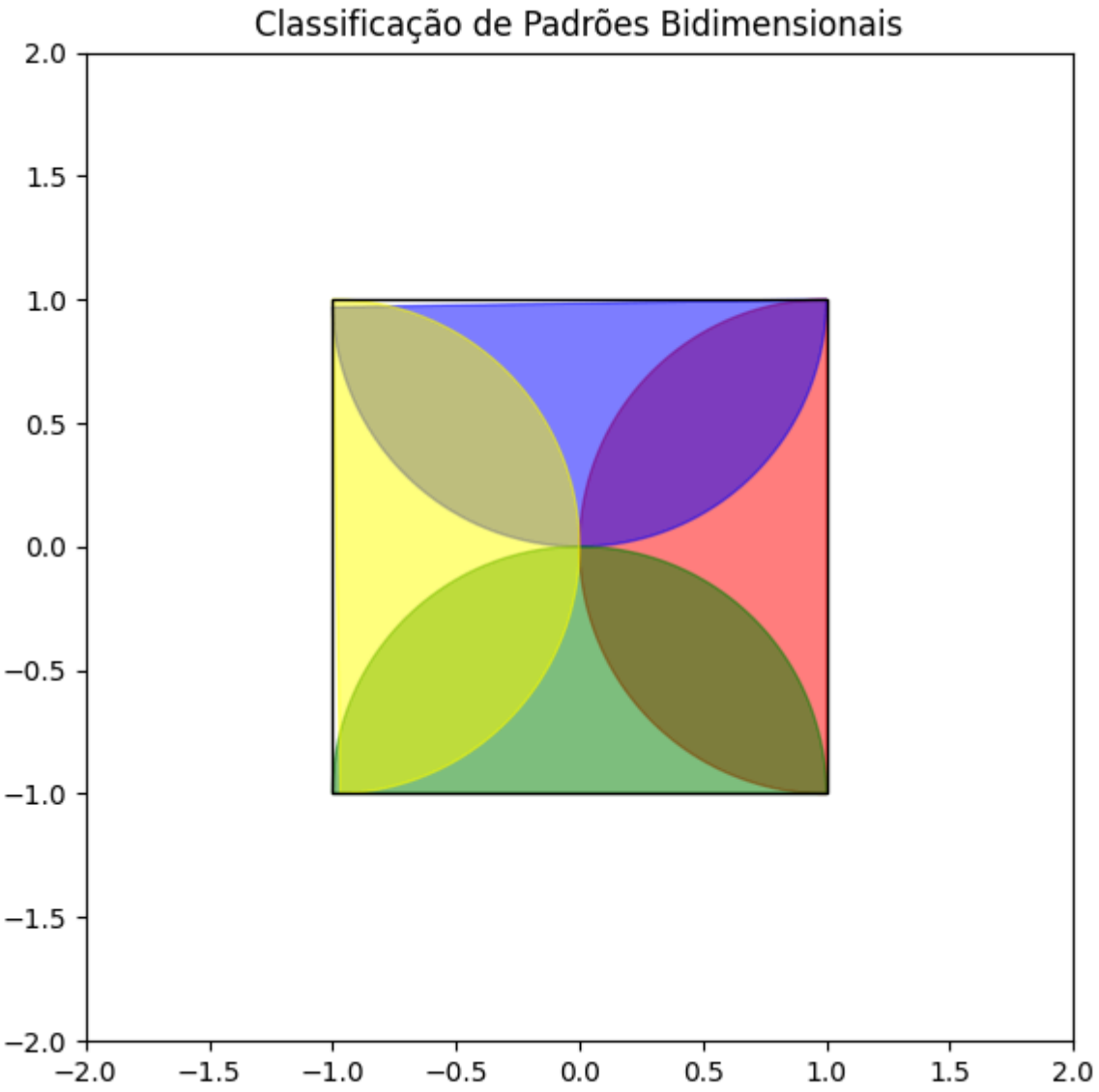
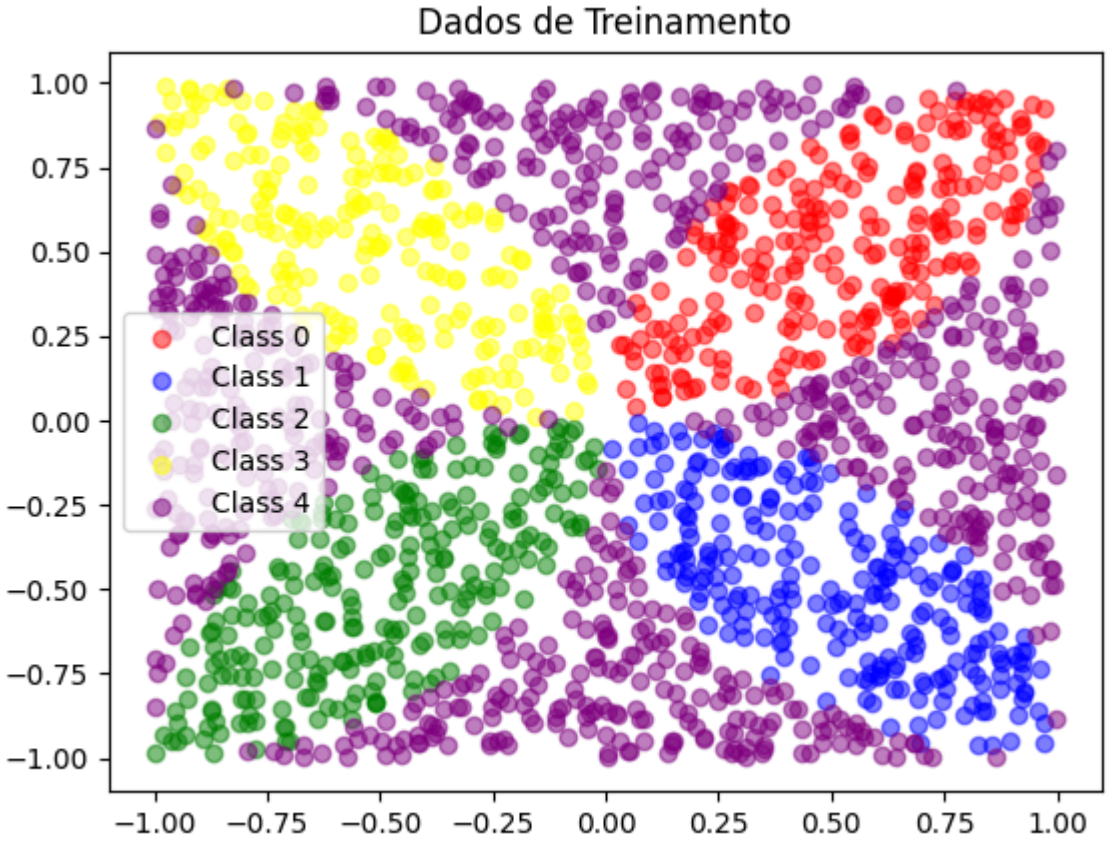
1) A região descrita pelo problema é a seguinte:



Para a próxima etapa foram gerados 500 dados para cada padrão e foram considerados um percentual de 30% para testes, após a geração dos dados temos o seguinte:



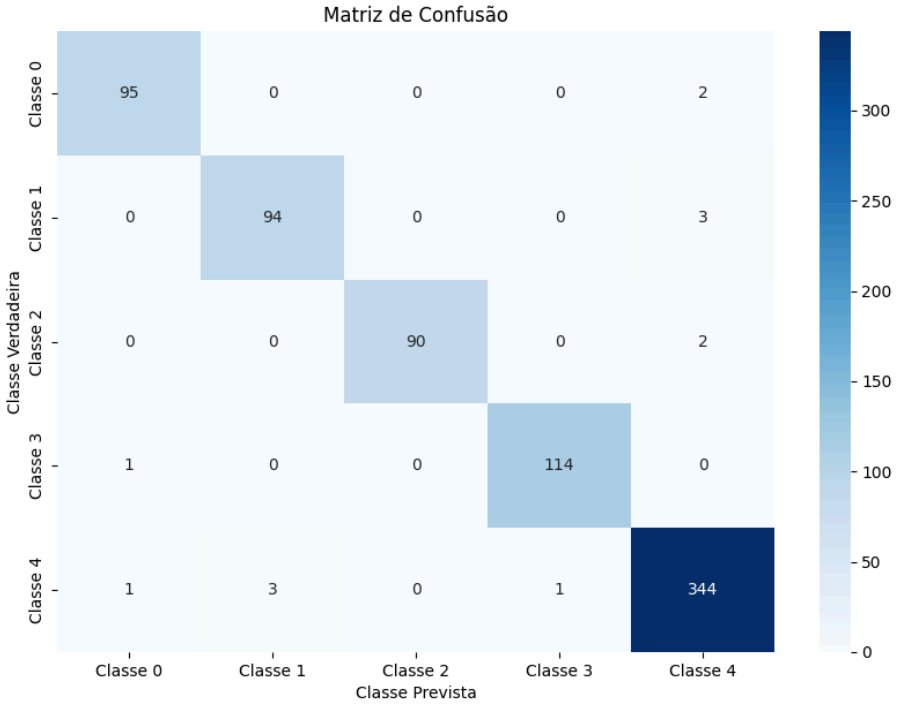
A MLP é definida no seguinte trecho do código:

```
mlp = MLPClassifier(hidden_layer_sizes=(100, 50), max_iter=1000, random_state=42)
```

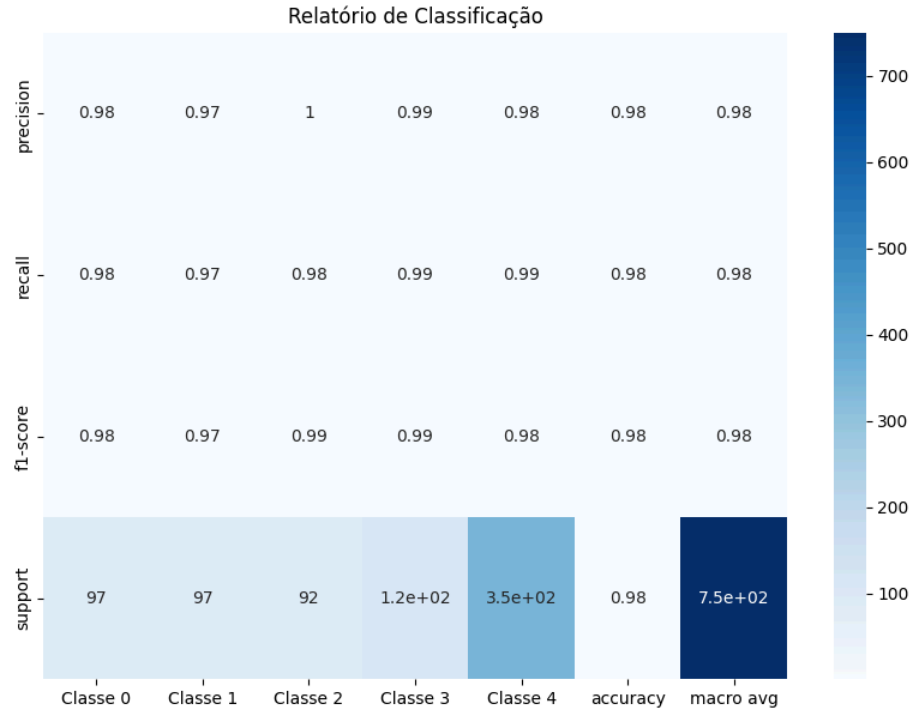
Onde a primeira camada tem 100 neurônios e a segunda 50, um critério de parada é o número máximo de iterações igual a 1000, além disso a rede utiliza a função softmax na camada de saída para realizar classificação multiclasse e a otimização utiliza o algoritmo Adam.

Os resultados obtidos foram os seguintes:

- Matriz de confusão:



- Relatório de classificação:



2) A função é: $f(x) = x_1^2 + x_2^2 + 2x_1x_2 + \cos(x_1 + x_2) - 1$, com isso vamos gerar 1000 pontos para x_1 e x_2 entre [-5 , 5]. Vamos também definir 20% dos dados para teste e 80% para treino.

Sobre a definição da arquitetura da rede neural utilizamos o modelo sequencial do Keras, foi adicionado também uma camada oculta com 10 neurônios e a função de ativação ReLU. Para a camada de saída foi adicionado 1 neurônio com função de ativação linear, além disso o modelo foi otimizado utilizando o algoritmo Adam.

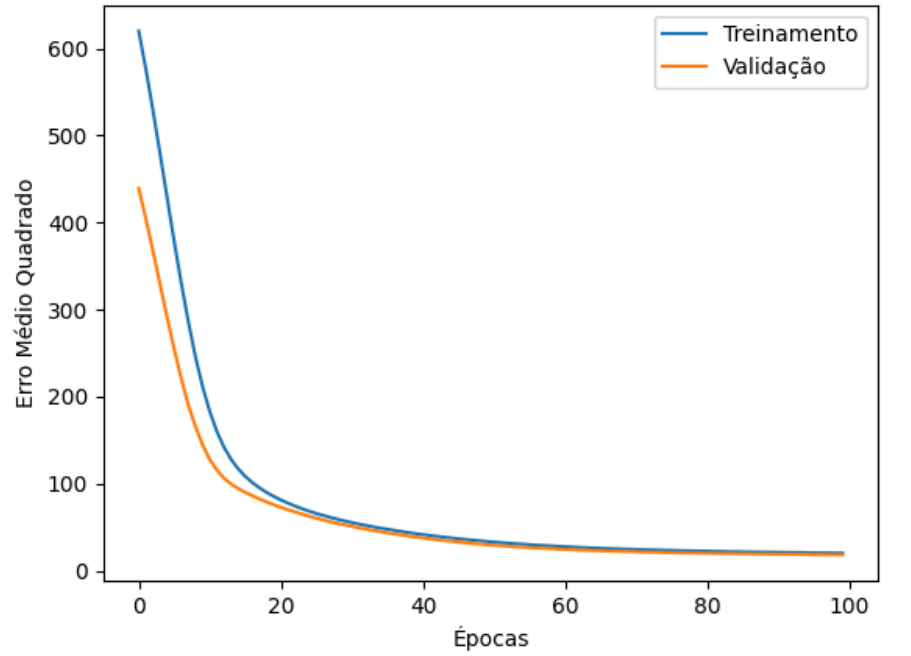
No processo de treinamento foram consideradas 100 épocas e tamanho de lotes igual a 20, utilizando os 20% dos dados de treino para a validação. O código vai estar no anexo.

Fizemos o treino para 2 redes distintas, uma mais simples e outra mais complexa, sendo elas:

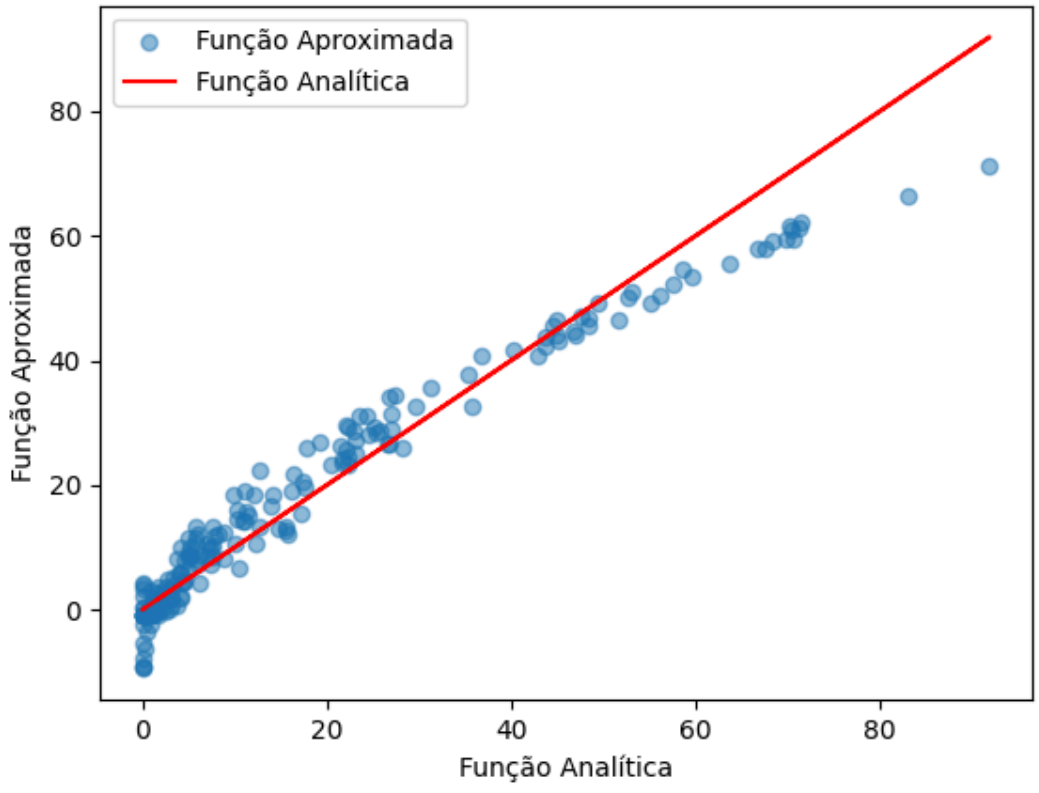
- **Simples**: Camada de entrada com 2 neurônios correspondentes a x1 e x2;
  Camada oculta com 1 com 10 neurônios e função de ativação ReLU;
  Camada de saída com 1 neurônio com função de ativação linear.

- **Complexa:** 2 neurônios na camada de entrada, correspondentes a x1 e x2;
  Primeira camada oculta com 20 neurônios e função de ativação ReLU;
  Segunda camada oculta com 10 neurônios e função de ativação ReLU;
  Camada de saída com 1 neurônio com função de ativação linear.

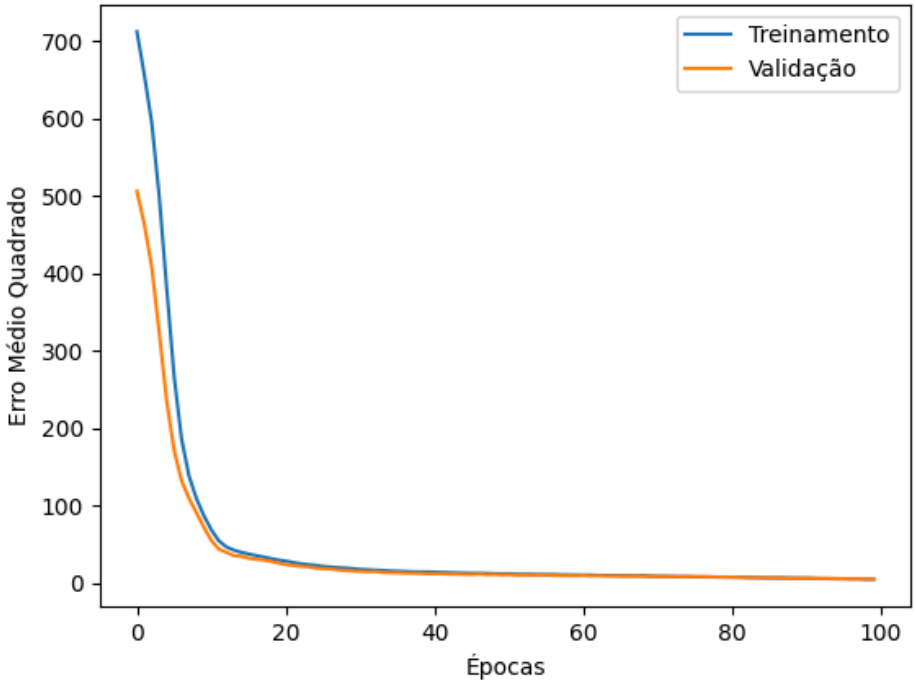
Para a rede **simples** o gráfico da função custo durante o treino foi o seguinte:



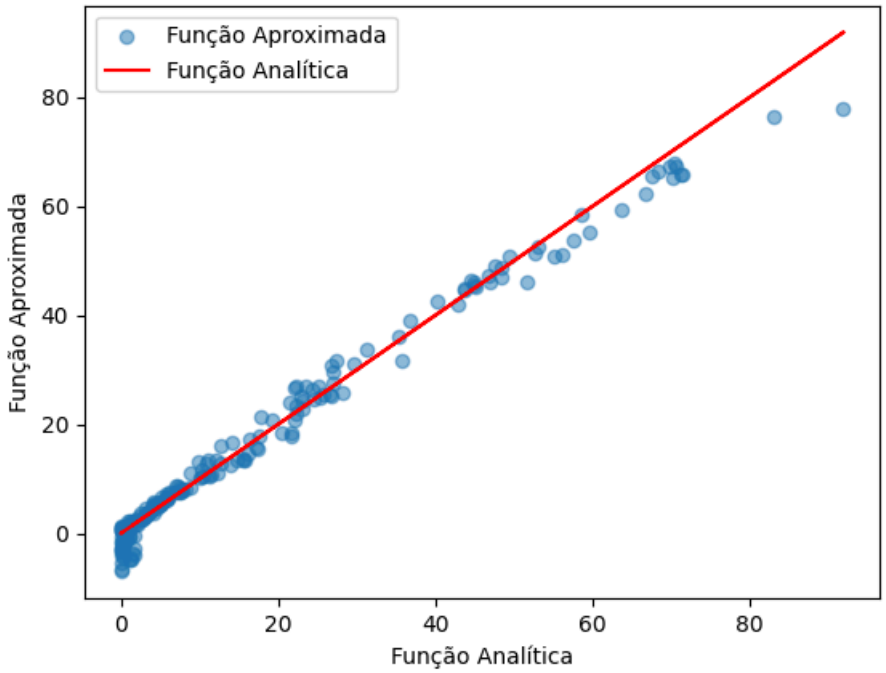
O erro médio quadrático no conjunto de teste foi de aproximadamente 20,59, e o gráfico da função aproximada foi:



Para a rede **complexa** o gráfico da função custo durante o treino foi o seguinte:



O erro médio quadrático no conjunto de teste foi de aproximadamente 6,53, e o gráfico da função da função aproximada foi:



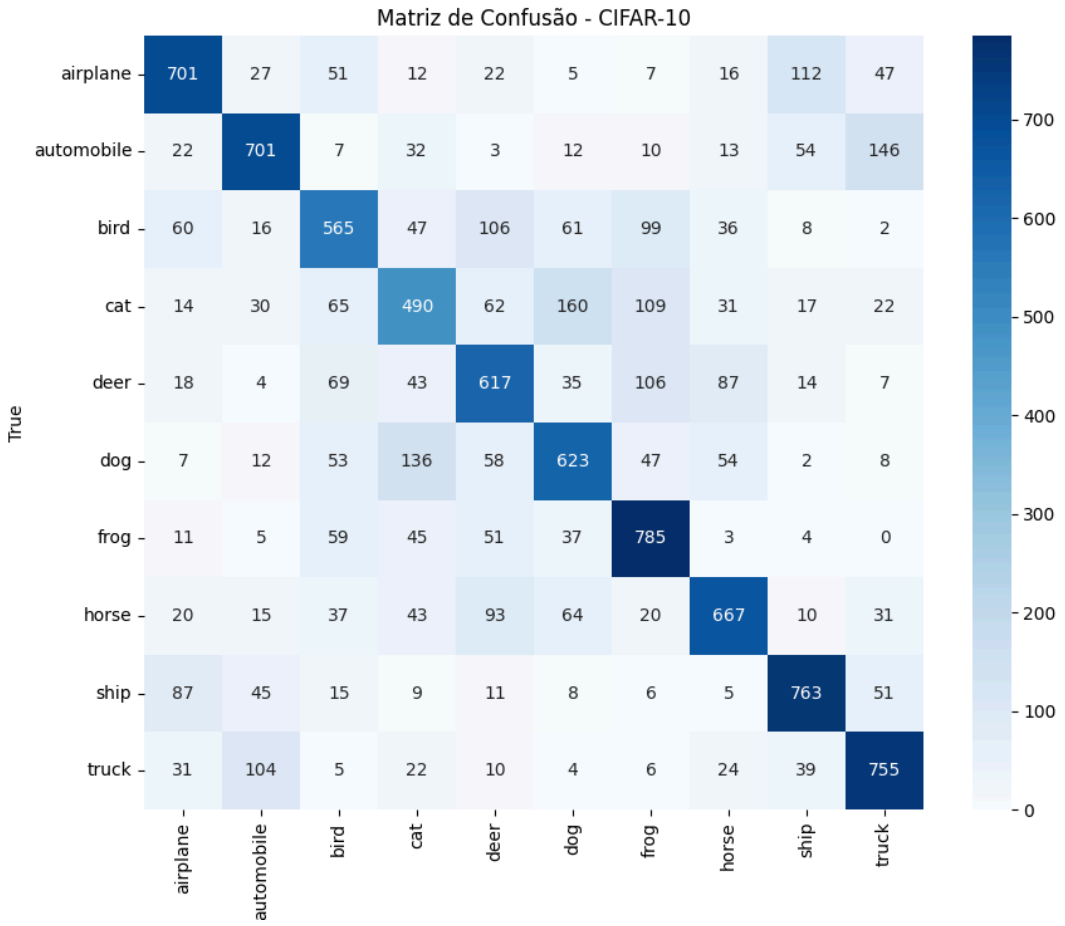
3)

CIFAR-10 consiste num conjunto de dados de imagens bastante utilizado em tarefas de classificação em machine learning e deep learning, especialmente usados em redes do tipo convolucionais CNN para a classificação de imagens, sua principal vantagem é sua é que ela é compacta e de fácil manipulação. No caso da questão temos um banco com 60 mil imagens coloridas e cada imagem contendo 32x32 pixels e 3 canais de cores RGB. Desse montante vamos considerar 50 mil para treino do modelo e 10 mil para testes.

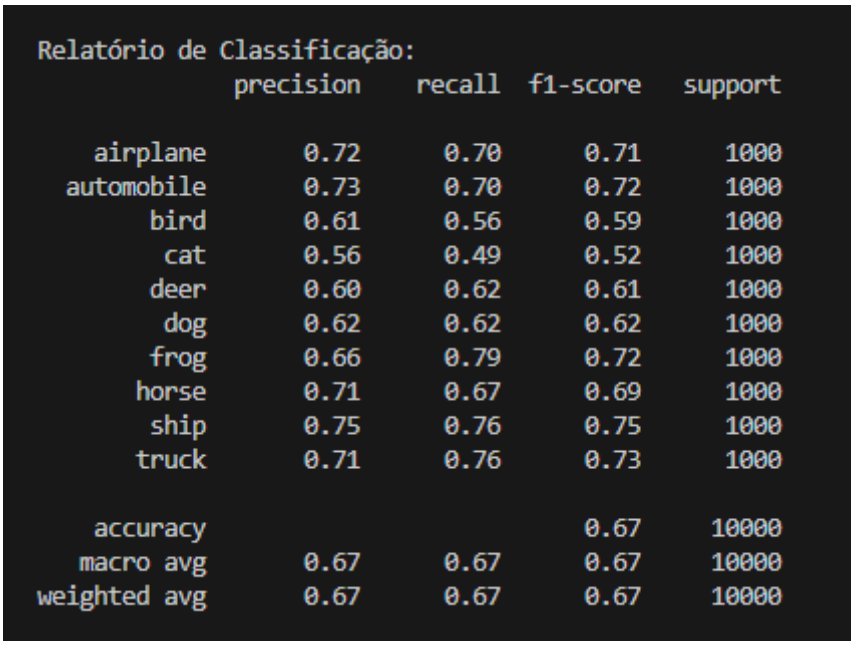
Devemos considerar também que cada imagem pode ter somente um objeto da classe podendo estar obstruído por outros objetos que não façam parte da classe de interesse.

O conjunto de dados utilizados, CIFAR-10, foi importado a partir da biblioteca TensorFlow, onde está disponível por padrão. O mesmo ocorre com a rede convolutiva pré-treinada, ResNet-50, também disponível na biblioteca TensorFlow.

Primeiramente o dataset é carregado e os dados são normalizados, enquanto os rótulos são convertidos para One-Hot encode. Então o modelo ResNet-50 é carregado e usado para criar o modelo que será treinado. Após o treinamento, utilizando 10 épocas, o modelo é avaliado utilizando o conjunto de teste e a matriz de confusão é plotada. Por fim, um relatório de classificação é impresso, mostrando as métricas de avaliação do modelo e o número de amostras reais disponíveis para os cálculos (support).



Matriz de confusão



Relatório de Classificação:

	precision	recall	f1-score	support
airplane	0.72	0.70	0.71	1000
automobile	0.73	0.70	0.72	1000
bird	0.61	0.56	0.59	1000
cat	0.56	0.49	0.52	1000
deer	0.60	0.62	0.61	1000
dog	0.62	0.62	0.62	1000
frog	0.66	0.79	0.72	1000
horse	0.71	0.67	0.69	1000
ship	0.75	0.76	0.75	1000
truck	0.71	0.76	0.73	1000
accuracy			0.67	10000
macro avg	0.67	0.67	0.67	10000
weighted avg	0.67	0.67	0.67	10000

Relatório de classificação

4)

A rede NARX é um tipo de rede recorrente utilizada para modelagem e previsão de séries temporais. Uma de suas principais características é utilizar dados passados da série e também dados externos para prever dados futuros. É bastante utilizada para prever preços de ações ou demanda de energia, por exemplo. Para o primeiro passo definimos a série temporal como sendo x(n)=sqrt(1+sin(n+sin2(n))). Em seguida montamos um array com valores entre 0 a 1000 com incrementos de 0,1, os valores de 'x' no código representam valores gerados pela série temporal. Definimos o número de passos anteriores como entrada igual a 4 e vamos utilizar a metodologia 80/20, 80% dos dados para treino e 20% para teste. Lembrando que utilizamos o vetor de entrada $x(n)=[x(n),x(n-1),x(n-2),x(n-3)]^t$, conforme sugerido no enunciado da questão, e para a saída definimos como desejada x(n+1).

Definimos também a arquitetura da rede como sendo um modelo sequencial, uma camada de 50 neurônios com uma função de ativação ReLU e o número de passos anteriores como entrada, para a saída temos 1 neurônio para a predição, a rede também utiliza o algoritmo Adam como otimizador.

Para o treino definimos que o números de épocas foi 100 e batch = 32 e o erro de predição definido como $e(n+1)=x(n+1)-x^{(n+1)}$. Após o treino da rede tivemos os seguintes resultados:

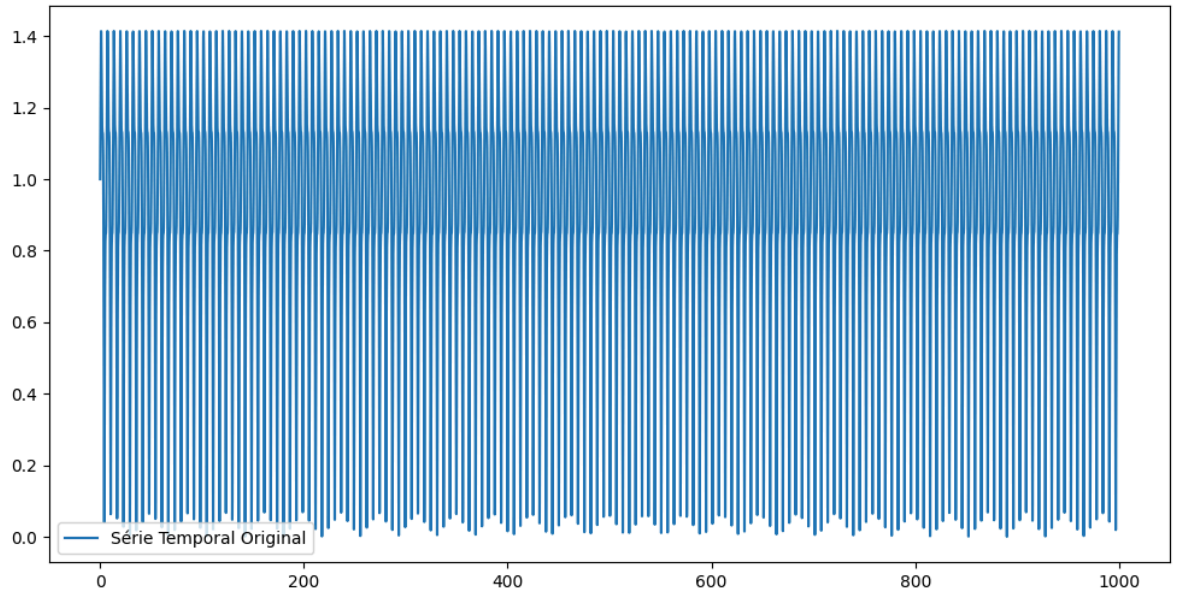


Figura XX - Gráfico da série temporal original

Da Figura XX vemos que a série temporal tem um padrão altamente oscilatório, sua regularidade sugere que há uma estrutura que possa ser modelada e prevista. Já na Figura YY vemos a comparação entre os valores reais e os previstos, podemos perceber que as linhas parecem se sobrepor indicando que os valores preditos seguem próximos dos reais, e que a rede foi bem montada e capaz de capturar bem a estrutura da série temporal. Por fim, na Figura ZZ vemos o gráfico do erro entre os valores previstos e reais, é possível notar que o erro varia dentro de uma faixa estreita e sua flutuação em torno do valor 0 indica que o modelo foi bastante preciso em suas previsões.

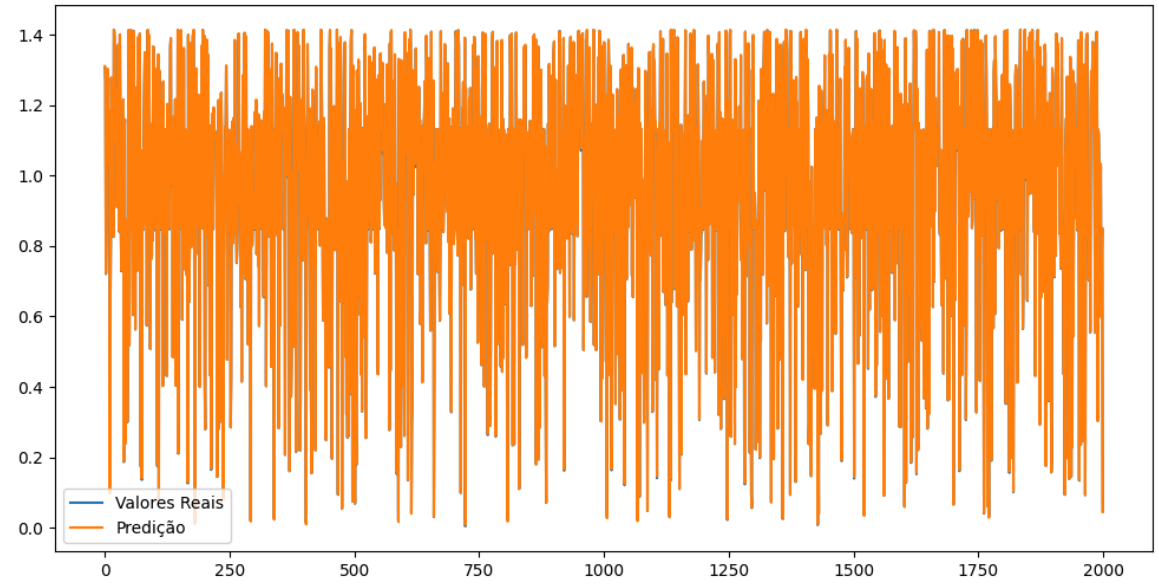


Figura YY - Gráfico da predição

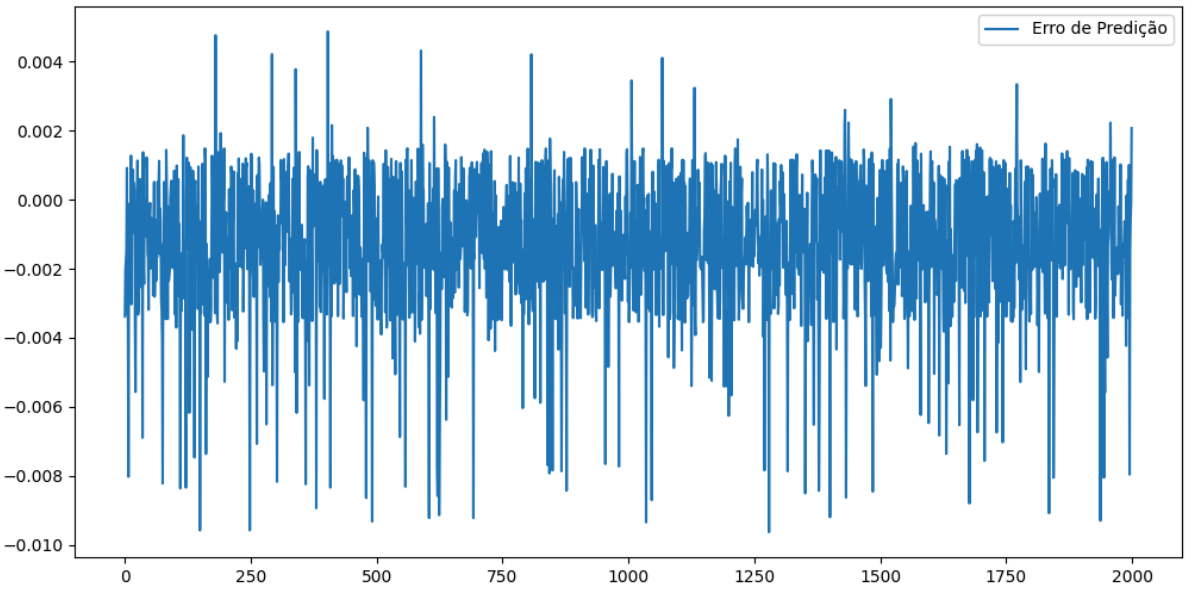


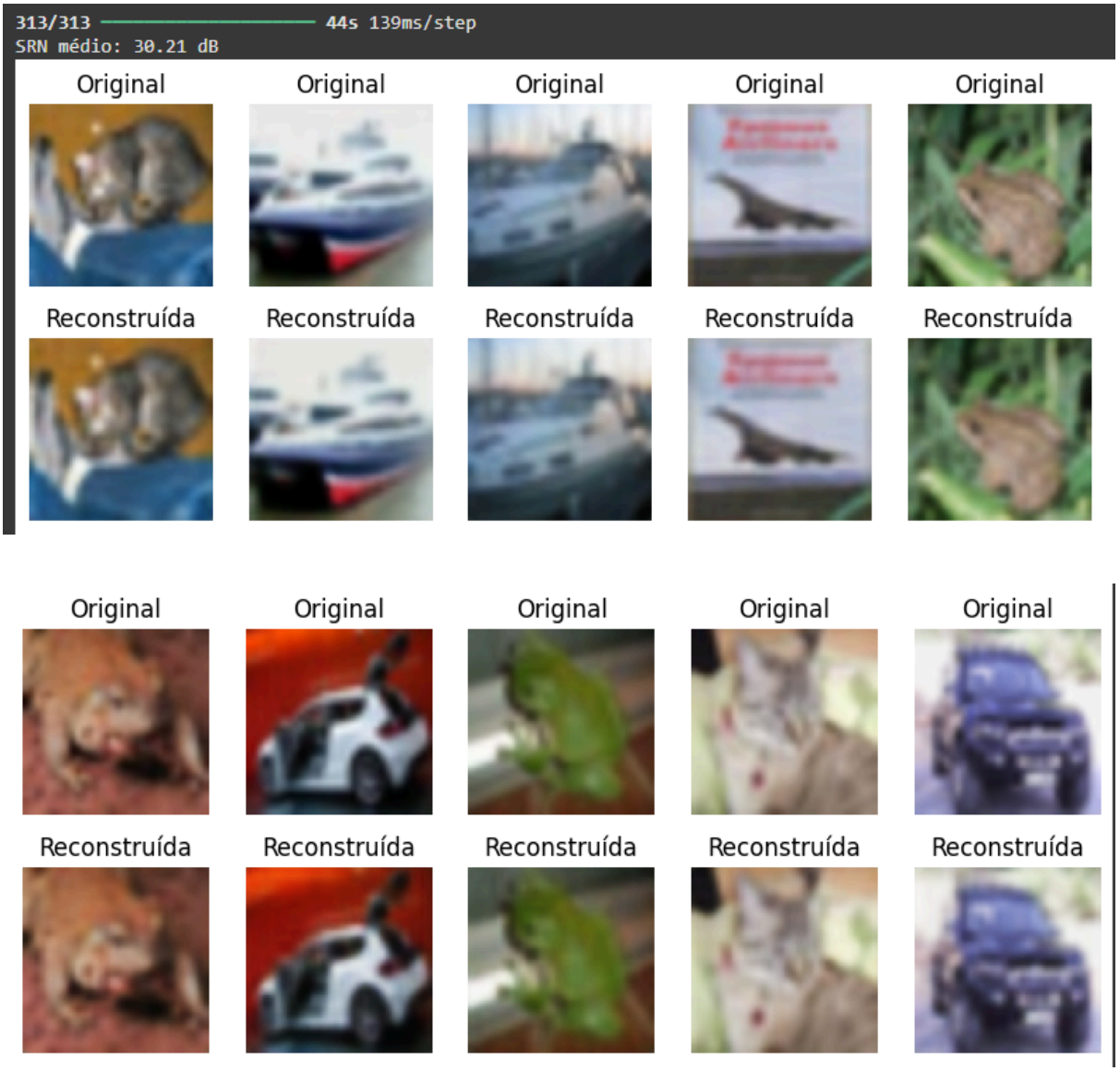
Figura ZZ - Gráfico do erro de predição

5)

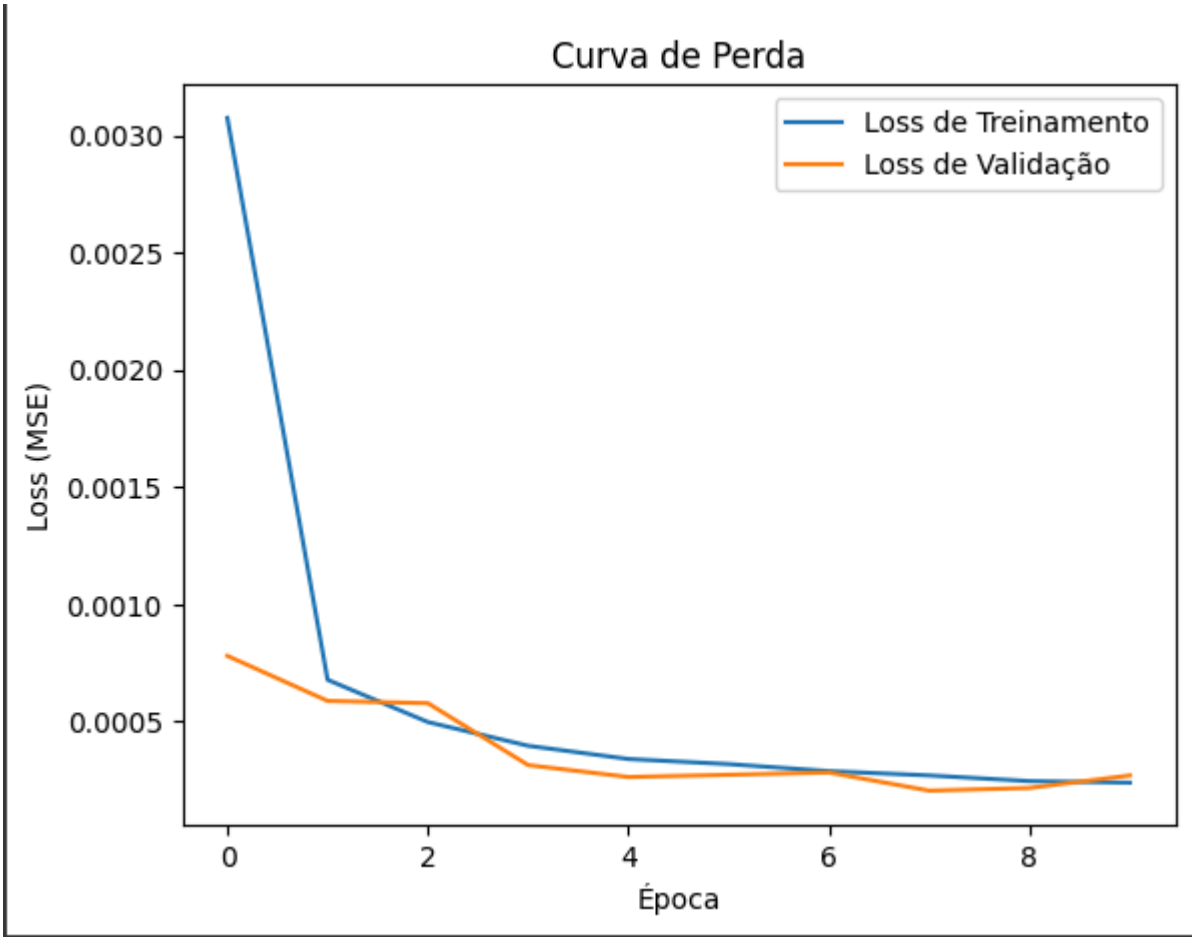
Similar à questão 3, tanto o dataset quanto o modelo utilizado foram importados a partir da biblioteca TensorFlow. Sendo o dataset, CIFAR-10 e o autoencoder um modelo convolucional pré-treinado.

Inicialmente as imagens são carregadas e redimensionadas para 64x64, se necessário. Após isso, o encoder é definido para reduzir as imagens para 32x32 e 16x16, enquanto o decoder é definido para ampliá-las de volta para 32x32 ou 64x64. Sendo assim, com todos os parâmetros definidos, o modelo do autoencoder pode ser treinado, utilizando 10 épocas.

Finalmente, é feita a predição usando o conjunto de teste e o SRN médio é calculado. O resultado é impresso na tela, juntamente com um conjunto de imagens para comparação, originais e reconstruídas, além de uma curva de perda que mostra a evolução do modelo em cada época de treinamento. Sendo possível observar a evolução do modelo, paralelo a diminuição de perdas.



Comparação entre imagens originais e reconstruídas



Curva de perda

6) No caso de problemas de geração de faces baseadas em GANs uma das soluções seria utilizar essas redes para criar imagens de rostos com muito realismo.

As redes generativas adversárias trata-se de um modelo composto por dois modelos que se complementam e aprendem juntos. O primeiro modelo seria um gerador que cria imagens falsas a partir de um vetor de ruídos aleatório ou de outras entradas controladas e segundo é um discriminador, que sua função é avaliar se as imagens recebidas são reais (vindas de um conjunto de treinamento) ou falsas, caso venham do gerador.

Esses dois modelos irão treinar de forma competitiva onde o gerador irá tentar criar imagens que possam enganar o discriminador tentando criar imagens cada vez mais realistas, e o discriminador que tentará melhorar sua capacidade de discernir entre uma imagem real ou falsa. O treinamento terminaria quando a capacidade do discriminador for tão aguçada que o gerador não conseguiria mais enganá-lo.

Um exemplo prático pode ser visto no estudo [1]. Nesse artigo as GANs são utilizadas para criação de rostos em 3D, uma aplicação desafiadora e inovadora na área de visão computacional. Apesar dos avanços das aplicações das GANs na criação de rostos 2D ainda há enormes obstáculos na reconstrução de rostos em 3D. O artigo discute a respeito de questões teóricas relacionadas ao uso das GANs para explorar aplicações em ambientes práticos como Realidade Virtual, jogos, videoconferências e efeitos especiais. Para isso os autores fazem uma série de testes e fornecem avaliações quantitativas e qualitativas, mostrando que os resultados obtidos com as GANs apresentam maior qualidade em relação a métodos tradicionais.

São apresentados no estudo diferentes modelos baseados em probabilidade como Variational Autoencoders (VAEs), que é um tipo modelo generativo baseado em probabilidade e aprendizado não supervisionado, elas utilizam um codificador, que irá reduzir os dados a um espaço latente, e um decodificador, que irá reconstruir os dados originais a partir do ponto do espaço latente. Além das VAEs são utilizados modelos implícitos de GANs. O estudo também faz uma descrição das técnicas que foram utilizadas como modelos morfáveis 3D, CNNs e variantes das GANs. É possível concluir a partir desse artigo que as GANs possuem uma grande versatilidade e potencial para avançar nos estudos de geração de rostos em 3D.

7) As redes recorrentes LSTM são um modelo de rede neural projetadas para resolver problemas de dependência de longo prazo em sequências. Para isso ela utiliza 3 entradas principais (entrada, esquecimento e saída) que irão decidir quais informações devem ser mantidas e quais devem ser descartadas. Normalmente são usadas em Processamento de Linguagem Natural como traduções, geração de texto, reconhecimento de fala e previsão de dados. Sua principal vantagem é a captura de padrões de difícil complexidade em sequências com alta eficácia. No contexto de PLN, por exemplo, elas atuam classificando os textos como positivos ou negativos com aplicações em redes sociais, por exemplo.

A Figura X mostra o modelo da LSTM esquematizada, nesse tipo de rede temos 2 principais componentes:

- Células de memória: responsáveis por manter ou descartar informações por longos períodos de tempo, que são úteis para aprender padrões e sequências de dados que dependem de informações distantes.
- Portões: que controlam a célula de memória. São classificados como portões de entrada, de esquecimento e de saída. Normalmente utilizam funções de ativação (sigmóide) para produzir valores entre 0 e 1, que são usadas para ajustar a quantidade de informação que passa por cada etapa.

Na figura temos x_t que representa a entrada para a LSTM (dados de entrada que podem ser uma palavra ou sequência qualquer), h_{t-1} que representa o estado oculto, ou seja, a saída gerada pela rede na iteração anterior, é nele onde estão as informações sobre o que a rede aprendeu até aquele ponto, e c_{t-1} que representa o estado da célula de memória, que irá armazenar as informações de longo prazo, e por fim uma função de ativação do tipo tangente hiperbólica.

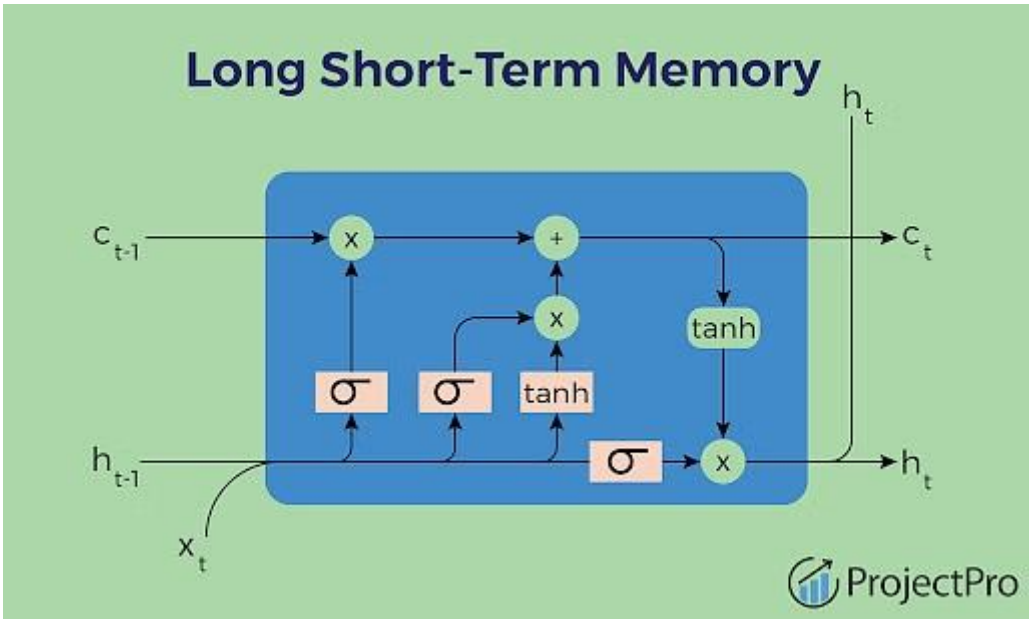


Figura X - LSTM esquematizada

O estudo encontrado referente ao uso de LSTMs em PLNs está descrito em [2]. O artigo "Natural Language Processing for the Analysis Sentiment using a LSTM Model" explora o uso de Redes Neurais Recorrentes com Memória de Longo Prazo (LSTM) para análise de sentimentos em dados textuais de clientes em plataformas de redes sociais. Ele propõe um pipeline robusto para processamento de linguagem natural (NLP), envolvendo

limpeza, tokenização e representação digital de texto, seguido por treinamento de uma LSTM para classificar sentimentos em categorias como "Muito Satisfeito" a "Muito Insatisfeito". O modelo foi treinado usando um dataset de mais de 50 mil observações e alcançou 96% de precisão em previsões, destacando sua eficácia na gestão de relacionamento com clientes.

No estudo, a arquitetura proposta para a rede LSTM inclui entradas representadas por sentenças codificadas com One Hot Encoding, geradas em lotes de 128 com 5% de diversidade. A topologia da rede consiste em uma camada de entrada, cinco camadas ocultas com 256 unidades LSTM e funções de ativação "relu", com normalização de lotes para melhorar o desempenho. A regularização é aplicada por meio de uma camada de exclusão para evitar o overfitting. A camada de saída é composta por cinco neurônios com a função de ativação "softmax" para classificação de cinco classes.

O modelo então é treinado com uma função de custo para medir a diferença entre as previsões da rede e os resultados reais. A função de custo escolhida foi a "cross-entropy". O algoritmo de otimização utilizado para minimizar a função de custo foi o Adam, com uma taxa de aprendizado de 0.001. A precisão foi definida como métrica de avaliação para o modelo.

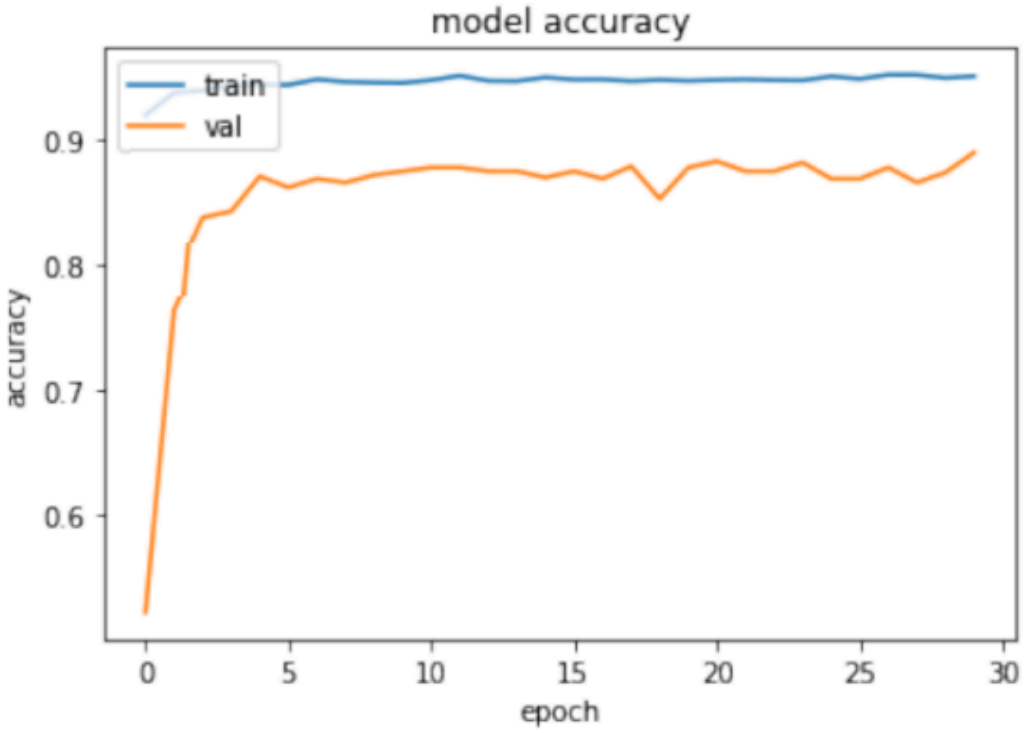


Figura Y - Desenvolvimento da pontuação de treino e validação por época

O estudo relata que a taxa de aprendizado foi de 96% e 89% para validação conforme consta a Figura Y.

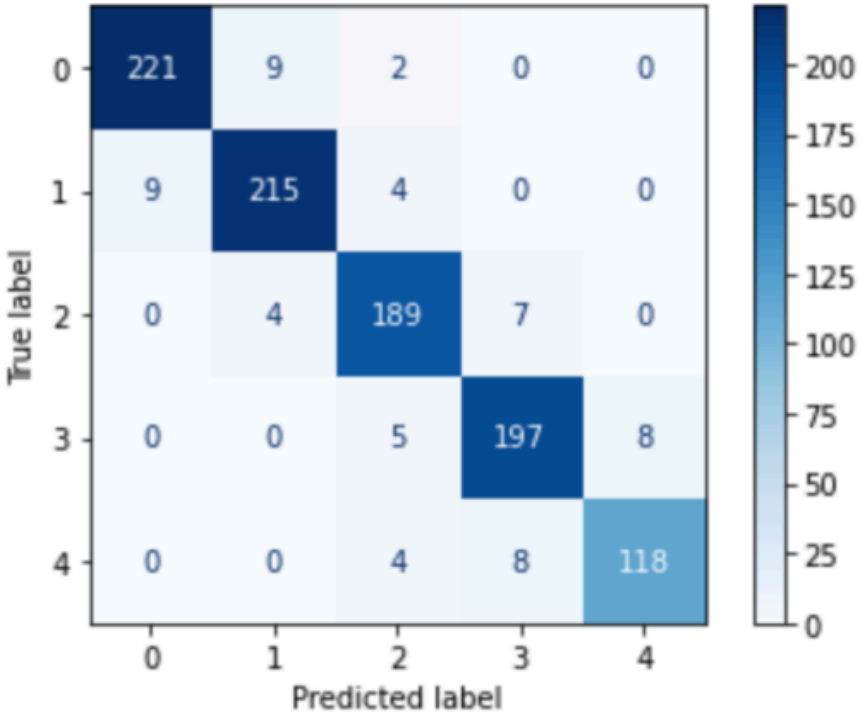


Figura ZZ - Matriz de confusão obtida do conjunto de teste

Ao final do artigo os autores concluem que, ao comparar os parâmetros de precisão da LSTM com os modelos de machine learning pode-se concluir que a taxa de aprendizado foi maior para a LSTM. Pela matriz de confusão mostrada na Figura ZZ a precisão obtida foi de 94%.

REFERÊNCIA

[1] TOSHPULATOV, Mukhiddin; LEE, Wookey; LEE, Suan. Generative adversarial networks (GANs) and their application to 3D face generation. Supported by the Ministry of Education of the Republic of Korea and the National Research Foundation of Korea (NRF − 019S1A5C2A03081234). [s.l.]: [s.n.], 20XX.

[2] BERRAJAA, Achraf. Natural Language Processing for the Analysis Sentiment using a LSTM Model. International Journal of Advanced Computer Science and Applications, v. 13, n. 5, 2022. Disponível em: https://www.ijacsa.thesai.org. Acesso em: 18 dez. 2024.

[3] APA: Dezyre. (n.d.). *Long Short Term Memory (LSTM) Models*. Dezyre. https://dezyre.gumlet.io/images/blog/lstm-model/Long_Short_Term_Memory_(LSTM)_Models.png?w=576&dpr=1.3

Códigos:

Q01)

```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split
from sklearn.neural_network import MLPClassifier
from sklearn.metrics import confusion_matrix, classification_report
# Função para verificar se um ponto está dentro de um semicírculo
def is_in_semicircle(x, y, center, radius, start_angle, end_angle):
angle = np.degrees(np.arctan2(y - center[1], x - center[0]))
angle = (angle + 360) % 360
distance = np.sqrt((x - center[0])**2 + (y - center[1])**2)
return (start_angle <= angle <= end_angle) and (distance <= radius)
# Função para gerar dados aleatórios dentro do quadrado
def generate_data(n_samples):
data = []
labels = []
while len(data) < n_samples:
x, y = np.random.uniform(-1, 1, 2)
if is_in_semicircle(x, y, (1, 0), 1, 0, 360) and is_in_semicircle(x, y, (0, 1), 1, 0, 360):
data.append([x, y])
labels.append(0) # Interseção entre semicirculo (1,0) e (0,1)
elif is_in_semicircle(x, y, (1, 0), 1, 0, 360) and is_in_semicircle(x, y, (0, -1), 1, 0, 360):
data.append([x, y])
labels.append(1) # Interseção entre semicirculo (0,1) e (0,-1)
```

```
elif is_in_semicircle(x, y, (0, -1), 1, 0, 360) and is_in_semicircle(x, y, (-1, 0), 1, 0, 360):

data.append([x, y])

labels.append(2) # Interseção entre semicirculo (0,-1) e (-1,0)

elif is_in_semicircle(x, y, (-1, 0), 1, 0, 360) and is_in_semicircle(x, y, (0, 1), 1, 0, 360):

data.append([x, y])

labels.append(3) # Interseção entre semicirculo (-1,0) e (0,1)

else:

data.append([x, y])

labels.append(4) # Fora das interseções

return np.array(data), np.array(labels)

# Gerando dados para as cinco classes

n_samples = 500

X, y = generate_data(n_samples * 5)

# Dividindo os dados em conjuntos de treinamento e teste

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

# Visualizando os dados

colors = ['red', 'blue', 'green', 'yellow', 'purple']

for i in range(5):

plt.scatter(X_train[y_train == i, 0], X_train[y_train == i, 1], c=colors[i], label=f'Class {i}',
alpha=0.5)

plt.title('Dados de Treinamento')

plt.legend()

plt.show()

# Definindo a arquitetura da rede MLP

mlp = MLPClassifier(hidden_layer_sizes=(100, 50), max_iter=1000, random_state=42)

# Treinando a rede neural

mlp.fit(X_train, y_train)
```

```
# Fazendo previsões no conjunto de teste

y_pred = mlp.predict(X_test)

# Calculando a matriz de confusão

conf_matrix = confusion_matrix(y_test, y_pred)

# Plotando a matriz de confusão

plt.figure(figsize=(10, 7))

sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues', xticklabels=['Classe 0', 'Classe 1', 'Classe 2', 'Classe 3', 'Classe 4'], yticklabels=['Classe 0', 'Classe 1', 'Classe 2', 'Classe 3', 'Classe 4'])

plt.xlabel('Classe Prevista')

plt.ylabel('Classe Verdadeira')

plt.title('Matriz de Confusão')

plt.show()

# Calculando e exibindo o relatório de classificação

class_report = classification_report(y_test, y_pred, target_names=['Classe 0', 'Classe 1', 'Classe 2', 'Classe 3', 'Classe 4'], output_dict=True)

# Convertendo o relatório de classificação em um DataFrame

import pandas as pd

df_class_report = pd.DataFrame(class_report).transpose()

# Plotando o relatório de classificação

plt.figure(figsize=(10, 7))

sns.heatmap(df_class_report.iloc[:-1, :].T, annot=True, cmap='Blues')

plt.title('Relatório de Classificação')

plt.show()
```

Q02)

```
import numpy as np
from sklearn.model_selection import train_test_split
from keras.models import Sequential
from keras.layers import Dense
import matplotlib.pyplot as plt

# 1. Gerar Conjuntos de Treinamento e Teste

# Definir a função
def func(x1, x2):
    return x1**2 + x2**2 + 2*x1*x2 + np.cos(x1 + x2) - 1

# Gerar dados aleatórios
np.random.seed(0)  # Para reprodutibilidade
x1 = np.random.uniform(-5, 5, 1000)
x2 = np.random.uniform(-5, 5, 1000)

# Calcular os valores da função
y = func(x1, x2)

# Dividir os dados em conjuntos de treinamento e teste
X = np.vstack((x1, x2)).T
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

print("Conjunto de treinamento:", X_train.shape, y_train.shape)
```

```
print("Conjunto de teste:", X_test.shape, y_test.shape)

# 2. Definir a Arquitetura da Rede Neural

# Definir a arquitetura da rede neural
model = Sequential()
model.add(Dense(10, input_dim=2, activation='relu'))  # Camada oculta com 10 neurônios
model.add(Dense(1, activation='linear'))  # Camada de saída

# Compilar o modelo
model.compile(optimizer='adam', loss='mean_squared_error')

# 3. Treinamento da Rede Neural

# Treinar a rede neural
history = model.fit(X_train, y_train, epochs=100, batch_size=10, validation_split=0.2)

# 4. Avaliação e Validação

# Avaliar a rede neural no conjunto de teste
loss = model.evaluate(X_test, y_test)
print(f'Erro médio quadrado no conjunto de teste: {loss}')

# Gerar gráficos da função custo durante o treinamento
plt.plot(history.history['loss'], label='Treinamento')
plt.plot(history.history['val_loss'], label='Validação')
```

```
plt.xlabel('Épocas')

plt.ylabel('Erro Médio Quadrado')

plt.legend()

plt.show()


# 5. Ajustes Finais


# Ajustar a arquitetura da rede neural (exemplo com duas camadas ocultas)

model = Sequential()

model.add(Dense(20, input_dim=2, activation='relu'))  # Primeira camada oculta com 20 neurônios

model.add(Dense(10, activation='relu'))  # Segunda camada oculta com 10 neurônios

model.add(Dense(1, activation='linear'))  # Camada de saída


# Compilar o modelo

model.compile(optimizer='adam', loss='mean_squared_error')


# Treinar a rede neural novamente

history = model.fit(X_train, y_train, epochs=100, batch_size=10, validation_split=0.2)


# Avaliar a rede neural no conjunto de teste novamente

loss = model.evaluate(X_test, y_test)

print(f'Erro médio quadrado no conjunto de teste: {loss}')


# Gerar gráficos da função custo durante o treinamento novamente

plt.plot(history.history['loss'], label='Treinamento')

plt.plot(history.history['val_loss'], label='Validação')
```

```
plt.xlabel('Épocas')

plt.ylabel('Erro Médio Quadrado')

plt.legend()

plt.show()
```

Q03)

```
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Flatten, Dropout
from tensorflow.keras.applications import ResNet50
from tensorflow.keras.applications.resnet50 import preprocess_input
from tensorflow.keras.utils import to_categorical
from sklearn.metrics import confusion_matrix, classification_report
import seaborn as sns
import matplotlib.pyplot as plt
import numpy as np

# 1. Carregar o dataset CIFAR-10
(x_train, y_train), (x_test, y_test) = tf.keras.datasets.cifar10.load_data()

# Normalizar os dados
x_train = preprocess_input(x_train)
x_test = preprocess_input(x_test)

# Converter os rótulos para one-hot encoding
y_train_onehot = to_categorical(y_train, num_classes=10)
y_test_onehot = to_categorical(y_test, num_classes=10)

# 2. Carregar o modelo ResNet50 pré-treinado
base_model = ResNet50(weights='imagenet', include_top=False, input_shape=(32, 32, 3))
```

```
# Congelar as camadas convolutivas pré-treinadas
base_model.trainable = False

# Criar o modelo
model = Sequential([
    base_model,
    Flatten(),
    Dense(256, activation='relu'),
    Dropout(0.5),
    Dense(10, activation='softmax')  # 10 classes do CIFAR-10
])

# Compilar o modelo
model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])

# 3. Treinar o modelo
history = model.fit(x_train, y_train_onehot, validation_split=0.2, epochs=10, batch_size=64)

# 4. Avaliar o modelo no conjunto de teste
y_pred = model.predict(x_test)
y_pred_classes = np.argmax(y_pred, axis=1)

# 5. Matriz de confusão
conf_matrix = confusion_matrix(y_test, y_pred_classes)

# Plotar a matriz de confusão
```

```
plt.figure(figsize=(10, 8))
sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues', xticklabels=[
    'airplane', 'automobile', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck'
], yticklabels=[
    'airplane', 'automobile', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck'
])
plt.xlabel('Predicted')
plt.ylabel('True')
plt.title('Matriz de Confusão - CIFAR-10')
plt.show()

# 6. Relatório de classificação
print("\nRelatório de Classificação:")
print(classification_report(y_test, y_pred_classes, target_names=[
    'airplane', 'automobile', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck'
]))
```

Q04)

```
import numpy as np
import matplotlib.pyplot as plt
from keras.models import Sequential
from keras.layers import Dense
from sklearn.model_selection import train_test_split

# 1. Gerar Conjunto de Amostras para Treinamento e Teste

# Definir a série temporal
def generate_series(n):
    return np.sqrt(1 + np.sin(n + np.sin(n)**2))

# Gerar dados
n = np.arange(0, 1000, 0.1)
x = generate_series(n)

# Preparar os dados de entrada e saída
def prepare_data(x, timesteps):
    X, y = [], []
    for i in range(len(x) - timesteps):
        X.append(x[i:i+timesteps])
        y.append(x[i+timesteps])
    return np.array(X), np.array(y)

timesteps = 4
```

```
X, y = prepare_data(x, timesteps)

# Dividir os dados em conjuntos de treinamento e teste
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# 2. Definir a Arquitetura da Rede Neural

model = Sequential()
model.add(Dense(50, input_dim=timesteps, activation='relu'))
model.add(Dense(1))

# Compilar o modelo
model.compile(optimizer='adam', loss='mean_squared_error')

# 3. Treinar a Rede Neural

history = model.fit(X_train, y_train, epochs=100, batch_size=32, validation_data=(X_test,
y_test))

# 4. Avaliar o Desempenho

# Predição
y_pred = model.predict(X_test)

# Curva da Série Temporal
plt.figure(figsize=(12, 6))
plt.plot(n, x, label='Série Temporal Original')
```

```
plt.legend()

plt.show()


# Curva de Predição

plt.figure(figsize=(12, 6))

plt.plot(y_test, label='Valores Reais')

plt.plot(y_pred, label='Predição')

plt.legend()

plt.show()


# Curva do Erro de Predição

erro = y_test - y_pred.flatten()

plt.figure(figsize=(12, 6))

plt.plot(erro, label='Erro de Predição')

plt.legend()

plt.show()
```

Q05)

```
import tensorflow as tf
from tensorflow.keras.layers import Conv2D, MaxPooling2D, UpSampling2D, Input
from tensorflow.keras.models import Model
import numpy as np
import matplotlib.pyplot as plt

# 1. Carregar e preparar os dados
from tensorflow.keras.datasets import cifar10

(x_train, _), (x_test, _) = cifar10.load_data()

# Normalizar as imagens
x_train = x_train.astype('float32') / 255.0
x_test = x_test.astype('float32') / 255.0

# Redimensionar as imagens para 64x64 (se necessário)
x_train = tf.image.resize(x_train, (64, 64))
x_test = tf.image.resize(x_test, (64, 64))

# Formato das imagens
input_shape = x_train.shape[1:]

# 2. Definir o Autoencoder
# Encoder
input_img = Input(shape=input_shape)

x = Conv2D(32, (3, 3), activation='relu', padding='same')(input_img)
x = MaxPooling2D((2, 2), padding='same')(x)  # Reduz para 32x32
x = Conv2D(64, (3, 3), activation='relu', padding='same')(x)
x = MaxPooling2D((2, 2), padding='same')(x)  # Reduz para 16x16

# Decoder
x = Conv2D(64, (3, 3), activation='relu', padding='same')(x)
x = UpSampling2D((2, 2))(x)  # Amplia para 32x32
x = Conv2D(32, (3, 3), activation='relu', padding='same')(x)
x = UpSampling2D((2, 2))(x)  # Amplia para 64x64
decoded = Conv2D(3, (3, 3), activation='sigmoid', padding='same')(x)

# Modelo Autoencoder
autoencoder = Model(input_img, decoded)
autoencoder.compile(optimizer='adam', loss='mse')
autoencoder.summary()

# 3. Treinar o Autoencoder
history = autoencoder.fit(
    x_train, x_train,
```

```
    epochs=10,
    batch_size=64,
    validation_data=(x_test, x_test)
)

# Predição no conjunto de teste
decoded_imgs = autoencoder.predict(x_test)

# Cálculo do SRN
def calculate_srn(original, reconstructed):
    Smed = np.mean(np.square(original))
    Emed = np.mean(np.square(original - reconstructed))
    SRN = 10 * np.log10(Smed / Emed)
    return SRN

srn_values = [calculate_srn(x_test[i], decoded_imgs[i]) for i in range(len(x_test))]
average_srn = np.mean(srn_values)
print(f"SRN médio: {average_srn:.2f} dB")

# Visualizar algumas imagens
n = 10
plt.figure(figsize=(20, 4))
for i in range(n):
    # Original
    ax = plt.subplot(2, n, i + 1)
    plt.imshow(x_test[i])
    plt.title("Original")
    plt.axis("off")

    # Reconstruída
    ax = plt.subplot(2, n, i + 1 + n)
    plt.imshow(decoded_imgs[i])
    plt.title("Reconstruída")
    plt.axis("off")
plt.show()

# Curva da perda
plt.plot(history.history['loss'], label='Loss de Treinamento')
plt.plot(history.history['val_loss'], label='Loss de Validação')
plt.title("Curva de Perda")
plt.xlabel("Época")
plt.ylabel("Loss (MSE)")
plt.legend()
plt.show()
```