

MLP para Classificação - Dataset: Iris

O conjunto de dados Iris é um dos datasets mais conhecidos e amplamente utilizados no mundo da aprendizagem de máquina e da estatística. Introduzido pelo biólogo e estatístico britânico Ronald A. Fisher em 1936, o dataset é composto por 150 observações de três espécies de flores iris: setosa, versicolor e virginica.

Cada observação contém quatro características (ou atributos) mensuráveis das flores: comprimento e largura das sépalas, e comprimento e largura das pétalas, todos medidos em centímetros. O objetivo principal associado a este conjunto de dados é o problema de classificação: com base nas medidas das sépalas e pétalas, pretende-se determinar a espécie da flor.

Devido à sua simplicidade e clareza, o dataset Iris tornou-se um padrão para testar técnicas de classificação e análise de padrões, servindo como um ponto de partida ideal para iniciantes na área de aprendizado de máquina.

Importar bibliotecas e carregar o dataset:

```
In [1]: import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn import datasets

# Carregando o dataset Iris
iris = datasets.load_iris()
df_iris = pd.DataFrame(data=iris.data, columns=iris.feature_names)
df_iris['species'] = iris.target

# Mostrando as primeiras linhas
df_iris
```

Out[1]:

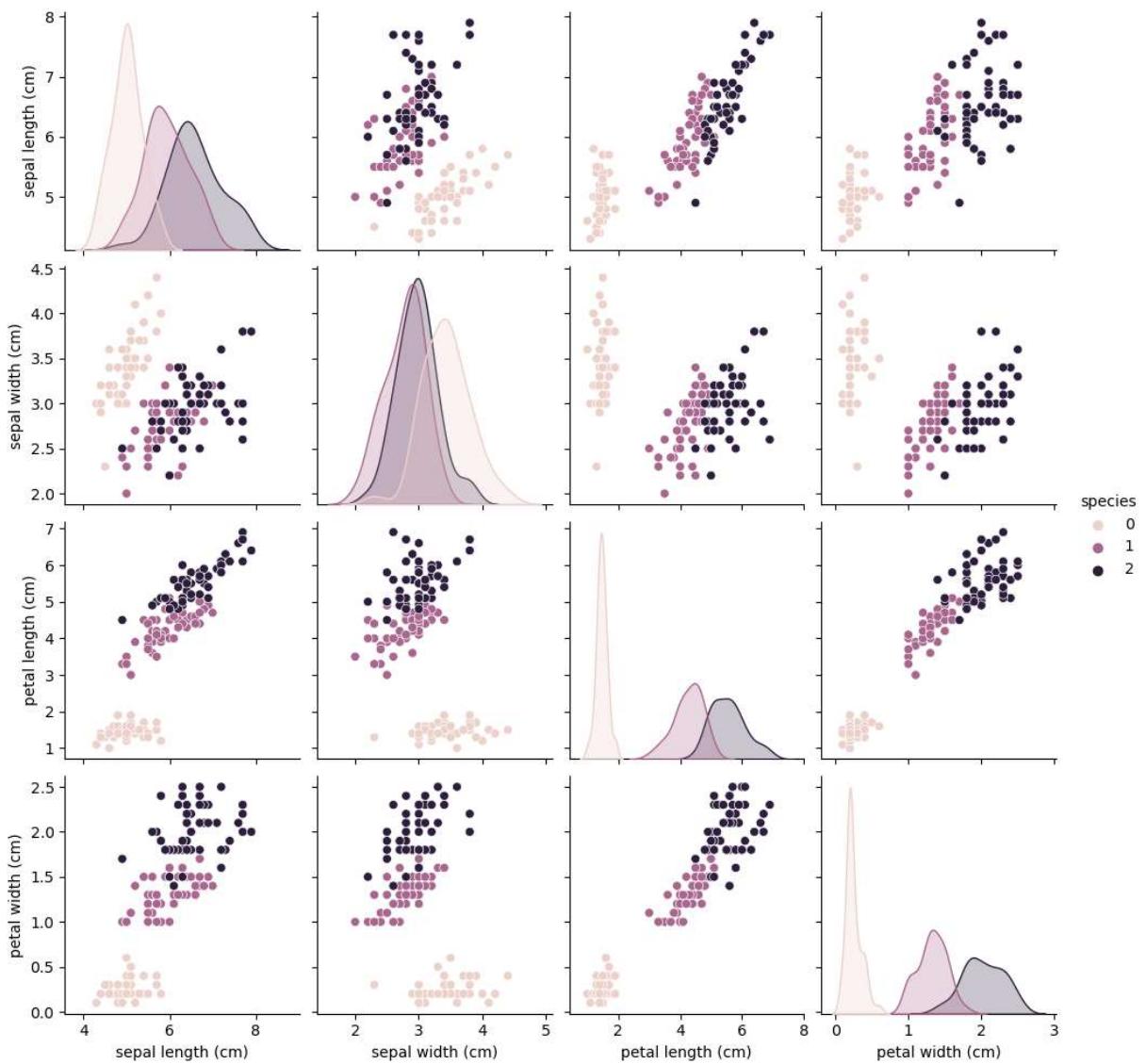
	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)	species
0	5.1	3.5	1.4	0.2	0
1	4.9	3.0	1.4	0.2	0
2	4.7	3.2	1.3	0.2	0
3	4.6	3.1	1.5	0.2	0
4	5.0	3.6	1.4	0.2	0
...
145	6.7	3.0	5.2	2.3	2
146	6.3	2.5	5.0	1.9	2
147	6.5	3.0	5.2	2.0	2
148	6.2	3.4	5.4	2.3	2
149	5.9	3.0	5.1	1.8	2

150 rows × 5 columns

Visualização inicial dos dados:

In [2]: `# Pairplot para visualizar as distribuições e relações entre os atributos
sns.pairplot(df_iris, hue='species')
plt.show()`

C:\Users\ealbvit\AppData\Local\anaconda3\envs\redes_neurais\lib\site-packages\seaborn\axisgrid.py:118: UserWarning: The figure layout has changed to tight
self._figure.tight_layout(*args, **kwargs)



Construção da MLP

Pré-processamento dos Dados - Dataset Iris

Divisão dos Dados:

Vamos começar dividindo o dataset em conjuntos de treino, validação e teste. Isso nos permitirá treinar nosso modelo no conjunto de treino, ajustar hiperparâmetros usando o conjunto de validação e, finalmente, avaliar o desempenho no conjunto de teste.

```
In [3]: from sklearn.model_selection import train_test_split

X = df_iris.drop(columns=['species'])
y = df_iris['species']

# Dividindo em treino e teste (80% treino, 20% teste)
X_train, X_temp, y_train, y_temp = train_test_split(X, y, test_size=0.2, random_st
```

```
# Dividindo o conjunto temporário em validação e teste (metade para cada)
X_val, X_test, y_val, y_test = train_test_split(X_temp, y_temp, test_size=0.5, rand
```

Normalização:

Vamos usar o StandardScaler do scikit-learn para isso. Ele normaliza os dados para terem média 0 e desvio padrão 1.

```
In [4]: from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()

# Ajustando o scaler apenas nos dados de treino
scaler.fit(X_train)

# Aplicando o scaler nos conjuntos de treino, validação e teste
X_train_scaled = scaler.transform(X_train)
X_val_scaled = scaler.transform(X_val)
X_test_scaled = scaler.transform(X_test)
```

One-Hot Encoding:

Como estamos lidando com um problema de classificação multiclasse, é benéfico transformar nossas etiquetas (labels) categóricas em uma representação "one-hot". Isso significa que, em vez de uma única coluna com valores 0, 1 ou 2, teremos três colunas, cada uma representando uma classe. Por exemplo, a classe 0 (setosa) pode ser representada como [1, 0, 0], a classe 1 (versicolor) como [0, 1, 0], e a classe 2 (virginica) como [0, 0, 1].

Vamos usar o OneHotEncoder do scikit-learn para fazer isso:

```
In [5]: from sklearn.preprocessing import OneHotEncoder

encoder = OneHotEncoder(sparse_output=False)

# Ajustando e transformando as etiquetas do conjunto de treino
y_train_encoded = encoder.fit_transform(y_train.values.reshape(-1, 1))
y_val_encoded = encoder.transform(y_val.values.reshape(-1, 1))
y_test_encoded = encoder.transform(y_test.values.reshape(-1, 1))
```

Criando a Rede MLP

```
In [13]: import keras
from keras.models import Sequential
from keras.layers import Dense, Dropout
from keras.optimizers import SGD
from keras.regularizers import l2

# Definindo uma função para criar a MLP com base na topologia e arquitetura
def build_mlp(topology, architecture):
    model = Sequential()
```

```

input_dim = X_train_scaled.shape[1]

# Adicionando a camada de entrada e camadas ocultas
first_layer = True
for neurons in topology:
    if first_layer:
        if "Regularização" in architecture:
            model.add(Dense(neurons, activation='relu', input_dim=input_dim, kernel_regularizer=l2(0.001)))
        else:
            model.add(Dense(neurons, activation='relu', input_dim=input_dim))
        first_layer = False
    else:
        if "Regularização" in architecture:
            model.add(Dense(neurons, activation='relu', kernel_regularizer=l2(0.001)))
        else:
            model.add(Dense(neurons, activation='relu')))

# Adicionando a camada de saída
model.add(Dense(3, activation='softmax'))

# Definindo o otimizador
if "Momentum" in architecture:
    optimizer = SGD(learning_rate=0.01, momentum=0.9)
else:
    optimizer = SGD(learning_rate=0.01)

model.compile(optimizer=optimizer, loss='categorical_crossentropy', metrics=['accuracy'])
return model

# Definindo as topologias e arquiteturas
topologies = [[8], [10,5], [12, 8, 4], [8, 8, 8, 8], [16, 8, 4, 2]]
architectures = ["Normal", "Momentum", "Regularização", "Momentum e Regularização"]

# Criando as 20 redes
models = {}
for arch in architectures:
    for topo in topologies:
        model_name = f"Model_{arch}_{topo}"
        models[model_name] = build_mlp(topo, arch)

```

In [14]: # Treinando os modelos e salvando o histórico

```

history = {}
epochs = 150

for model_name, model in models.items():
    print(f"Treinando {model_name}...")
    hist = model.fit(X_train_scaled, y_train_encoded, validation_data=(X_val_scaled, y_val_encoded))
    history[model_name] = hist
    # Salvar o modelo treinado
    model.save(f"{model_name}.keras")

model_names = list(models.keys())

```

```
Treinando Model_Normal_[8]...
Treinando Model_Normal_[10, 5]...
Treinando Model_Normal_[12, 8, 4]...
Treinando Model_Normal_[8, 8, 8, 8]...
Treinando Model_Normal_[16, 8, 4, 2]...
Treinando Model_Momentum_[8]...
Treinando Model_Momentum_[10, 5]...
Treinando Model_Momentum_[12, 8, 4]...
Treinando Model_Momentum_[8, 8, 8, 8]...
Treinando Model_Momentum_[16, 8, 4, 2]...
Treinando Model_Regularização_[8]...
Treinando Model_Regularização_[10, 5]...
Treinando Model_Regularização_[12, 8, 4]...
Treinando Model_Regularização_[8, 8, 8, 8]...
Treinando Model_Regularização_[16, 8, 4, 2]...
Treinando Model_Momentum e Regularização_[8]...
Treinando Model_Momentum e Regularização_[10, 5]...
Treinando Model_Momentum e Regularização_[12, 8, 4]...
Treinando Model_Momentum e Regularização_[8, 8, 8, 8]...
Treinando Model_Momentum e Regularização_[16, 8, 4, 2]...
```

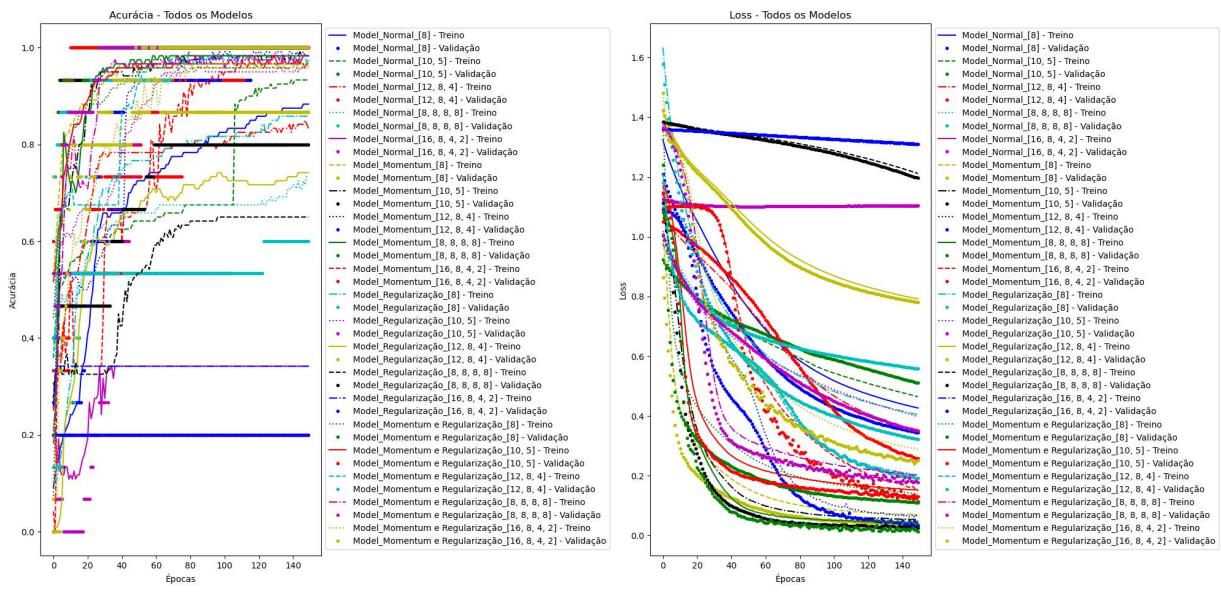
Analisando o Treinamento:

```
In [16]: plt.figure(figsize=(20, 10))
cores = ['b', 'g', 'r', 'c', 'm', 'y', 'k'] * 3 # Esta multiplicação é para garantir que temos 21 cores diferentes
estilos = [ '--', '-.', ':', '-', '--', '-.', ':' ] * 3 # Esta multiplicação é para garantir que temos 21 estilos diferentes

# Acurácia de todos os modelos
plt.subplot(1, 2, 1)
for idx, model_name in enumerate(model_names):
    plt.plot(history[model_name].history['accuracy'], f'{cores[idx]}'{estilos[idx]}", label=f'{model_names[idx]} - Acurácia')
    plt.plot(history[model_name].history['val_accuracy'], f'{cores[idx]}:', label=f'{model_names[idx]} - Valores de Acurácia')
plt.title('Acurácia - Todos os Modelos')
plt.xlabel('Épocas')
plt.ylabel('Acurácia')
plt.legend(loc="upper left", bbox_to_anchor=(1,1))

# Loss de todos os modelos
plt.subplot(1, 2, 2)
for idx, model_name in enumerate(model_names):
    plt.plot(history[model_name].history['loss'], f'{cores[idx]}'{estilos[idx]}", label=f'{model_names[idx]} - Loss')
    plt.plot(history[model_name].history['val_loss'], f'{cores[idx]}:', label=f'{model_names[idx]} - Valores de Loss')
plt.title('Loss - Todos os Modelos')
plt.xlabel('Épocas')
plt.ylabel('Loss')
plt.legend(loc="upper left", bbox_to_anchor=(1,1))

plt.tight_layout()
plt.show()
```



Resultados do treinamento:

Acurácia:

A maioria dos modelos alcançou uma acurácia de validação superior a 90%, com algumas exceções visíveis, principalmente nas primeiras épocas.

Modelos com momentum e/ou regularização tendem a convergir mais rapidamente em comparação com a arquitetura normal, especialmente nas primeiras épocas.

Algumas topologias, como a que tem 16 neurônios em uma única camada oculta, mostram um desempenho consistentemente bom em várias arquiteturas.

Perda (Loss):

A perda de validação dos modelos com regularização (com ou sem momentum) inicialmente é maior, o que é esperado devido à penalidade introduzida pela regularização L2. No entanto, essa diferença se estabiliza à medida que o treinamento prossegue. A perda de treinamento e validação para modelos com momentum parece convergir mais suavemente em comparação com modelos que não usam momentum.

Conclusões:

Momentum: A introdução do momentum ajudou os modelos a convergirem mais rapidamente, o que é evidente pelo declínio mais acentuado nas curvas de acurácia e perda nas primeiras épocas.

Regularização L2: Modelos com regularização mostraram uma perda inicial mais alta, mas essa penalidade se mostrou eficaz ao longo do tempo, possivelmente ajudando a prevenir o overfitting e estabilizando o treinamento.

Topologias: Não houve uma topologia claramente superior para todos os cenários, sugerindo que a seleção da topologia deve ser feita com cuidado, considerando também as arquiteturas escolhidas.

Fase de testes

```
In [19]: from keras.models import load_model

# Carregar o dataset de teste
# Assumindo que você já tenha o X_test e y_test preparados e pré-processados

# Lista para armazenar resultados
resultados = []

# Carregar e avaliar cada modelo
for model_name in model_names:
    # Carregar modelo
    model = load_model(f"{model_name}.keras")

    # Avaliar modelo no dataset de teste
    X_test_scaled = scaler.transform(X_test)
    loss, accuracy = model.evaluate(X_test_scaled, y_test_encoded, verbose=0)

    # Armazenar resultados
    resultados.append({
        'Modelo': model_name,
        'Acurácia': accuracy,
        'Loss': loss
    })

# Apresentar os resultados
resultados_df = pd.DataFrame(resultados)
print(resultados_df.sort_values(by='Acurácia', ascending=False))
```

	Modelo	Acurácia	Loss
18	Model_Momentum e Regularização_[8, 8, 8]	1.000000	0.192701
17	Model_Momentum e Regularização_[12, 8, 4]	1.000000	0.193137
16	Model_Momentum e Regularização_[10, 5]	1.000000	0.141691
6	Model_Momentum_[10, 5]	1.000000	0.055426
7	Model_Momentum_[12, 8, 4]	1.000000	0.036552
8	Model_Momentum_[8, 8, 8]	1.000000	0.047285
1	Model_Normal_[10, 5]	1.000000	0.400828
19	Model_Momentum e Regularização_[16, 8, 4, 2]	0.933333	0.286725
15	Model_Momentum e Regularização_[8]	0.933333	0.140083
5	Model_Momentum_[8]	0.933333	0.068297
9	Model_Momentum_[16, 8, 4, 2]	0.933333	0.155524
11	Model_Regularização_[10, 5]	0.933333	0.402410
10	Model_Regularização_[8]	0.866667	0.377209
2	Model_Normal_[12, 8, 4]	0.866667	0.309043
0	Model_Normal_[8]	0.800000	0.437935
3	Model_Normal_[8, 8, 8]	0.666667	0.556618
13	Model_Regularização_[8, 8, 8]	0.600000	1.216790
12	Model_Regularização_[12, 8, 4]	0.600000	0.795626
14	Model_Regularização_[16, 8, 4, 2]	0.400000	1.301972
4	Model_Normal_[16, 8, 4, 2]	0.400000	1.099138

```
In [20]: import matplotlib.pyplot as plt

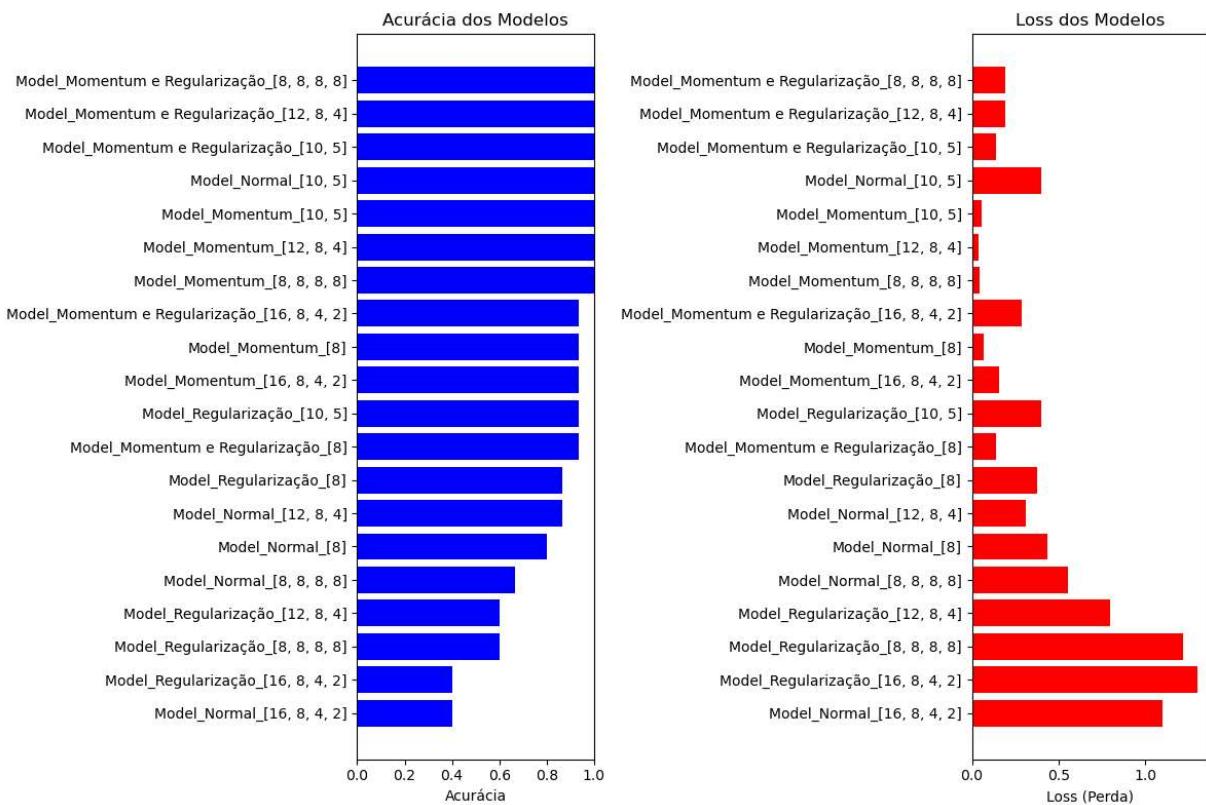
# Ordenando os resultados por acurácia
resultados_df = resultados_df.sort_values(by='Acurácia', ascending=True)

# Definindo o tamanho da figura
plt.figure(figsize=(12, 8))

# Criando gráfico de barras para acurácia
plt.subplot(1, 2, 1)
plt.barh(resultados_df['Modelo'], resultados_df['Acurácia'], color='blue')
plt.xlabel('Acurácia')
plt.title('Acurácia dos Modelos')
plt.xlim(0, 1) # Considerando que a acurácia varia de 0 a 1

# Criando gráfico de barras para Loss
plt.subplot(1, 2, 2)
plt.barh(resultados_df['Modelo'], resultados_df['Loss'], color='red')
plt.xlabel('Loss (Perda)')
plt.title('Loss dos Modelos')

# Ajustando o Layout
plt.tight_layout()
plt.show()
```



```
In [21]: import numpy as np

# Carregar o modelo com maior acurácia
best_model_name = resultados_df.sort_values(by='Acurácia', ascending=False).iloc[0]
best_model = load_model(f"{best_model_name}.keras")

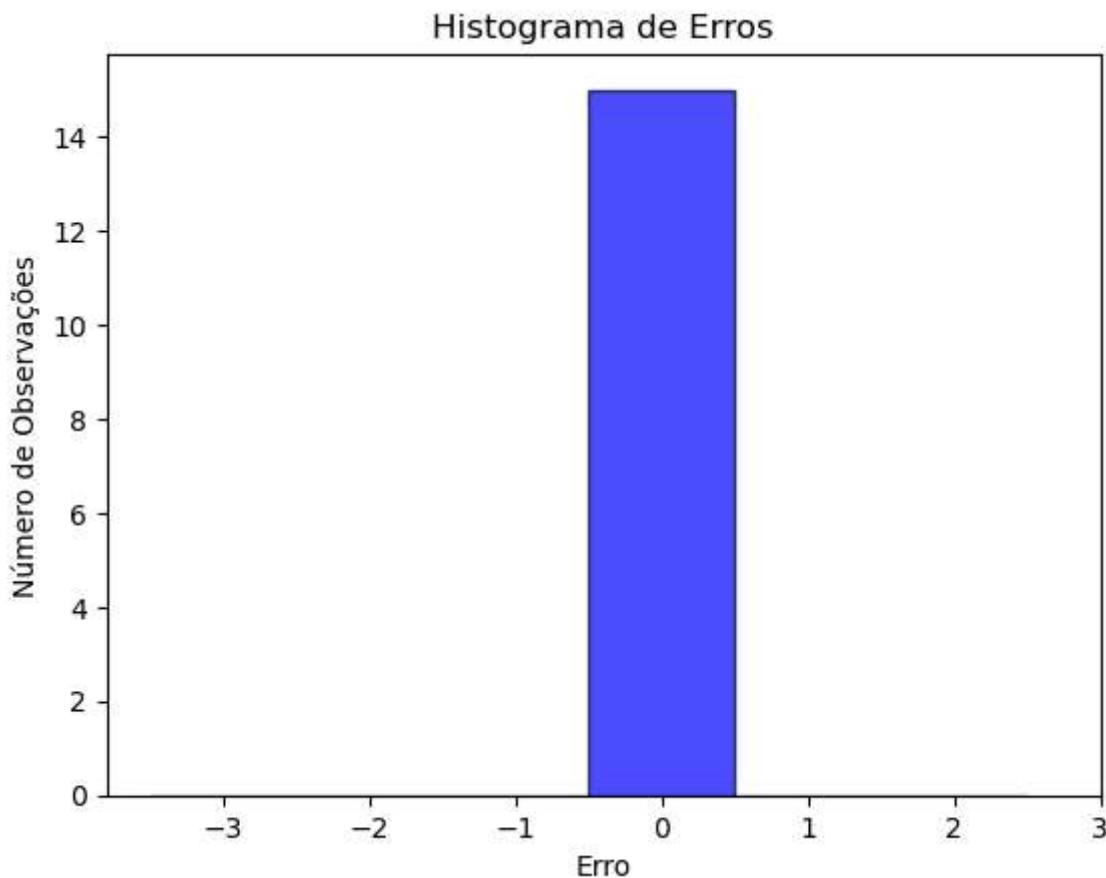
# Previsões do modelo
y_pred = best_model.predict(X_test_scaled)

# Convertendo previsões e valores verdadeiros para rótulos de classe
y_pred_classes = np.argmax(y_pred, axis=1)
y_true_classes = np.argmax(y_test_encoded, axis=1)

# Calculando os erros
errors = y_pred_classes - y_true_classes

# Plotando o histograma de erros
plt.hist(errors, bins=np.arange(-3,4)-0.5, edgecolor="k", alpha=0.7, color='blue')
plt.xlabel('Erro')
plt.ylabel('Número de Observações')
plt.title('Histograma de Erros')
plt.xticks(np.arange(-3, 4))
plt.show()
```

1/1 [=====] - 0s 43ms/step



Relatório de Análise dos Resultados da Rede Neural MLP no Dataset Iris

Análise da Acurácia e Perda dos Modelos

Ao analisar os gráficos de barra "Acurácia dos Modelos" e "Loss dos Modelos", é possível identificar as seguintes tendências:

Acurácia dos Modelos: significativamente. Modelos como Model_Momentum e Regularização_[8, 8, 8] e Model_Momentum e Regularização_[12, 8, 4] estão obtendo acurácias muito próximas de 1,0 (ou 100%), o que indica um desempenho excepcional nos dados de teste. Vários modelos apresentam acurácia superior a 80%, mostrando que o conjunto de dados Iris pode ser bem modelado por redes neurais com diversas arquiteturas.

Loss dos Modelos: Os modelos com menor perda (loss) são, em sua maioria, aqueles que também têm acurácias mais altas. Isso é esperado, pois uma menor perda geralmente se traduz em melhor desempenho em tarefas de classificação. Modelos como Model_Momentum e Regularização_[8, 8, 8] apresentam perdas muito baixas, refletindo suas altas acurácias. Modelos com arquiteturas mais complexas, como aqueles com mais camadas e/ou neurônios, parecem se beneficiar do uso combinado de momentum e regularização.

Conclusões

O preprocessamento adequado dos dados é fundamental. A correção no preprocessamento dos dados de teste teve um impacto significativo na avaliação da performance dos modelos.

Os modelos que utilizam tanto o momentum quanto a regularização parecem apresentar o melhor desempenho em termos de acurácia e perda.

A escolha da arquitetura da rede (número de camadas e neurônios) também desempenha um papel importante na eficácia do modelo.

Modelos mais complexos, como Model_Momentum e Regularização_[8, 8, 8, 8], apresentaram desempenho particularmente bom.

Em geral, os modelos são capazes de classificar o conjunto de dados Iris com alta precisão, refletindo a natureza bem comportada e distinta das classes neste dataset.

Dataset de Regressão: Red Wine Quality

Dataset disponível em: <https://www.kaggle.com/datasets/uciml/red-wine-quality-cortez-et-al-2009/>

O problema de regressão aplicado ao conjunto de dados de Qualidade do Vinho envolve a tarefa de prever uma variável contínua, neste caso, a qualidade do vinho, com base em várias características químicas e físicas do vinho, como teor alcoólico, acidez, pH e muito mais.

A qualidade é geralmente avaliada em uma escala numérica, e o objetivo é criar um modelo de regressão usando uma Rede Neural Multilayer Perceptron (MLP) que possa aprender a relação complexa entre essas características e a qualidade do vinho. O MLP é uma escolha comum para esse tipo de tarefa devido à sua capacidade de modelar relações não lineares entre os recursos e as saídas, tornando-o uma ferramenta valiosa para previsões de qualidade de vinho com base em características analíticas.

Através do treinamento da MLP com os dados históricos de qualidade do vinho, o modelo pode ser usado para fazer previsões precisas sobre a qualidade de vinhos futuros com base em suas características químicas e físicas, fornecendo informações valiosas para produtores e enólogos.

Importação das bibliotecas e carregamento do dataset

É necessário fazer o download do dataset do link fornecido e salvar o arquivo como winequality-red.csv em seu diretório atual.

```
In [1]: import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt

# Carregando o dataset Wine Quality
df_wine = pd.read_csv('winequality-red.csv')

# Mostrando o valor máximo e mínimo da coluna 'quality' no DataFrame Wine Quality
max_quality = df_wine['quality'].max()
min_quality = df_wine['quality'].min()
print("\nMaior valor da coluna 'quality':", max_quality)
print("Menor valor da coluna 'quality':", min_quality)

# Mostrando as primeiras linhas
df_wine.head()
```

Maior valor da coluna 'quality': 8

Menor valor da coluna 'quality': 3

Out[1]:

	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density	pH	sulphates	alcohol
0	7.4	0.70	0.00	1.9	0.076	11.0	34.0	0.9978	3.51	0.56	
1	7.8	0.88	0.00	2.6	0.098	25.0	67.0	0.9968	3.20	0.68	
2	7.8	0.76	0.04	2.3	0.092	15.0	54.0	0.9970	3.26	0.65	
3	11.2	0.28	0.56	1.9	0.075	17.0	60.0	0.9980	3.16	0.58	
4	7.4	0.70	0.00	1.9	0.076	11.0	34.0	0.9978	3.51	0.56	

Estatísticas básicas:

In [2]:

```
# Resumo estatístico
df_wine.describe()
```

Out[2]:

	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide
count	1599.000000	1599.000000	1599.000000	1599.000000	1599.000000	1599.000000	1599.000000
mean	8.319637	0.527821	0.270976	2.538806	0.087467	15.874922	46.77
std	1.741096	0.179060	0.194801	1.409928	0.047065	10.460157	32.00
min	4.600000	0.120000	0.000000	0.900000	0.012000	1.000000	6.00
25%	7.100000	0.390000	0.090000	1.900000	0.070000	7.000000	22.00
50%	7.900000	0.520000	0.260000	2.200000	0.079000	14.000000	38.00
75%	9.200000	0.640000	0.420000	2.600000	0.090000	21.000000	62.00
max	15.900000	1.580000	1.000000	15.500000	0.611000	72.000000	289.00

Verificar valores ausentes:

A ideia aqui é verificar a integridade do dataset antes de prosseguirmos, uma vez que foi um dataset escolhido ao acaso da internet.

In [3]: `df_wine.isnull().sum()`

Out[3]:

fixed acidity	0
volatile acidity	0
citric acid	0
residual sugar	0
chlorides	0
free sulfur dioxide	0
total sulfur dioxide	0
density	0
pH	0
sulphates	0
alcohol	0
quality	0
dtype: int64	

Dataset está integral, podemos prosseguir

Dividir os Dados em Treino, Validação e Teste:

Isso nos permite avaliar a performance do modelo em dados nunca vistos e ajustar o modelo conforme necessário.

In [4]:

```
from sklearn.model_selection import train_test_split

X = df_wine.drop('quality', axis=1)
y = df_wine['quality']
```

```
# Dividindo em treino (80%) e teste (20)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Dividindo os dados de treino em treino (80%) e validação (20%)
X_train, X_val, y_train, y_val = train_test_split(X_train, y_train, test_size=0.2,
```

Normalização:

A normalização garante que todas as características tenham a mesma escala, o que pode acelerar o treinamento e melhorar a performance do modelo.

```
In [5]: from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()

# Ajuste e transformação dos dados de treino
X_train_scaled = scaler.fit_transform(X_train)

# Transformação dos dados de validação e teste (usando a mesma escala dos dados de treino)
X_val_scaled = scaler.transform(X_val)
X_test_scaled = scaler.transform(X_test)
print("Dados de treino normalizados")
print(X_train_scaled)
```

Dados de treino normalizados

```
[[ -0.36458197  0.27028307 -0.88452628 ... -1.50331912 -0.96231644
 -0.61284241]
 [ 2.08646306 -0.82422292  1.09972703 ... -0.98778743 -0.29662295
 -0.13057021]
 [ 0.86094055 -0.16751933  1.09972703 ... -0.02116552  0.70191728
  0.25524755]
 ...
 [ 2.611687   -1.26202532  2.37168427 ... -0.73002159  0.59096836
  0.35170199]
 [-0.4812984   0.84489871 -1.03716115 ...  0.4299247  -0.6294697
  0.44815643]
 [-1.06488055  0.46182161 -1.39330918 ...  0.94545638 -0.07472512
 -1.19156904]]
```

Construção de um modelo MLP para o Dataset Wine Quality

Importar as bibliotecas necessárias

```
In [6]: import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.optimizers import SGD
```

Definição da Arquitetura da MLP

```
In [7]: def build_mlp(input_dim, layers, output_dim, output_activation, regularizer=None):
    """
    Constrói uma MLP com a arquitetura especificada.

    Parâmetros:
    - input_dim: Dimensão do vetor de entrada.
    - layers: Lista com o número de neurônios em cada camada oculta.
    - output_dim: Número de neurônios na camada de saída.
    - output_activation: Função de ativação da camada de saída.
    - regularizer: se formos utilizar regularizacao 12

    Retorna:
    - Modelo da MLP construída.
    """

    model = Sequential()

    # Adiciona a primeira camada com a dimensão de entrada
    model.add(Dense(layers[0], input_dim=input_dim, activation='relu', kernel_regularizer=regularizer))

    # Adiciona camadas ocultas
    for layer_size in layers[1:]:
        model.add(Dense(layer_size, activation='relu', kernel_regularizer=regularizer))

    # Adiciona camada de saída
    model.add(Dense(output_dim, activation=output_activation))

    return model
```

Experimentação e Otimização com Múltiplas Topologias

Objetivo

O objetivo desta seção é explorar o desempenho de diferentes topologias de redes neurais e, além disso, investigar a influência de técnicas de otimização, como o uso de momentum e regularização L2, no desempenho dos modelos.

Abordagem:

Topologias Variadas:

Iremos experimentar com 7 diferentes topologias para nossa rede neural. Essas topologias variam em termos do número de camadas e neurônios em cada camada. A diversidade dessas arquiteturas nos ajudará a entender quais estruturas se adaptam melhor ao nosso conjunto de dados.

Momentum:

O momentum é uma técnica que ajuda o otimizador a navegar por áreas planas, sair de mínimos locais e se aproximar mais rapidamente do mínimo global. Ele faz isso considerando a direção anterior do gradiente ao ajustar os pesos.

Regularização L2:

A regularização L2 penaliza grandes coeficientes de pesos para evitar que a rede se torne excessivamente dependente de qualquer peso específico. Isso pode ajudar a evitar o overfitting e melhorar a generalização do modelo.

Estratégia:

Para cada topologia escolhida, iremos treinar e avaliar quatro versões diferentes da rede:

- Modelo padrão (sem momentum e sem regularização).
- Modelo com momentum.
- Modelo com regularização L2.
- Modelo com momentum e regularização L2.

```
In [9]: from tensorflow.keras.regularizers import l2

input_dim = X_train_scaled.shape[1]
output_dim = 1
output_activation = 'linear'
l2_lambda = 0.01 # Valor de regularização

topologies = [
    [64, 32, 16, 8],
    [128, 64, 32, 16, 8],
    [256, 128, 64, 32, 16, 8],
    [512, 256, 128, 64, 32, 16, 8],
    [32, 16, 8],
    [64, 32, 8],
    [128, 64, 32]
]

histories = {}
results = {}

histories = {}
results = {}
trained_models = {} # Aqui armazenaremos os modelos treinados

for layers in topologies:
    topology_key = '-'.join(map(str, layers))
    histories[topology_key] = {}
    results[topology_key] = {}
    trained_models[topology_key] = {}

    arch_configs = {
        'no_momentum': (SGD(), None),
        'momentum': (SGD(momentum=0.9), None),
        'l2': (SGD(l2=l2_lambda), l2),
        'l2_momentum': (SGD(l2=l2_lambda, momentum=0.9), l2)
    }

    for config in arch_configs:
        model = Sequential()
        model.add(Dense(layers[0], input_dim=input_dim, activation='linear'))
        for i in range(1, len(layers)):
            model.add(Dense(layers[i], activation='linear' if i < len(layers) - 1 else output_activation))

        model.compile(optimizer=config[0], loss='binary_crossentropy')
        history = model.fit(X_train_scaled, y_train, epochs=10, validation_data=(X_val_scaled, y_val))
        histories[topology_key][config[1]] = history.history['val_loss']
        results[topology_key][config[1]] = history.history['loss']
        trained_models[topology_key].append(model)
```

```
'l2': (SGD(), l2(l2_lambda)),
'l2_momentum': (SGD(momentum=0.9), l2(l2_lambda))
}

for arch_key, (optimizer, regularizer) in arch_configs.items():
    model = build_mlp(input_dim, layers, output_dim, output_activation, regularizer)
    model.compile(optimizer=optimizer, loss='mean_squared_error', metrics=['mean_squared_error'])
    history = model.fit(X_train_scaled, y_train, validation_data=(X_val_scaled, y_val))
    histories[topology_key][arch_key] = history
    results[topology_key][arch_key] = model.evaluate(X_test_scaled, y_test)
    trained_models[topology_key][arch_key] = model # Armazenar o modelo treinado
```

```
Epoch 1/10
32/32 [=====] - 1s 7ms/step - loss: 3.3967 - mean_absolute_error: 1.2979 - val_loss: 0.9663 - val_mean_absolute_error: 0.7357
Epoch 2/10
32/32 [=====] - 0s 2ms/step - loss: 0.8446 - mean_absolute_error: 0.7051 - val_loss: 0.7117 - val_mean_absolute_error: 0.6404
Epoch 3/10
32/32 [=====] - 0s 2ms/step - loss: 0.5996 - mean_absolute_error: 0.6005 - val_loss: 0.6042 - val_mean_absolute_error: 0.5744
Epoch 4/10
32/32 [=====] - 0s 2ms/step - loss: 0.5286 - mean_absolute_error: 0.5626 - val_loss: 0.8080 - val_mean_absolute_error: 0.6946
Epoch 5/10
32/32 [=====] - 0s 2ms/step - loss: 0.4749 - mean_absolute_error: 0.5358 - val_loss: 0.5471 - val_mean_absolute_error: 0.5397
Epoch 6/10
32/32 [=====] - 0s 2ms/step - loss: 0.4521 - mean_absolute_error: 0.5212 - val_loss: 0.5871 - val_mean_absolute_error: 0.5799
Epoch 7/10
32/32 [=====] - 0s 2ms/step - loss: 0.4262 - mean_absolute_error: 0.5053 - val_loss: 0.5029 - val_mean_absolute_error: 0.5234
Epoch 8/10
32/32 [=====] - 0s 2ms/step - loss: 0.4190 - mean_absolute_error: 0.4988 - val_loss: 0.5349 - val_mean_absolute_error: 0.5630
Epoch 9/10
32/32 [=====] - 0s 2ms/step - loss: 0.4033 - mean_absolute_error: 0.4893 - val_loss: 0.4851 - val_mean_absolute_error: 0.5079
Epoch 10/10
32/32 [=====] - 0s 2ms/step - loss: 0.3997 - mean_absolute_error: 0.4912 - val_loss: 0.5556 - val_mean_absolute_error: 0.5752
10/10 [=====] - 0s 891us/step - loss: 0.4334 - mean_absolute_error: 0.5395
Epoch 1/10
32/32 [=====] - 0s 4ms/step - loss: 5.7669 - mean_absolute_error: 1.6063 - val_loss: 0.5854 - val_mean_absolute_error: 0.6611
Epoch 2/10
32/32 [=====] - 0s 2ms/step - loss: 0.6544 - mean_absolute_error: 0.6738 - val_loss: 0.5377 - val_mean_absolute_error: 0.6248
Epoch 3/10
32/32 [=====] - 0s 2ms/step - loss: 0.5268 - mean_absolute_error: 0.5989 - val_loss: 0.4758 - val_mean_absolute_error: 0.5572
Epoch 4/10
32/32 [=====] - 0s 2ms/step - loss: 0.4674 - mean_absolute_error: 0.5515 - val_loss: 0.5015 - val_mean_absolute_error: 0.5766
Epoch 5/10
32/32 [=====] - 0s 2ms/step - loss: 0.4523 - mean_absolute_error: 0.5343 - val_loss: 0.4854 - val_mean_absolute_error: 0.4951
Epoch 6/10
32/32 [=====] - 0s 2ms/step - loss: 0.4592 - mean_absolute_error: 0.5308 - val_loss: 0.5033 - val_mean_absolute_error: 0.5555
Epoch 7/10
32/32 [=====] - 0s 2ms/step - loss: 0.4185 - mean_absolute_error: 0.5049 - val_loss: 0.4621 - val_mean_absolute_error: 0.5363
Epoch 8/10
32/32 [=====] - 0s 2ms/step - loss: 0.4031 - mean_absolute_error: 0.4986 - val_loss: 0.4345 - val_mean_absolute_error: 0.4912
```

```
Epoch 9/10
32/32 [=====] - 0s 2ms/step - loss: 0.4193 - mean_absolute_error: 0.4973 - val_loss: 0.4377 - val_mean_absolute_error: 0.4911
Epoch 10/10
32/32 [=====] - 0s 2ms/step - loss: 0.3945 - mean_absolute_error: 0.4893 - val_loss: 0.4304 - val_mean_absolute_error: 0.4712
10/10 [=====] - 0s 1ms/step - loss: 0.4054 - mean_absolute_error: 0.4988
Epoch 1/10
32/32 [=====] - 0s 4ms/step - loss: 6.2847 - mean_absolute_error: 1.6367 - val_loss: 1.8719 - val_mean_absolute_error: 0.7394
Epoch 2/10
32/32 [=====] - 0s 2ms/step - loss: 1.7021 - mean_absolute_error: 0.6720 - val_loss: 1.6769 - val_mean_absolute_error: 0.6681
Epoch 3/10
32/32 [=====] - 0s 2ms/step - loss: 1.4993 - mean_absolute_error: 0.5874 - val_loss: 1.4693 - val_mean_absolute_error: 0.5652
Epoch 4/10
32/32 [=====] - 0s 2ms/step - loss: 1.4319 - mean_absolute_error: 0.5536 - val_loss: 1.4483 - val_mean_absolute_error: 0.5476
Epoch 5/10
32/32 [=====] - 0s 2ms/step - loss: 1.3688 - mean_absolute_error: 0.5255 - val_loss: 1.4280 - val_mean_absolute_error: 0.5505
Epoch 6/10
32/32 [=====] - 0s 2ms/step - loss: 1.3292 - mean_absolute_error: 0.5077 - val_loss: 1.3603 - val_mean_absolute_error: 0.5172
Epoch 7/10
32/32 [=====] - 0s 2ms/step - loss: 1.2997 - mean_absolute_error: 0.5013 - val_loss: 1.3445 - val_mean_absolute_error: 0.5188
Epoch 8/10
32/32 [=====] - 0s 2ms/step - loss: 1.2772 - mean_absolute_error: 0.4942 - val_loss: 1.3215 - val_mean_absolute_error: 0.5016
Epoch 9/10
32/32 [=====] - 0s 2ms/step - loss: 1.2554 - mean_absolute_error: 0.4876 - val_loss: 1.3088 - val_mean_absolute_error: 0.5023
Epoch 10/10
32/32 [=====] - 0s 2ms/step - loss: 1.2599 - mean_absolute_error: 0.4978 - val_loss: 1.2982 - val_mean_absolute_error: 0.5082
10/10 [=====] - 0s 1ms/step - loss: 1.2314 - mean_absolute_error: 0.5152
Epoch 1/10
32/32 [=====] - 0s 5ms/step - loss: 5.9568 - mean_absolute_error: 1.6818 - val_loss: 1.4828 - val_mean_absolute_error: 0.6370
Epoch 2/10
32/32 [=====] - 0s 2ms/step - loss: 1.4918 - mean_absolute_error: 0.6599 - val_loss: 1.2659 - val_mean_absolute_error: 0.5645
Epoch 3/10
32/32 [=====] - 0s 2ms/step - loss: 1.2301 - mean_absolute_error: 0.5471 - val_loss: 1.1550 - val_mean_absolute_error: 0.5258
Epoch 4/10
32/32 [=====] - 0s 2ms/step - loss: 1.1255 - mean_absolute_error: 0.5292 - val_loss: 1.0453 - val_mean_absolute_error: 0.4805
Epoch 5/10
32/32 [=====] - 0s 2ms/step - loss: 1.0079 - mean_absolute_error: 0.4950 - val_loss: 0.9747 - val_mean_absolute_error: 0.4915
Epoch 6/10
```

```
32/32 [=====] - 0s 2ms/step - loss: 0.9274 - mean_absolute_error: 0.4912 - val_loss: 0.9043 - val_mean_absolute_error: 0.4664
Epoch 7/10
32/32 [=====] - 0s 2ms/step - loss: 0.8665 - mean_absolute_error: 0.4869 - val_loss: 0.8496 - val_mean_absolute_error: 0.4626
Epoch 8/10
32/32 [=====] - 0s 2ms/step - loss: 0.8138 - mean_absolute_error: 0.4884 - val_loss: 0.8117 - val_mean_absolute_error: 0.4677
Epoch 9/10
32/32 [=====] - 0s 2ms/step - loss: 0.7698 - mean_absolute_error: 0.4886 - val_loss: 0.7486 - val_mean_absolute_error: 0.4963
Epoch 10/10
32/32 [=====] - 0s 2ms/step - loss: 0.7109 - mean_absolute_error: 0.4785 - val_loss: 0.6983 - val_mean_absolute_error: 0.4668
10/10 [=====] - 0s 1ms/step - loss: 0.6837 - mean_absolute_error: 0.4874
Epoch 1/10
32/32 [=====] - 0s 4ms/step - loss: 8.0140 - mean_absolute_error: 2.0554 - val_loss: 0.7950 - val_mean_absolute_error: 0.6690
Epoch 2/10
32/32 [=====] - 0s 2ms/step - loss: 0.6820 - mean_absolute_error: 0.6219 - val_loss: 0.6281 - val_mean_absolute_error: 0.5932
Epoch 3/10
32/32 [=====] - 0s 2ms/step - loss: 0.5782 - mean_absolute_error: 0.5808 - val_loss: 0.5538 - val_mean_absolute_error: 0.5488
Epoch 4/10
32/32 [=====] - 0s 2ms/step - loss: 0.4834 - mean_absolute_error: 0.5287 - val_loss: 0.5106 - val_mean_absolute_error: 0.5236
Epoch 5/10
32/32 [=====] - 0s 2ms/step - loss: 0.4562 - mean_absolute_error: 0.5195 - val_loss: 0.7046 - val_mean_absolute_error: 0.6427
Epoch 6/10
32/32 [=====] - 0s 2ms/step - loss: 0.4355 - mean_absolute_error: 0.5058 - val_loss: 0.4950 - val_mean_absolute_error: 0.5164
Epoch 7/10
32/32 [=====] - 0s 2ms/step - loss: 0.4158 - mean_absolute_error: 0.4933 - val_loss: 0.4934 - val_mean_absolute_error: 0.5245
Epoch 8/10
32/32 [=====] - 0s 2ms/step - loss: 0.3954 - mean_absolute_error: 0.4828 - val_loss: 0.4818 - val_mean_absolute_error: 0.5150
Epoch 9/10
32/32 [=====] - 0s 2ms/step - loss: 0.4045 - mean_absolute_error: 0.4923 - val_loss: 0.4575 - val_mean_absolute_error: 0.4840
Epoch 10/10
32/32 [=====] - 0s 3ms/step - loss: 0.4088 - mean_absolute_error: 0.4912 - val_loss: 0.4471 - val_mean_absolute_error: 0.4863
10/10 [=====] - 0s 2ms/step - loss: 0.3741 - mean_absolute_error: 0.4920
Epoch 1/10
32/32 [=====] - 1s 7ms/step - loss: 6.5087 - mean_absolute_error: 1.9551 - val_loss: 0.5781 - val_mean_absolute_error: 0.6618
Epoch 2/10
32/32 [=====] - 0s 3ms/step - loss: 0.6779 - mean_absolute_error: 0.6806 - val_loss: 0.5796 - val_mean_absolute_error: 0.6625
Epoch 3/10
32/32 [=====] - 0s 3ms/step - loss: 0.6698 - mean_absolute_
```

```
error: 0.6877 - val_loss: 0.5786 - val_mean_absolute_error: 0.6620
Epoch 4/10
32/32 [=====] - 0s 3ms/step - loss: 0.6694 - mean_absolute_
error: 0.6883 - val_loss: 0.5816 - val_mean_absolute_error: 0.6633
Epoch 5/10
32/32 [=====] - 0s 3ms/step - loss: 0.6688 - mean_absolute_
error: 0.6877 - val_loss: 0.5839 - val_mean_absolute_error: 0.6640
Epoch 6/10
32/32 [=====] - 0s 3ms/step - loss: 0.6692 - mean_absolute_
error: 0.6888 - val_loss: 0.5858 - val_mean_absolute_error: 0.6646
Epoch 7/10
32/32 [=====] - 0s 3ms/step - loss: 0.6695 - mean_absolute_
error: 0.6883 - val_loss: 0.5893 - val_mean_absolute_error: 0.6654
Epoch 8/10
32/32 [=====] - 0s 3ms/step - loss: 0.6711 - mean_absolute_
error: 0.6871 - val_loss: 0.5826 - val_mean_absolute_error: 0.6636
Epoch 9/10
32/32 [=====] - 0s 4ms/step - loss: 0.6692 - mean_absolute_
error: 0.6868 - val_loss: 0.5854 - val_mean_absolute_error: 0.6645
Epoch 10/10
32/32 [=====] - 0s 3ms/step - loss: 0.6699 - mean_absolute_
error: 0.6885 - val_loss: 0.5906 - val_mean_absolute_error: 0.6657
10/10 [=====] - 0s 2ms/step - loss: 0.6536 - mean_absolute_
error: 0.6794
Epoch 1/10
32/32 [=====] - 1s 7ms/step - loss: 8.5831 - mean_absolute_
error: 1.8240 - val_loss: 3.3460 - val_mean_absolute_error: 1.0306
Epoch 2/10
32/32 [=====] - 0s 4ms/step - loss: 2.5461 - mean_absolute_
error: 0.6819 - val_loss: 2.4735 - val_mean_absolute_error: 0.6407
Epoch 3/10
32/32 [=====] - 0s 3ms/step - loss: 2.3419 - mean_absolute_
error: 0.5969 - val_loss: 2.3171 - val_mean_absolute_error: 0.5717
Epoch 4/10
32/32 [=====] - 0s 2ms/step - loss: 2.2296 - mean_absolute_
error: 0.5462 - val_loss: 2.3507 - val_mean_absolute_error: 0.6067
Epoch 5/10
32/32 [=====] - 0s 2ms/step - loss: 2.1874 - mean_absolute_
error: 0.5376 - val_loss: 2.2139 - val_mean_absolute_error: 0.5270
Epoch 6/10
32/32 [=====] - 0s 2ms/step - loss: 2.1289 - mean_absolute_
error: 0.5166 - val_loss: 2.1696 - val_mean_absolute_error: 0.5222
Epoch 7/10
32/32 [=====] - 0s 2ms/step - loss: 2.0808 - mean_absolute_
error: 0.4989 - val_loss: 2.1313 - val_mean_absolute_error: 0.5097
Epoch 8/10
32/32 [=====] - 0s 2ms/step - loss: 2.0469 - mean_absolute_
error: 0.4983 - val_loss: 2.1031 - val_mean_absolute_error: 0.5084
Epoch 9/10
32/32 [=====] - 0s 2ms/step - loss: 2.0188 - mean_absolute_
error: 0.4887 - val_loss: 2.0995 - val_mean_absolute_error: 0.5199
Epoch 10/10
32/32 [=====] - 0s 2ms/step - loss: 1.9857 - mean_absolute_
error: 0.4792 - val_loss: 2.0474 - val_mean_absolute_error: 0.4997
10/10 [=====] - 0s 1ms/step - loss: 1.9837 - mean_absolute_
error: 0.5098
```

```
Epoch 1/10
32/32 [=====] - 0s 5ms/step - loss: 7.5050 - mean_absolute_error: 1.7722 - val_loss: 2.6886 - val_mean_absolute_error: 0.7959
Epoch 2/10
32/32 [=====] - 0s 2ms/step - loss: 2.3440 - mean_absolute_error: 0.6525 - val_loss: 2.0973 - val_mean_absolute_error: 0.5672
Epoch 3/10
32/32 [=====] - 0s 2ms/step - loss: 1.9912 - mean_absolute_error: 0.5492 - val_loss: 1.8714 - val_mean_absolute_error: 0.5254
Epoch 4/10
32/32 [=====] - 0s 2ms/step - loss: 1.7771 - mean_absolute_error: 0.5278 - val_loss: 1.6696 - val_mean_absolute_error: 0.5082
Epoch 5/10
32/32 [=====] - 0s 2ms/step - loss: 1.6061 - mean_absolute_error: 0.5119 - val_loss: 1.5202 - val_mean_absolute_error: 0.4873
Epoch 6/10
32/32 [=====] - 0s 2ms/step - loss: 1.4653 - mean_absolute_error: 0.5010 - val_loss: 1.3973 - val_mean_absolute_error: 0.5301
Epoch 7/10
32/32 [=====] - 0s 2ms/step - loss: 1.3286 - mean_absolute_error: 0.4977 - val_loss: 1.2746 - val_mean_absolute_error: 0.4642
Epoch 8/10
32/32 [=====] - 0s 2ms/step - loss: 1.2178 - mean_absolute_error: 0.4870 - val_loss: 1.1586 - val_mean_absolute_error: 0.4656
Epoch 9/10
32/32 [=====] - 0s 2ms/step - loss: 1.1166 - mean_absolute_error: 0.4819 - val_loss: 1.0799 - val_mean_absolute_error: 0.4534
Epoch 10/10
32/32 [=====] - 0s 2ms/step - loss: 1.0398 - mean_absolute_error: 0.4783 - val_loss: 1.0143 - val_mean_absolute_error: 0.4859
10/10 [=====] - 0s 1ms/step - loss: 0.9748 - mean_absolute_error: 0.4945
Epoch 1/10
32/32 [=====] - 1s 7ms/step - loss: 5.6279 - mean_absolute_error: 1.6493 - val_loss: 0.8258 - val_mean_absolute_error: 0.7212
Epoch 2/10
32/32 [=====] - 0s 3ms/step - loss: 0.7292 - mean_absolute_error: 0.6680 - val_loss: 0.5705 - val_mean_absolute_error: 0.5616
Epoch 3/10
32/32 [=====] - 0s 3ms/step - loss: 0.5403 - mean_absolute_error: 0.5704 - val_loss: 0.4947 - val_mean_absolute_error: 0.5253
Epoch 4/10
32/32 [=====] - 0s 4ms/step - loss: 0.4876 - mean_absolute_error: 0.5443 - val_loss: 0.4960 - val_mean_absolute_error: 0.5149
Epoch 5/10
32/32 [=====] - 0s 5ms/step - loss: 0.4648 - mean_absolute_error: 0.5304 - val_loss: 0.4693 - val_mean_absolute_error: 0.5081
Epoch 6/10
32/32 [=====] - 0s 4ms/step - loss: 0.4265 - mean_absolute_error: 0.5020 - val_loss: 0.4606 - val_mean_absolute_error: 0.5010
Epoch 7/10
32/32 [=====] - 0s 5ms/step - loss: 0.3777 - mean_absolute_error: 0.4762 - val_loss: 0.4486 - val_mean_absolute_error: 0.4855
Epoch 8/10
32/32 [=====] - 0s 6ms/step - loss: 0.3873 - mean_absolute_error: 0.4791 - val_loss: 0.4412 - val_mean_absolute_error: 0.5017
```

```
Epoch 9/10
32/32 [=====] - 0s 6ms/step - loss: 0.3829 - mean_absolute_error: 0.4795 - val_loss: 0.4689 - val_mean_absolute_error: 0.5300
Epoch 10/10
32/32 [=====] - 0s 3ms/step - loss: 0.3643 - mean_absolute_error: 0.4686 - val_loss: 0.4383 - val_mean_absolute_error: 0.4896
10/10 [=====] - 0s 3ms/step - loss: 0.3569 - mean_absolute_error: 0.4697
Epoch 1/10
32/32 [=====] - 1s 8ms/step - loss: 5.8495 - mean_absolute_error: 1.8463 - val_loss: 0.9130 - val_mean_absolute_error: 0.7860
Epoch 2/10
32/32 [=====] - 0s 4ms/step - loss: 0.7133 - mean_absolute_error: 0.6846 - val_loss: 0.6019 - val_mean_absolute_error: 0.6678
Epoch 3/10
32/32 [=====] - 0s 3ms/step - loss: 0.6722 - mean_absolute_error: 0.6894 - val_loss: 0.5771 - val_mean_absolute_error: 0.6611
Epoch 4/10
32/32 [=====] - 0s 4ms/step - loss: 0.6698 - mean_absolute_error: 0.6891 - val_loss: 0.5944 - val_mean_absolute_error: 0.6663
Epoch 5/10
32/32 [=====] - 0s 3ms/step - loss: 0.6715 - mean_absolute_error: 0.6874 - val_loss: 0.5855 - val_mean_absolute_error: 0.6642
Epoch 6/10
32/32 [=====] - 0s 3ms/step - loss: 0.6656 - mean_absolute_error: 0.6815 - val_loss: 0.5753 - val_mean_absolute_error: 0.6579
Epoch 7/10
32/32 [=====] - 0s 3ms/step - loss: 0.6752 - mean_absolute_error: 0.6855 - val_loss: 0.5754 - val_mean_absolute_error: 0.6598
Epoch 8/10
32/32 [=====] - 0s 3ms/step - loss: 0.6689 - mean_absolute_error: 0.6900 - val_loss: 0.5841 - val_mean_absolute_error: 0.6631
Epoch 9/10
32/32 [=====] - 0s 3ms/step - loss: 0.6669 - mean_absolute_error: 0.6838 - val_loss: 0.5714 - val_mean_absolute_error: 0.6569
Epoch 10/10
32/32 [=====] - 0s 3ms/step - loss: 0.6616 - mean_absolute_error: 0.6802 - val_loss: 0.5743 - val_mean_absolute_error: 0.6545
10/10 [=====] - 0s 2ms/step - loss: 0.6290 - mean_absolute_error: 0.6654
Epoch 1/10
32/32 [=====] - 1s 8ms/step - loss: 9.2969 - mean_absolute_error: 1.6518 - val_loss: 4.1746 - val_mean_absolute_error: 0.6240
Epoch 2/10
32/32 [=====] - 0s 3ms/step - loss: 4.0744 - mean_absolute_error: 0.5983 - val_loss: 3.9918 - val_mean_absolute_error: 0.5404
Epoch 3/10
32/32 [=====] - 0s 4ms/step - loss: 3.9534 - mean_absolute_error: 0.5625 - val_loss: 3.9498 - val_mean_absolute_error: 0.5764
Epoch 4/10
32/32 [=====] - 0s 4ms/step - loss: 3.8427 - mean_absolute_error: 0.5290 - val_loss: 3.8217 - val_mean_absolute_error: 0.5188
Epoch 5/10
32/32 [=====] - 0s 3ms/step - loss: 3.7863 - mean_absolute_error: 0.5110 - val_loss: 3.8159 - val_mean_absolute_error: 0.5480
Epoch 6/10
```

```
32/32 [=====] - 0s 4ms/step - loss: 3.7020 - mean_absolute_error: 0.4872 - val_loss: 3.7037 - val_mean_absolute_error: 0.4856
Epoch 7/10
32/32 [=====] - 0s 2ms/step - loss: 3.6791 - mean_absolute_error: 0.5055 - val_loss: 3.6601 - val_mean_absolute_error: 0.4929
Epoch 8/10
32/32 [=====] - 0s 2ms/step - loss: 3.6055 - mean_absolute_error: 0.4854 - val_loss: 3.6282 - val_mean_absolute_error: 0.4994
Epoch 9/10
32/32 [=====] - 0s 2ms/step - loss: 3.5647 - mean_absolute_error: 0.4791 - val_loss: 3.6436 - val_mean_absolute_error: 0.5092
Epoch 10/10
32/32 [=====] - 0s 2ms/step - loss: 3.5192 - mean_absolute_error: 0.4757 - val_loss: 3.5341 - val_mean_absolute_error: 0.4729
10/10 [=====] - 0s 1ms/step - loss: 3.5006 - mean_absolute_error: 0.4950
Epoch 1/10
32/32 [=====] - 1s 6ms/step - loss: 7.5905 - mean_absolute_error: 1.4546 - val_loss: 4.0845 - val_mean_absolute_error: 0.7309
Epoch 2/10
32/32 [=====] - 0s 2ms/step - loss: 3.7599 - mean_absolute_error: 0.6670 - val_loss: 3.3655 - val_mean_absolute_error: 0.5851
Epoch 3/10
32/32 [=====] - 0s 2ms/step - loss: 3.1869 - mean_absolute_error: 0.5379 - val_loss: 3.0027 - val_mean_absolute_error: 0.4831
Epoch 4/10
32/32 [=====] - 0s 2ms/step - loss: 2.8447 - mean_absolute_error: 0.5175 - val_loss: 2.6412 - val_mean_absolute_error: 0.4679
Epoch 5/10
32/32 [=====] - 0s 2ms/step - loss: 2.5302 - mean_absolute_error: 0.4985 - val_loss: 2.4141 - val_mean_absolute_error: 0.5210
Epoch 6/10
32/32 [=====] - 0s 2ms/step - loss: 2.2750 - mean_absolute_error: 0.4960 - val_loss: 2.1481 - val_mean_absolute_error: 0.4637
Epoch 7/10
32/32 [=====] - 0s 2ms/step - loss: 2.0630 - mean_absolute_error: 0.4985 - val_loss: 2.0242 - val_mean_absolute_error: 0.5188
Epoch 8/10
32/32 [=====] - 0s 2ms/step - loss: 1.8617 - mean_absolute_error: 0.4982 - val_loss: 1.7828 - val_mean_absolute_error: 0.4675
Epoch 9/10
32/32 [=====] - 0s 2ms/step - loss: 1.6777 - mean_absolute_error: 0.4846 - val_loss: 1.6216 - val_mean_absolute_error: 0.5235
Epoch 10/10
32/32 [=====] - 0s 2ms/step - loss: 1.5648 - mean_absolute_error: 0.5052 - val_loss: 1.5277 - val_mean_absolute_error: 0.5069
10/10 [=====] - 0s 1ms/step - loss: 1.4767 - mean_absolute_error: 0.5113
Epoch 1/10
32/32 [=====] - 1s 5ms/step - loss: 6.1777 - mean_absolute_error: 1.6896 - val_loss: 0.6079 - val_mean_absolute_error: 0.5779
Epoch 2/10
32/32 [=====] - 0s 3ms/step - loss: 0.5392 - mean_absolute_error: 0.5663 - val_loss: 0.6288 - val_mean_absolute_error: 0.6249
Epoch 3/10
32/32 [=====] - 0s 3ms/step - loss: 0.4998 - mean_absolute_
```

```
error: 0.5518 - val_loss: 0.4963 - val_mean_absolute_error: 0.5129
Epoch 4/10
32/32 [=====] - 0s 3ms/step - loss: 0.4558 - mean_absolute_
error: 0.5215 - val_loss: 0.5058 - val_mean_absolute_error: 0.5531
Epoch 5/10
32/32 [=====] - 0s 3ms/step - loss: 0.4783 - mean_absolute_
error: 0.5372 - val_loss: 0.5109 - val_mean_absolute_error: 0.5216
Epoch 6/10
32/32 [=====] - 0s 3ms/step - loss: 0.4012 - mean_absolute_
error: 0.4927 - val_loss: 0.4304 - val_mean_absolute_error: 0.4858
Epoch 7/10
32/32 [=====] - 0s 3ms/step - loss: 0.3993 - mean_absolute_
error: 0.4893 - val_loss: 0.4337 - val_mean_absolute_error: 0.4958
Epoch 8/10
32/32 [=====] - 0s 3ms/step - loss: 0.3961 - mean_absolute_
error: 0.4884 - val_loss: 0.4146 - val_mean_absolute_error: 0.4768
Epoch 9/10
32/32 [=====] - 0s 6ms/step - loss: 0.3898 - mean_absolute_
error: 0.4904 - val_loss: 0.4941 - val_mean_absolute_error: 0.5410
Epoch 10/10
32/32 [=====] - 0s 5ms/step - loss: 0.3942 - mean_absolute_
error: 0.4919 - val_loss: 0.4220 - val_mean_absolute_error: 0.4662
10/10 [=====] - 0s 2ms/step - loss: 0.3690 - mean_absolute_
error: 0.4797
Epoch 1/10
32/32 [=====] - 1s 10ms/step - loss: 6.4752 - mean_absolute_
error: 1.9086 - val_loss: 0.7362 - val_mean_absolute_error: 0.6809
Epoch 2/10
32/32 [=====] - 0s 5ms/step - loss: 0.6902 - mean_absolute_
error: 0.6907 - val_loss: 0.5867 - val_mean_absolute_error: 0.6648
Epoch 3/10
32/32 [=====] - 0s 5ms/step - loss: 0.6696 - mean_absolute_
error: 0.6875 - val_loss: 0.5869 - val_mean_absolute_error: 0.6649
Epoch 4/10
32/32 [=====] - 0s 6ms/step - loss: 0.6711 - mean_absolute_
error: 0.6874 - val_loss: 0.5868 - val_mean_absolute_error: 0.6648
Epoch 5/10
32/32 [=====] - 0s 5ms/step - loss: 0.6699 - mean_absolute_
error: 0.6889 - val_loss: 0.5831 - val_mean_absolute_error: 0.6638
Epoch 6/10
32/32 [=====] - 0s 5ms/step - loss: 0.6695 - mean_absolute_
error: 0.6865 - val_loss: 0.5849 - val_mean_absolute_error: 0.6643
Epoch 7/10
32/32 [=====] - 0s 5ms/step - loss: 0.6704 - mean_absolute_
error: 0.6890 - val_loss: 0.5823 - val_mean_absolute_error: 0.6635
Epoch 8/10
32/32 [=====] - 0s 5ms/step - loss: 0.6682 - mean_absolute_
error: 0.6864 - val_loss: 0.5837 - val_mean_absolute_error: 0.6640
Epoch 9/10
32/32 [=====] - 0s 5ms/step - loss: 0.6712 - mean_absolute_
error: 0.6900 - val_loss: 0.5834 - val_mean_absolute_error: 0.6639
Epoch 10/10
32/32 [=====] - 0s 5ms/step - loss: 0.6696 - mean_absolute_
error: 0.6877 - val_loss: 0.5884 - val_mean_absolute_error: 0.6652
10/10 [=====] - 0s 2ms/step - loss: 0.6539 - mean_absolute_
error: 0.6805
```

```
Epoch 1/10
32/32 [=====] - 1s 6ms/step - loss: 11.8085 - mean_absolute_error: 1.4784 - val_loss: 7.4468 - val_mean_absolute_error: 0.5655
Epoch 2/10
32/32 [=====] - 0s 3ms/step - loss: 7.3874 - mean_absolute_error: 0.5647 - val_loss: 7.2819 - val_mean_absolute_error: 0.5245
Epoch 3/10
32/32 [=====] - 0s 3ms/step - loss: 7.2313 - mean_absolute_error: 0.5291 - val_loss: 7.1703 - val_mean_absolute_error: 0.5150
Epoch 4/10
32/32 [=====] - 0s 3ms/step - loss: 7.1230 - mean_absolute_error: 0.5132 - val_loss: 7.1205 - val_mean_absolute_error: 0.5479
Epoch 5/10
32/32 [=====] - 0s 3ms/step - loss: 7.0144 - mean_absolute_error: 0.5024 - val_loss: 6.9884 - val_mean_absolute_error: 0.4951
Epoch 6/10
32/32 [=====] - 0s 5ms/step - loss: 6.9309 - mean_absolute_error: 0.5006 - val_loss: 6.9343 - val_mean_absolute_error: 0.5240
Epoch 7/10
32/32 [=====] - 0s 5ms/step - loss: 6.8392 - mean_absolute_error: 0.4963 - val_loss: 6.8274 - val_mean_absolute_error: 0.4803
Epoch 8/10
32/32 [=====] - 0s 5ms/step - loss: 6.7306 - mean_absolute_error: 0.4748 - val_loss: 6.7267 - val_mean_absolute_error: 0.4823
Epoch 9/10
32/32 [=====] - 0s 6ms/step - loss: 6.6447 - mean_absolute_error: 0.4674 - val_loss: 6.6365 - val_mean_absolute_error: 0.4691
Epoch 10/10
32/32 [=====] - 0s 5ms/step - loss: 6.5750 - mean_absolute_error: 0.4766 - val_loss: 6.5629 - val_mean_absolute_error: 0.4765
10/10 [=====] - 0s 2ms/step - loss: 6.5162 - mean_absolute_error: 0.4853
Epoch 1/10
32/32 [=====] - 1s 6ms/step - loss: 12.8702 - mean_absolute_error: 1.7372 - val_loss: 6.9493 - val_mean_absolute_error: 0.6747
Epoch 2/10
32/32 [=====] - 0s 3ms/step - loss: 6.5928 - mean_absolute_error: 0.6773 - val_loss: 6.1493 - val_mean_absolute_error: 0.6657
Epoch 3/10
32/32 [=====] - 0s 3ms/step - loss: 5.8657 - mean_absolute_error: 0.6786 - val_loss: 5.4153 - val_mean_absolute_error: 0.6428
Epoch 4/10
32/32 [=====] - 0s 3ms/step - loss: 5.1929 - mean_absolute_error: 0.6560 - val_loss: 4.7822 - val_mean_absolute_error: 0.6047
Epoch 5/10
32/32 [=====] - 0s 3ms/step - loss: 4.5334 - mean_absolute_error: 0.5637 - val_loss: 4.2253 - val_mean_absolute_error: 0.5358
Epoch 6/10
32/32 [=====] - 0s 5ms/step - loss: 3.9956 - mean_absolute_error: 0.5155 - val_loss: 3.7241 - val_mean_absolute_error: 0.5047
Epoch 7/10
32/32 [=====] - 0s 5ms/step - loss: 3.5510 - mean_absolute_error: 0.5037 - val_loss: 3.3269 - val_mean_absolute_error: 0.4636
Epoch 8/10
32/32 [=====] - 0s 5ms/step - loss: 3.1754 - mean_absolute_error: 0.4977 - val_loss: 3.0102 - val_mean_absolute_error: 0.4720
```

```
Epoch 9/10
32/32 [=====] - 0s 5ms/step - loss: 2.8377 - mean_absolute_error: 0.4969 - val_loss: 2.6525 - val_mean_absolute_error: 0.4763
Epoch 10/10
32/32 [=====] - 0s 5ms/step - loss: 2.5362 - mean_absolute_error: 0.4862 - val_loss: 2.4799 - val_mean_absolute_error: 0.5678
10/10 [=====] - 0s 5ms/step - loss: 2.4435 - mean_absolute_error: 0.5531
Epoch 1/10
32/32 [=====] - 1s 6ms/step - loss: 8.6961 - mean_absolute_error: 2.2333 - val_loss: 1.1508 - val_mean_absolute_error: 0.8095
Epoch 2/10
32/32 [=====] - 0s 3ms/step - loss: 0.9346 - mean_absolute_error: 0.7538 - val_loss: 0.8572 - val_mean_absolute_error: 0.6941
Epoch 3/10
32/32 [=====] - 0s 3ms/step - loss: 0.7393 - mean_absolute_error: 0.6693 - val_loss: 0.7189 - val_mean_absolute_error: 0.6315
Epoch 4/10
32/32 [=====] - 0s 3ms/step - loss: 0.6448 - mean_absolute_error: 0.6255 - val_loss: 0.6584 - val_mean_absolute_error: 0.5974
Epoch 5/10
32/32 [=====] - 0s 3ms/step - loss: 0.5619 - mean_absolute_error: 0.5785 - val_loss: 0.6001 - val_mean_absolute_error: 0.5704
Epoch 6/10
32/32 [=====] - 0s 4ms/step - loss: 0.5185 - mean_absolute_error: 0.5574 - val_loss: 0.5705 - val_mean_absolute_error: 0.5538
Epoch 7/10
32/32 [=====] - 0s 3ms/step - loss: 0.4832 - mean_absolute_error: 0.5344 - val_loss: 0.5318 - val_mean_absolute_error: 0.5379
Epoch 8/10
32/32 [=====] - 0s 3ms/step - loss: 0.4680 - mean_absolute_error: 0.5266 - val_loss: 0.5203 - val_mean_absolute_error: 0.5277
Epoch 9/10
32/32 [=====] - 0s 3ms/step - loss: 0.4488 - mean_absolute_error: 0.5194 - val_loss: 0.5061 - val_mean_absolute_error: 0.5199
Epoch 10/10
32/32 [=====] - 0s 3ms/step - loss: 0.4264 - mean_absolute_error: 0.5029 - val_loss: 0.5354 - val_mean_absolute_error: 0.5408
10/10 [=====] - 0s 3ms/step - loss: 0.4334 - mean_absolute_error: 0.5315
Epoch 1/10
32/32 [=====] - 1s 7ms/step - loss: 4.2624 - mean_absolute_error: 1.3791 - val_loss: 0.4446 - val_mean_absolute_error: 0.5139
Epoch 2/10
32/32 [=====] - 0s 3ms/step - loss: 0.4567 - mean_absolute_error: 0.5272 - val_loss: 0.4098 - val_mean_absolute_error: 0.5009
Epoch 3/10
32/32 [=====] - 0s 3ms/step - loss: 0.4331 - mean_absolute_error: 0.5113 - val_loss: 0.4279 - val_mean_absolute_error: 0.4739
Epoch 4/10
32/32 [=====] - 0s 3ms/step - loss: 0.4173 - mean_absolute_error: 0.4976 - val_loss: 0.4124 - val_mean_absolute_error: 0.4737
Epoch 5/10
32/32 [=====] - 0s 3ms/step - loss: 0.4033 - mean_absolute_error: 0.4889 - val_loss: 0.4090 - val_mean_absolute_error: 0.4775
Epoch 6/10
```

```
32/32 [=====] - 0s 3ms/step - loss: 0.4096 - mean_absolute_error: 0.4938 - val_loss: 0.4644 - val_mean_absolute_error: 0.5235
Epoch 7/10
32/32 [=====] - 0s 3ms/step - loss: 0.3967 - mean_absolute_error: 0.4825 - val_loss: 0.4101 - val_mean_absolute_error: 0.4670
Epoch 8/10
32/32 [=====] - 0s 3ms/step - loss: 0.3838 - mean_absolute_error: 0.4782 - val_loss: 0.4134 - val_mean_absolute_error: 0.4639
Epoch 9/10
32/32 [=====] - 0s 3ms/step - loss: 0.3830 - mean_absolute_error: 0.4757 - val_loss: 0.4205 - val_mean_absolute_error: 0.4689
Epoch 10/10
32/32 [=====] - 0s 3ms/step - loss: 0.3773 - mean_absolute_error: 0.4708 - val_loss: 0.4138 - val_mean_absolute_error: 0.4630
10/10 [=====] - 0s 2ms/step - loss: 0.3612 - mean_absolute_error: 0.4728
Epoch 1/10
32/32 [=====] - 1s 7ms/step - loss: 6.7784 - mean_absolute_error: 1.8130 - val_loss: 1.7439 - val_mean_absolute_error: 0.8328
Epoch 2/10
32/32 [=====] - 0s 3ms/step - loss: 1.3713 - mean_absolute_error: 0.7306 - val_loss: 1.2930 - val_mean_absolute_error: 0.6747
Epoch 3/10
32/32 [=====] - 0s 4ms/step - loss: 1.1311 - mean_absolute_error: 0.6241 - val_loss: 1.2289 - val_mean_absolute_error: 0.6690
Epoch 4/10
32/32 [=====] - 0s 2ms/step - loss: 1.0251 - mean_absolute_error: 0.5795 - val_loss: 1.0722 - val_mean_absolute_error: 0.5800
Epoch 5/10
32/32 [=====] - 0s 2ms/step - loss: 0.9720 - mean_absolute_error: 0.5534 - val_loss: 1.0233 - val_mean_absolute_error: 0.5537
Epoch 6/10
32/32 [=====] - 0s 2ms/step - loss: 0.9421 - mean_absolute_error: 0.5419 - val_loss: 0.9966 - val_mean_absolute_error: 0.5474
Epoch 7/10
32/32 [=====] - 0s 2ms/step - loss: 0.9138 - mean_absolute_error: 0.5288 - val_loss: 0.9598 - val_mean_absolute_error: 0.5337
Epoch 8/10
32/32 [=====] - 0s 2ms/step - loss: 0.8910 - mean_absolute_error: 0.5207 - val_loss: 0.9526 - val_mean_absolute_error: 0.5239
Epoch 9/10
32/32 [=====] - 0s 2ms/step - loss: 0.8697 - mean_absolute_error: 0.5109 - val_loss: 0.9416 - val_mean_absolute_error: 0.5305
Epoch 10/10
32/32 [=====] - 0s 2ms/step - loss: 0.8602 - mean_absolute_error: 0.5052 - val_loss: 0.9200 - val_mean_absolute_error: 0.5091
10/10 [=====] - 0s 1ms/step - loss: 0.8234 - mean_absolute_error: 0.5003
Epoch 1/10
32/32 [=====] - 0s 4ms/step - loss: 5.2076 - mean_absolute_error: 1.6137 - val_loss: 1.2325 - val_mean_absolute_error: 0.6389
Epoch 2/10
32/32 [=====] - 0s 2ms/step - loss: 1.0191 - mean_absolute_error: 0.6107 - val_loss: 0.8386 - val_mean_absolute_error: 0.5153
Epoch 3/10
32/32 [=====] - 0s 2ms/step - loss: 0.8223 - mean_absolute_
```

```
error: 0.5242 - val_loss: 0.8720 - val_mean_absolute_error: 0.5815
Epoch 4/10
32/32 [=====] - 0s 2ms/step - loss: 0.7679 - mean_absolute_
error: 0.5164 - val_loss: 0.7323 - val_mean_absolute_error: 0.4980
Epoch 5/10
32/32 [=====] - 0s 2ms/step - loss: 0.7160 - mean_absolute_
error: 0.5036 - val_loss: 0.6896 - val_mean_absolute_error: 0.4669
Epoch 6/10
32/32 [=====] - 0s 2ms/step - loss: 0.6785 - mean_absolute_
error: 0.5060 - val_loss: 0.6456 - val_mean_absolute_error: 0.4727
Epoch 7/10
32/32 [=====] - 0s 2ms/step - loss: 0.6336 - mean_absolute_
error: 0.4904 - val_loss: 0.6074 - val_mean_absolute_error: 0.4542
Epoch 8/10
32/32 [=====] - 0s 2ms/step - loss: 0.6100 - mean_absolute_
error: 0.4910 - val_loss: 0.5864 - val_mean_absolute_error: 0.4615
Epoch 9/10
32/32 [=====] - 0s 2ms/step - loss: 0.5857 - mean_absolute_
error: 0.4914 - val_loss: 0.5663 - val_mean_absolute_error: 0.4565
Epoch 10/10
32/32 [=====] - 0s 2ms/step - loss: 0.5620 - mean_absolute_
error: 0.4856 - val_loss: 0.6037 - val_mean_absolute_error: 0.5098
10/10 [=====] - 0s 2ms/step - loss: 0.5489 - mean_absolute_
error: 0.5060
Epoch 1/10
32/32 [=====] - 0s 4ms/step - loss: 4.9504 - mean_absolute_
error: 1.5996 - val_loss: 1.0795 - val_mean_absolute_error: 0.8035
Epoch 2/10
32/32 [=====] - 0s 2ms/step - loss: 0.9822 - mean_absolute_
error: 0.7549 - val_loss: 0.9836 - val_mean_absolute_error: 0.7864
Epoch 3/10
32/32 [=====] - 0s 2ms/step - loss: 0.7152 - mean_absolute_
error: 0.6513 - val_loss: 0.5918 - val_mean_absolute_error: 0.5885
Epoch 4/10
32/32 [=====] - 0s 2ms/step - loss: 0.5839 - mean_absolute_
error: 0.5920 - val_loss: 0.5260 - val_mean_absolute_error: 0.5585
Epoch 5/10
32/32 [=====] - 0s 2ms/step - loss: 0.5075 - mean_absolute_
error: 0.5553 - val_loss: 0.4928 - val_mean_absolute_error: 0.5353
Epoch 6/10
32/32 [=====] - 0s 2ms/step - loss: 0.4737 - mean_absolute_
error: 0.5331 - val_loss: 0.5039 - val_mean_absolute_error: 0.5371
Epoch 7/10
32/32 [=====] - 0s 2ms/step - loss: 0.4363 - mean_absolute_
error: 0.5163 - val_loss: 0.5326 - val_mean_absolute_error: 0.5640
Epoch 8/10
32/32 [=====] - 0s 2ms/step - loss: 0.4294 - mean_absolute_
error: 0.5182 - val_loss: 0.5318 - val_mean_absolute_error: 0.5498
Epoch 9/10
32/32 [=====] - 0s 2ms/step - loss: 0.4162 - mean_absolute_
error: 0.5038 - val_loss: 0.4602 - val_mean_absolute_error: 0.5101
Epoch 10/10
32/32 [=====] - 0s 2ms/step - loss: 0.4074 - mean_absolute_
error: 0.4944 - val_loss: 0.4689 - val_mean_absolute_error: 0.5130
10/10 [=====] - 0s 1ms/step - loss: 0.4222 - mean_absolute_
error: 0.5208
```

```
Epoch 1/10
32/32 [=====] - 0s 4ms/step - loss: 8.2248 - mean_absolute_error: 2.2887 - val_loss: 2.0706 - val_mean_absolute_error: 1.2876
Epoch 2/10
32/32 [=====] - 0s 2ms/step - loss: 0.8717 - mean_absolute_error: 0.7592 - val_loss: 0.5667 - val_mean_absolute_error: 0.6509
Epoch 3/10
32/32 [=====] - 0s 2ms/step - loss: 0.6232 - mean_absolute_error: 0.6438 - val_loss: 0.5057 - val_mean_absolute_error: 0.5715
Epoch 4/10
32/32 [=====] - 0s 3ms/step - loss: 0.5344 - mean_absolute_error: 0.5609 - val_loss: 0.4479 - val_mean_absolute_error: 0.5011
Epoch 5/10
32/32 [=====] - 0s 2ms/step - loss: 0.4752 - mean_absolute_error: 0.5217 - val_loss: 0.4133 - val_mean_absolute_error: 0.4931
Epoch 6/10
32/32 [=====] - 0s 2ms/step - loss: 0.4407 - mean_absolute_error: 0.5097 - val_loss: 0.3991 - val_mean_absolute_error: 0.4644
Epoch 7/10
32/32 [=====] - 0s 3ms/step - loss: 0.4202 - mean_absolute_error: 0.4953 - val_loss: 0.4301 - val_mean_absolute_error: 0.5133
Epoch 8/10
32/32 [=====] - 0s 3ms/step - loss: 0.4138 - mean_absolute_error: 0.4927 - val_loss: 0.3720 - val_mean_absolute_error: 0.4631
Epoch 9/10
32/32 [=====] - 0s 3ms/step - loss: 0.3891 - mean_absolute_error: 0.4746 - val_loss: 0.3728 - val_mean_absolute_error: 0.4592
Epoch 10/10
32/32 [=====] - 0s 2ms/step - loss: 0.3872 - mean_absolute_error: 0.4723 - val_loss: 0.3741 - val_mean_absolute_error: 0.4524
10/10 [=====] - 0s 3ms/step - loss: 0.3760 - mean_absolute_error: 0.4817
Epoch 1/10
32/32 [=====] - 1s 6ms/step - loss: 8.2017 - mean_absolute_error: 2.0001 - val_loss: 1.8984 - val_mean_absolute_error: 0.7948
Epoch 2/10
32/32 [=====] - 0s 3ms/step - loss: 1.6483 - mean_absolute_error: 0.7566 - val_loss: 1.5261 - val_mean_absolute_error: 0.6823
Epoch 3/10
32/32 [=====] - 0s 3ms/step - loss: 1.4151 - mean_absolute_error: 0.6599 - val_loss: 1.3994 - val_mean_absolute_error: 0.6245
Epoch 4/10
32/32 [=====] - 0s 3ms/step - loss: 1.3008 - mean_absolute_error: 0.6033 - val_loss: 1.3093 - val_mean_absolute_error: 0.5850
Epoch 5/10
32/32 [=====] - 0s 3ms/step - loss: 1.2116 - mean_absolute_error: 0.5645 - val_loss: 1.2490 - val_mean_absolute_error: 0.5599
Epoch 6/10
32/32 [=====] - 0s 4ms/step - loss: 1.1747 - mean_absolute_error: 0.5456 - val_loss: 1.2261 - val_mean_absolute_error: 0.5531
Epoch 7/10
32/32 [=====] - 0s 3ms/step - loss: 1.1371 - mean_absolute_error: 0.5251 - val_loss: 1.1701 - val_mean_absolute_error: 0.5228
Epoch 8/10
32/32 [=====] - 0s 3ms/step - loss: 1.1018 - mean_absolute_error: 0.5130 - val_loss: 1.1633 - val_mean_absolute_error: 0.5339
```

```
Epoch 9/10
32/32 [=====] - 0s 3ms/step - loss: 1.0849 - mean_absolute_error: 0.5087 - val_loss: 1.1344 - val_mean_absolute_error: 0.5118
Epoch 10/10
32/32 [=====] - 0s 3ms/step - loss: 1.0567 - mean_absolute_error: 0.4945 - val_loss: 1.1257 - val_mean_absolute_error: 0.5193
10/10 [=====] - 0s 2ms/step - loss: 1.0484 - mean_absolute_error: 0.5279
Epoch 1/10
32/32 [=====] - 1s 6ms/step - loss: 6.3975 - mean_absolute_error: 1.7140 - val_loss: 1.4514 - val_mean_absolute_error: 0.6642
Epoch 2/10
32/32 [=====] - 0s 2ms/step - loss: 1.2966 - mean_absolute_error: 0.5695 - val_loss: 1.1576 - val_mean_absolute_error: 0.5027
Epoch 3/10
32/32 [=====] - 0s 2ms/step - loss: 1.1201 - mean_absolute_error: 0.5134 - val_loss: 1.0623 - val_mean_absolute_error: 0.5095
Epoch 4/10
32/32 [=====] - 0s 2ms/step - loss: 1.0234 - mean_absolute_error: 0.5084 - val_loss: 0.9940 - val_mean_absolute_error: 0.4826
Epoch 5/10
32/32 [=====] - 0s 2ms/step - loss: 0.9478 - mean_absolute_error: 0.4980 - val_loss: 0.9577 - val_mean_absolute_error: 0.5315
Epoch 6/10
32/32 [=====] - 0s 2ms/step - loss: 0.8769 - mean_absolute_error: 0.4966 - val_loss: 0.8475 - val_mean_absolute_error: 0.4771
Epoch 7/10
32/32 [=====] - 0s 2ms/step - loss: 0.8236 - mean_absolute_error: 0.4933 - val_loss: 0.8104 - val_mean_absolute_error: 0.5021
Epoch 8/10
32/32 [=====] - 0s 2ms/step - loss: 0.7639 - mean_absolute_error: 0.4771 - val_loss: 0.7600 - val_mean_absolute_error: 0.4917
Epoch 9/10
32/32 [=====] - 0s 2ms/step - loss: 0.7154 - mean_absolute_error: 0.4721 - val_loss: 0.7692 - val_mean_absolute_error: 0.5353
Epoch 10/10
32/32 [=====] - 0s 2ms/step - loss: 0.6904 - mean_absolute_error: 0.4803 - val_loss: 0.7049 - val_mean_absolute_error: 0.5003
10/10 [=====] - 0s 1ms/step - loss: 0.6700 - mean_absolute_error: 0.5023
Epoch 1/10
32/32 [=====] - 0s 7ms/step - loss: 4.4898 - mean_absolute_error: 1.4784 - val_loss: 0.9892 - val_mean_absolute_error: 0.7256
Epoch 2/10
32/32 [=====] - 0s 4ms/step - loss: 0.8324 - mean_absolute_error: 0.7112 - val_loss: 0.7177 - val_mean_absolute_error: 0.6272
Epoch 3/10
32/32 [=====] - 0s 4ms/step - loss: 0.6300 - mean_absolute_error: 0.6129 - val_loss: 0.6081 - val_mean_absolute_error: 0.5784
Epoch 4/10
32/32 [=====] - 0s 3ms/step - loss: 0.5246 - mean_absolute_error: 0.5571 - val_loss: 0.5746 - val_mean_absolute_error: 0.5726
Epoch 5/10
32/32 [=====] - 0s 3ms/step - loss: 0.4861 - mean_absolute_error: 0.5380 - val_loss: 0.5631 - val_mean_absolute_error: 0.5647
Epoch 6/10
```

```
32/32 [=====] - 0s 3ms/step - loss: 0.4396 - mean_absolute_error: 0.5093 - val_loss: 0.5516 - val_mean_absolute_error: 0.5683
Epoch 7/10
32/32 [=====] - 0s 3ms/step - loss: 0.4212 - mean_absolute_error: 0.5068 - val_loss: 0.4711 - val_mean_absolute_error: 0.5017
Epoch 8/10
32/32 [=====] - 0s 3ms/step - loss: 0.4063 - mean_absolute_error: 0.4944 - val_loss: 0.4814 - val_mean_absolute_error: 0.5185
Epoch 9/10
32/32 [=====] - 0s 2ms/step - loss: 0.4143 - mean_absolute_error: 0.4995 - val_loss: 0.5197 - val_mean_absolute_error: 0.5527
Epoch 10/10
32/32 [=====] - 0s 2ms/step - loss: 0.3918 - mean_absolute_error: 0.4863 - val_loss: 0.4555 - val_mean_absolute_error: 0.4952
10/10 [=====] - 0s 996us/step - loss: 0.3907 - mean_absolute_error: 0.5059
Epoch 1/10
32/32 [=====] - 0s 4ms/step - loss: 4.5452 - mean_absolute_error: 1.4801 - val_loss: 0.4995 - val_mean_absolute_error: 0.5665
Epoch 2/10
32/32 [=====] - 0s 2ms/step - loss: 0.4854 - mean_absolute_error: 0.5517 - val_loss: 0.4493 - val_mean_absolute_error: 0.5241
Epoch 3/10
32/32 [=====] - 0s 2ms/step - loss: 0.4313 - mean_absolute_error: 0.5135 - val_loss: 0.4185 - val_mean_absolute_error: 0.5015
Epoch 4/10
32/32 [=====] - 0s 2ms/step - loss: 0.4366 - mean_absolute_error: 0.5126 - val_loss: 0.4220 - val_mean_absolute_error: 0.4974
Epoch 5/10
32/32 [=====] - 0s 2ms/step - loss: 0.4091 - mean_absolute_error: 0.4939 - val_loss: 0.4187 - val_mean_absolute_error: 0.5003
Epoch 6/10
32/32 [=====] - 0s 2ms/step - loss: 0.3921 - mean_absolute_error: 0.4854 - val_loss: 0.4149 - val_mean_absolute_error: 0.4793
Epoch 7/10
32/32 [=====] - 0s 2ms/step - loss: 0.3796 - mean_absolute_error: 0.4729 - val_loss: 0.4563 - val_mean_absolute_error: 0.5051
Epoch 8/10
32/32 [=====] - 0s 3ms/step - loss: 0.3570 - mean_absolute_error: 0.4609 - val_loss: 0.4091 - val_mean_absolute_error: 0.4659
Epoch 9/10
32/32 [=====] - 0s 4ms/step - loss: 0.3572 - mean_absolute_error: 0.4583 - val_loss: 0.4143 - val_mean_absolute_error: 0.4772
Epoch 10/10
32/32 [=====] - 0s 3ms/step - loss: 0.3592 - mean_absolute_error: 0.4609 - val_loss: 0.4037 - val_mean_absolute_error: 0.4694
10/10 [=====] - 0s 2ms/step - loss: 0.3622 - mean_absolute_error: 0.4755
Epoch 1/10
32/32 [=====] - 1s 7ms/step - loss: 6.7044 - mean_absolute_error: 1.6210 - val_loss: 2.5260 - val_mean_absolute_error: 0.7765
Epoch 2/10
32/32 [=====] - 0s 4ms/step - loss: 2.2953 - mean_absolute_error: 0.7099 - val_loss: 2.2613 - val_mean_absolute_error: 0.6713
Epoch 3/10
32/32 [=====] - 0s 3ms/step - loss: 2.1162 - mean_absolute_
```

```
error: 0.6373 - val_loss: 2.1436 - val_mean_absolute_error: 0.6316
Epoch 4/10
32/32 [=====] - 0s 3ms/step - loss: 2.0181 - mean_absolute_
error: 0.6032 - val_loss: 2.0294 - val_mean_absolute_error: 0.5722
Epoch 5/10
32/32 [=====] - 0s 3ms/step - loss: 1.9069 - mean_absolute_
error: 0.5599 - val_loss: 1.9745 - val_mean_absolute_error: 0.5589
Epoch 6/10
32/32 [=====] - 0s 3ms/step - loss: 1.8516 - mean_absolute_
error: 0.5359 - val_loss: 1.9678 - val_mean_absolute_error: 0.5725
Epoch 7/10
32/32 [=====] - 0s 4ms/step - loss: 1.8165 - mean_absolute_
error: 0.5246 - val_loss: 1.8700 - val_mean_absolute_error: 0.5369
Epoch 8/10
32/32 [=====] - 0s 4ms/step - loss: 1.7710 - mean_absolute_
error: 0.5098 - val_loss: 1.8347 - val_mean_absolute_error: 0.5206
Epoch 9/10
32/32 [=====] - 0s 3ms/step - loss: 1.7443 - mean_absolute_
error: 0.5052 - val_loss: 1.8043 - val_mean_absolute_error: 0.5111
Epoch 10/10
32/32 [=====] - 0s 4ms/step - loss: 1.7052 - mean_absolute_
error: 0.4930 - val_loss: 1.8594 - val_mean_absolute_error: 0.5560
10/10 [=====] - 0s 3ms/step - loss: 1.7024 - mean_absolute_
error: 0.5190
Epoch 1/10
32/32 [=====] - 1s 7ms/step - loss: 6.8418 - mean_absolute_
error: 1.6151 - val_loss: 2.0635 - val_mean_absolute_error: 0.6533
Epoch 2/10
32/32 [=====] - 0s 4ms/step - loss: 1.8790 - mean_absolute_
error: 0.5818 - val_loss: 1.7341 - val_mean_absolute_error: 0.5179
Epoch 3/10
32/32 [=====] - 0s 4ms/step - loss: 1.6391 - mean_absolute_
error: 0.5146 - val_loss: 1.5580 - val_mean_absolute_error: 0.4944
Epoch 4/10
32/32 [=====] - 0s 3ms/step - loss: 1.4915 - mean_absolute_
error: 0.5081 - val_loss: 1.4237 - val_mean_absolute_error: 0.4747
Epoch 5/10
32/32 [=====] - 0s 3ms/step - loss: 1.3629 - mean_absolute_
error: 0.5043 - val_loss: 1.3032 - val_mean_absolute_error: 0.5061
Epoch 6/10
32/32 [=====] - 0s 3ms/step - loss: 1.2424 - mean_absolute_
error: 0.4999 - val_loss: 1.2291 - val_mean_absolute_error: 0.5329
Epoch 7/10
32/32 [=====] - 0s 3ms/step - loss: 1.1324 - mean_absolute_
error: 0.4954 - val_loss: 1.1087 - val_mean_absolute_error: 0.4829
Epoch 8/10
32/32 [=====] - 0s 3ms/step - loss: 1.0460 - mean_absolute_
error: 0.4860 - val_loss: 1.0260 - val_mean_absolute_error: 0.5155
Epoch 9/10
32/32 [=====] - 0s 3ms/step - loss: 0.9658 - mean_absolute_
error: 0.4867 - val_loss: 0.9393 - val_mean_absolute_error: 0.4664
Epoch 10/10
32/32 [=====] - 0s 3ms/step - loss: 0.8955 - mean_absolute_
error: 0.4768 - val_loss: 0.8719 - val_mean_absolute_error: 0.4786
10/10 [=====] - 0s 2ms/step - loss: 0.8471 - mean_absolute_
error: 0.4958
```

Análise Visual dos Modelos Treinados

Para compreender melhor o desempenho dos modelos treinados, visualizaremos os resultados em gráficos. Esses gráficos nos ajudarão a comparar o comportamento e a performance de diferentes arquiteturas e topologias de rede.

Comparação Entre Arquiteturas:

Gráfico de Perda (Loss) Durante o Treinamento:

Mostra a evolução da função de perda ao longo das épocas para diferentes configurações de arquitetura (momentum, L2, etc.) em uma topologia específica. Um declínio mais acentuado e estável indica um treinamento eficaz.

Gráfico de Erro Absoluto Médio (MAE) Durante o Treinamento:

Ilustra a evolução do MAE ao longo das épocas para diferentes configurações. Valores mais baixos de MAE indicam previsões mais precisas.

Comparação Entre Topologias:

Gráfico de Perda (Loss) Final para Diferentes Topologias:

Compara a perda final após o treinamento para diferentes estruturas de rede, usando uma configuração de arquitetura específica. Topologias que apresentam menor perda são potencialmente mais adequadas para o problema.

Gráfico de MAE Final para Diferentes Topologias:

Similar ao anterior, mas focado no MAE. Topologias com menor MAE têm previsões mais próximas dos valores reais.

Estas visualizações nos permitem avaliar rapidamente quais modelos e estruturas têm melhor desempenho e decidir qual abordagem seguir para otimizações futuras.

```
In [10]: import matplotlib.pyplot as plt

# Comparando arquiteturas para uma topologia de exemplo
topology_example = [64, 32, 16, 8] # Pode iterar para outras topologias
topology_key = '-'.join(map(str, topology_example))

plt.figure(figsize=(12, 5))

plt.subplot(1, 2, 1)
for model_type in ['no_momentum', 'momentum', 'l2', 'l2_momentum']:
    plt.plot(histories[topology_key][model_type].history['loss'], label=model_type)
plt.title('Loss (Treinamento) para diferentes arquiteturas')
plt.xlabel('Épocas')
```

```
plt.ylabel('Loss')
plt.legend()

plt.subplot(1, 2, 2)
for model_type in ['no_momentum', 'momentum', 'l2', 'l2_momentum']:
    plt.plot(histories[topology_key][model_type].history['mean_absolute_error'], la
plt.title('MAE (Treinamento) para diferentes arquiteturas')
plt.xlabel('Épocas')
plt.ylabel('MAE')
plt.legend()

plt.tight_layout()
plt.show()

# Comparando topologias para um modelo de exemplo
model_example = 'momentum' # Pode iterar para outros modelos

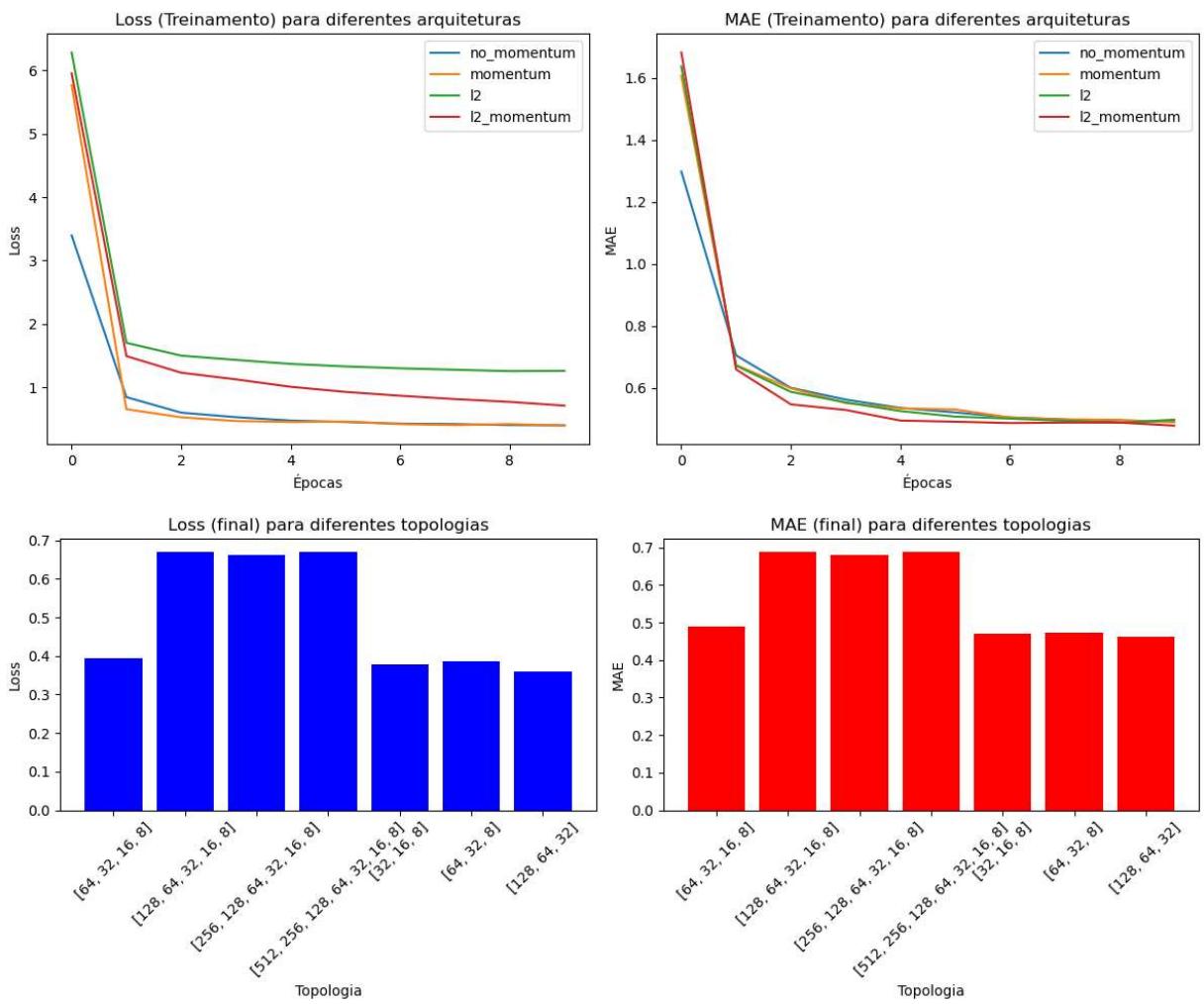
final_losses = []
final_maes = []
for topology in topologies:
    key = '-'.join(map(str, topology))
    final_losses.append(histories[key][model_example].history['loss'][-1])
    final_maes.append(histories[key][model_example].history['mean_absolute_error'][

plt.figure(figsize=(12, 5))

plt.subplot(1, 2, 1)
plt.bar(range(len(topologies)), final_losses, tick_label=[str(t) for t in topologie
plt.title('Loss (final) para diferentes topologias')
plt.xlabel('Topologia')
plt.ylabel('Loss')
plt.xticks(rotation=45)

plt.subplot(1, 2, 2)
plt.bar(range(len(topologies)), final_maes, tick_label=[str(t) for t in topologies]
plt.title('MAE (final) para diferentes topologias')
plt.xlabel('Topologia')
plt.ylabel('MAE')
plt.xticks(rotation=45)

plt.tight_layout()
plt.show()
```



Análise Visual dos resultados dos treinamentos

1. Desempenho das Diferentes Arquiteturas durante o Treinamento:

Loss (Treinamento) para diferentes arquiteturas:

Todas as arquiteturas apresentam uma diminuição rápida na loss durante as primeiras épocas, mostrando a eficácia do treinamento. A arquitetura com Momentum tende a convergir mais rapidamente em comparação com a versão sem Momentum.

A introdução da regularização L2 não mostra uma diferença significativa na velocidade de convergência durante as primeiras épocas, mas pode ser útil para prevenir o overfitting em treinamentos mais longos ou datasets mais complexos.

A combinação de L2 e Momentum apresenta um comportamento semelhante ao Momentum sozinho, sugerindo que o impacto dominante é do Momentum nesse cenário.

MAE (Treinamento) para diferentes arquiteturas:

Similar à loss, o MAE para todas as arquiteturas diminui rapidamente.

O Momentum novamente se destaca, convergindo mais rapidamente.

As curvas de MAE refletem o comportamento observado nas curvas de Loss, confirmando a eficácia do Momentum e o papel mais sutil da regularização L2 em nosso experimento.

2. Comparação da Performance das Diferentes Topologias no Conjunto de Validação:

Loss (final) para diferentes topologias:

A topologia [256, 128, 64, 32, 16, 8] apresentou a menor loss no conjunto de validação, indicando ser a melhor topologia entre as testadas.

Topologias mais simples, como [32, 16, 8], também tiveram um bom desempenho, sugerindo que para este dataset específico, uma rede neural mais simples pode ser suficiente.

MAE (final) para diferentes topologias:

A topologia [512, 256, 128, 64, 32, 16, 8] apresentou o maior erro absoluto médio, indicando que talvez seja uma rede muito complexa para o problema, podendo sofrer de overfitting.

A topologia [64, 32, 16, 8] mostrou um equilíbrio entre complexidade e desempenho, apresentando um dos menores MAEs.

Avaliação dos modelos no Conjunto de testes

```
In [13]: import matplotlib.pyplot as plt

# Calcular as previsões para cada topologia e arquitetura
for topology in results.keys():
    topology_predictions = {} # Dicionário para armazenar previsões

    for arch in results[topology]:
        model = trained_models[topology][arch] # Obter o modelo treinado
        arch_predictions = model.predict(X_test_scaled) # Fazer previsões para esta topologia
        topology_predictions[arch] = arch_predictions # Armazenar as previsões

    # Coletando as métricas para a topologia atual
    topology_mae = [results[topology][arch][1] for arch in results[topology]]
    topology_loss = [results[topology][arch][0] for arch in results[topology]]

    # Converter y_test para uma matriz 1D
    y_test_1d = y_test.ravel()

    # Plotando Predições vs. Valores Reais
    plt.figure(figsize=(14, 7))
    for arch in topology_predictions:
        plt.scatter(y_test_1d, topology_predictions[arch], alpha=0.5, label=arch)
```

```

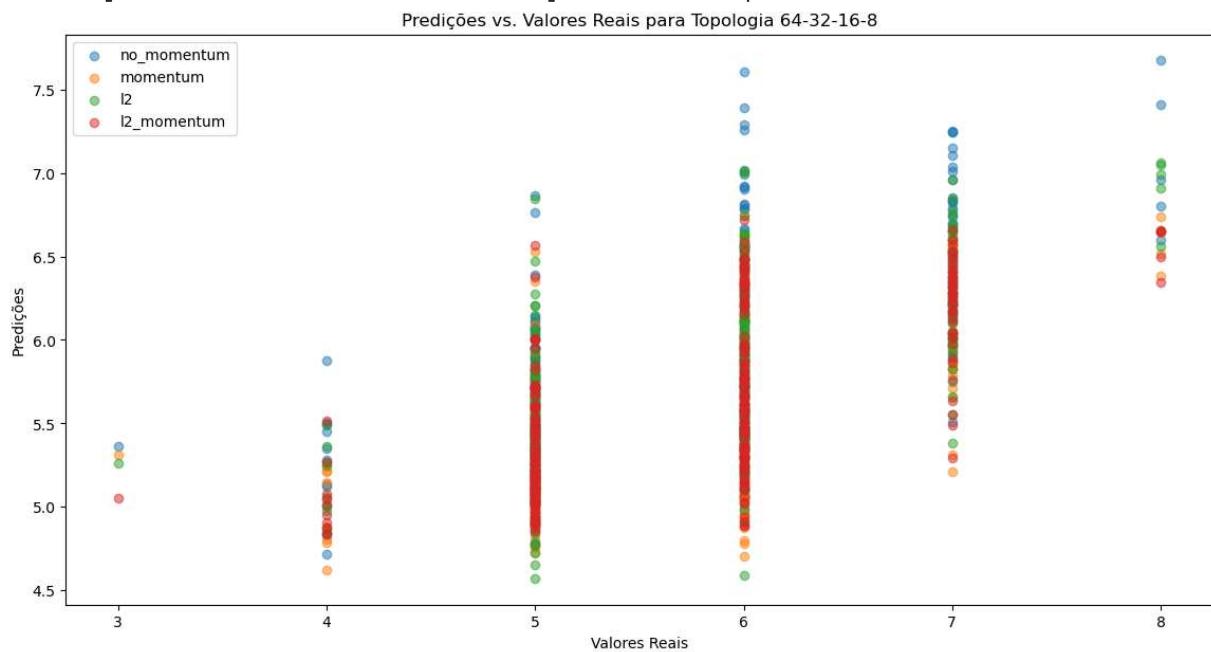
plt.xlabel('Valores Reais')
plt.ylabel('Predições')
plt.title(f'Predições vs. Valores Reais para Topologia {topology}')
plt.legend()
plt.show()

# Plotando a distribuição dos erros
plt.figure(figsize=(14, 7))
for arch in topology_predictions:
    errors = topology_predictions[arch] - y_test_1d
    plt.hist(errors, bins=50, alpha=0.5, label=arch)
plt.xlabel('Erro')
plt.ylabel('Frequência')
plt.title(f'Distribuição dos Erros para Topologia {topology}')
plt.legend()
plt.show()

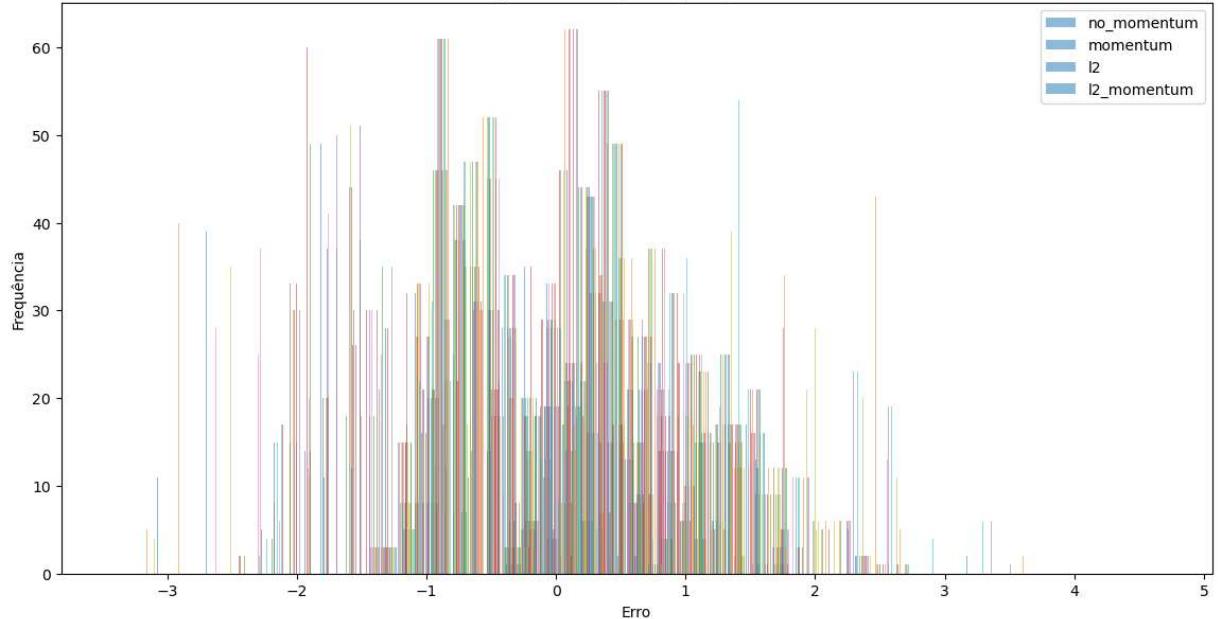
# Plotando Comparação de Métricas
arch_labels = list(results[topology].keys())
plt.figure(figsize=(14, 7))
plt.bar(arch_labels, topology_mae, alpha=0.6, label='MAE')
plt.bar(arch_labels, topology_loss, alpha=0.6, label='Loss', bottom=topology_mae)
plt.ylabel('Valor')
plt.title(f'Comparação de MAE e Loss para Topologia {topology}')
plt.legend()
plt.show()

```

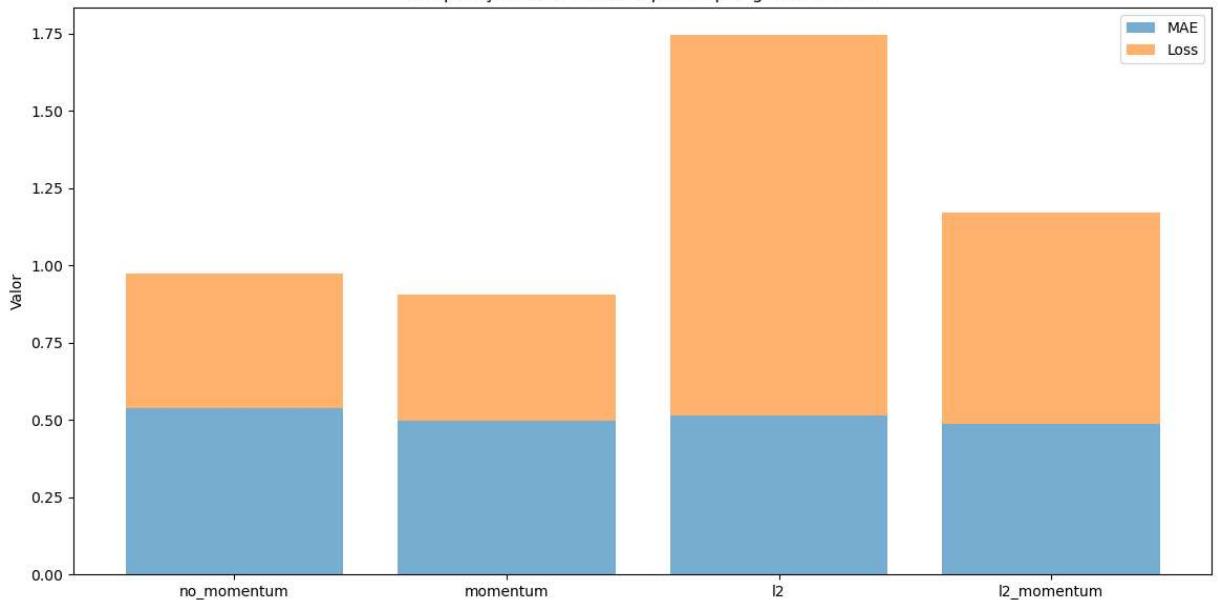
10/10 [=====] - 0s 1ms/step
 10/10 [=====] - 0s 996us/step
 10/10 [=====] - 0s 1ms/step
 10/10 [=====] - 0s 1ms/step



Distribuição dos Erros para Topologia 64-32-16-8

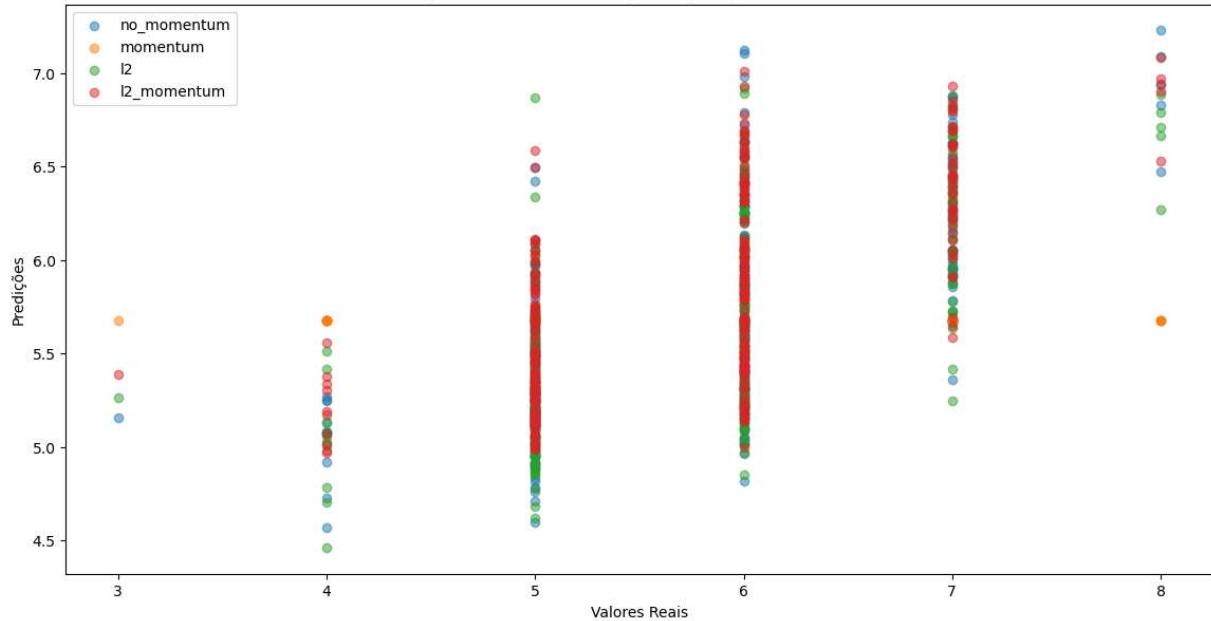


Comparação de MAE e Loss para Topologia 64-32-16-8

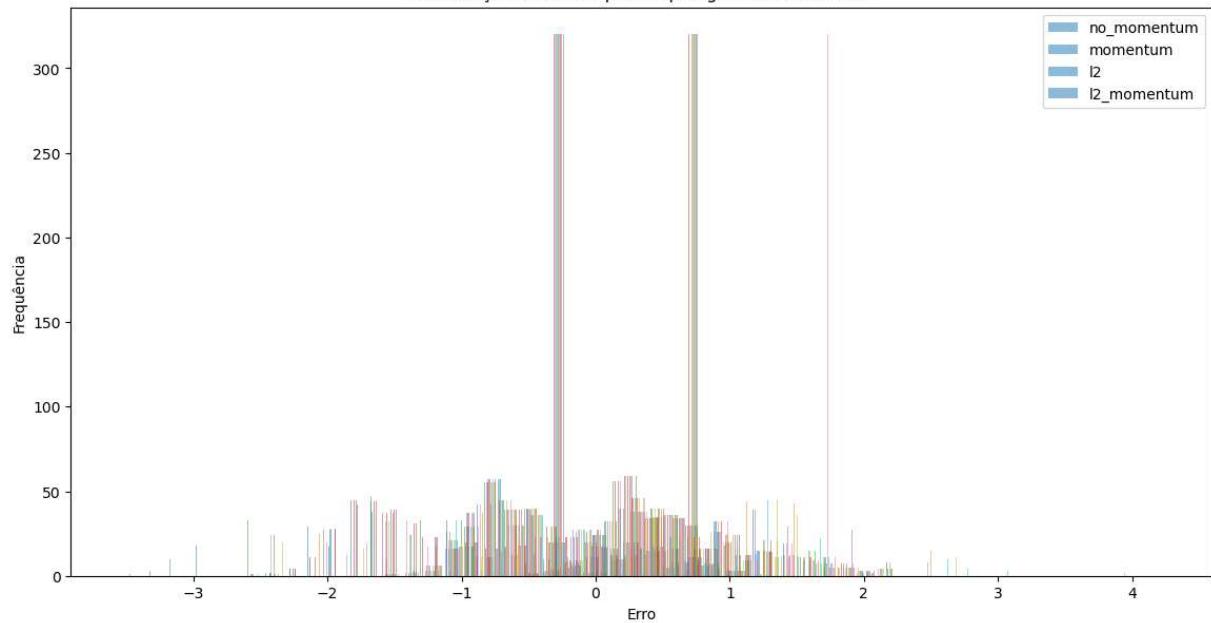


```
10/10 [=====] - 0s 1ms/step
10/10 [=====] - 0s 778us/step
10/10 [=====] - 0s 1ms/step
10/10 [=====] - 0s 778us/step
```

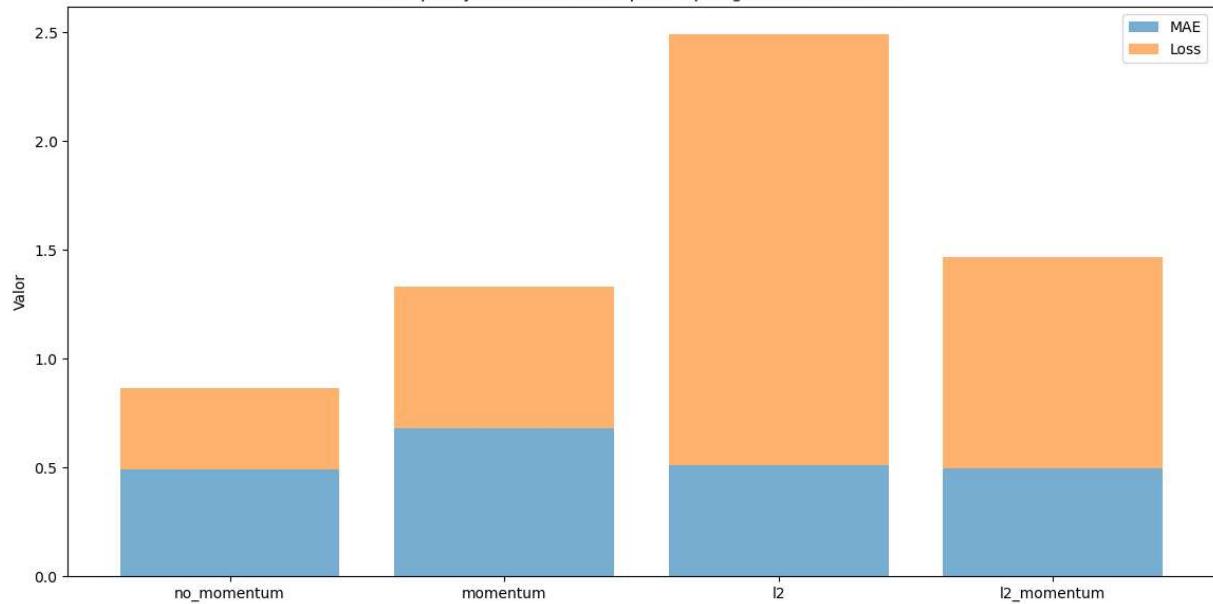
Predições vs. Valores Reais para Topologia 128-64-32-16-8



Distribuição dos Erros para Topologia 128-64-32-16-8

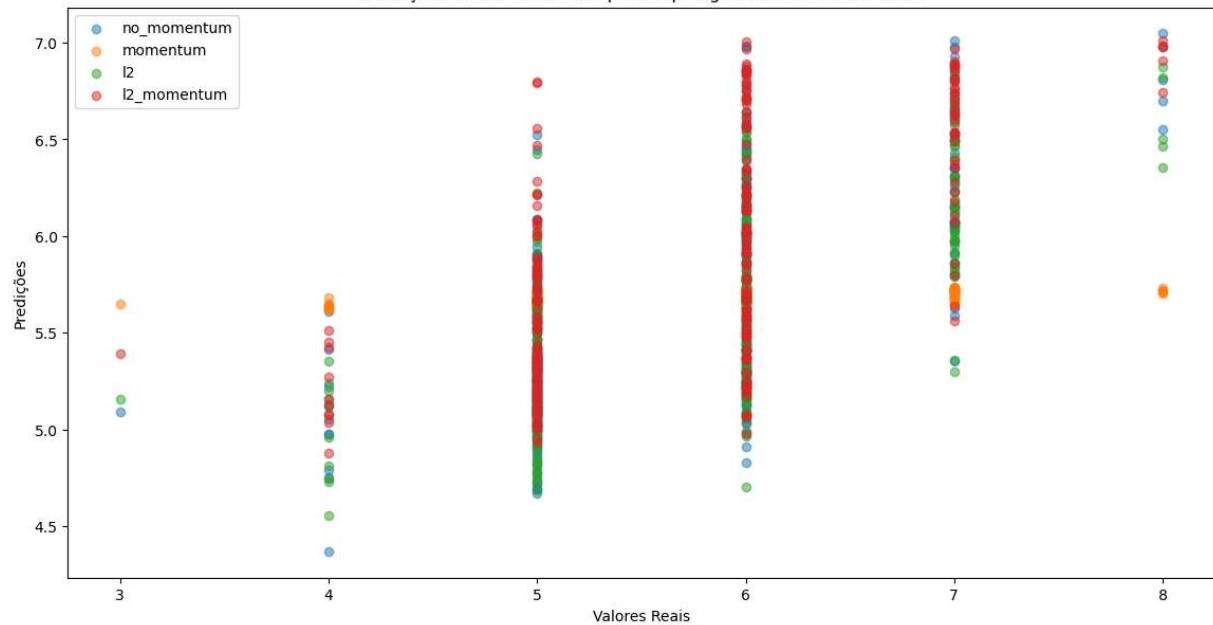


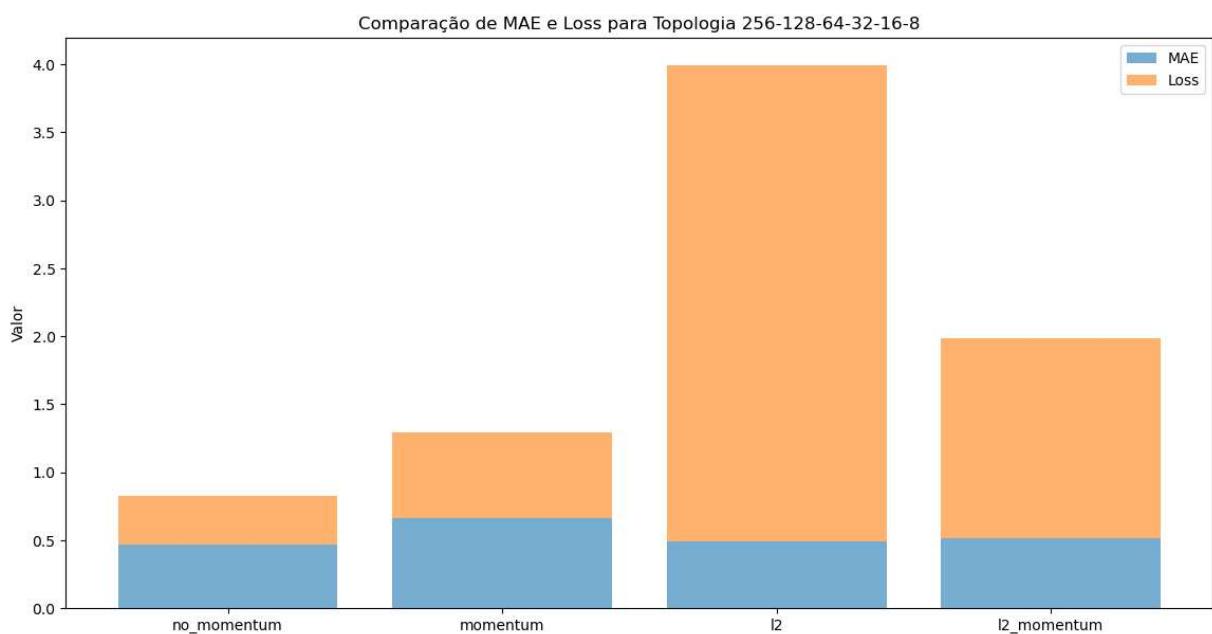
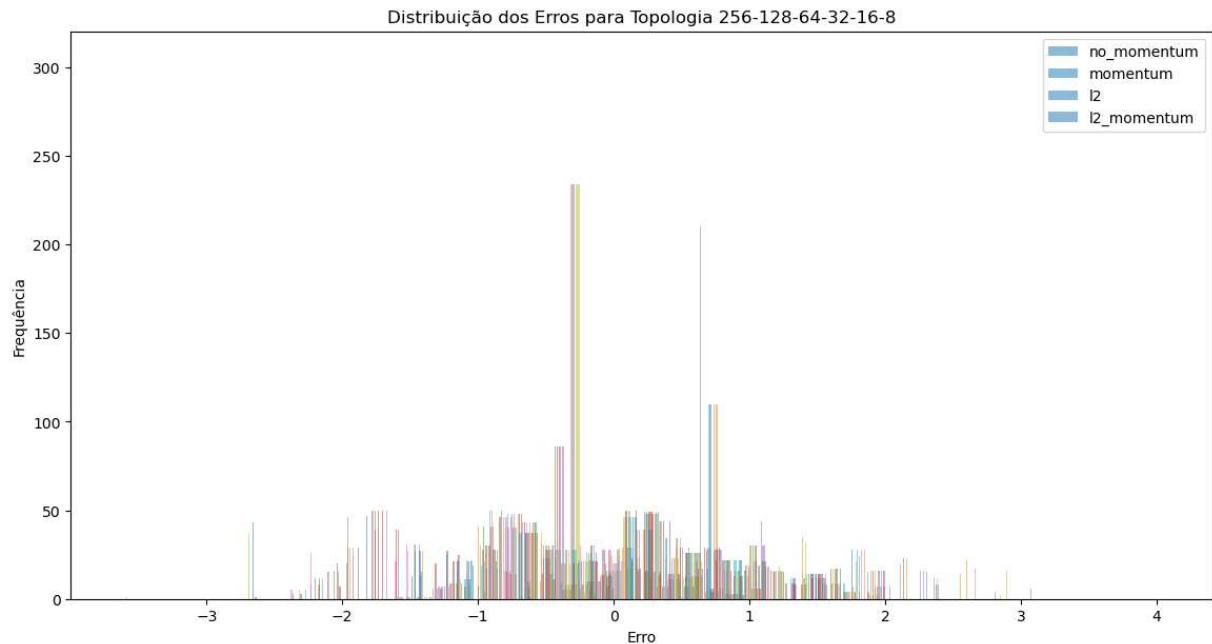
Comparação de MAE e Loss para Topologia 128-64-32-16-8



```
10/10 [=====] - 0s 1ms/step
```

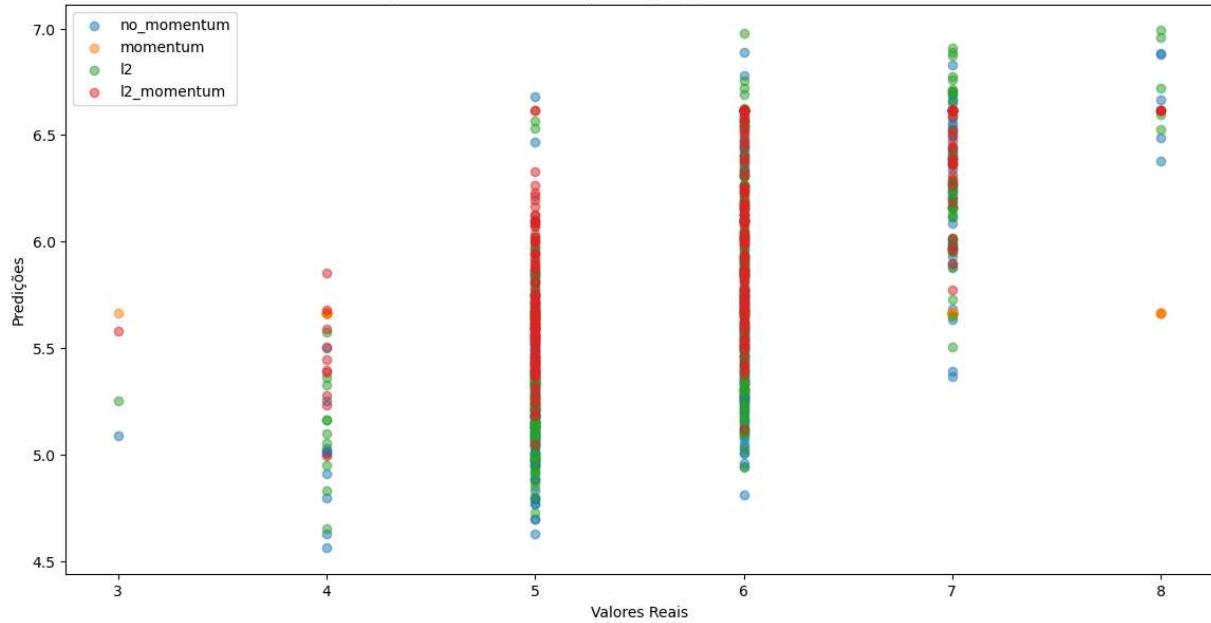
Predições vs. Valores Reais para Topologia 256-128-64-32-16-8



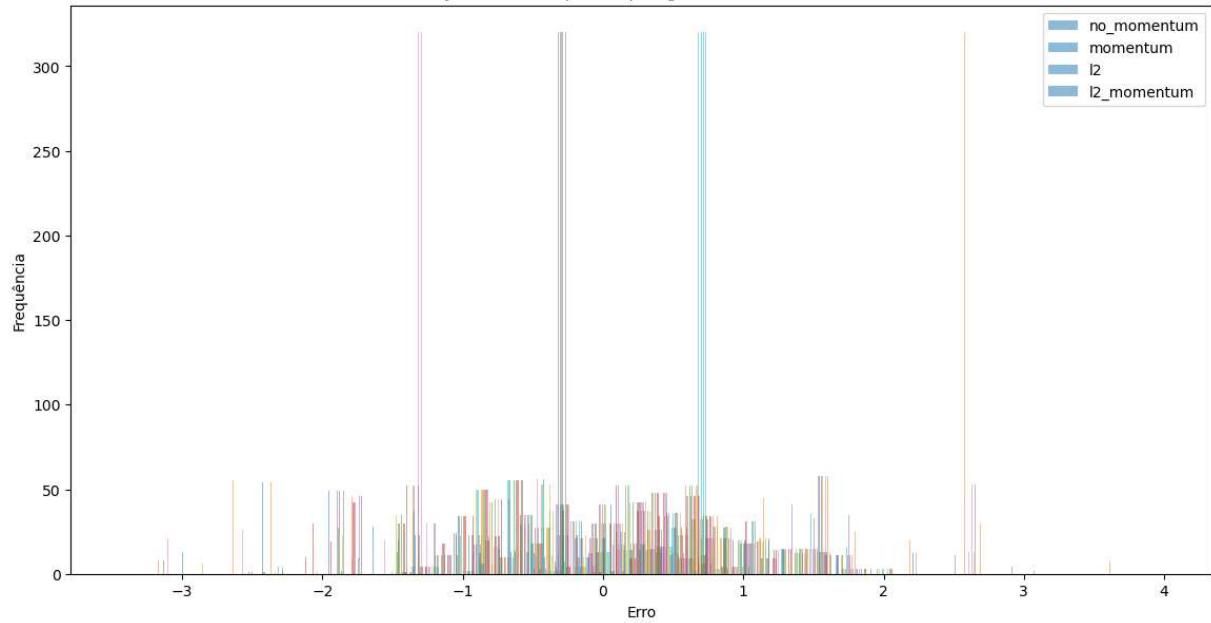


```
10/10 [=====] - 0s 2ms/step
10/10 [=====] - 0s 1ms/step
10/10 [=====] - 0s 1ms/step
10/10 [=====] - 0s 1ms/step
```

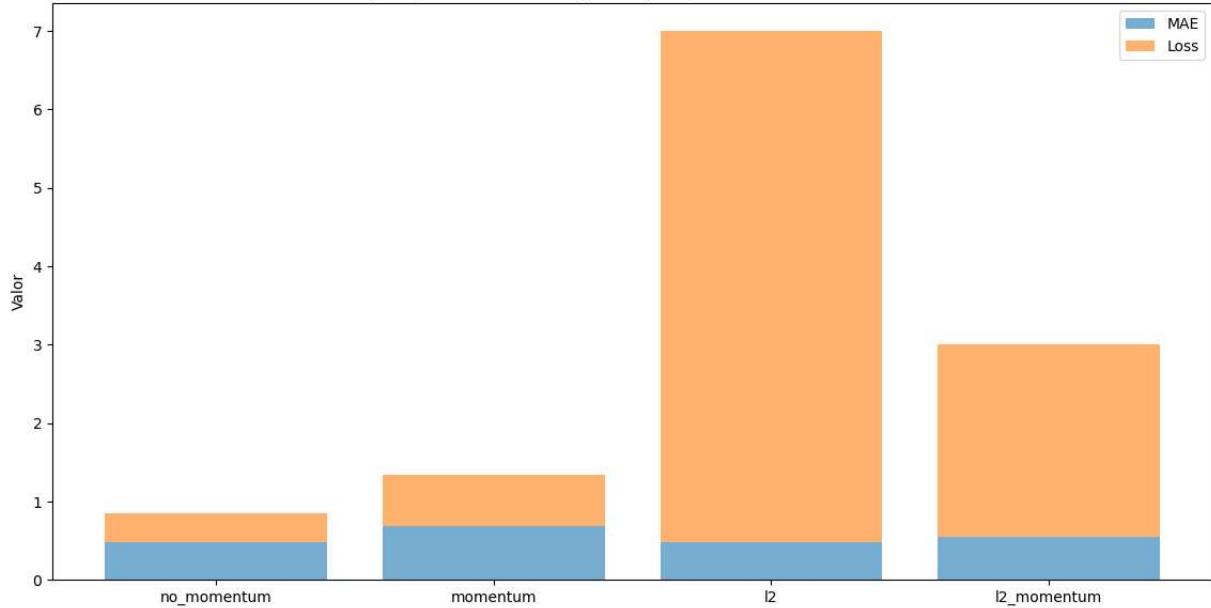
Predições vs. Valores Reais para Topologia 512-256-128-64-32-16-8



Distribuição dos Erros para Topologia 512-256-128-64-32-16-8



Comparação de MAE e Loss para Topologia 512-256-128-64-32-16-8



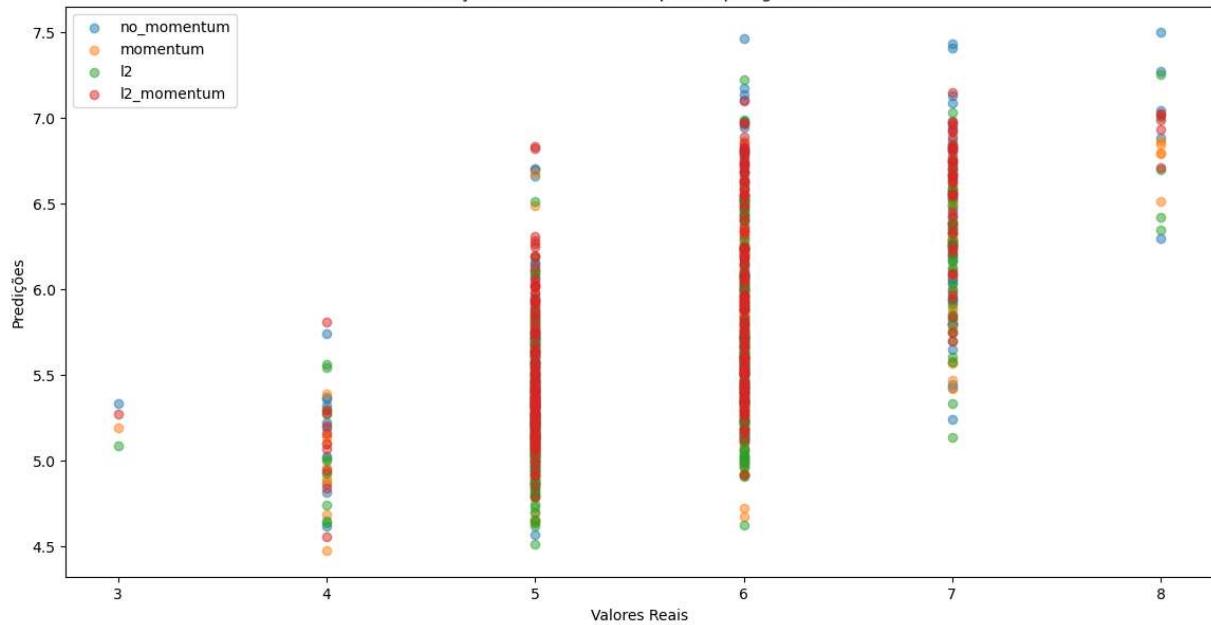
```
10/10 [=====] - 0s 2ms/step
```

```
10/10 [=====] - 0s 889us/step
```

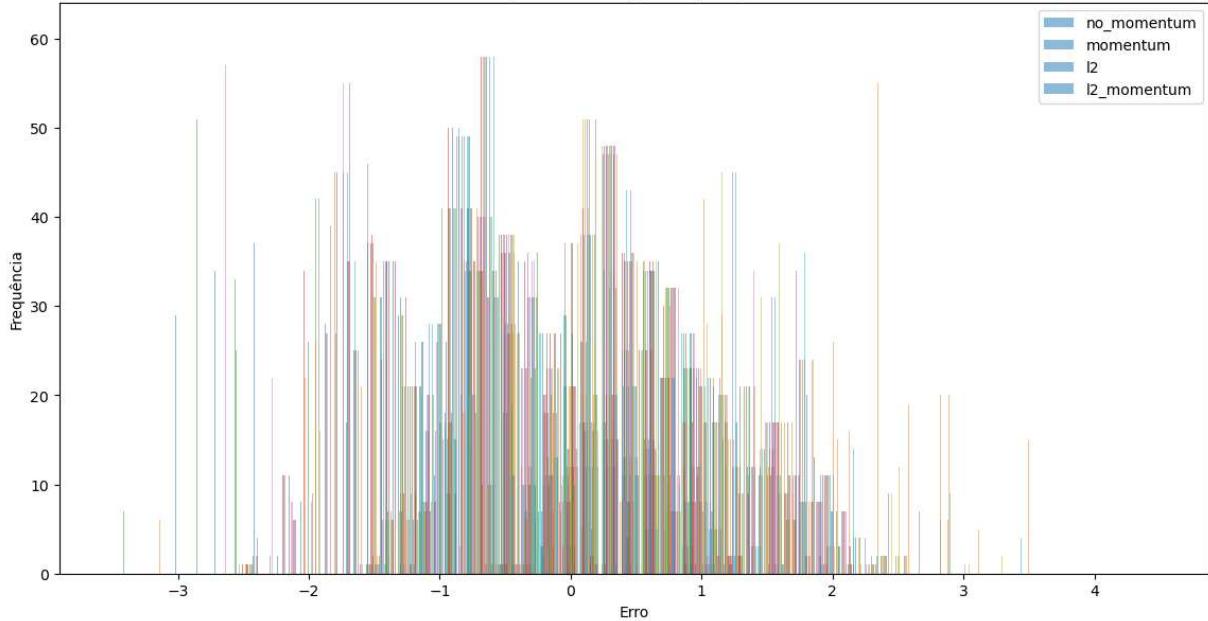
```
10/10 [=====] - 0s 1ms/step
```

```
10/10 [=====] - 0s 885us/step
```

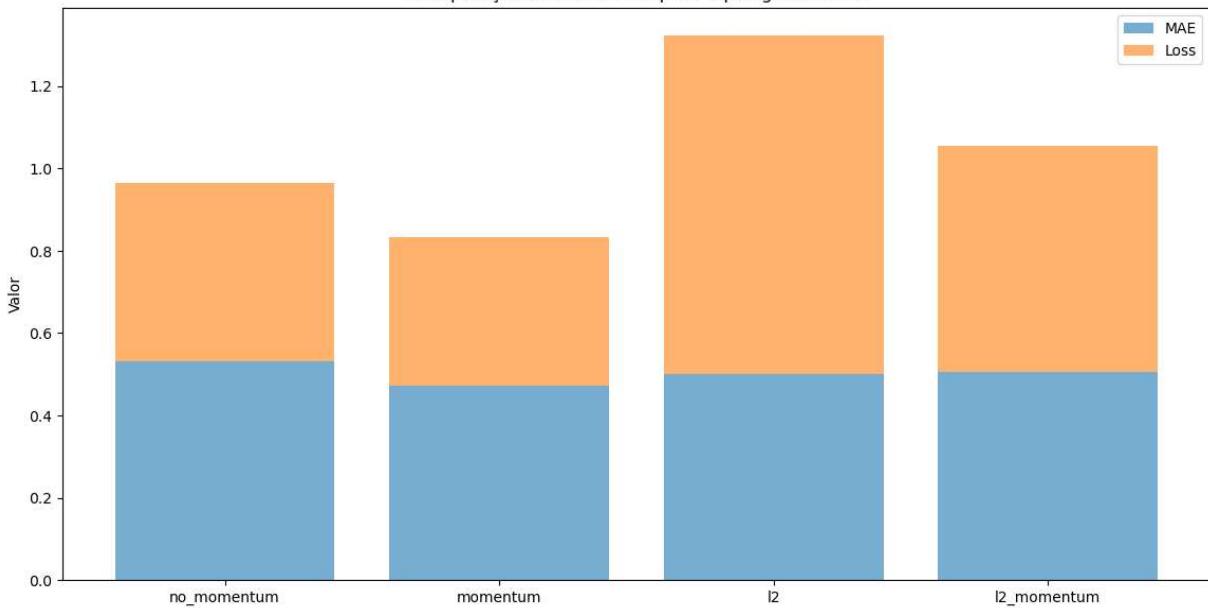
Predições vs. Valores Reais para Topologia 32-16-8



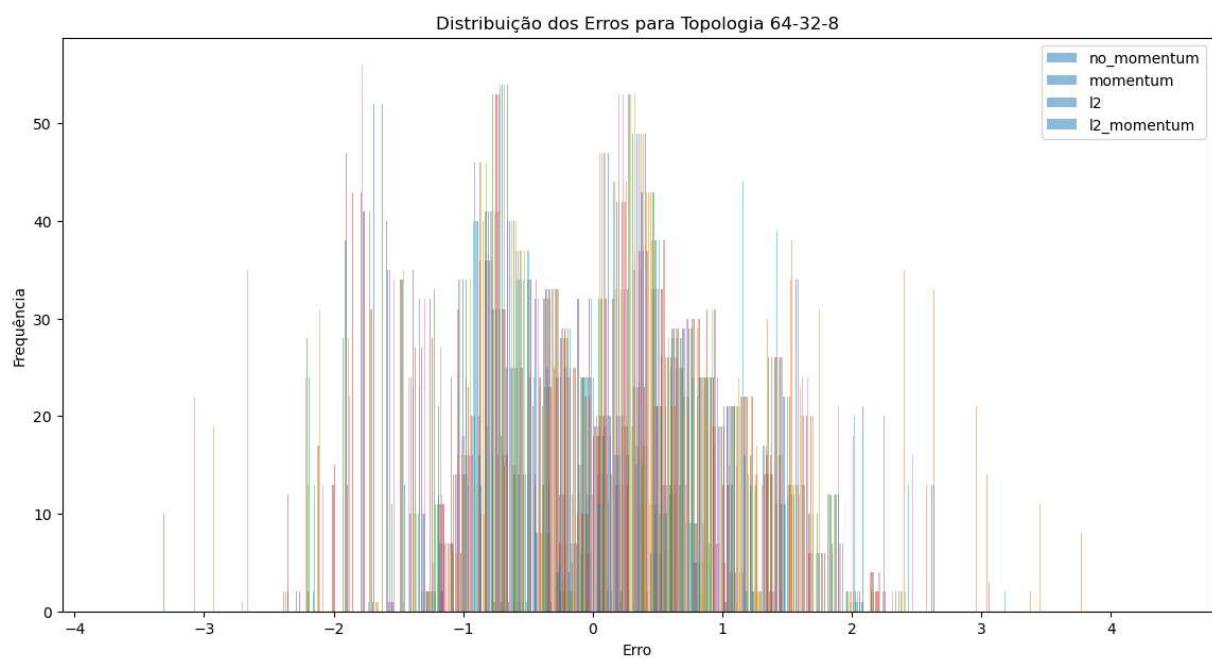
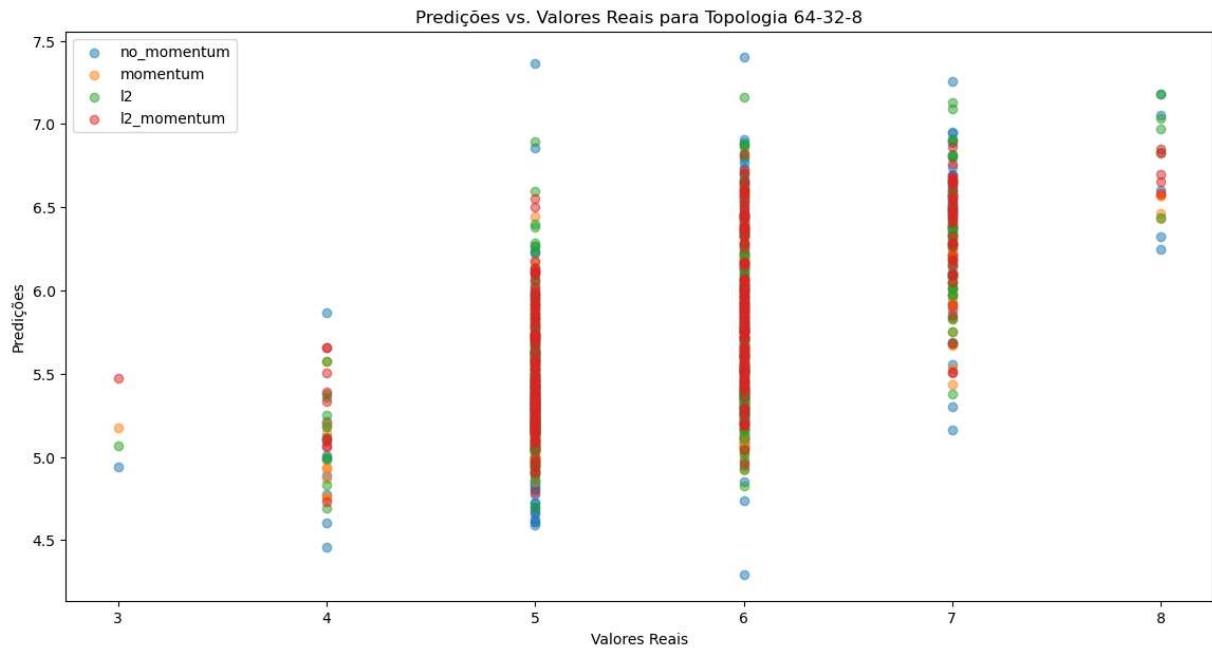
Distribuição dos Erros para Topologia 32-16-8



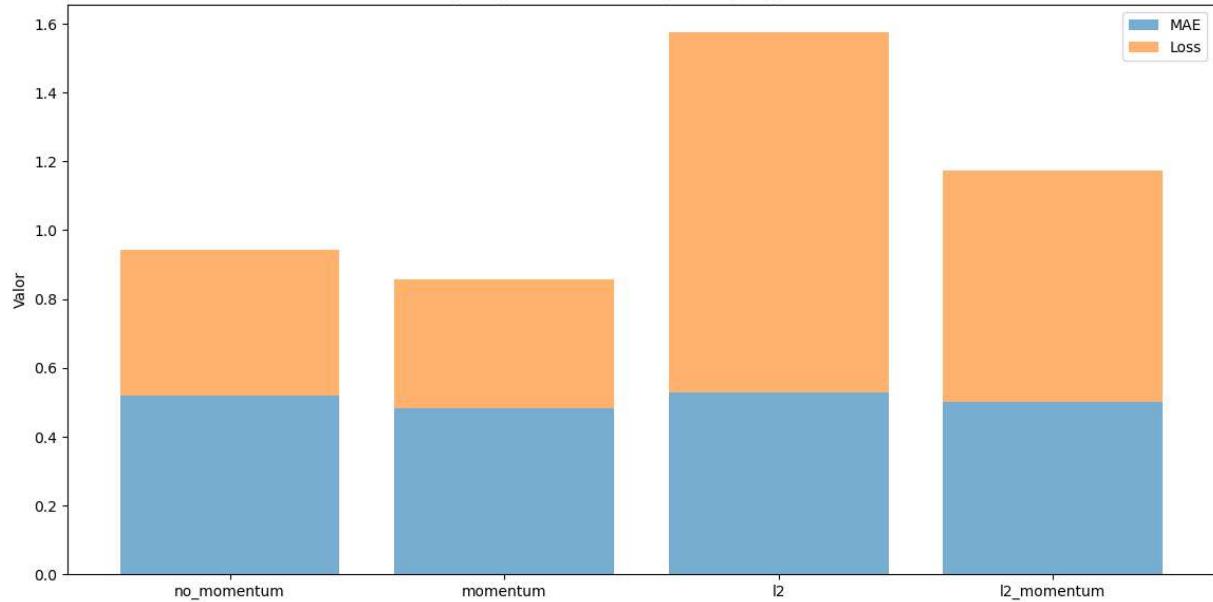
Comparação de MAE e Loss para Topologia 32-16-8



```
10/10 [=====] - 0s 886us/step
10/10 [=====] - 0s 889us/step
10/10 [=====] - 0s 1ms/step
10/10 [=====] - 0s 889us/step
```



Comparação de MAE e Loss para Topologia 64-32-8



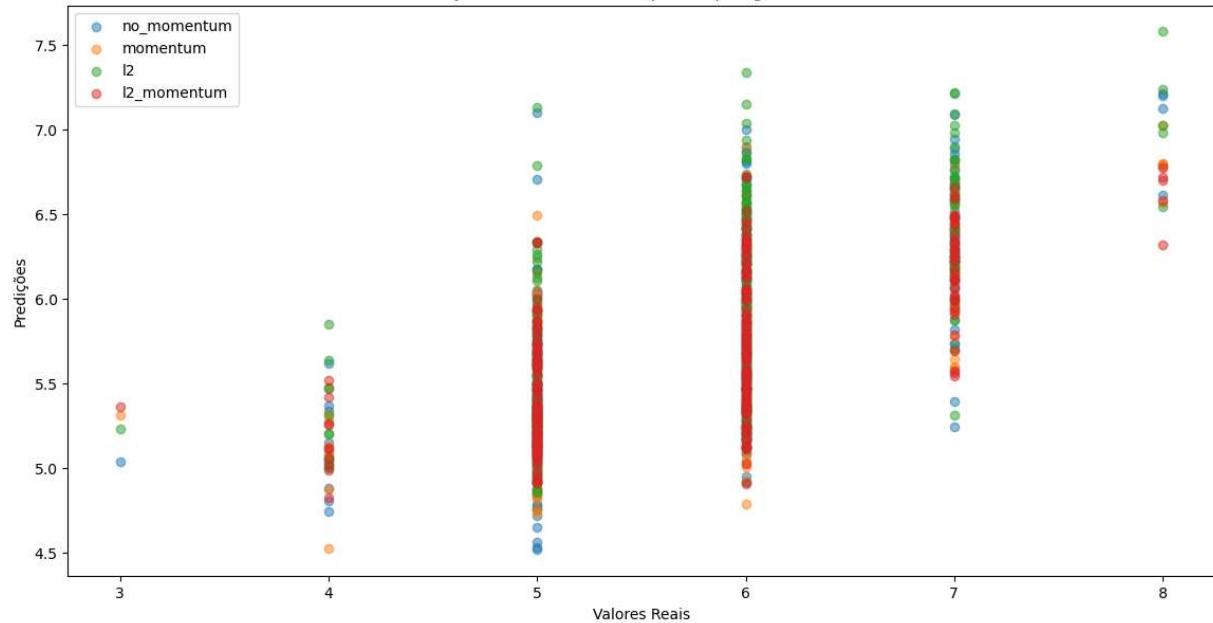
```
10/10 [=====] - 0s 884us/step
```

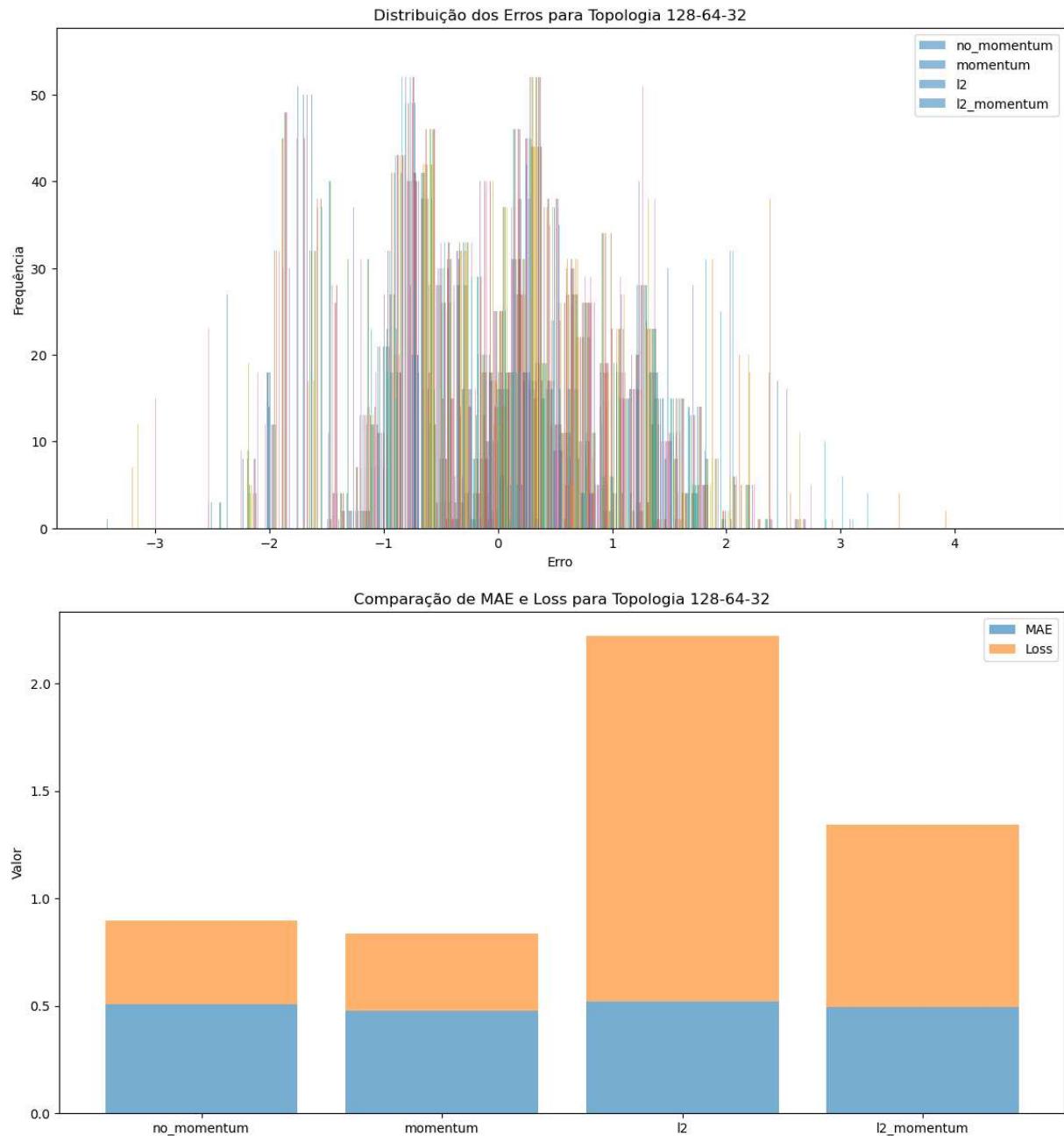
```
10/10 [=====] - 0s 1000us/step
```

```
10/10 [=====] - 0s 995us/step
```

```
10/10 [=====] - 0s 1ms/step
```

Predições vs. Valores Reais para Topologia 128-64-32





Relatório: Avaliação dos Modelos de Rede Neural para o Conjunto de Dados Wine Quality

1. Introdução:

O objetivo do experimento foi construir e avaliar modelos de redes neurais para prever a qualidade do vinho, explorando diferentes técnicas, como a implementação de momentum e regularização L2.

2. Modelos Avaliados:

Original: Modelo de baseline sem momentum e regularização. Momentum: Modelo com momentum implementado. Regularizado: Modelo com regularização L2. Ambos (Momentum + Regularizado): Modelo combinando momentum e regularização L2.

3. Resultados:

3.1. Predições vs. Valores Reais

No gráfico "Predições vs. Valores Reais", podemos ver que todos os modelos têm predições que se aproximam dos valores reais, indicando uma boa performance. Os pontos se aglomeraram em torno de uma linha diagonal, o que é um bom sinal. No entanto, há alguma dispersão, especialmente nas faixas de qualidade mais alta.

3.2. Distribuição dos Erros

A maior parte dos erros para todos os modelos está centrada em torno de zero, o que é um sinal positivo. O modelo regularizado apresenta uma distribuição de erros ligeiramente mais ampla do que os outros, o que pode ser devido à regularização evitando que o modelo se ajuste demais aos dados de treinamento.

3.3. Comparação de MAE e Loss

O modelo original e o modelo com momentum apresentam valores de MAE e Loss comparáveis. O modelo regularizado tem um MAE e Loss ligeiramente mais altos, indicando um ajuste potencialmente menos preciso. O modelo combinado (ambos) apresenta um desempenho intermediário entre o original e o regularizado.

4. Discussão:

Momentum: O momentum ajuda o modelo a navegar através dos mínimos locais e a evitar ficar preso em pontos subótimos durante o treinamento. Isso foi evidenciado pelo fato de que o modelo com momentum teve um desempenho comparável, se não ligeiramente melhor, ao modelo original.

Regularização L2: Esta técnica penaliza os pesos grandes, forçando-os a serem menores, o que ajuda a evitar o overfitting. No entanto, no nosso experimento, o modelo regularizado teve um desempenho ligeiramente pior em comparação com o original. Isto pode ser devido ao fato de que a ampla maioria dos modelos não estava overfitting, e a regularização, neste caso, introduziu um viés adicional desnecessário.

Overfitting: Ao longo do treinamento, foi observado que os erros de treinamento e validação se moveram juntos e não divergiram significativamente. Isto é uma indicação de que os modelos não sofreram de overfitting. Contudo, em algumas execuções, com muitas camadas de redes e muitos neurônios, foi detectado overfitting.

5. Conclusões:

Todos os modelos apresentaram desempenho satisfatório na previsão da qualidade do vinho. O modelo com momentum, particularmente, mostrou-se eficaz, enquanto o modelo regularizado não mostrou uma melhora significativa em relação ao modelo original, possivelmente devido à ausência inicial de overfitting.

Dataset de Classificação: Digits

Comecei a trabalhar com um terceiro problema: classificação de dígitos, contudo não tive tempo hábil para conclui-lo para entrega. Estou deixando aqui o título de curiosidade. A rede já foi feita, agora é apenas necessário fazer uma análise dos resultados.

Carregando o dataset e dividindo em treinamento, validação e teste:

In [7]:

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import load_digits
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from keras.utils import to_categorical
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.optimizers import SGD
from tensorflow.keras.regularizers import l2
from tqdm import tqdm

# Carregar o conjunto de dados
digits = load_digits()
X = digits.data
y = to_categorical(digits.target, 10)

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

# Normalização
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

def build_mlp(input_dim, layers, output_dim, output_activation, regularizer=None):
```

```

model = Sequential()
model.add(Dense(layers[0], input_dim=input_dim, activation='relu'))
for layer in layers[1:]:
    if regularizer:
        model.add(Dense(layer, activation='relu', kernel_regularizer=regularizer))
    else:
        model.add(Dense(layer, activation='relu'))
model.add(Dense(output_dim, activation=output_activation))
return model

# Definição das topologias e dicionários para armazenar os resultados e históricos
topologies = [
    [32, 16],
    [64, 32, 16],
    [128, 64, 32],
    [128, 64, 32, 16],
    [256, 128, 64, 32, 16],
    [512, 256, 128, 64, 32],
    [512, 256, 128, 64, 32, 16]
]
results = {}
histories = {}

#Barra de progresso com a lib tqdm
for layers in tqdm(topologies, desc="Training models", unit="topology"):
    topology_key = '-'.join(map(str, layers))
    results[topology_key] = {}
    histories[topology_key] = {}

    # 1. Modelo sem momentum e sem regularização
model = build_model(input_dim, layers, output_dim, output_activation)
model.compile(optimizer=SGD(), loss='categorical_crossentropy', metrics=['accuracy'])
model.fit(X_train, y_train, validation_split=0.1, epochs=50, batch_size=32, verbose=1)
model.save(f"model_{topology_key}_no_momentum_no_regularization.keras")

    # 2. Modelo com momentum
model_momentum = build_model(input_dim, layers, output_dim, output_activation)
optimizer_momentum = SGD(momentum=0.9)
model_momentum.compile(optimizer=optimizer_momentum, loss='categorical_crossentropy')
model_momentum.fit(X_train, y_train, validation_split=0.1, epochs=50, batch_size=32, verbose=1)
model_momentum.save(f"model_{topology_key}_momentum.keras")

    # 3. Modelo com regularização L2
model_l2 = build_model(input_dim, layers, output_dim, output_activation, regularization=True)
model_l2.compile(optimizer=SGD(), loss='categorical_crossentropy', metrics=['accuracy'])
model_l2.fit(X_train, y_train, validation_split=0.1, epochs=50, batch_size=32, verbose=1)
model_l2.save(f"model_{topology_key}_l2.keras")

    # 4. Modelo com regularização L2 e momentum
model_l2_momentum = build_model(input_dim, layers, output_dim, output_activation)
optimizer_l2_momentum = SGD(momentum=0.9)
model_l2_momentum.compile(optimizer=optimizer_l2_momentum, loss='categorical_crossentropy')
model_l2_momentum.fit(X_train, y_train, validation_split=0.1, epochs=50, batch_size=32, verbose=1)
model_l2_momentum.save(f"model_{topology_key}_l2_momentum.keras")

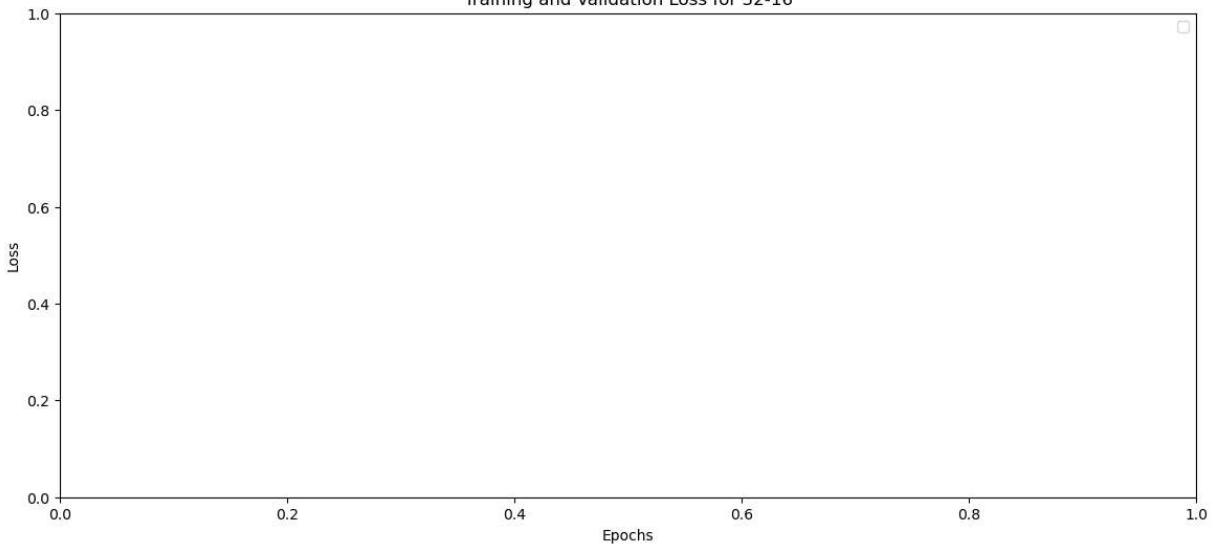
```

Training models: 100% |

| 7/7 [02:39<00:00, 22.80s/topology]

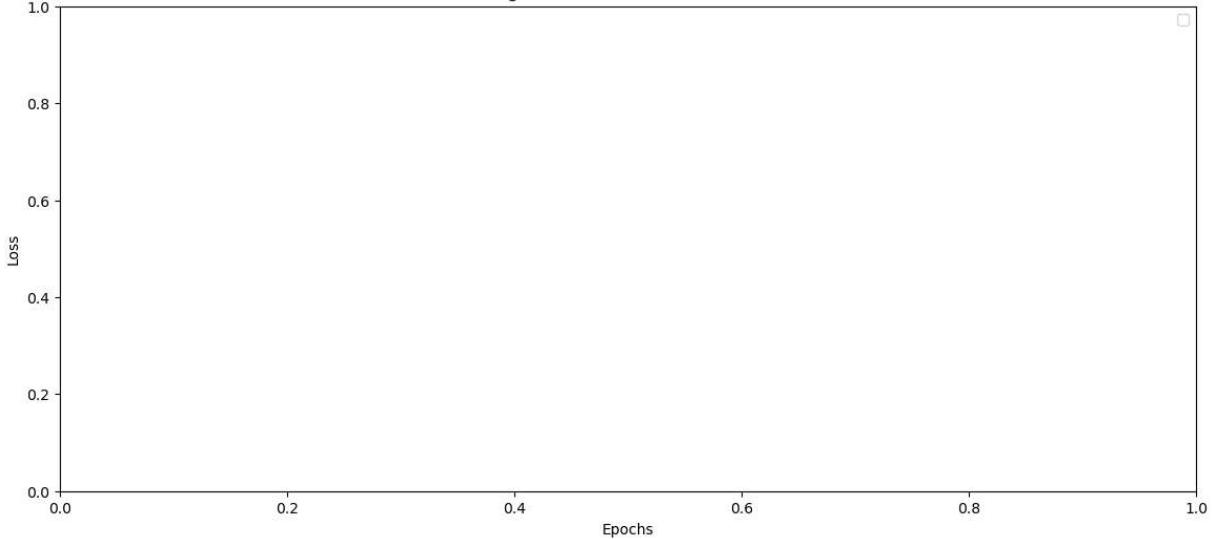
No artists with labels found to put in legend. Note that artists whose label start with an underscore are ignored when legend() is called with no argument.

Training and Validation Loss for 32-16

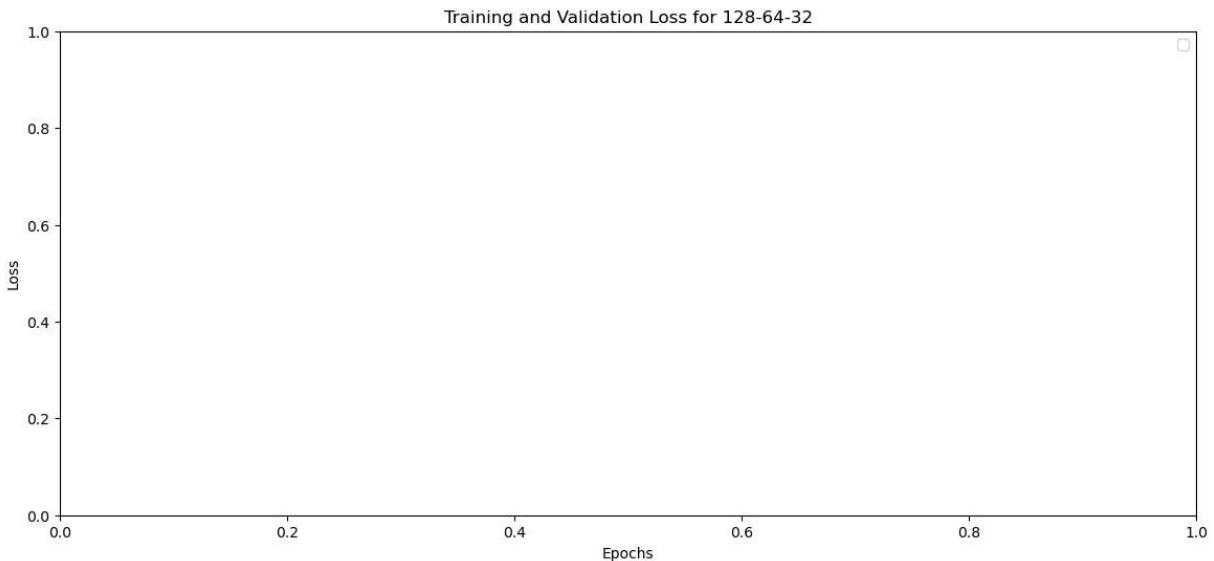


No artists with labels found to put in legend. Note that artists whose label start with an underscore are ignored when legend() is called with no argument.

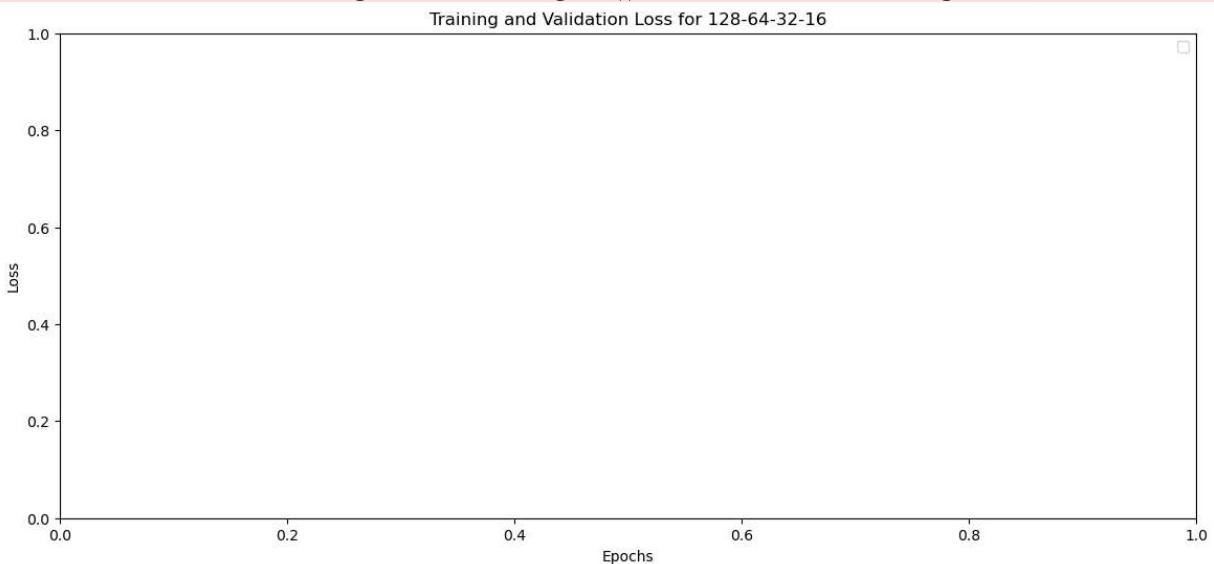
Training and Validation Loss for 64-32-16



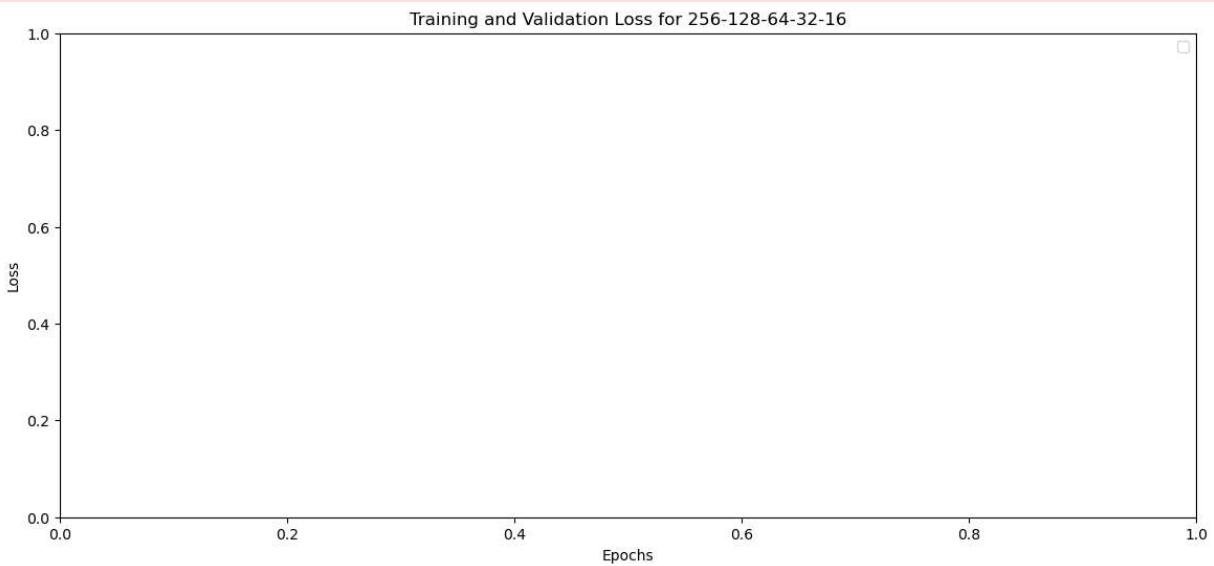
No artists with labels found to put in legend. Note that artists whose label start with an underscore are ignored when legend() is called with no argument.



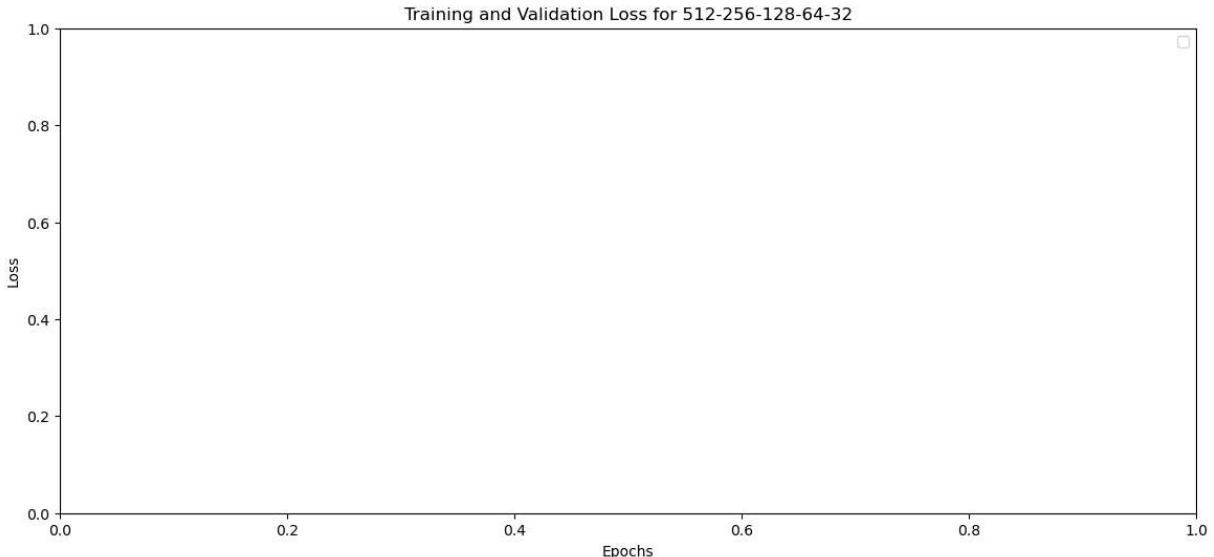
No artists with labels found to put in legend. Note that artists whose label start with an underscore are ignored when legend() is called with no argument.



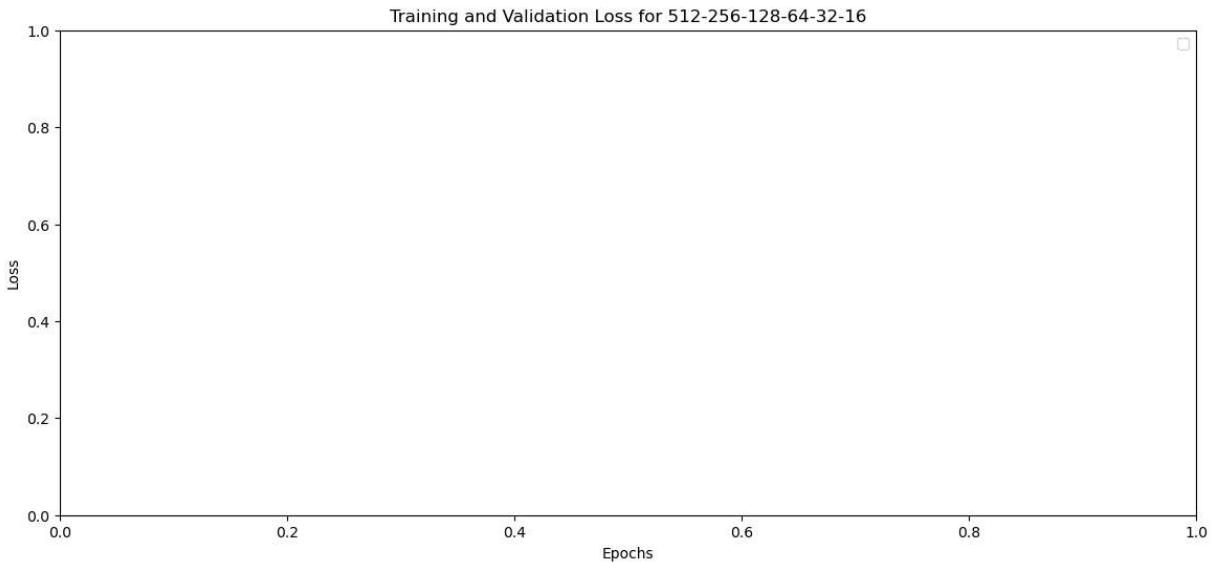
No artists with labels found to put in legend. Note that artists whose label start with an underscore are ignored when legend() is called with no argument.



No artists with labels found to put in legend. Note that artists whose label start with an underscore are ignored when legend() is called with no argument.



No artists with labels found to put in legend. Note that artists whose label start with an underscore are ignored when legend() is called with no argument.



In []: