early stopping com paciência de 5 épocas, como no segundo exemplo, para otimizar o processo de treinamento.

3. Relatório de avaliação de métricas

Após a criação desses 3 modelos, abaixo segue a documentação do relatório de avaliação de métricas primeiramente para o primeiro modelo, e em seguida para o terceiro modelo.

3.1 Relatório de Métricas do Modelo 1

O modelo foi treinado durante **30 épocas** com um **batch size de 64**, utilizando o conjunto de dados de treino e validando com o conjunto de teste. Durante o treinamento, houve uma melhora progressiva tanto na **perda (loss)** quanto na **acurácia (accuracy)** em ambos os conjuntos de treino e validação.

Desempenho ao longo das épocas

- **Perda de Treinamento (loss):** Iniciou em 0.6855 e terminou em 0.6753.
- **Acurácia de Treinamento (accuracy):** Iniciou em 0.5514 e aumentou gradualmente para 0.5736.
- **Perda de Validação (val_loss):** Iniciou em 0.6825 e finalizou em 0.6736.
- **Acurácia de Validação (val_accuracy):** Iniciou em 0.5564 e finalizou em 0.5782.

Métricas de avaliação no conjunto de teste

Após o treinamento, o modelo foi avaliado no conjunto de teste, utilizando um threshold de 0.3 para classificação. Os resultados de precisão, recall e F1-Score foram gerados para as duas classes (0 e 1).

Desempenho por Classe

- Classe 0 (negativa):
    - Precisão: 0.77
    - Recall: 0.06
    - F1-Score: 0.11
    - Total de exemplos (support): 192,101
- Classe 1 (positiva):
    - Precisão: 0.51
    - Recall: 0.98
    - F1-Score: 0.67
    - Total de exemplos (support): 191,917

Desempenho Geral

- Acurácia Geral: 0.52
- Média Macro:
    - Precisão: 0.64
    - Recall: 0.52
    - F1-Score: 0.39
- Média Ponderada:
    - Precisão: 0.64
    - Recall: 0.52
    - F1-Score: 0.39

Observações

- **Acurácia geral:** é relativamente baixa, situando-se em 0.52. Isso sugere que o modelo está tendo dificuldades em classificar corretamente os dados, apesar de um bom **recall** para a classe positiva.
- **Desempenho nas classes:** a precisão para a **classe 0** é alta (0.77), mas o **recall** é extremamente baixo (0.06), indicando que o modelo quase não identifica verdadeiros negativos. Para a **classe 1**, o recall é excelente (0.98), mas a precisão é mais baixa (0.51), indicando que o modelo classifica muitos exemplos como positivos, mesmo que sejam falsos positivos.
- **F1-Score:** para a classe 0 é bastante baixo (0.11), o que confirma o fraco desempenho na detecção dessa classe.

Logo, este modelo tem um desempenho claramente enviesado para a classe 1 (positiva), onde obtém um recall elevado, mas à custa de uma baixa precisão. O modelo pode estar enfrentando dificuldades com um dataset desbalanceado, onde a classe 0 está sub-representada em termos de verdadeira detecção. Estratégias como

ajuste do threshold, balanceamento de classes ou tuning do modelo podem melhorar o desempenho geral.

3.2 Relatório de Métricas do Modelo 3

Este relatório apresenta o desempenho do modelo baseado em redes GRU, que foi treinado e avaliado em um conjunto de dados com uma divisão de 80% para treino e 20% para validação. O modelo foi treinado com early stopping, configurado para monitorar a métrica de perda (val_loss) com uma paciência de 5 épocas, e utilizou uma arquitetura com duas camadas GRU (64 e 32 unidades, respectivamente) e Dropout para regularização.

Métricas de Treinamento

- **Loss (Perda):** 0.0187
- **Accuracy (Acurácia):** 99.34%
- **Precision (Precisão):** 99.45%
- **Recall:** 99.23%
- **AUC (Área sob a Curva ROC):** 0.9995

As métricas de treinamento indicam que o modelo apresenta uma perda muito baixa (0.0187) e uma acurácia extremamente alta (99.34%), o que sugere que o modelo está capturando muito bem os padrões presentes nos dados de treino.

Além disso, a precisão de 99.45% indica que o modelo realiza previsões corretas na maior parte das vezes, enquanto o recall de 99.23% demonstra que ele consegue identificar a maioria dos casos positivos.

O AUC próximo de 1 (0.9995) indica uma excelente capacidade de discriminar entre classes positivas e negativas.

Métricas de Validação

- **Validation Loss (Perda de Validação):** 0.0431
- **Validation Accuracy (Acurácia de Validação):** 98.52%
- **Validation Precision (Precisão de Validação):** 98.98%
- **Validation Recall (Recall de Validação):** 98.05%
- **Validation AUC (Área sob a Curva ROC de Validação):** 0.9980
- **Validation F1 Score:** 98.51%

Os resultados de validação mostram que o modelo se generaliza bem para dados não vistos, com uma perda de validação de 0.0431, um leve aumento em relação à perda de treino, o que é esperado.

A acurácia de validação de 98.52% também indica um excelente desempenho, embora seja ligeiramente inferior à acurácia de treino.

A precisão de 98.98% e o recall de 98.05% mostram que o modelo tem um excelente equilíbrio entre prever corretamente as classes positivas e capturar a maioria dos exemplos positivos.

O F1-Score de 98.51%, uma métrica que combina precisão e recall, confirma esse bom desempenho geral.

O AUC de 0.9980 continua muito próximo de 1, confirmando que o modelo tem uma excelente capacidade de distinção entre as classes.

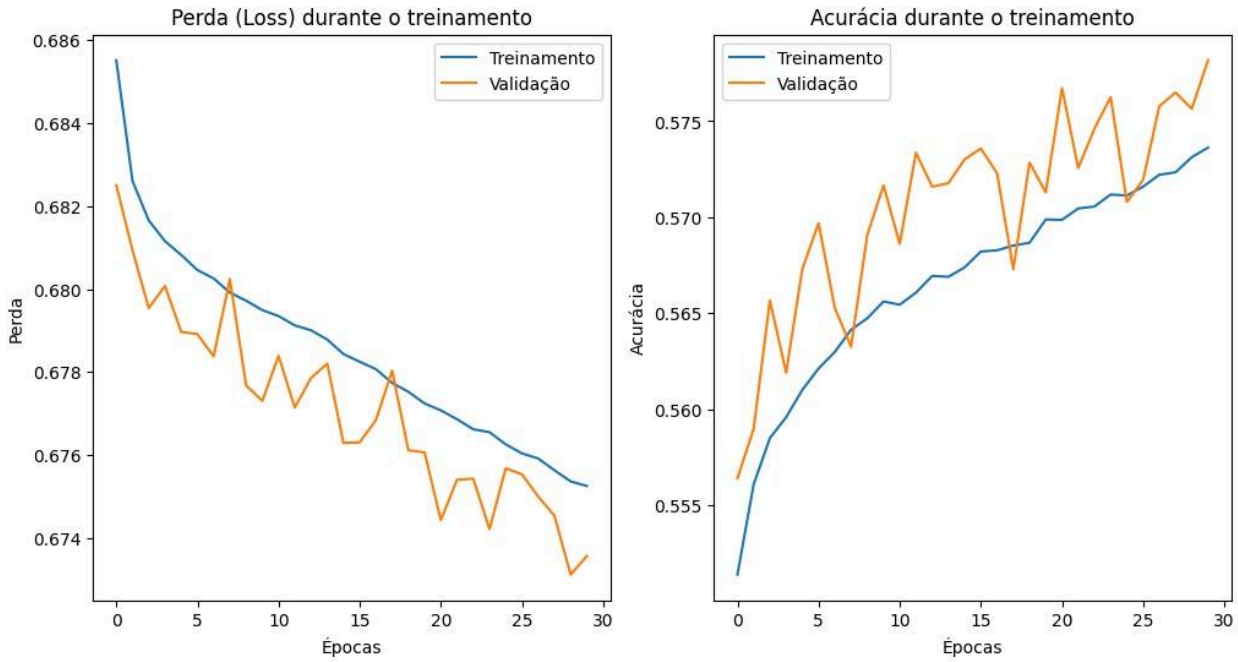
Os resultados gerais indicam que o modelo treinado é altamente preciso e robusto, tanto nos dados de treino quanto de validação. A diferença mínima entre as métricas de treino e validação sugere que o modelo não está superajustado (overfitting), o que é evidenciado pelo uso eficaz de dropout para regularização e early stopping. O alto valor de AUC (> 0.99) reforça que o modelo consegue discriminar com alta eficiência entre as classes, enquanto o F1-Score alto (98.51%) demonstra um excelente balanço entre precisão e recall.

Com base nas métricas observadas, este modelo GRU é uma solução altamente eficaz para o problema em questão, oferecendo alta precisão e generalização. Ele é adequado para aplicações em cenários onde uma excelente discriminação entre classes e um equilíbrio entre precisão e recall são essenciais, como em sistemas de detecção de fraudes, análise preditiva e outros problemas binários, sendo assim está dentro dos critérios do projeto e representa o melhor modelo para a Sprint em questão.

4. Comparação dos Modelos

O modelo que teve o desempenho foi o terceiro e é possível notar pelos seguintes gráficos a grande diferença em comparação com o modelo 1:

Modelo 1 - RNNs com LSTM (rede neural recorrente com long short term memory) com 30 épocas com um batch size de 64.



Acima, observa-se o desempenho de uma RNN com LSTM durante o treinamento e validação, com duas métricas principais: perda (loss) e acurácia ao longo das épocas.

1. **Perda (Loss):**

A curva de perda diminui constantemente tanto para os dados de treinamento quanto para os dados de validação, indicando que o modelo está aprendendo a reduzir o erro ao longo do tempo.

As curvas de treinamento e validação estão bem próximas, o que sugere que não há um overfitting significativo até o momento.

2. **Acurácia:**

A acurácia aumenta tanto no conjunto de treinamento quanto no de validação, o que é um bom sinal.

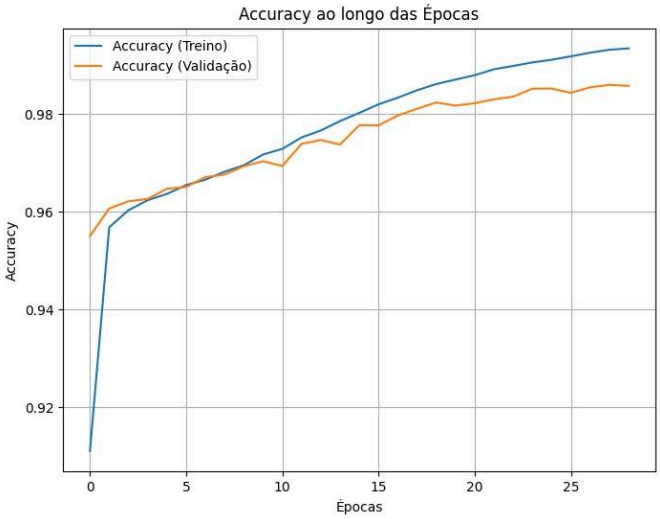
Há variações mais acentuadas na curva de validação, indicando oscilações nas predições de validação, possivelmente devido a uma sensibilidade maior aos dados de validação (podem estar menos balanceados ou mais diversos).

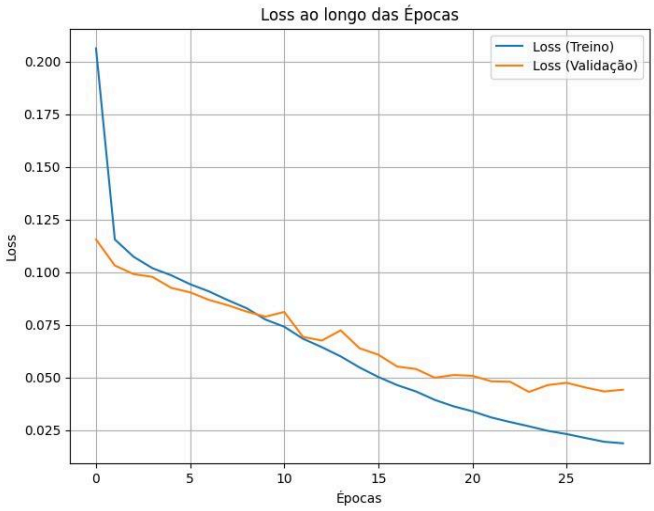
Para melhorar o modelo, podemos fazer o seguinte:

1. Adicionar camadas como Dropout ou L2 regularization pode ajudar a estabilizar a performance e reduzir oscilações.
2. Ajuste de hiperparâmetros, ou seja, alterar a taxa de aprendizado ou o número de unidades LSTM pode melhorar o aprendizado.

Logo, pode-se concluir que o modelo de melhor aprendizado é o terceiro.

Modelo 3 - Modelo de rede neural recorrente com long short term memory (GRU)





O gráfico de perda (loss) ao longo das épocas para o Modelo 3, mostra tanto a perda de treinamento quanto a perda de validação.

A perda de treinamento e validação diminuem de forma consistente ao longo das épocas, sugerindo que o modelo está aprendendo com os dados e está ajustando seus parâmetros de forma adequada.

As curvas de treinamento e validação são bastante próximas, indicando que o modelo não está superajustando aos dados de treinamento, ou seja, não há overfitting.

A perda para ambos os conjuntos (treinamento e validação) estabiliza nas últimas épocas, o que indica que o modelo está se aproximando do seu limite de aprendizado, mas sem grandes flutuações, o que sugere um comportamento estável.