Introduction

Our project "Exploring European Football Players - A deep learning approach" explores the use of deep learning and neural networks to predict and classify the performance of European football players. We have considered the dataset from Kaggle, we applied various data mining techniques as part of Module A of the course, followed by building and training multiple neural network models for both regression and classification tasks.

Our goal was to predict players overall ratings and categorize their performance levels.

Through extensive experimentation with different architectures and hyperparameter tuning, we aimed to find the optimal model that can accurately predict player performance while exploring advanced neural network concepts such as regularization, dropout, and learning rate tuning.

Project Overview:

We have divided the project into two parts,

Regression: Predicting player performance (overall rating) using numerical features.

Classification: Classifying players into performance categories using numerical and categorical features.

Importing Data

The data being imported is the file obtained after data transformation, imputation and feature engineering from Data Mining Module A.

Original Data Source: FIFA Football Players Dataset of 17000 Players From sofifa.com

from google.colab import drive drive.mount('/content/drive')

```
import pandas as pd
df = pd.read_csv('CSV_Files/Player_Attributes_with_Names.csv')
df.head()
```

```
           name                        full_name birth_date  age
height_cm  \
0      L. Messi  Lionel Andrés Messi Cuccittini  6/24/1987   31
170.18
1    C. Eriksen    Christian  Dannemann Eriksen  2/14/1992   27
154.94
2      P. Pogba                      Paul Pogba  3/15/1993   25
190.50
3    L. Insigne                 Lorenzo Insigne   6/4/1991   27
162.56
4  K. Koulibaly               Kalidou Koulibaly  6/20/1991   27
187.96

   weight_kgs  positions nationality  overall_rating
potential  ...  \
0        72.1   CF,RW,ST   Argentina              94         94  ...

1        76.2  CAM,RM,CM     Denmark              88         89  ...

2        83.9     CM,CAM      France              88         91  ...

3        59.0      LW,ST       Italy              88         88  ...

4        88.9         CB     Senegal              88         91  ...


   standing_tackle  sliding_tackle age_category  height_m
BMI  \
0               28              26          30+    1.7018  24.895349

1               57              22        25-30    1.5494  31.741531

2               67              67        20-25    1.9050  23.119157

3               24              22        25-30    1.6256  22.326705

4               88              87        25-30    1.8796  25.163491


    BMI_category offensive_score  defensive_score  offensive_category
\
0  Normal weight             828              227                 NaN

1        Obesity             777              292                High
```

```
2  Normal weight              760               414                  High

3  Normal weight              747               213                  High

4     Overweight              382               524                Medium


   defensive_category
0              Medium
1              Medium
2              Medium
3              Medium
4                High

[5 rows x 54 columns]
```

Objectives

Our objective is to create a comprehensive project to be able to predict and classify the players.

1. Predict Player Performance Ratings: Build and optimize deep learning models to accurately predict the overall performance ratings of European football players using regression techniques. The goal is to explore different model architectures and hyperparameter settings to achieve the lowest prediction error.
2. Classify Players into Performance Categories: Classify players into predefined categories (low, medium, high) based on their overall ratings by using deep learning classification models. The objective is to test various loss functions and optimize the model for the highest classification accuracy.

Methodology

Since the project is divided into parts we had different appraoches to do the tasks,

For the regression task, we predicted players' overall ratings using deep learning models. We began by preprocessing the dataset, applying feature selection, and normalizing the data. Two different frameworks, PyTorch and Keras, were used to experiment with model architectures and hyperparameters, such as batch size, epochs, and learning rates.

For the classification task, we converted the overall ratings into performance categories (low, medium, high) and applied categorical crossentropy and sparse categorical crossentropy as loss functions. We iterated through different configurations of batch sizes and epochs to find the optimal settings. Throughout the process, we tracked metrics such as MSE, RMSE, and accuracy, and visualized the training process to monitor overfitting or underfitting. Finally, we selected the best-

performing models and validated their predictions using confusion matrices and error analysis.

Theoritical Framework

Below in the table are the metrics of evaluation we used in our project tabulated with ideal values.

Metric	Definition	High or Low	Explanation
MSE (Mean Squared Error)	Average of the squared differences between predicted and actual values.	Low	Lower MSE means the model's predictions are closer to the actual values.
RMSE (Root Mean Squared Error)	Square root of MSE, providing an error measure in the same units as the target variable.	Low	Lower RMSE indicates better fit; good when closer to zero.
Training Loss	Error or cost of the model during training, calculated on the training dataset.	Low	Lower training loss suggests the model is learning well from the training data.
Validation Loss	Error or cost of the model during validation, calculated on the validation dataset.	Low	Lower validation loss suggests better generalization to unseen data.
Loss	Overall error measurement used to update model weights, generally a function like MSE.	Low	Lower loss means the model is learning well; difference between loss and val_loss shows overfitting.
Root Mean Squared Error	Another name for RMSE, showing error in the same scale as the output variable.	Low	A lower value indicates better model performance.
Val Loss	Validation set loss, used to check how well the model generalizes on unseen data.	Low	Should be close to training loss; large difference implies overfitting.
Val RMSE	RMSE for the validation dataset, showing prediction errors on unseen data.	Low	Lower value indicates better validation performance.

Metric	Definition	High or Low	Explanation
Training Accuracy	Percentage of correct predictions on the training data.	High	Higher accuracy means the model is correctly classifying more training samples.
Validation Accuracy	Percentage of correct predictions on the validation data.	High	Higher validation accuracy shows better performance on unseen data.

Epochs: The number of complete passes through the training dataset during model training.Epochs The number of complete passes through the training dataset during model training.

Batch Size: Number of samples processed before the model updates its weights.Larger batch size gives smoother gradient updates, smaller size gives noisy updates.

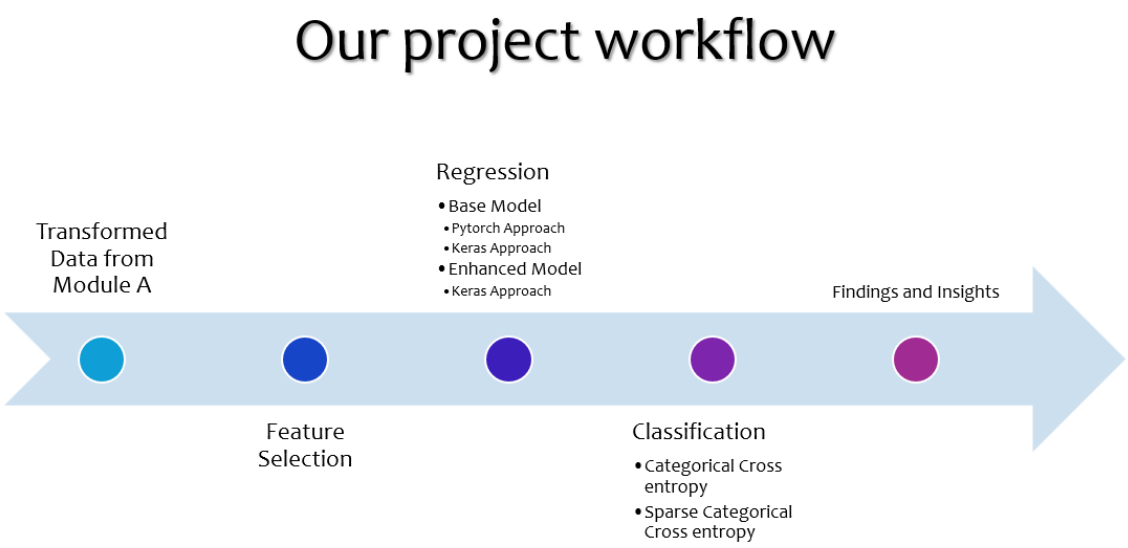
Steps per Epoch: Number of batches of data passed through the model in one epoch.More steps per epoch give finer-grained updates but may increase training time.

Reference :

Important Model Evaluation metrics

Metrics of Evaluation

Workflow of the Project



Regression

In the context of neural networks, regression refers to predicting continuous numerical values by learning a mapping from input features to a continuous output, typically using a loss function like Mean Squared Error (MSE).

Primary Objective: Predict the overall rating of football players based on their attributes

Libraries Required

Library	Short Description	Where & Why It Is Used
torch	Core library of PyTorch for building and running deep learning models.	Used for tensor operations, which are the building blocks of neural networks, enabling GPU acceleration.
torch.nn	Provides neural network layers and operations (e.g., loss functions, activations).	Used to define and build neural network architectures like layers, activations, and loss functions.
torch.optim	Contains optimization algorithms (e.g., SGD, Adam) for updating model parameters.	Used during training to optimize the model by adjusting weights based on the computed gradients.

The following are the lbraries required for Pytorch approach.

```
!pip install tensorflow
!pip install torchsummary
!pip install torchviz
!pip install torch torchvision
!pip install pydot
!pip install graphviz
!pip install tensorboard
```

```
Requirement already satisfied: tensorflow in
/home/unina/anaconda3/lib/python3.12/site-packages (2.17.0)
Requirement already satisfied: absl-py>=1.0.0 in
/home/unina/anaconda3/lib/python3.12/site-packages (from tensorflow)
(2.1.0)
Requirement already satisfied: astunparse>=1.6.0 in
/home/unina/anaconda3/lib/python3.12/site-packages (from tensorflow)
(1.6.3)
Requirement already satisfied: flatbuffers>=24.3.25 in
/home/unina/anaconda3/lib/python3.12/site-packages (from tensorflow)
(24.3.25)
Requirement already satisfied: gast!=0.5.0,!=0.5.1,!=0.5.2,>=0.2.1
in /home/unina/anaconda3/lib/python3.12/site-packages (from
tensorflow) (0.6.0)
Requirement already satisfied: google-pasta>=0.1.1 in
/home/unina/anaconda3/lib/python3.12/site-packages (from tensorflow)
(0.2.0)
Requirement already satisfied: h5py>=3.10.0 in
/home/unina/anaconda3/lib/python3.12/site-packages (from tensorflow)
(3.11.0)
Requirement already satisfied: libclang>=13.0.0 in
/home/unina/anaconda3/lib/python3.12/site-packages (from tensorflow)
```

```
(18.1.1)
Requirement already satisfied: ml-dtypes<0.5.0,>=0.3.1 in
/home/unina/anaconda3/lib/python3.12/site-packages (from tensorflow)
(0.4.0)
Requirement already satisfied: opt-einsum>=2.3.2 in
/home/unina/anaconda3/lib/python3.12/site-packages (from tensorflow)
(3.3.0)
Requirement already satisfied: packaging in
/home/unina/anaconda3/lib/python3.12/site-packages (from tensorflow)
(23.2)
Requirement already satisfied: protobuf!=4.21.0,!=4.21.1,!=4.21.2,!
=4.21.3,!=4.21.4,!=4.21.5,<5.0.0dev,>=3.20.3 in
/home/unina/anaconda3/lib/python3.12/site-packages (from tensorflow)
(3.20.3)
Requirement already satisfied: requests<3,>=2.21.0 in
/home/unina/anaconda3/lib/python3.12/site-packages (from tensorflow)
(2.32.2)
Requirement already satisfied: setuptools in
/home/unina/anaconda3/lib/python3.12/site-packages (from tensorflow)
(69.5.1)
Requirement already satisfied: six>=1.12.0 in
/home/unina/anaconda3/lib/python3.12/site-packages (from tensorflow)
(1.16.0)
Requirement already satisfied: termcolor>=1.1.0 in
/home/unina/anaconda3/lib/python3.12/site-packages (from tensorflow)
(2.4.0)
Requirement already satisfied: typing-extensions>=3.6.6 in
/home/unina/anaconda3/lib/python3.12/site-packages (from tensorflow)
(4.11.0)
Requirement already satisfied: wrapt>=1.11.0 in
/home/unina/anaconda3/lib/python3.12/site-packages (from tensorflow)
(1.14.1)
Requirement already satisfied: grpcio<2.0,>=1.24.3 in
/home/unina/anaconda3/lib/python3.12/site-packages (from tensorflow)
(1.66.1)
Requirement already satisfied: tensorboard<2.18,>=2.17 in
/home/unina/anaconda3/lib/python3.12/site-packages (from tensorflow)
(2.17.1)
Requirement already satisfied: keras>=3.2.0 in
/home/unina/anaconda3/lib/python3.12/site-packages (from tensorflow)
(3.5.0)
Requirement already satisfied: numpy<2.0.0,>=1.26.0 in
/home/unina/anaconda3/lib/python3.12/site-packages (from tensorflow)
(1.26.4)
Requirement already satisfied: wheel<1.0,>=0.23.0 in
/home/unina/anaconda3/lib/python3.12/site-packages (from
astunparse>=1.6.0->tensorflow) (0.43.0)
Requirement already satisfied: rich in
/home/unina/anaconda3/lib/python3.12/site-packages (from keras>=3.2.0-
```

```
>tensorflow) (13.3.5)
Requirement already satisfied: namex in
/home/unina/anaconda3/lib/python3.12/site-packages (from keras>=3.2.0-
>tensorflow) (0.0.8)
Requirement already satisfied: optree in
/home/unina/anaconda3/lib/python3.12/site-packages (from keras>=3.2.0-
>tensorflow) (0.12.1)
Requirement already satisfied: charset-normalizer<4,>=2 in
/home/unina/anaconda3/lib/python3.12/site-packages (from
requests<3,>=2.21.0->tensorflow) (2.0.4)
Requirement already satisfied: idna<4,>=2.5 in
/home/unina/anaconda3/lib/python3.12/site-packages (from
requests<3,>=2.21.0->tensorflow) (3.7)
Requirement already satisfied: urllib3<3,>=1.21.1 in
/home/unina/anaconda3/lib/python3.12/site-packages (from
requests<3,>=2.21.0->tensorflow) (2.2.2)
Requirement already satisfied: certifi>=2017.4.17 in
/home/unina/anaconda3/lib/python3.12/site-packages (from
requests<3,>=2.21.0->tensorflow) (2024.8.30)
Requirement already satisfied: markdown>=2.6.8 in
/home/unina/anaconda3/lib/python3.12/site-packages (from
tensorboard<2.18,>=2.17->tensorflow) (3.4.1)
Requirement already satisfied: tensorboard-data-server<0.8.0,>=0.7.0
in /home/unina/anaconda3/lib/python3.12/site-packages (from
tensorboard<2.18,>=2.17->tensorflow) (0.7.2)
Requirement already satisfied: werkzeug>=1.0.1 in
/home/unina/anaconda3/lib/python3.12/site-packages (from
tensorboard<2.18,>=2.17->tensorflow) (3.0.3)
Requirement already satisfied: MarkupSafe>=2.1.1 in
/home/unina/anaconda3/lib/python3.12/site-packages (from
werkzeug>=1.0.1->tensorboard<2.18,>=2.17->tensorflow) (2.1.3)
Requirement already satisfied: markdown-it-py<3.0.0,>=2.2.0 in
/home/unina/anaconda3/lib/python3.12/site-packages (from rich-
>keras>=3.2.0->tensorflow) (2.2.0)
Requirement already satisfied: pygments<3.0.0,>=2.13.0 in
/home/unina/anaconda3/lib/python3.12/site-packages (from rich-
>keras>=3.2.0->tensorflow) (2.15.1)
Requirement already satisfied: mdurl~=0.1 in
/home/unina/anaconda3/lib/python3.12/site-packages (from markdown-it-
py<3.0.0,>=2.2.0->rich->keras>=3.2.0->tensorflow) (0.1.0)
Requirement already satisfied: torchsummary in
/home/unina/anaconda3/lib/python3.12/site-packages (1.5.1)
Requirement already satisfied: torchviz in
/home/unina/anaconda3/lib/python3.12/site-packages (0.0.2)
Requirement already satisfied: torch in
/home/unina/anaconda3/lib/python3.12/site-packages (from torchviz)
(2.4.1)
Requirement already satisfied: graphviz in
/home/unina/anaconda3/lib/python3.12/site-packages (from torchviz)
```

```
(0.20.3)
Requirement already satisfied: filelock in
/home/unina/anaconda3/lib/python3.12/site-packages (from torch-
>torchviz) (3.13.1)
Requirement already satisfied: typing-extensions>=4.8.0 in
/home/unina/anaconda3/lib/python3.12/site-packages (from torch-
>torchviz) (4.11.0)
Requirement already satisfied: sympy in
/home/unina/anaconda3/lib/python3.12/site-packages (from torch-
>torchviz) (1.12)
Requirement already satisfied: networkx in
/home/unina/anaconda3/lib/python3.12/site-packages (from torch-
>torchviz) (3.2.1)
Requirement already satisfied: jinja2 in
/home/unina/anaconda3/lib/python3.12/site-packages (from torch-
>torchviz) (3.1.4)
Requirement already satisfied: fsspec in
/home/unina/anaconda3/lib/python3.12/site-packages (from torch-
>torchviz) (2024.3.1)
Requirement already satisfied: setuptools in
/home/unina/anaconda3/lib/python3.12/site-packages (from torch-
>torchviz) (69.5.1)
Requirement already satisfied: nvidia-cuda-nvrtc-cu12==12.1.105 in
/home/unina/anaconda3/lib/python3.12/site-packages (from torch-
>torchviz) (12.1.105)
Requirement already satisfied: nvidia-cuda-runtime-cu12==12.1.105
in /home/unina/anaconda3/lib/python3.12/site-packages (from torch-
>torchviz) (12.1.105)
Requirement already satisfied: nvidia-cuda-cupti-cu12==12.1.105 in
/home/unina/anaconda3/lib/python3.12/site-packages (from torch-
>torchviz) (12.1.105)
Requirement already satisfied: nvidia-cudnn-cu12==9.1.0.70 in
/home/unina/anaconda3/lib/python3.12/site-packages (from torch-
>torchviz) (9.1.0.70)
Requirement already satisfied: nvidia-cublas-cu12==12.1.3.1 in
/home/unina/anaconda3/lib/python3.12/site-packages (from torch-
>torchviz) (12.1.3.1)
Requirement already satisfied: nvidia-cufft-cu12==11.0.2.54 in
/home/unina/anaconda3/lib/python3.12/site-packages (from torch-
>torchviz) (11.0.2.54)
Requirement already satisfied: nvidia-curand-cu12==10.3.2.106 in
/home/unina/anaconda3/lib/python3.12/site-packages (from torch-
>torchviz) (10.3.2.106)
Requirement already satisfied: nvidia-cusolver-cu12==11.4.5.107 in
/home/unina/anaconda3/lib/python3.12/site-packages (from torch-
>torchviz) (11.4.5.107)
Requirement already satisfied: nvidia-cusparse-cu12==12.1.0.106 in
/home/unina/anaconda3/lib/python3.12/site-packages (from torch-
>torchviz) (12.1.0.106)
```

```
Requirement already satisfied: nvidia-nccl-cu12==2.20.5 in
/home/unina/anaconda3/lib/python3.12/site-packages (from torch-
>torchviz) (2.20.5)
Requirement already satisfied: nvidia-nvtx-cu12==12.1.105 in
/home/unina/anaconda3/lib/python3.12/site-packages (from torch-
>torchviz) (12.1.105)
Requirement already satisfied: triton==3.0.0 in
/home/unina/anaconda3/lib/python3.12/site-packages (from torch-
>torchviz) (3.0.0)
Requirement already satisfied: nvidia-nvjitlink-cu12 in
/home/unina/anaconda3/lib/python3.12/site-packages (from nvidia-
cusolver-cu12==11.4.5.107->torch->torchviz) (12.6.68)
Requirement already satisfied: MarkupSafe>=2.0 in
/home/unina/anaconda3/lib/python3.12/site-packages (from jinja2-
>torch->torchviz) (2.1.3)
Requirement already satisfied: mpmath>=0.19 in
/home/unina/anaconda3/lib/python3.12/site-packages (from sympy->torch-
>torchviz) (1.3.0)
Requirement already satisfied: torch in
/home/unina/anaconda3/lib/python3.12/site-packages (2.4.1)
Requirement already satisfied: torchvision in
/home/unina/anaconda3/lib/python3.12/site-packages (0.19.1)
Requirement already satisfied: filelock in
/home/unina/anaconda3/lib/python3.12/site-packages (from torch)
(3.13.1)
Requirement already satisfied: typing-extensions>=4.8.0 in
/home/unina/anaconda3/lib/python3.12/site-packages (from torch)
(4.11.0)
Requirement already satisfied: sympy in
/home/unina/anaconda3/lib/python3.12/site-packages (from torch) (1.12)
Requirement already satisfied: networkx in
/home/unina/anaconda3/lib/python3.12/site-packages (from torch)
(3.2.1)
Requirement already satisfied: jinja2 in
/home/unina/anaconda3/lib/python3.12/site-packages (from torch)
(3.1.4)
Requirement already satisfied: fsspec in
/home/unina/anaconda3/lib/python3.12/site-packages (from torch)
(2024.3.1)
Requirement already satisfied: setuptools in
/home/unina/anaconda3/lib/python3.12/site-packages (from torch)
(69.5.1)
Requirement already satisfied: nvidia-cuda-nvrtc-cu12==12.1.105 in
/home/unina/anaconda3/lib/python3.12/site-packages (from torch)
(12.1.105)
Requirement already satisfied: nvidia-cuda-runtime-cu12==12.1.105
in /home/unina/anaconda3/lib/python3.12/site-packages (from torch)
(12.1.105)
Requirement already satisfied: nvidia-cuda-cupti-cu12==12.1.105 in
```

```
/home/unina/anaconda3/lib/python3.12/site-packages (from torch)
(12.1.105)
Requirement already satisfied: nvidia-cudnn-cu12==9.1.0.70 in
/home/unina/anaconda3/lib/python3.12/site-packages (from torch)
(9.1.0.70)
Requirement already satisfied: nvidia-cublas-cu12==12.1.3.1 in
/home/unina/anaconda3/lib/python3.12/site-packages (from torch)
(12.1.3.1)
Requirement already satisfied: nvidia-cufft-cu12==11.0.2.54 in
/home/unina/anaconda3/lib/python3.12/site-packages (from torch)
(11.0.2.54)
Requirement already satisfied: nvidia-curand-cu12==10.3.2.106 in
/home/unina/anaconda3/lib/python3.12/site-packages (from torch)
(10.3.2.106)
Requirement already satisfied: nvidia-cusolver-cu12==11.4.5.107 in
/home/unina/anaconda3/lib/python3.12/site-packages (from torch)
(11.4.5.107)
Requirement already satisfied: nvidia-cusparse-cu12==12.1.0.106 in
/home/unina/anaconda3/lib/python3.12/site-packages (from torch)
(12.1.0.106)
Requirement already satisfied: nvidia-nccl-cu12==2.20.5 in
/home/unina/anaconda3/lib/python3.12/site-packages (from torch)
(2.20.5)
Requirement already satisfied: nvidia-nvtx-cu12==12.1.105 in
/home/unina/anaconda3/lib/python3.12/site-packages (from torch)
(12.1.105)
Requirement already satisfied: triton==3.0.0 in
/home/unina/anaconda3/lib/python3.12/site-packages (from torch)
(3.0.0)
Requirement already satisfied: nvidia-nvjitlink-cu12 in
/home/unina/anaconda3/lib/python3.12/site-packages (from nvidia-
cusolver-cu12==11.4.5.107->torch) (12.6.68)
Requirement already satisfied: numpy in
/home/unina/anaconda3/lib/python3.12/site-packages (from torchvision)
(1.26.4)
Requirement already satisfied: pillow!=8.3.*,>=5.3.0 in
/home/unina/anaconda3/lib/python3.12/site-packages (from torchvision)
(10.3.0)
Requirement already satisfied: MarkupSafe>=2.0 in
/home/unina/anaconda3/lib/python3.12/site-packages (from jinja2-
>torch) (2.1.3)
Requirement already satisfied: mpmath>=0.19 in
/home/unina/anaconda3/lib/python3.12/site-packages (from sympy->torch)
(1.3.0)
Requirement already satisfied: pydot in
/home/unina/anaconda3/lib/python3.12/site-packages (3.0.1)
Requirement already satisfied: pyparsing>=3.0.9 in
/home/unina/anaconda3/lib/python3.12/site-packages (from pydot)
(3.0.9)
```

```
Requirement already satisfied: graphviz in
/home/unina/anaconda3/lib/python3.12/site-packages (0.20.3)
Requirement already satisfied: tensorboard in
/home/unina/anaconda3/lib/python3.12/site-packages (2.17.1)
Requirement already satisfied: absl-py>=0.4 in
/home/unina/anaconda3/lib/python3.12/site-packages (from tensorboard)
(2.1.0)
Requirement already satisfied: grpcio>=1.48.2 in
/home/unina/anaconda3/lib/python3.12/site-packages (from tensorboard)
(1.66.1)
Requirement already satisfied: markdown>=2.6.8 in
/home/unina/anaconda3/lib/python3.12/site-packages (from tensorboard)
(3.4.1)
Requirement already satisfied: numpy>=1.12.0 in
/home/unina/anaconda3/lib/python3.12/site-packages (from tensorboard)
(1.26.4)
Requirement already satisfied: packaging in
/home/unina/anaconda3/lib/python3.12/site-packages (from tensorboard)
(23.2)
Requirement already satisfied: protobuf!=4.24.0,>=3.19.6 in
/home/unina/anaconda3/lib/python3.12/site-packages (from tensorboard)
(3.20.3)
Requirement already satisfied: setuptools>=41.0.0 in
/home/unina/anaconda3/lib/python3.12/site-packages (from tensorboard)
(69.5.1)
Requirement already satisfied: six>1.9 in
/home/unina/anaconda3/lib/python3.12/site-packages (from tensorboard)
(1.16.0)
Requirement already satisfied: tensorboard-data-server<0.8.0,>=0.7.0
in /home/unina/anaconda3/lib/python3.12/site-packages (from
tensorboard) (0.7.2)
Requirement already satisfied: werkzeug>=1.0.1 in
/home/unina/anaconda3/lib/python3.12/site-packages (from tensorboard)
(3.0.3)
Requirement already satisfied: MarkupSafe>=2.1.1 in
/home/unina/anaconda3/lib/python3.12/site-packages (from
werkzeug>=1.0.1->tensorboard) (2.1.3)
```

```python
# Importing the libraries
import torch
import torch.nn as nn
import torch.optim as optim
```

Then these are for the tensorflow appraoch

Library	Short Description	Where & Why It Is Used
tensorflow	Open-source deep learning library for building and training neural networks.	Used for machine learning and deep learning tasks, supporting both CPU and GPU acceleration.
tensorflow.keras	High-level API in TensorFlow for building and training neural networks.	Provides an easy-to-use interface for creating neural networks with layers, models, and more.
tensorflow.keras.layers	Contains predefined neural network layers (e.g., Dense, Conv2D).	Used to define different types of layers in a neural network model.
tensorflow.keras.models	Provides classes to build and manage neural network models (e.g., Sequential, Model).	Used to create and compile neural network architectures, supporting functional and sequential models.

```
# Import necessary libraries
import tensorflow as tf
from tensorflow.keras import layers, models
```

Then these are the general ones

```
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
import pandas as pd
import matplotlib.pyplot as plt

#Model Architecture
from torchsummary import summary
# Displaying the model plot in Colab
from IPython.display import Image
from torchviz import make_dot
from tensorflow.keras.utils import plot_model

#Keras approach
#Base Model
from tensorflow.keras.callbacks import EarlyStopping
from tensorflow.keras.models import Sequential
from tensorflow.keras import layers, regularizers
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.metrics import RootMeanSquaredError
#Enhanced Model
import pickle

#Testing on Best model
from tensorflow.keras.models import load_model
import numpy as np
```

```
import pandas as pd
import joblib

#Findings
import numpy as np
from tensorflow.keras.models import load_model
import joblib

#tensorboard
import numpy as np
from tensorflow.keras.models import Sequential
from tensorflow.keras import layers, regularizers
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.callbacks import EarlyStopping, TensorBoard
import datetime
import joblib
```

Feature Selection

We choose **overall_rating** of the players as the target variable because it represents a comprehensive measure of a player's performance, skill, and abilities. This rating integrates multiple attributes (e.g., speed, strength, technique), making it an ideal variable for predicting a player's overall capability or value in various analyses, such as performance evaluation or scouting in data-driven sports analytics. It provides a continuous numeric target suitable for regression model.

For the regression appraoch we will choose the following features and then set our target variable as overall rating

```
# Feature selection for regression task
X = df[['age', 'height_cm', 'weight_kgs', 'value_euro', 'wage_euro',
'crossing', 'finishing',
        'dribbling', 'ball_control', 'acceleration', 'sprint_speed',
'stamina', 'strength']]

# Target variable: Overall Rating
y = df['overall_rating']
```

Preprocessing

Preprocessing is essential for neural network regression tasks because it helps ensure that the input data is clean, consistent, and in the right format for the model to learn effectively. Techniques like normalization and scaling are needed to ensure that features are on similar scales, which prevents larger values from dominating the learning process. Additionally, handling missing or categorical data ensures that the model receives meaningful and complete inputs.

To make sure we dont have any missing values

```
# Handling missing data
X.fillna(X.mean(), inplace=True)
y.fillna(y.mean(), inplace=True)
```

```
/tmp/ipykernel_6489/1341224913.py:2: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame

See the caveats in the documentation:
https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#
returning-a-view-versus-a-copy
  X.fillna(X.mean(), inplace=True)
```

Scaling the data

StandardScaler is a preprocessing tool that standardizes features by removing the mean and scaling them to unit variance, resulting in a mean of 0 and a standard deviation of 1. It helps ensure that features are on a similar scale, improving model performance and training stability.

```
# Normalizing the data
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)
```

Train and test splitting the data for both the approaches

train_test_split divides the dataset into 80% for training and 20% for testing (to evaluate model performance), while random_state=42 ensures consistent data splits across runs for reproducibility. A consistent split means that using the same random_state will yield the same training and testing datasets every time you run the code, which is essential for reproducibility and debugging in machine learning experiments.

```
# Train-test splitting
X_train, X_test, y_train, y_test = train_test_split(X_scaled, y,
test_size=0.2, random_state=42)
```

Base Model

Types of Layers in our Base Model

- **Dense Layer**: Fully connected layer where each neuron receives input from all neurons in the previous layer.Captures complex relationships and patterns in the data by learning weights and biases.
- **Hidden Layer**: Any layer between the input and output layers, which can also be a dense layer.Processes inputs to extract features and representations, enabling the model to learn more abstract concepts.

Purpose of Each Layer

- **Dense Layer**: Used for both hidden and output layers to perform computations based on learned weights, making it essential for learning from data.
- **Hidden Layer**: Enhances the model's capacity to learn by adding depth, allowing for more complex transformations of the input data.

Name of Function	Use	Advantage
Linear Cost Function	Used for regression tasks to measure the difference between predicted and actual values.	Simple to compute; provides a clear gradient for optimization.
ReLU Activation Function	Used in hidden layers to introduce non-linearity and allow models to learn complex patterns.	Computationally efficient; mitigates the vanishing gradient problem.

Use Case	Pytorch	Keras
Model Training	Dynamic computation graphs allow for easy experimentation with architectures.	Simplified, user-friendly API for quick prototyping and model building.
Data Preprocessing	Built-in tools for handling tensors and datasets streamline data manipulation.	High-level preprocessing layers and utilities for easy data handling.
Performance Prediction	Flexible model design enables rapid iteration and optimization of predictive models.	Faster model deployment with integrated model-building and evaluation tools.
Feature Extraction	Pre-trained models can be fine-tuned for specific player metrics and performance indicators.	Offers integration with pre-trained models (e.g., in Applications module) for easy fine-tuning.
Visualization	Integration with libraries like Matplotlib for visualizing training progress and results.	Built-in support for TensorBoard to track metrics, loss, and performance graphs.

Pytorch approach

The below PyTorch model is a fully connected feedforward neural network designed for regression tasks, specifically to predict football players' overall performance ratings based on their attributes. The model consists of three layers: the input layer (fc1), which takes the player features and outputs 128 neurons, followed by a hidden layer (fc2) with 64 neurons, and finally, the output layer (fc3) that produces a single continuous value as the predicted rating.

ReLU activation functions are applied after the first two layers to introduce non-linearity, helping the model capture complex relationships in the data. No activation function is used in the output layer, as this is a regression task and we want to predict a continuous value.

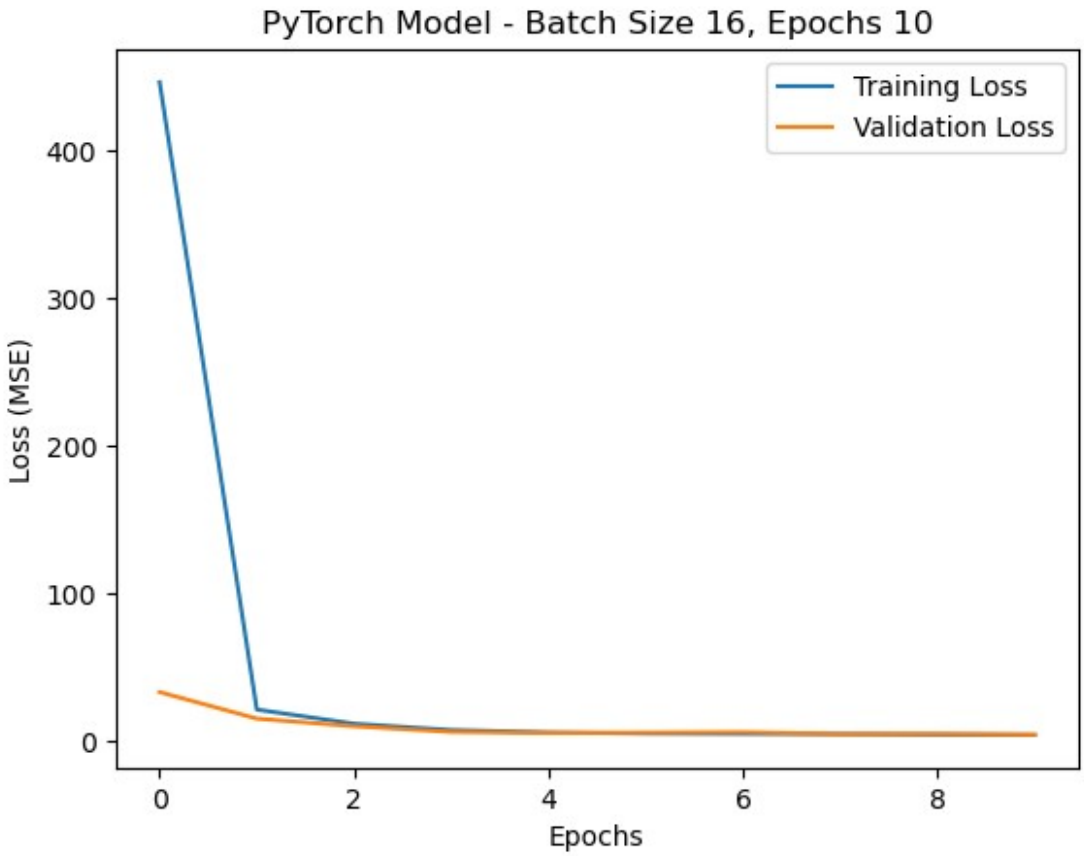
```
# Converting to PyTorch tensors
X_train_tensor = torch.tensor(X_train, dtype=torch.float32)
X_test_tensor = torch.tensor(X_test, dtype=torch.float32)
y_train_tensor = torch.tensor(y_train.values,
dtype=torch.float32).view(-1, 1)
y_test_tensor = torch.tensor(y_test.values,
dtype=torch.float32).view(-1, 1)

# Defining the PyTorch model
class PlayerPerformanceModel(nn.Module):
    def __init__(self):
        super(PlayerPerformanceModel, self).__init__()
        self.fc1 = nn.Linear(X_train.shape[1], 128)
        self.fc2 = nn.Linear(128, 64)
        self.fc3 = nn.Linear(64, 1)
        self.relu = nn.ReLU()

    def forward(self, x):
        x = self.relu(self.fc1(x))
        x = self.relu(self.fc2(x))
        x = self.fc3(x)  # No activation in the output layer for
regression
        return x

# Functioning to train the PyTorch model and track loss values
def train_pytorch_model(model, X_train, y_train, X_test, y_test,
epochs_list, batch_size_list):
    results = []

    for batch_size in batch_size_list:
        # Converting training data into batches
        train_data = torch.utils.data.TensorDataset(X_train, y_train)
        train_loader = torch.utils.data.DataLoader(dataset=train_data,
batch_size=batch_size, shuffle=True)

        for epochs in epochs_list:
            model = PlayerPerformanceModel()  # Reinitialize the model
            optimizer = optim.Adam(model.parameters(), lr=0.001)
            criterion = nn.MSELoss()  # Loss function for regression

            # store losses
            train_losses = []
            val_losses = []

            print(f"\nTraining with Batch Size: {batch_size}, Epochs:
{epochs}")
```

```
            # Training loop
            for epoch in range(epochs):
                model.train()  # Set the model to training mode
                epoch_train_loss = 0

                for batch_X, batch_y in train_loader:
                    outputs = model(batch_X)
                    loss = criterion(outputs, batch_y)

                    # Backward pass and optimization
                    optimizer.zero_grad()
                    loss.backward()
                    optimizer.step()

                    epoch_train_loss += loss.item()

                # Calculating the validation loss at the end of each
epoch
                model.eval()
                with torch.no_grad():
                    y_pred_test = model(X_test)
                    val_loss = criterion(y_pred_test, y_test).item()

                # Average train loss over batches and append to lists
                avg_epoch_train_loss = epoch_train_loss /
len(train_loader)
                train_losses.append(avg_epoch_train_loss)
                val_losses.append(val_loss)

                # Printing every 10th epoch
                if (epoch + 1) % 10 == 0 or epoch == epochs - 1:
                    print(f"Epoch [{epoch + 1}/{epochs}], Training
Loss: {avg_epoch_train_loss:.4f}, Validation Loss: {val_loss:.4f}")

            # Storing the results
            results.append({'Epochs': epochs, 'Batch Size':
batch_size, 'Training Loss': train_losses[-1], 'Validation Loss':
val_losses[-1]})

            # Ploting the training and validation loss curves
            plt.plot(train_losses, label='Training Loss')
            plt.plot(val_losses, label='Validation Loss')
            plt.xlabel('Epochs')
            plt.ylabel('Loss (MSE)')
            plt.title(f'PyTorch Model - Batch Size {batch_size},
Epochs {epochs}')
            plt.legend()
            plt.show()
```

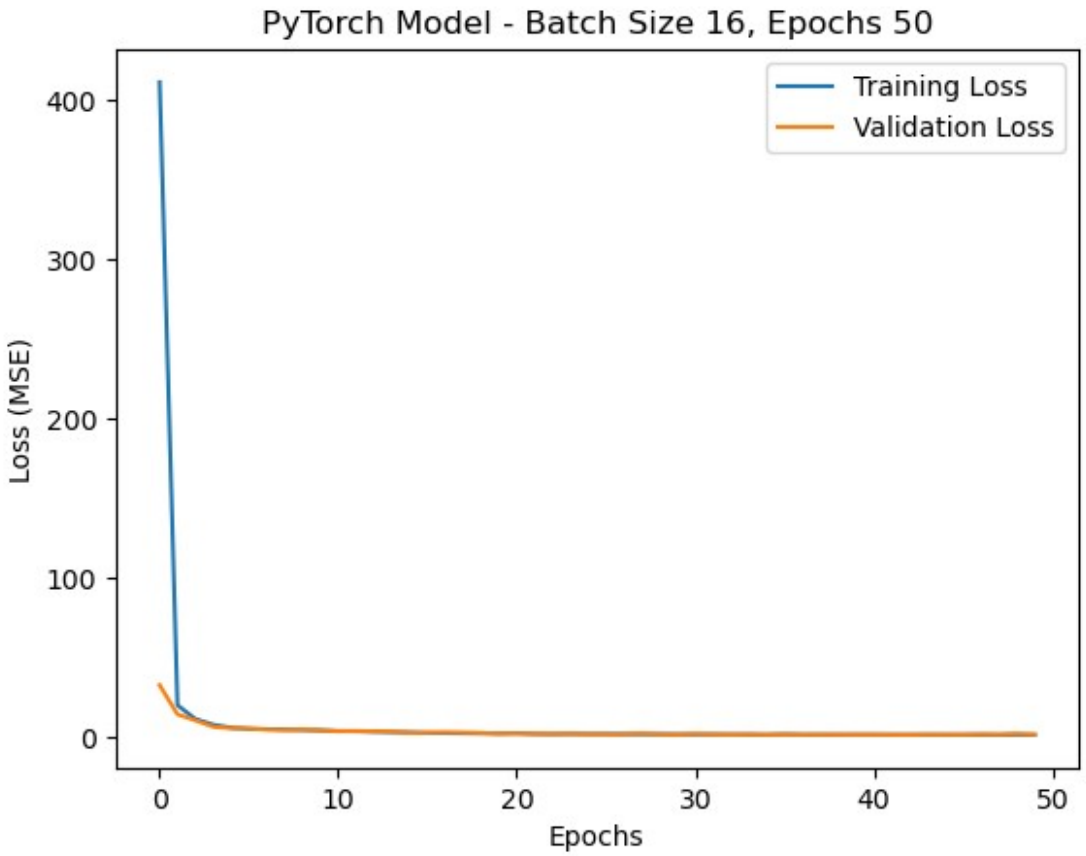
```
    return results
```

The primary goal of this function was to automate the training process, allowing us to evaluate the model's performance across multiple configurations. We used the Adam optimizer with Mean Squared Error (MSE) as our loss function for this regression task. By tracking both training and validation losses, we aimed to identify the best-performing model while preventing overfitting. Additionally, the function generates loss curves after each run, giving us clear visual feedback on the model's learning progress.

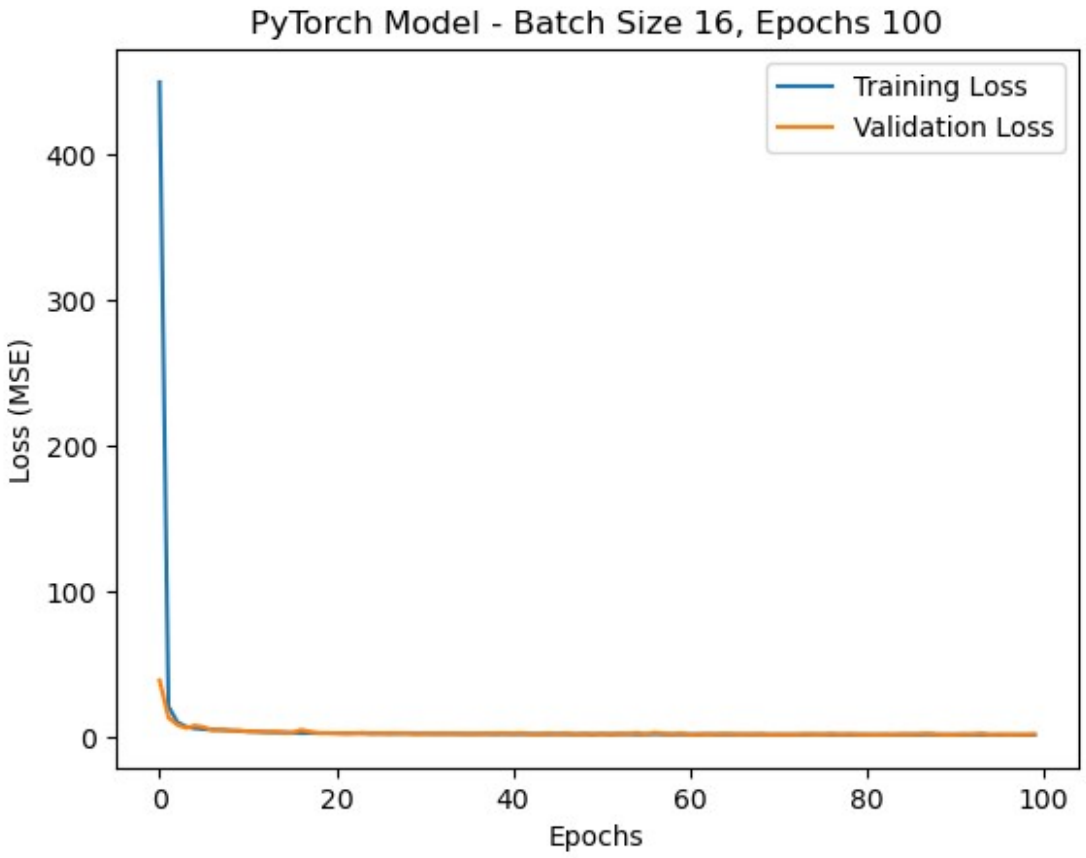
```
# Defining the different epochs and batch sizes to experiment with
epochs_list = [10, 50, 100]
batch_size_list = [16, 32, 64, 128]

# Train and collecting results
pytorch_results = train_pytorch_model(PlayerPerformanceModel(),
X_train_tensor, y_train_tensor, X_test_tensor, y_test_tensor,
epochs_list, batch_size_list)

# Converting the results to a DataFrame
pytorch_results_df = pd.DataFrame(pytorch_results)
pytorch_results_df


Training with Batch Size: 16, Epochs: 10
Epoch [10/10], Training Loss: 4.2540, Validation Loss: 4.4665
```



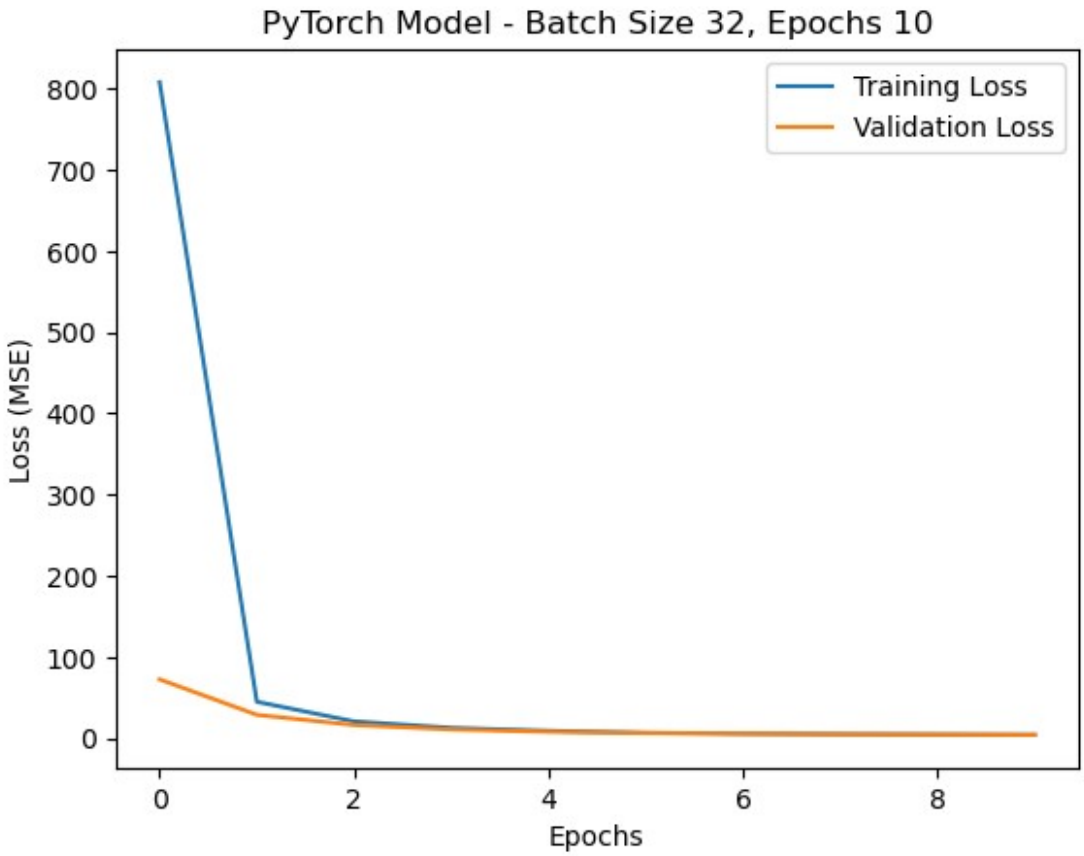

```
Training with Batch Size: 16, Epochs: 50
Epoch [10/50], Training Loss: 4.3191, Validation Loss: 4.4954
Epoch [20/50], Training Loss: 2.2921, Validation Loss: 2.2878
Epoch [30/50], Training Loss: 1.8302, Validation Loss: 1.7737
Epoch [40/50], Training Loss: 1.7504, Validation Loss: 1.6442
Epoch [50/50], Training Loss: 1.5655, Validation Loss: 1.8187
```



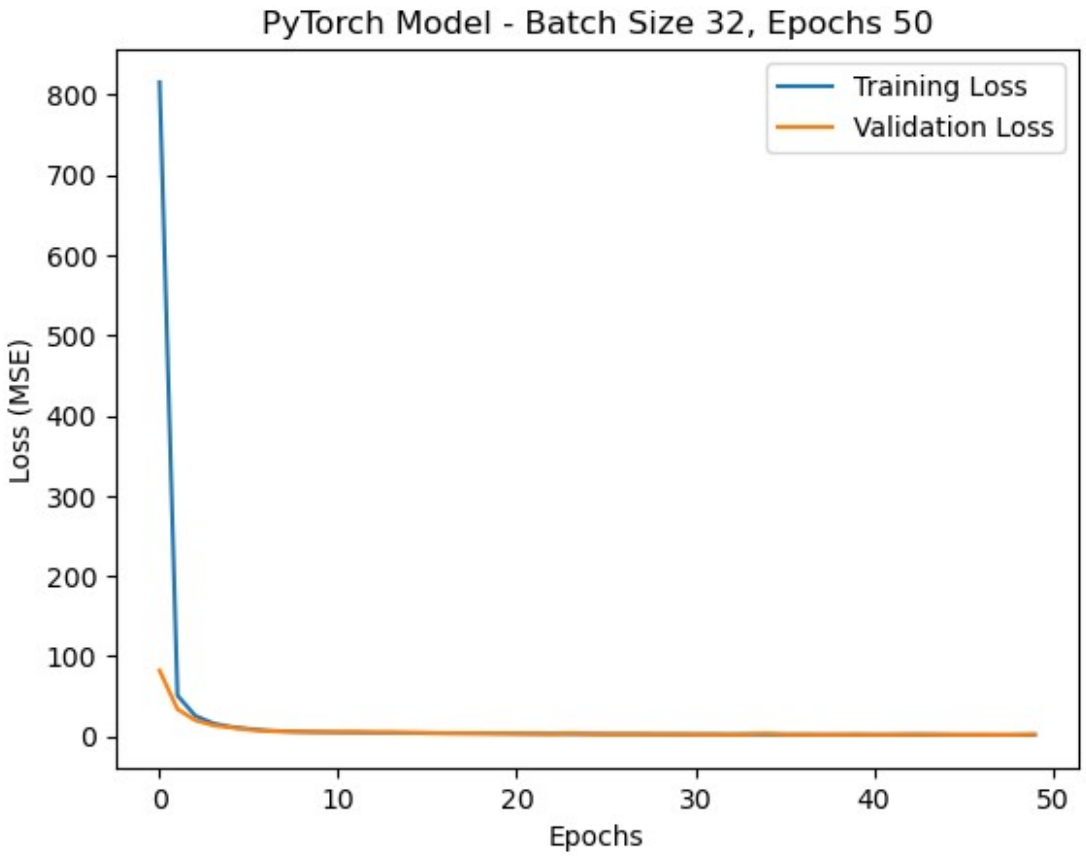

```
Training with Batch Size: 16, Epochs: 100
Epoch [10/100], Training Loss: 3.8316, Validation Loss: 4.2479
Epoch [20/100], Training Loss: 2.3779, Validation Loss: 2.4394
Epoch [30/100], Training Loss: 1.9100, Validation Loss: 1.7907
Epoch [40/100], Training Loss: 1.7438, Validation Loss: 2.3607
Epoch [50/100], Training Loss: 1.4323, Validation Loss: 1.8245
Epoch [60/100], Training Loss: 1.5360, Validation Loss: 2.2458
Epoch [70/100], Training Loss: 1.3939, Validation Loss: 1.3234
Epoch [80/100], Training Loss: 1.3571, Validation Loss: 1.5398
Epoch [90/100], Training Loss: 1.3873, Validation Loss: 1.2453
Epoch [100/100], Training Loss: 1.2012, Validation Loss: 1.6882
```



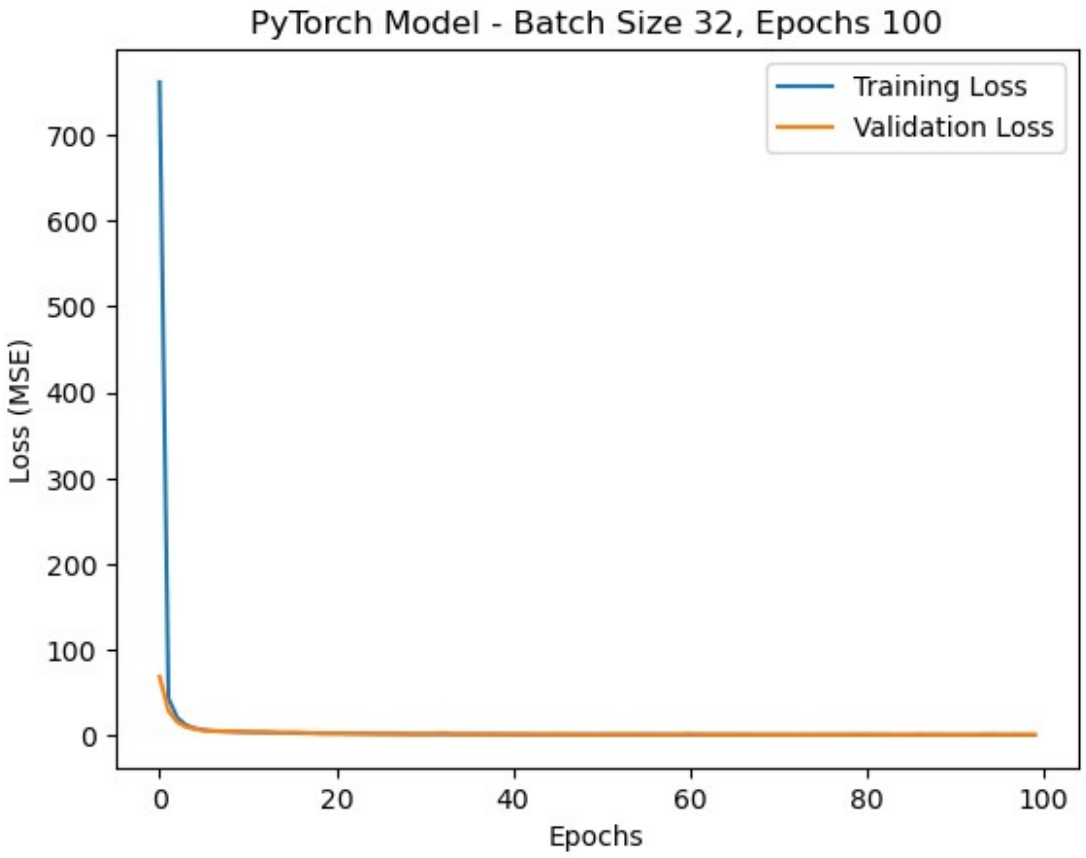

```
Training with Batch Size: 32, Epochs: 10
Epoch [10/10], Training Loss: 4.6627, Validation Loss: 4.4346
```



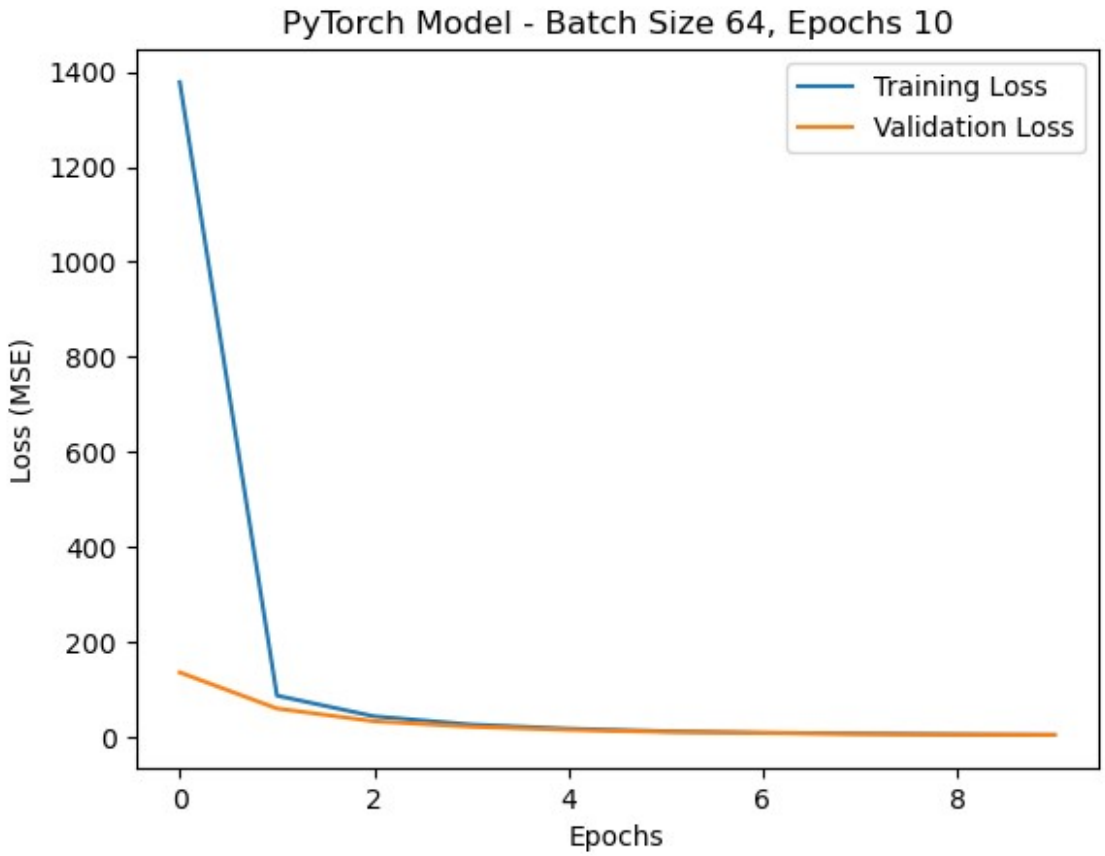

```
Training with Batch Size: 32, Epochs: 50
Epoch [10/50], Training Loss: 5.0473, Validation Loss: 5.0121
Epoch [20/50], Training Loss: 3.2448, Validation Loss: 3.4395
Epoch [30/50], Training Loss: 2.1814, Validation Loss: 2.6041
Epoch [40/50], Training Loss: 1.8023, Validation Loss: 2.2259
Epoch [50/50], Training Loss: 1.5932, Validation Loss: 2.3816
```



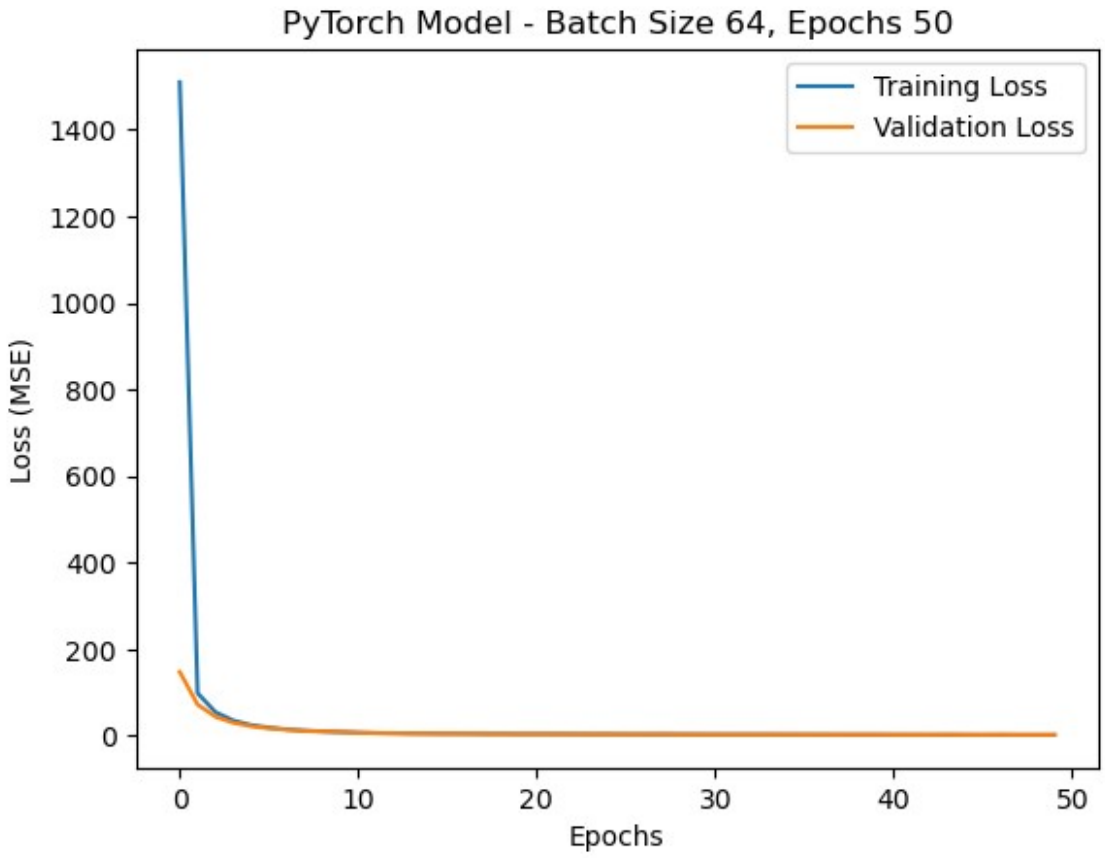

```
Training with Batch Size: 32, Epochs: 100
Epoch [10/100], Training Loss: 4.6386, Validation Loss: 4.4379
Epoch [20/100], Training Loss: 2.7564, Validation Loss: 2.7640
Epoch [30/100], Training Loss: 1.8554, Validation Loss: 1.9572
Epoch [40/100], Training Loss: 1.6105, Validation Loss: 1.7932
Epoch [50/100], Training Loss: 1.4767, Validation Loss: 1.5734
Epoch [60/100], Training Loss: 1.2734, Validation Loss: 1.5607
Epoch [70/100], Training Loss: 1.2769, Validation Loss: 1.6026
Epoch [80/100], Training Loss: 1.2157, Validation Loss: 1.6498
Epoch [90/100], Training Loss: 1.1561, Validation Loss: 1.4373
Epoch [100/100], Training Loss: 1.0926, Validation Loss: 1.4526
```



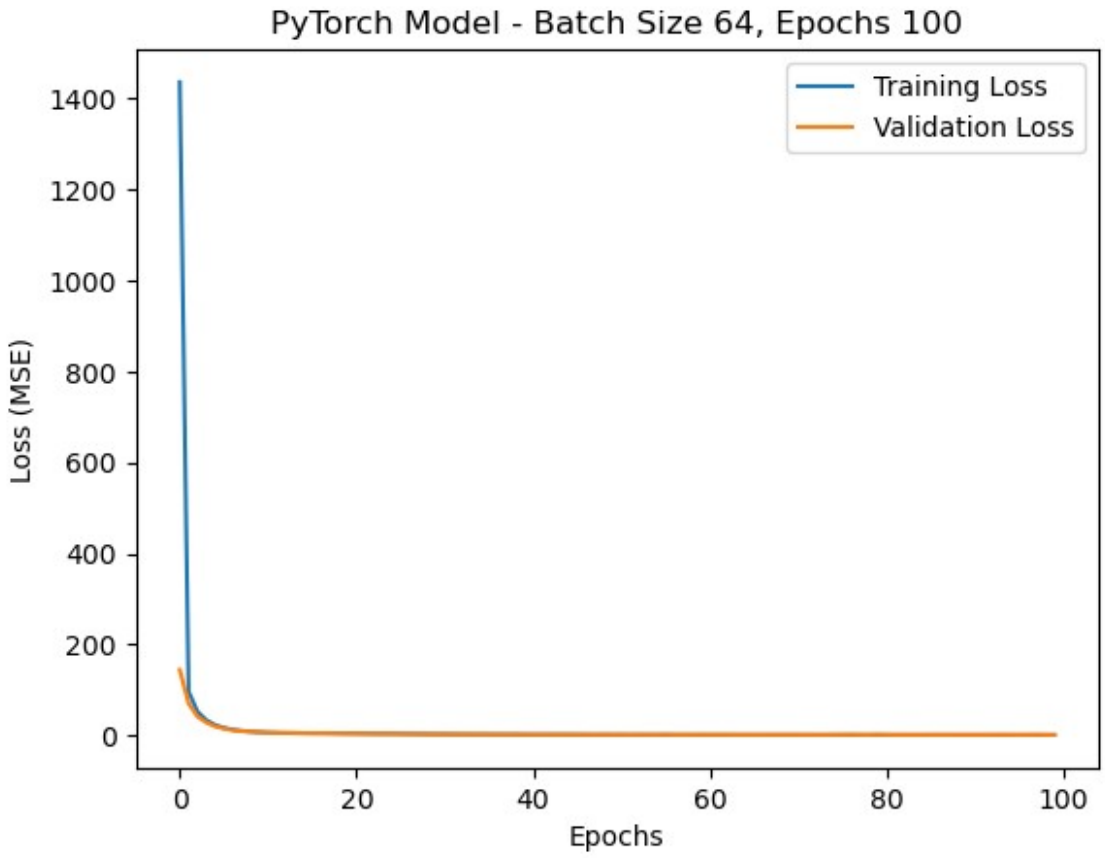

```
Training with Batch Size: 64, Epochs: 10
Epoch [10/10], Training Loss: 6.2818, Validation Loss: 6.2055
```



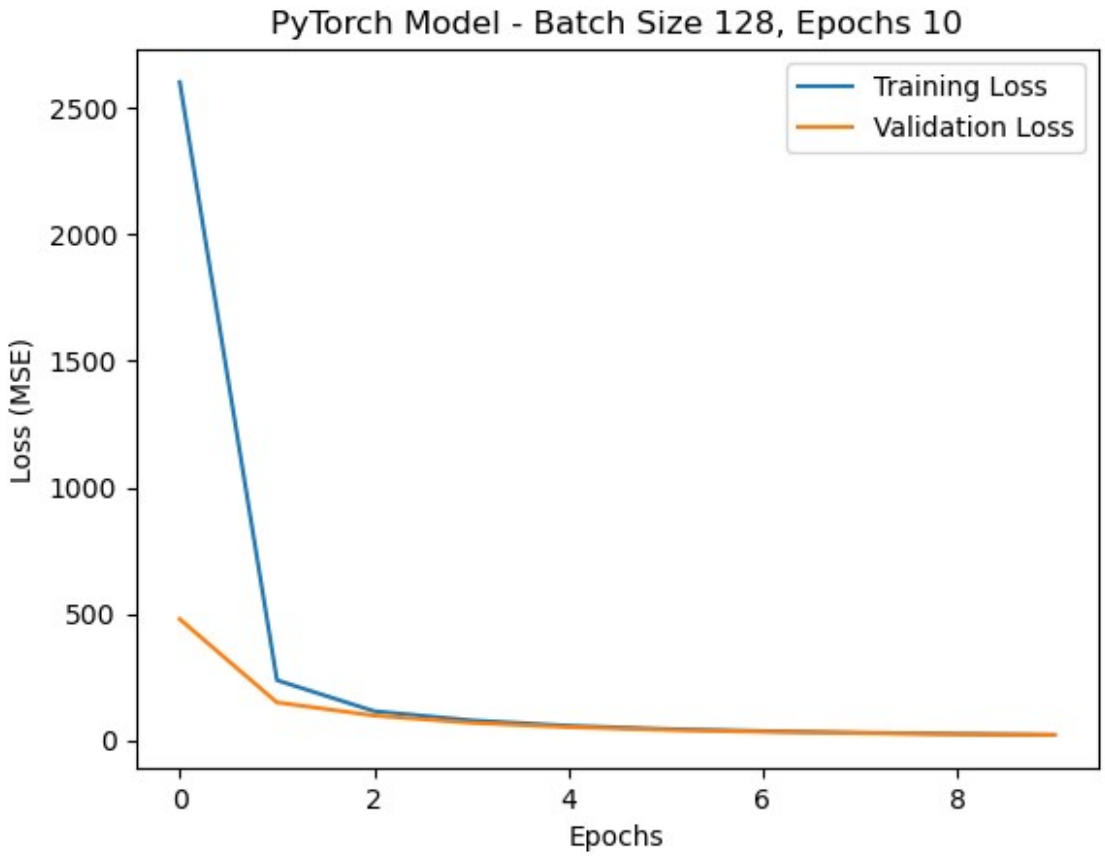

```
Training with Batch Size: 64, Epochs: 50
Epoch [10/50], Training Loss: 8.8716, Validation Loss: 8.4542
Epoch [20/50], Training Loss: 4.0693, Validation Loss: 4.2959
Epoch [30/50], Training Loss: 3.2601, Validation Loss: 3.2945
Epoch [40/50], Training Loss: 2.7173, Validation Loss: 3.1742
Epoch [50/50], Training Loss: 2.2045, Validation Loss: 2.5601
```



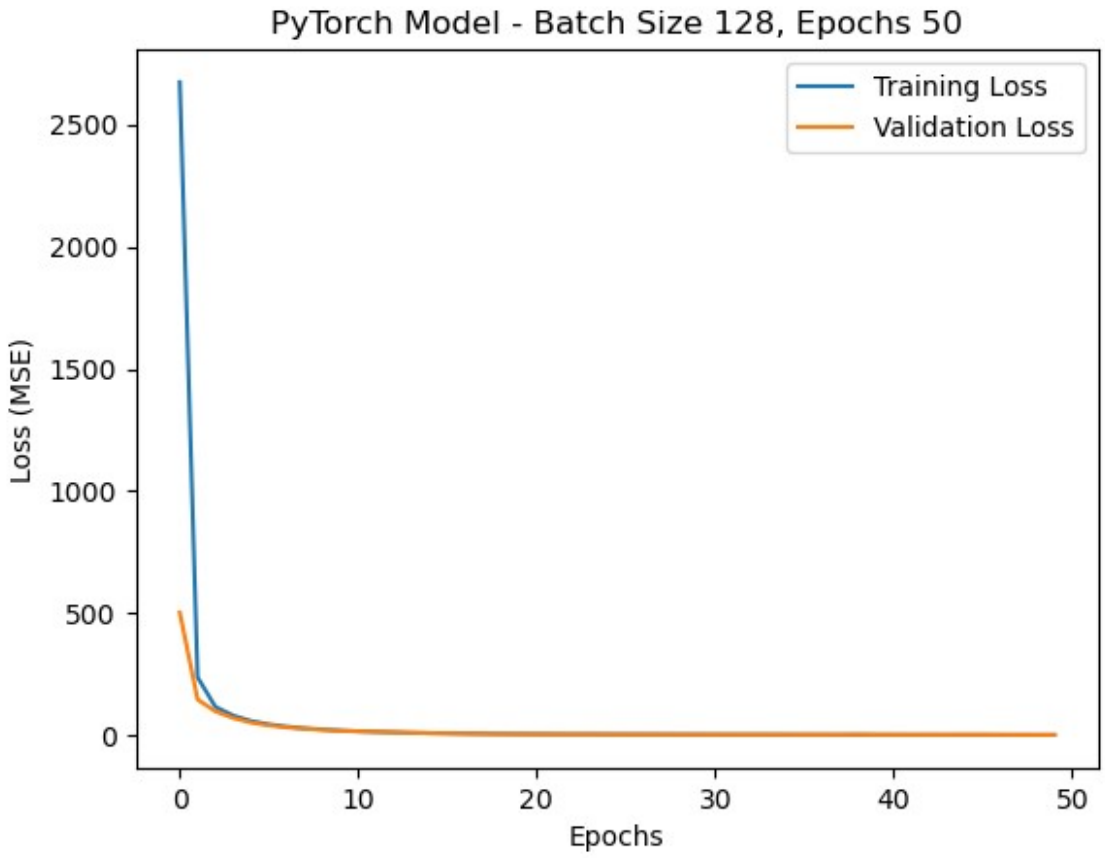

```
Training with Batch Size: 64, Epochs: 100
Epoch [10/100], Training Loss: 7.2343, Validation Loss: 6.9667
Epoch [20/100], Training Loss: 4.1251, Validation Loss: 4.2265
Epoch [30/100], Training Loss: 2.7426, Validation Loss: 2.8069
Epoch [40/100], Training Loss: 2.0141, Validation Loss: 2.2192
Epoch [50/100], Training Loss: 1.6996, Validation Loss: 1.9912
Epoch [60/100], Training Loss: 1.5181, Validation Loss: 1.8937
Epoch [70/100], Training Loss: 1.3921, Validation Loss: 1.6944
Epoch [80/100], Training Loss: 1.3945, Validation Loss: 2.0563
Epoch [90/100], Training Loss: 1.3332, Validation Loss: 1.5292
Epoch [100/100], Training Loss: 1.2643, Validation Loss: 1.3823
```



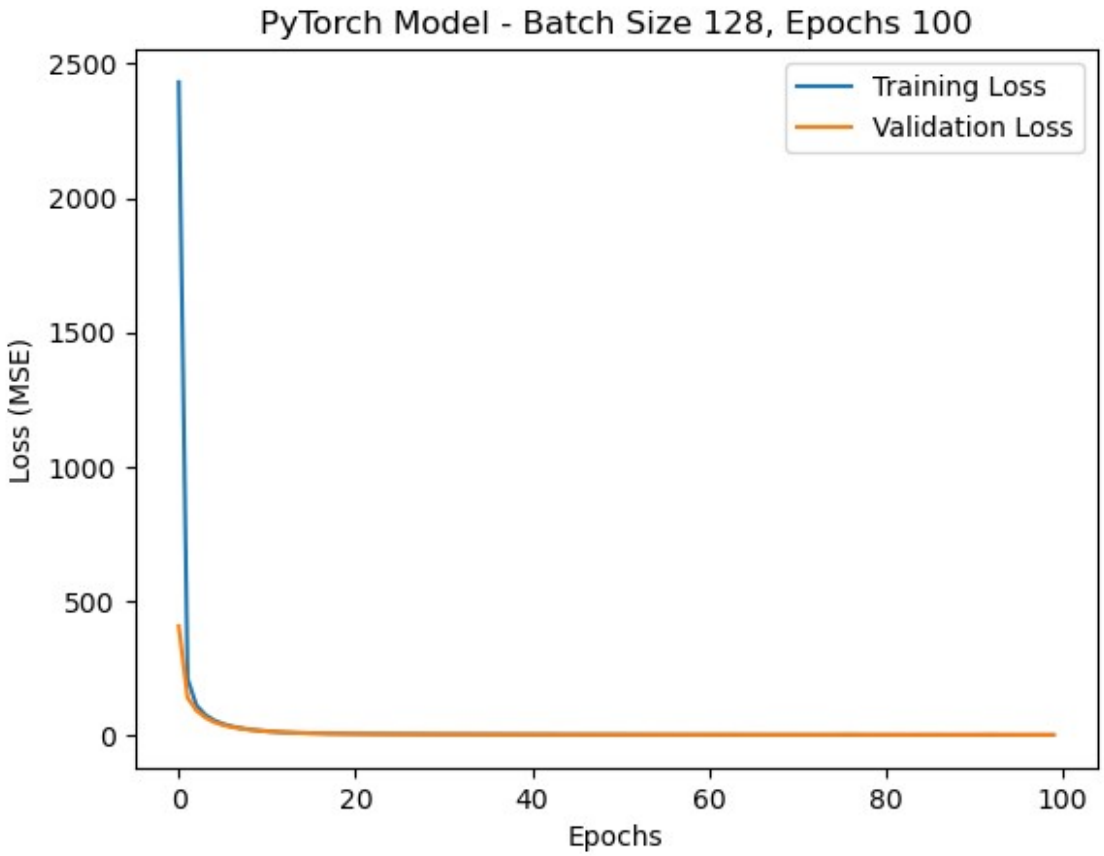

```
Training with Batch Size: 128, Epochs: 10
Epoch [10/10], Training Loss: 20.9826, Validation Loss: 20.9833
```




```
Training with Batch Size: 128, Epochs: 50
Epoch [10/50], Training Loss: 20.5174, Validation Loss: 19.8154
Epoch [20/50], Training Loss: 6.1739, Validation Loss: 6.3322
Epoch [30/50], Training Loss: 4.5592, Validation Loss: 4.6197
Epoch [40/50], Training Loss: 3.6489, Validation Loss: 3.7752
Epoch [50/50], Training Loss: 2.9080, Validation Loss: 3.1837
```




```
Training with Batch Size: 128, Epochs: 100
Epoch [10/100], Training Loss: 17.9095, Validation Loss: 17.5570
Epoch [20/100], Training Loss: 5.6039, Validation Loss: 5.6347
Epoch [30/100], Training Loss: 3.9738, Validation Loss: 4.0903
Epoch [40/100], Training Loss: 3.1880, Validation Loss: 3.7118
Epoch [50/100], Training Loss: 2.6738, Validation Loss: 3.0542
Epoch [60/100], Training Loss: 2.2862, Validation Loss: 2.5019
Epoch [70/100], Training Loss: 2.0241, Validation Loss: 2.2225
Epoch [80/100], Training Loss: 1.8028, Validation Loss: 2.1393
Epoch [90/100], Training Loss: 1.6846, Validation Loss: 2.0532
Epoch [100/100], Training Loss: 1.5776, Validation Loss: 1.8566
```



	Epochs	Batch Size	Training Loss	Validation Loss
0	10	16	4.253976	4.466458
1	50	16	1.565505	1.818725
2	100	16	1.201197	1.688236
3	10	32	4.662750	4.434649
4	50	32	1.593202	2.381625
5	100	32	1.092650	1.452606
6	10	64	6.281777	6.205473
7	50	64	2.204512	2.560088
8	100	64	1.264285	1.382258
9	10	128	20.982613	20.983322
10	50	128	2.908017	3.183661
11	100	128	1.577614	1.856592

Keras approach

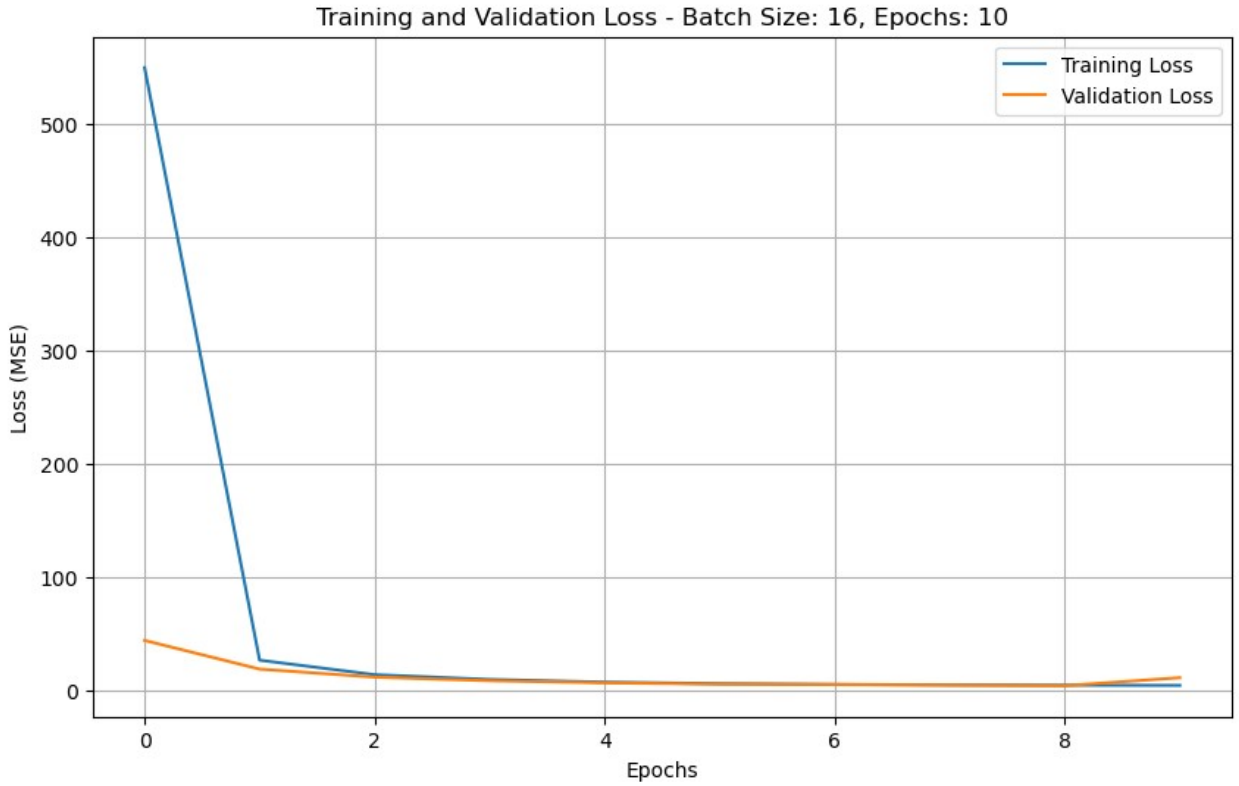
```
# Function to train Keras model with progress updates and plot the
training/validation loss curves
def train_keras_model(X_train, y_train, X_test, y_test, epochs_list,
batch_size_list):
    results = []

    for batch_size in batch_size_list:
```

```
        for epochs in epochs_list:
            print(f"\nTraining with Batch Size: {batch_size}, Epochs:
{epochs}")

            # Building a simple regression model
            model = models.Sequential([
                layers.Dense(128, activation='relu',
input_shape=(X_train.shape[1],)),
                layers.Dense(64, activation='relu'),
                layers.Dense(1)  # Output layer for regression
            ])

            # Compiling the model
            model.compile(optimizer='adam', loss='mse',
metrics=[tf.keras.metrics.RootMeanSquaredError()])

            # Training the model and printing the progress
            history = model.fit(X_train, y_train,
validation_split=0.2, epochs=epochs, batch_size=batch_size, verbose=0)

            # Printing the loss at the end of each epoch
            for epoch in range(epochs):
                if (epoch + 1) % 10 == 0 or epoch == epochs - 1:
                    avg_loss = history.history['loss'][epoch]
                    print(f"Epoch [{epoch + 1}/{epochs}], Loss:
{avg_loss:.4f}")

            # Evaluating the model on the test set
            test_loss, test_rmse = model.evaluate(X_test, y_test,
verbose=0)
            results.append({'Epochs': epochs, 'Batch Size':
batch_size, 'MSE': test_loss, 'RMSE': test_rmse})
            print(f"Completed training with Batch Size: {batch_size},
Epochs: {epochs}. Test MSE: {test_loss:.4f}, Test RMSE:
{test_rmse:.4f}")

            # Ploting the training and validation loss curves
            plt.figure(figsize=(10, 6))
            plt.plot(history.history['loss'], label='Training Loss')
            plt.plot(history.history['val_loss'], label='Validation
Loss')
            plt.title(f'Training and Validation Loss - Batch Size:
{batch_size}, Epochs: {epochs}')
            plt.xlabel('Epochs')
            plt.ylabel('Loss (MSE)')
            plt.legend()
            plt.grid(True)
            plt.show()

    return results
```

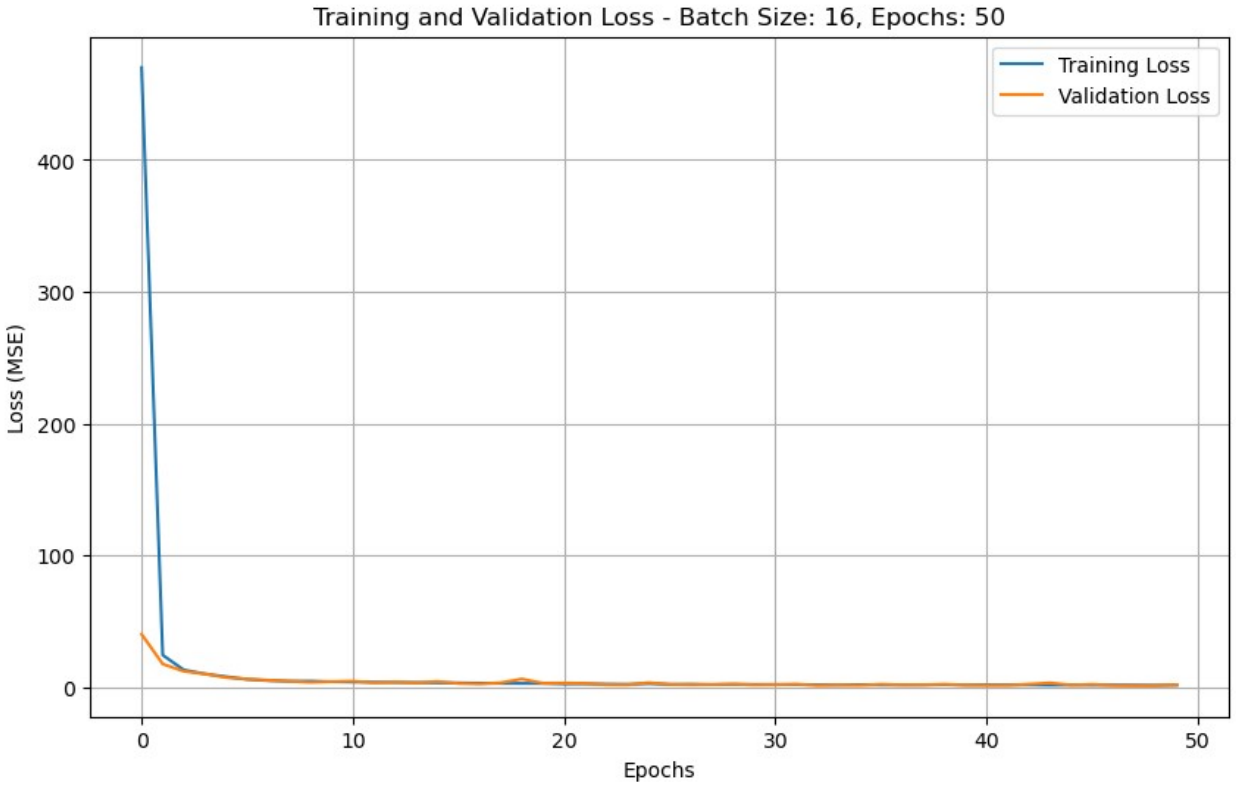
This function is designed to train a Keras regression model across various batch sizes and epochs, with the goal of identifying the optimal parameters for player performance prediction. The function builds a simple neural network with two hidden layers, using ReLU activations for non-linearity and Mean Squared Error (MSE) as the loss function. Throughout the training process, progress updates are provided, and both training and validation loss curves are plotted for each configuration. By evaluating the model on the test set, we gather results like MSE and Root Mean Squared Error (RMSE) to determine the model's effectiveness, helping us choose the best setup for further model refinement.

```
# Defining different epochs and batch sizes to experiment with
epochs_list = [10, 50, 100]
batch_size_list = [16, 32, 64, 128]

# Training and collecting results
keras_results = train_keras_model(X_train, y_train, X_test, y_test,
epochs_list, batch_size_list)

# Converting the results to a DataFrame for easy viewing
keras_results_df = pd.DataFrame(keras_results)
keras_results_df
```

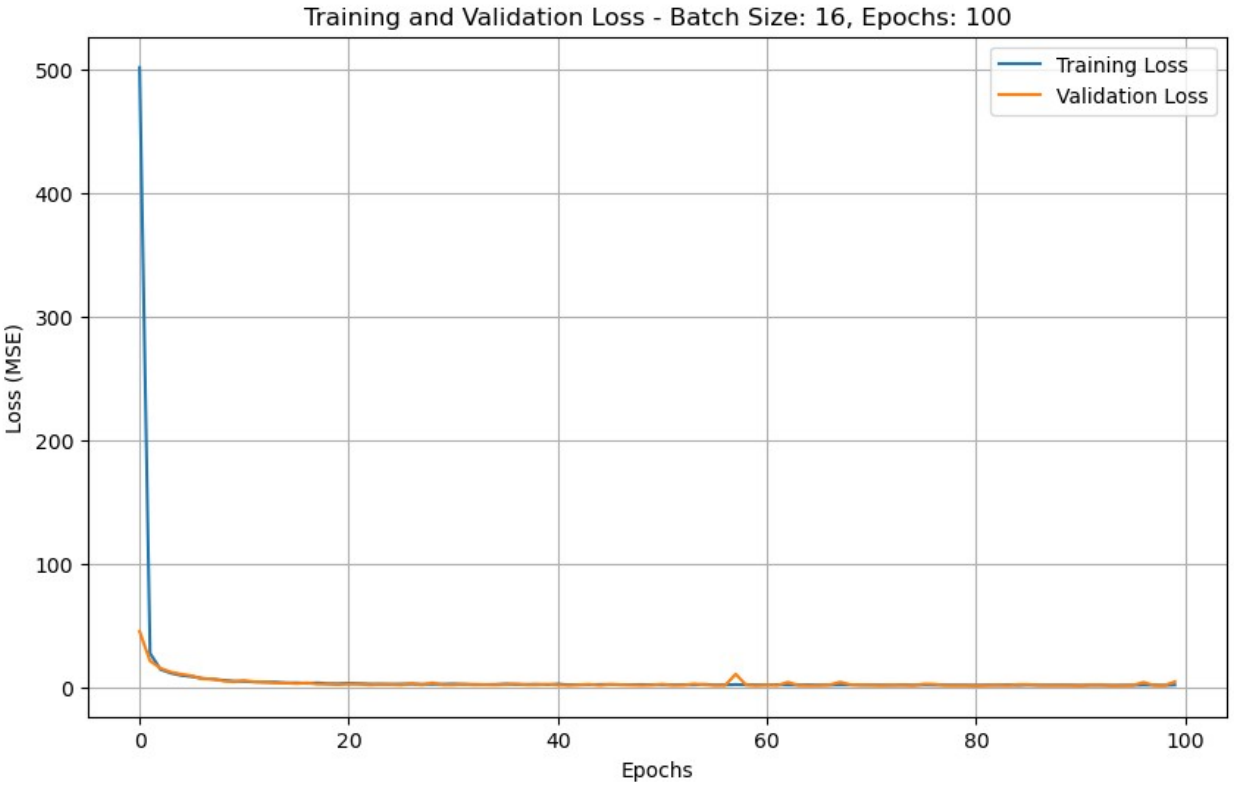
```
Training with Batch Size: 16, Epochs: 10

/home/unina/anaconda3/lib/python3.12/site-packages/keras/src/layers/
core/dense.py:87: UserWarning: Do not pass an
`input_shape`/`input_dim` argument to a layer. When using Sequential
models, prefer using an `Input(shape)` object as the first layer in
the model instead.
  super().__init__(activity_regularizer=activity_regularizer,
**kwargs)
WARNING: All log messages before absl::InitializeLog() is called are
written to STDERR
I0000 00:00:1728894978.612256    6489 cuda_executor.cc:1015]
successful NUMA node read from SysFS had negative value (-1), but
there must be at least one NUMA node, so returning NUMA node zero. See
more at
https://github.com/torvalds/linux/blob/v6.0/Documentation/ABI/testing/
sysfs-bus-pci#L344-L355
2024-10-14 10:36:18.614732: W
tensorflow/core/common_runtime/gpu/gpu_device.cc:2343] Cannot dlopen
some GPU libraries. Please make sure the missing libraries mentioned
above are installed properly if you would like to use GPU. Follow the
guide at https://www.tensorflow.org/install/gpu for how to download
and setup the required libraries for your platform.
Skipping registering GPU devices...

Epoch [10/10], Loss: 4.7738
Completed training with Batch Size: 16, Epochs: 10. Test MSE: 12.2087,
Test RMSE: 3.4941
```



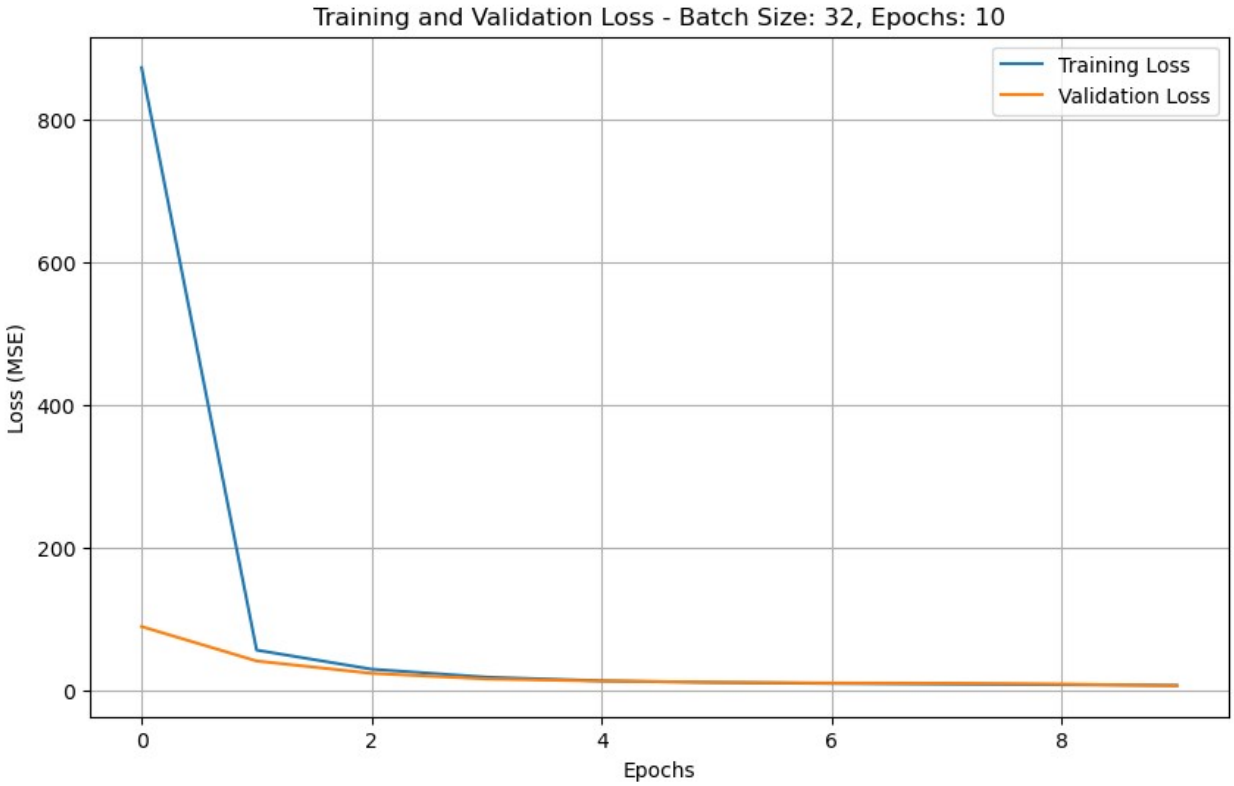

```
Training with Batch Size: 16, Epochs: 50
Epoch [10/50], Loss: 4.4685
Epoch [20/50], Loss: 3.1751
Epoch [30/50], Loss: 2.3903
Epoch [40/50], Loss: 1.9522
Epoch [50/50], Loss: 1.8627
Completed training with Batch Size: 16, Epochs: 50. Test MSE: 2.2612,
Test RMSE: 1.5037
```



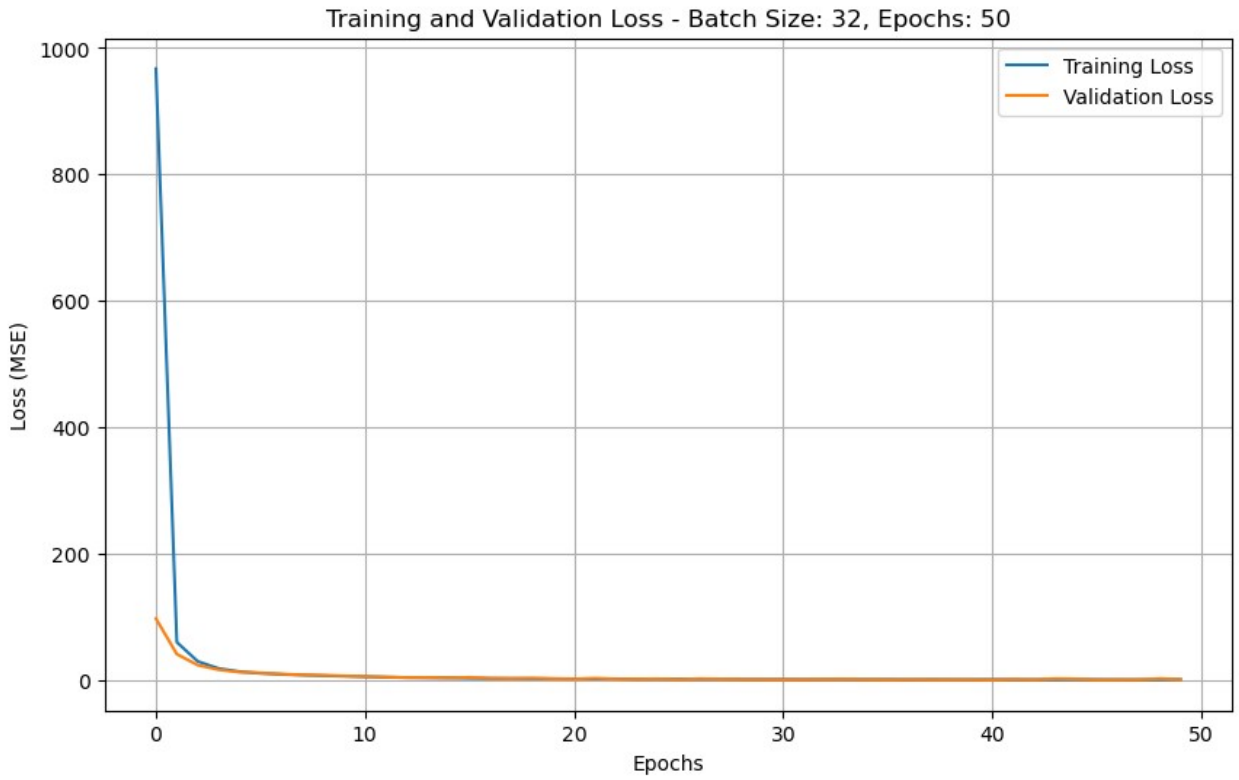

```
Training with Batch Size: 16, Epochs: 100
Epoch [10/100], Loss: 4.6516
Epoch [20/100], Loss: 2.5761
Epoch [30/100], Loss: 2.2712
Epoch [40/100], Loss: 1.8331
Epoch [50/100], Loss: 1.7695
Epoch [60/100], Loss: 1.6095
Epoch [70/100], Loss: 1.4573
Epoch [80/100], Loss: 1.4259
Epoch [90/100], Loss: 1.4255
Epoch [100/100], Loss: 1.5662
Completed training with Batch Size: 16, Epochs: 100. Test MSE: 4.4645,
Test RMSE: 2.1129
```



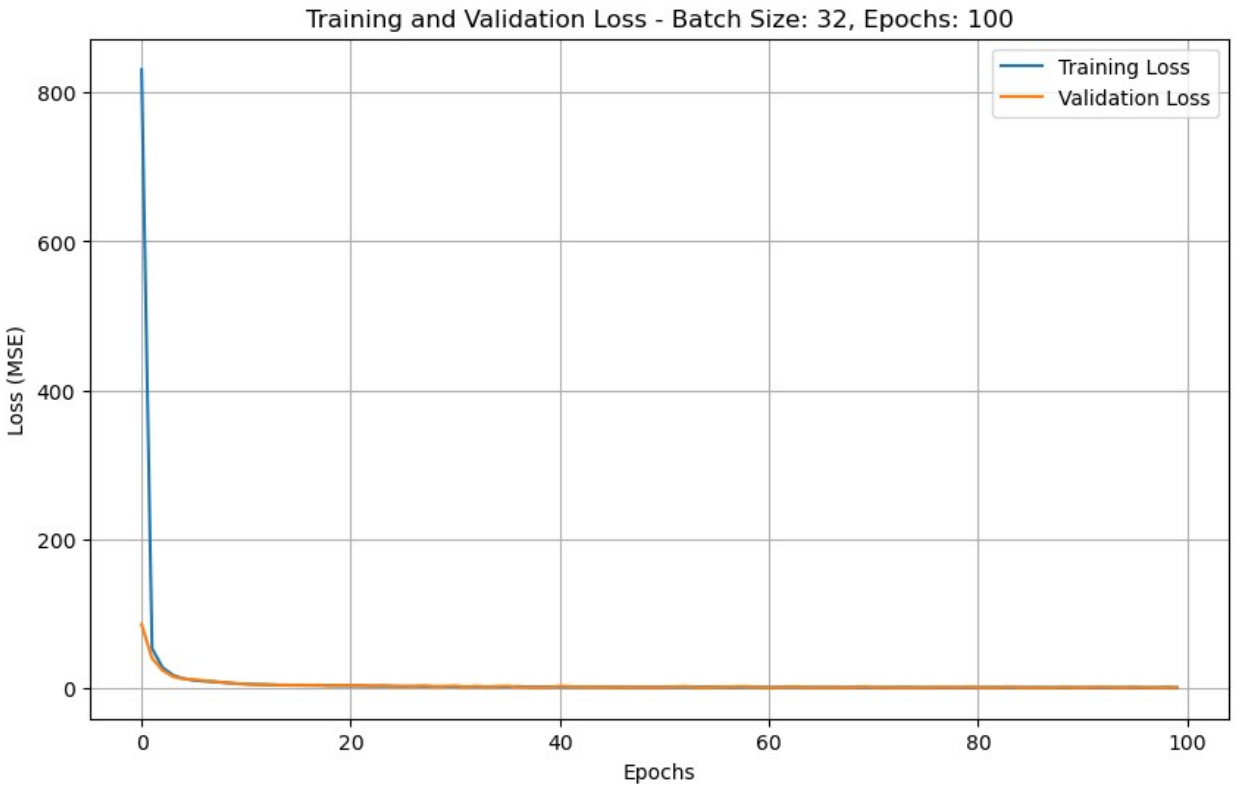

```
Training with Batch Size: 32, Epochs: 10
Epoch [10/10], Loss: 7.4379
Completed training with Batch Size: 32, Epochs: 10. Test MSE: 6.9861,
Test RMSE: 2.6431
```



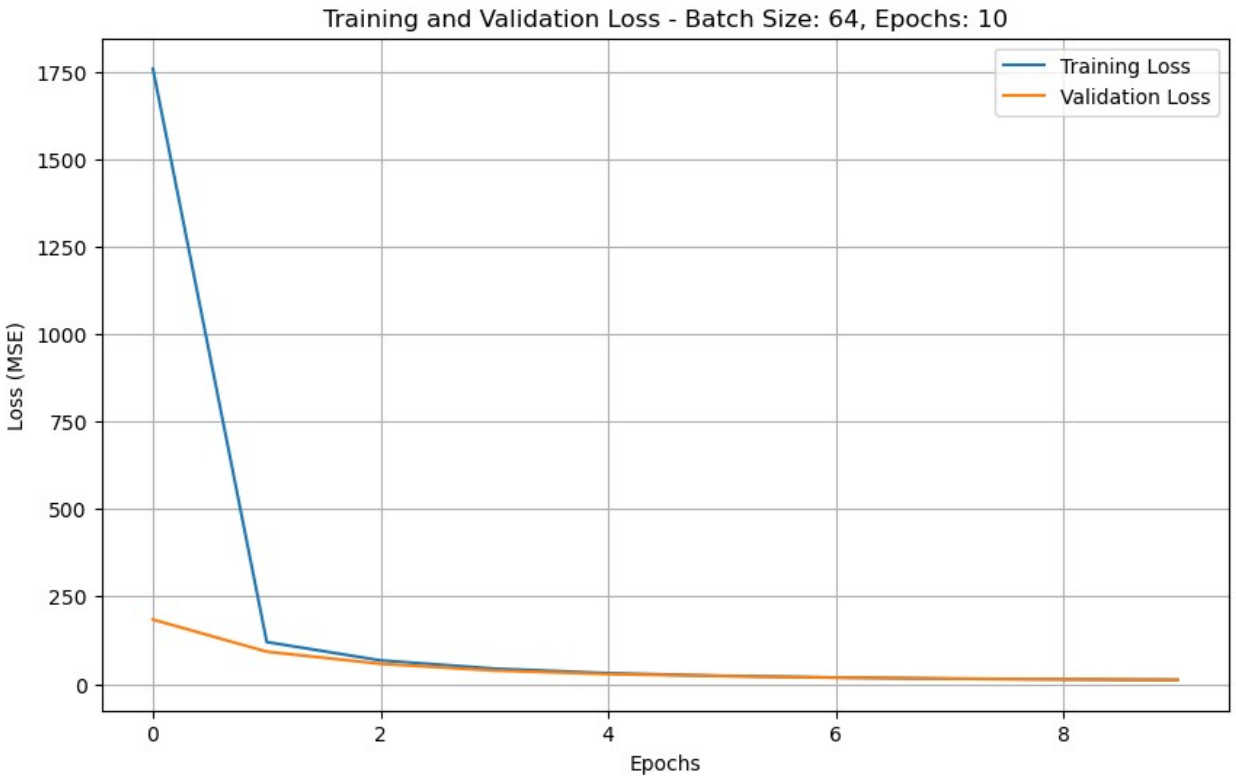

```
Training with Batch Size: 32, Epochs: 50
Epoch [10/50], Loss: 7.2796
Epoch [20/50], Loss: 3.3041
Epoch [30/50], Loss: 2.2070
Epoch [40/50], Loss: 1.9680
Epoch [50/50], Loss: 2.1641
Completed training with Batch Size: 32, Epochs: 50. Test MSE: 2.3549,
Test RMSE: 1.5346
```



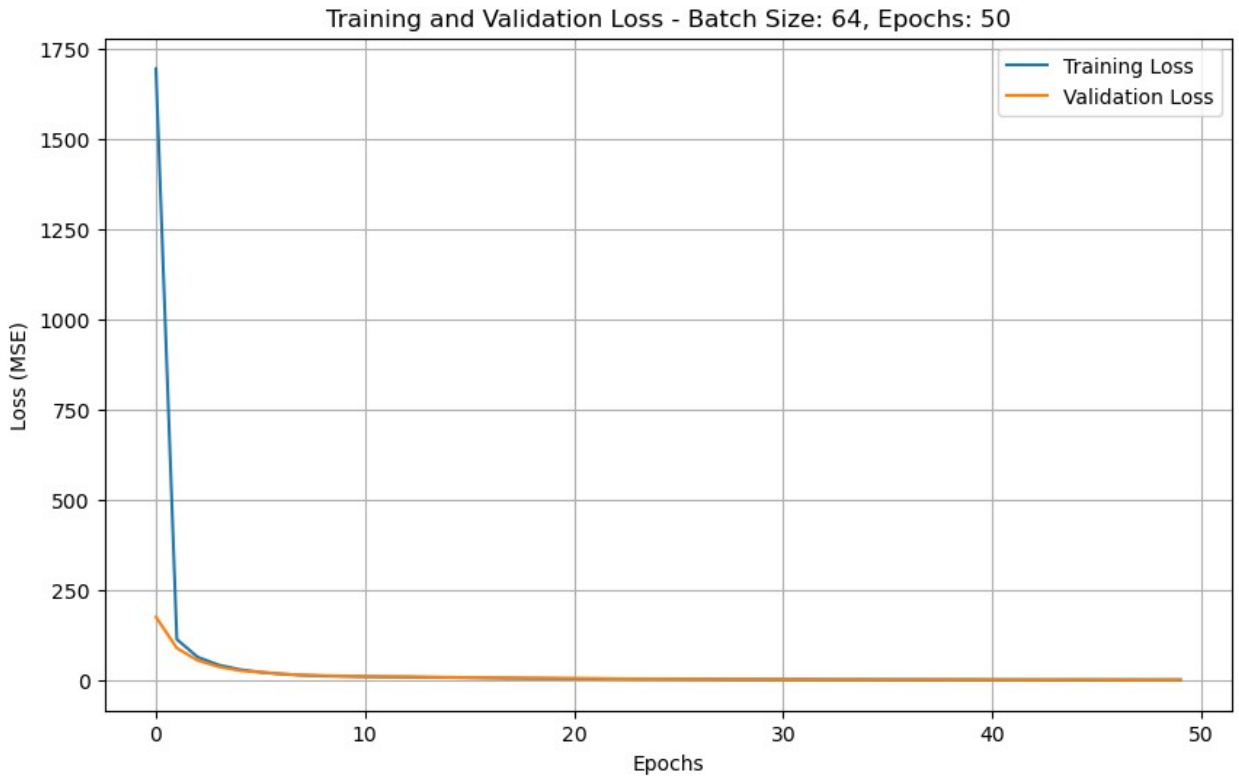

```
Training with Batch Size: 32, Epochs: 100
Epoch [10/100], Loss: 6.7384
Epoch [20/100], Loss: 3.7701
Epoch [30/100], Loss: 2.7800
Epoch [40/100], Loss: 2.1985
Epoch [50/100], Loss: 1.9752
Epoch [60/100], Loss: 1.7967
Epoch [70/100], Loss: 1.6651
Epoch [80/100], Loss: 1.6009
Epoch [90/100], Loss: 1.6790
Epoch [100/100], Loss: 1.3566
Completed training with Batch Size: 32, Epochs: 100. Test MSE: 1.9811,
Test RMSE: 1.4075
```



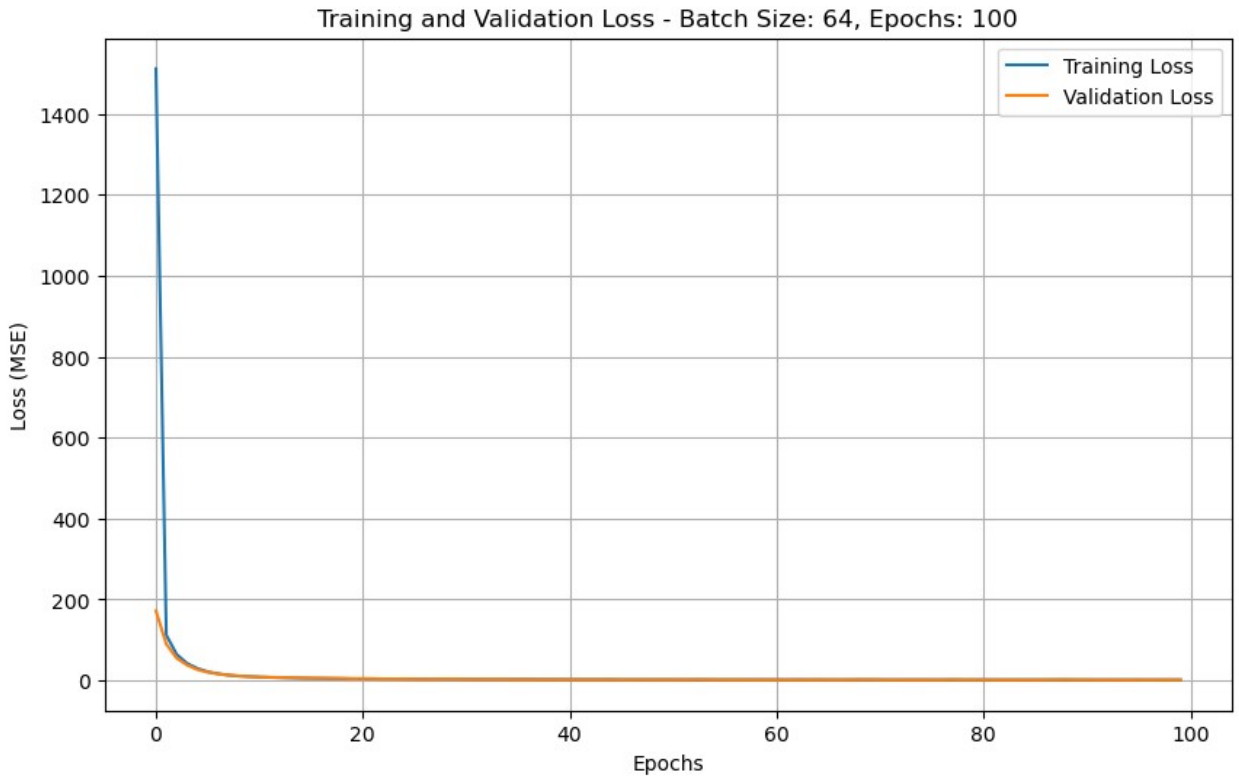

```
Training with Batch Size: 64, Epochs: 10
Epoch [10/10], Loss: 11.6595
Completed training with Batch Size: 64, Epochs: 10. Test MSE: 11.8237,
Test RMSE: 3.4386
```



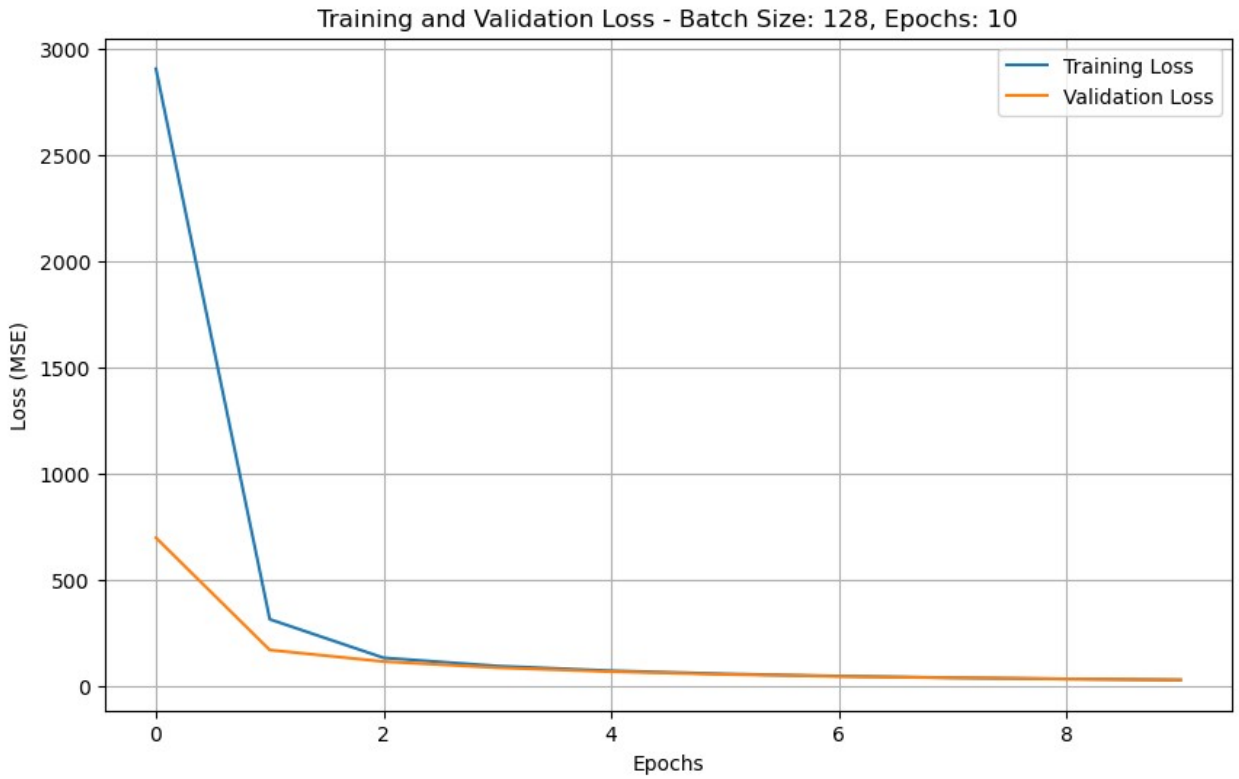

```
Training with Batch Size: 64, Epochs: 50
Epoch [10/50], Loss: 11.7131
Epoch [20/50], Loss: 5.8224
Epoch [30/50], Loss: 3.2539
Epoch [40/50], Loss: 2.5201
Epoch [50/50], Loss: 2.0088
Completed training with Batch Size: 64, Epochs: 50. Test MSE: 2.4732,
Test RMSE: 1.5726
```



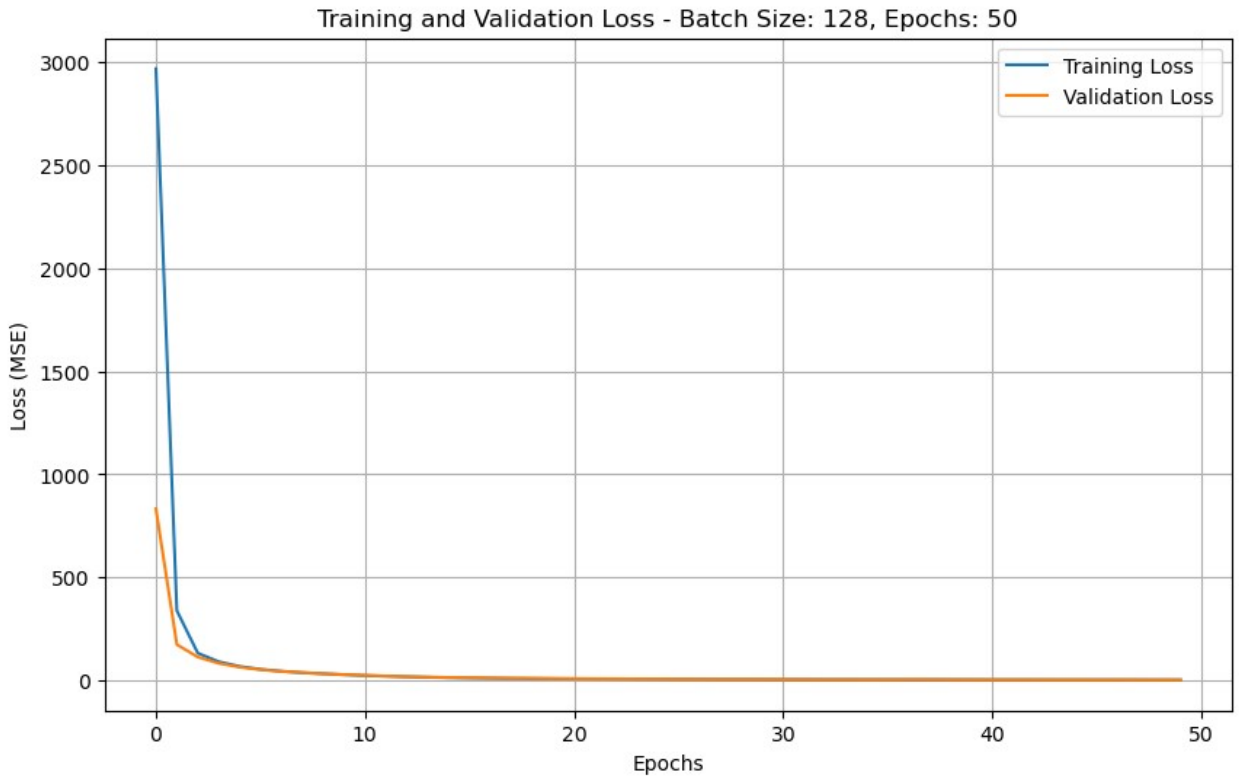

```
Training with Batch Size: 64, Epochs: 100
Epoch [10/100], Loss: 9.5960
Epoch [20/100], Loss: 4.2706
Epoch [30/100], Loss: 3.0498
Epoch [40/100], Loss: 2.4903
Epoch [50/100], Loss: 2.0765
Epoch [60/100], Loss: 2.0069
Epoch [70/100], Loss: 1.7927
Epoch [80/100], Loss: 1.6135
Epoch [90/100], Loss: 1.6247
Epoch [100/100], Loss: 1.5503
Completed training with Batch Size: 64, Epochs: 100. Test MSE: 1.8878,
Test RMSE: 1.3740
```



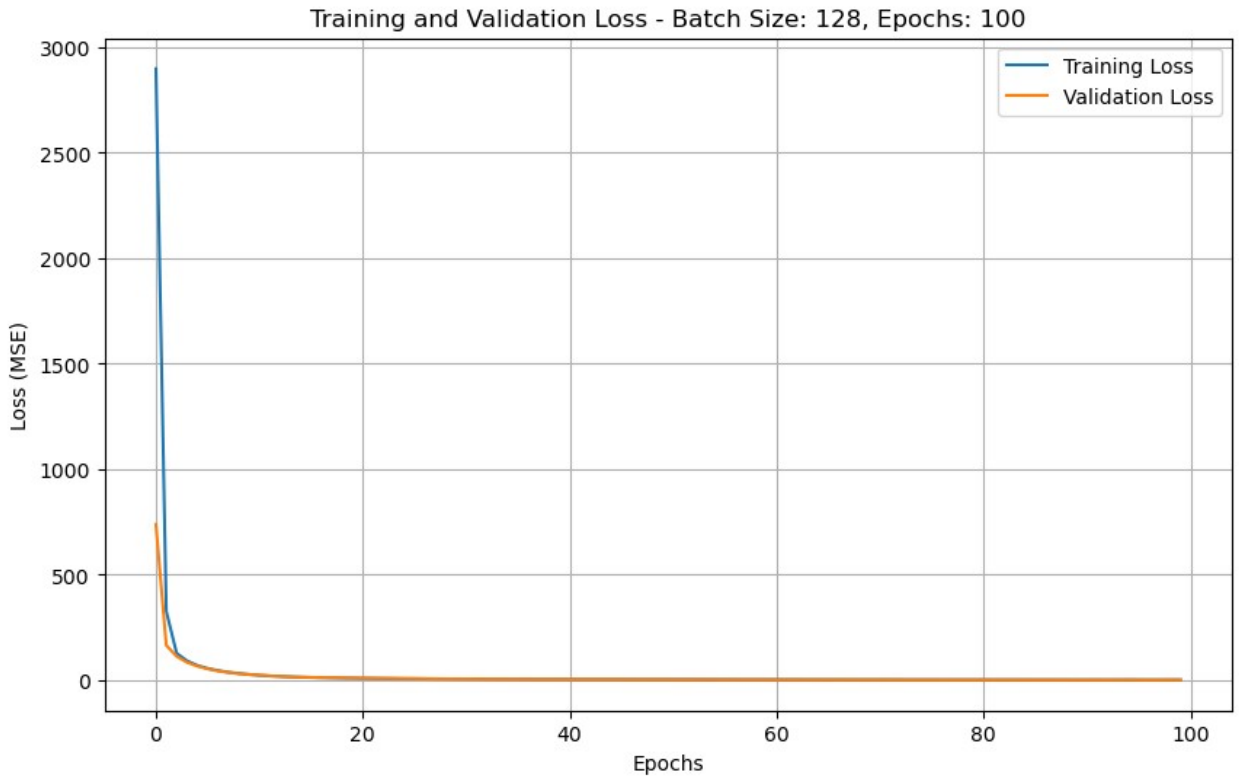

```
Training with Batch Size: 128, Epochs: 10
Epoch [10/10], Loss: 26.8325
Completed training with Batch Size: 128, Epochs: 10. Test MSE:
26.9795, Test RMSE: 5.1942
```




```
Training with Batch Size: 128, Epochs: 50
Epoch [10/50], Loss: 28.4600
Epoch [20/50], Loss: 8.8692
Epoch [30/50], Loss: 5.2076
Epoch [40/50], Loss: 3.9030
Epoch [50/50], Loss: 3.0871
Completed training with Batch Size: 128, Epochs: 50. Test MSE: 3.3735,
Test RMSE: 1.8367
```




```
Training with Batch Size: 128, Epochs: 100
Epoch [10/100], Loss: 27.4866
Epoch [20/100], Loss: 9.9823
Epoch [30/100], Loss: 5.9286
Epoch [40/100], Loss: 4.5215
Epoch [50/100], Loss: 3.6457
Epoch [60/100], Loss: 2.9750
Epoch [70/100], Loss: 2.5079
Epoch [80/100], Loss: 2.2465
Epoch [90/100], Loss: 1.8651
Epoch [100/100], Loss: 1.7906
Completed training with Batch Size: 128, Epochs: 100. Test MSE:
2.1277, Test RMSE: 1.4587
```



	Epochs	Batch Size	MSE	RMSE
0	10	16	12.208668	3.494090
1	50	16	2.261165	1.503717
2	100	16	4.464502	2.112937
3	10	32	6.986111	2.643125
4	50	32	2.354912	1.534572
5	100	32	1.981055	1.407500
6	10	64	11.823679	3.438558
7	50	64	2.473211	1.572644
8	100	64	1.887822	1.373981
9	10	128	26.979492	5.194179
10	50	128	3.373538	1.836719
11	100	128	2.127723	1.458672

Model architecture

```
# Defining the model instantiated
model = PlayerPerformanceModel()

# Providing the input shape (without the batch size), here 13 features
# summary(model, input_size=(X_train_tensor.shape[1],))
```

```
---------------------------------------------------------------------------
-----
RuntimeError                              Traceback (most recent call
last)
```

```
Cell In[24], line 5
      2 model = PlayerPerformanceModel()
      4 # Providing the input shape (without the batch size), here 13
features
----> 5 summary(model, input_size=(X_train_tensor.shape[1],))

File
~/anaconda3/lib/python3.12/site-packages/torchsummary/torchsummary.py:
72, in summary(model, input_size, batch_size, device)
     68 model.apply(register_hook)
     70 # make a forward pass
     71 # print(x.shape)
---> 72 model(*x)
     74 # remove these hooks
     75 for h in hooks:

File
~/anaconda3/lib/python3.12/site-packages/torch/nn/modules/module.py:15
53, in Module._wrapped_call_impl(self, *args, **kwargs)
   1551     return self._compiled_call_impl(*args, **kwargs)  # type:
ignore[misc]
   1552 else:
-> 1553     return self._call_impl(*args, **kwargs)

File
~/anaconda3/lib/python3.12/site-packages/torch/nn/modules/module.py:15
62, in Module._call_impl(self, *args, **kwargs)
   1557 # If we don't have any hooks, we want to skip the rest of the
logic in
   1558 # this function, and just call forward.
   1559 if not (self._backward_hooks or self._backward_pre_hooks or
self._forward_hooks or self._forward_pre_hooks
   1560         or _global_backward_pre_hooks or
_global_backward_hooks
   1561         or _global_forward_hooks or
_global_forward_pre_hooks):
-> 1562     return forward_call(*args, **kwargs)
   1564 try:
   1565     result = None

Cell In[23], line 11, in PlayerPerformanceModel.forward(self, x)
     10 def forward(self, x):
---> 11     x = self.relu(self.fc1(x))
     12     x = self.relu(self.fc2(x))
     13     x = self.fc3(x)  # No activation in the output layer for
regression

File
~/anaconda3/lib/python3.12/site-packages/torch/nn/modules/module.py:15
53, in Module._wrapped_call_impl(self, *args, **kwargs)
```

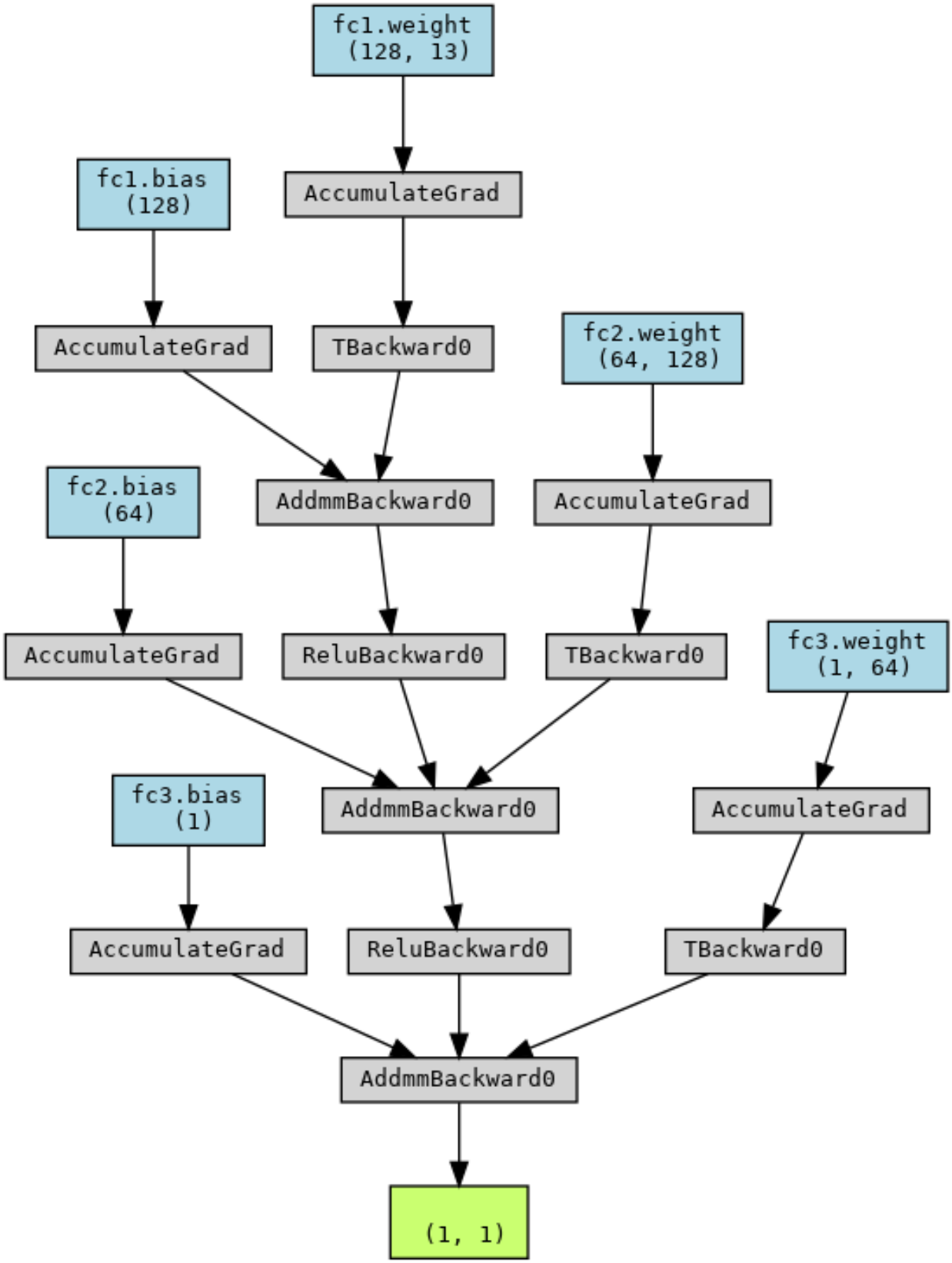
```
   1551     return self._compiled_call_impl(*args, **kwargs)  # type:
ignore[misc]
   1552 else:
-> 1553     return self._call_impl(*args, **kwargs)

File
~/anaconda3/lib/python3.12/site-packages/torch/nn/modules/module.py:16
03, in Module._call_impl(self, *args, **kwargs)
   1600     bw_hook = hooks.BackwardHook(self, full_backward_hooks,
backward_pre_hooks)
   1601     args = bw_hook.setup_input_hook(args)
-> 1603 result = forward_call(*args, **kwargs)
   1604 if _global_forward_hooks or self._forward_hooks:
   1605     for hook_id, hook in (
   1606         *_global_forward_hooks.items(),
   1607         *self._forward_hooks.items(),
   1608     ):
   1609         # mark that always called hook is run

File
~/anaconda3/lib/python3.12/site-packages/torch/nn/modules/linear.py:11
7, in Linear.forward(self, input)
    116 def forward(self, input: Tensor) -> Tensor:
--> 117     return F.linear(input, self.weight, self.bias)

RuntimeError: Expected all tensors to be on the same device, but found
at least two devices, cpu and cuda:0! (when checking argument for
argument mat1 in method wrapper_CUDA_addmm)
```

torchsummary library has been used to display a concise summary of the neural network architecture for the PlayerPerformanceModel in the Pytorch approach. It provides details about each layer in the model, including the output shape, the number of trainable parameters, and the total memory required. By using summary(), we can quickly inspect the model's structure, showing three fully connected layers (with ReLU activations) and the total parameter count of 10,113.

```
# Defining an input of tensor based on the input features
sample_input = torch.randn(1, X_train_tensor.shape[1])

# Passing the input through the model to get the output graph
output = model(sample_input)

# Generating a visualization of the model graph
make_dot(output,
params=dict(model.named_parameters())).render("model_architecture",
format="png")
```

```
'model_architecture.png'
```

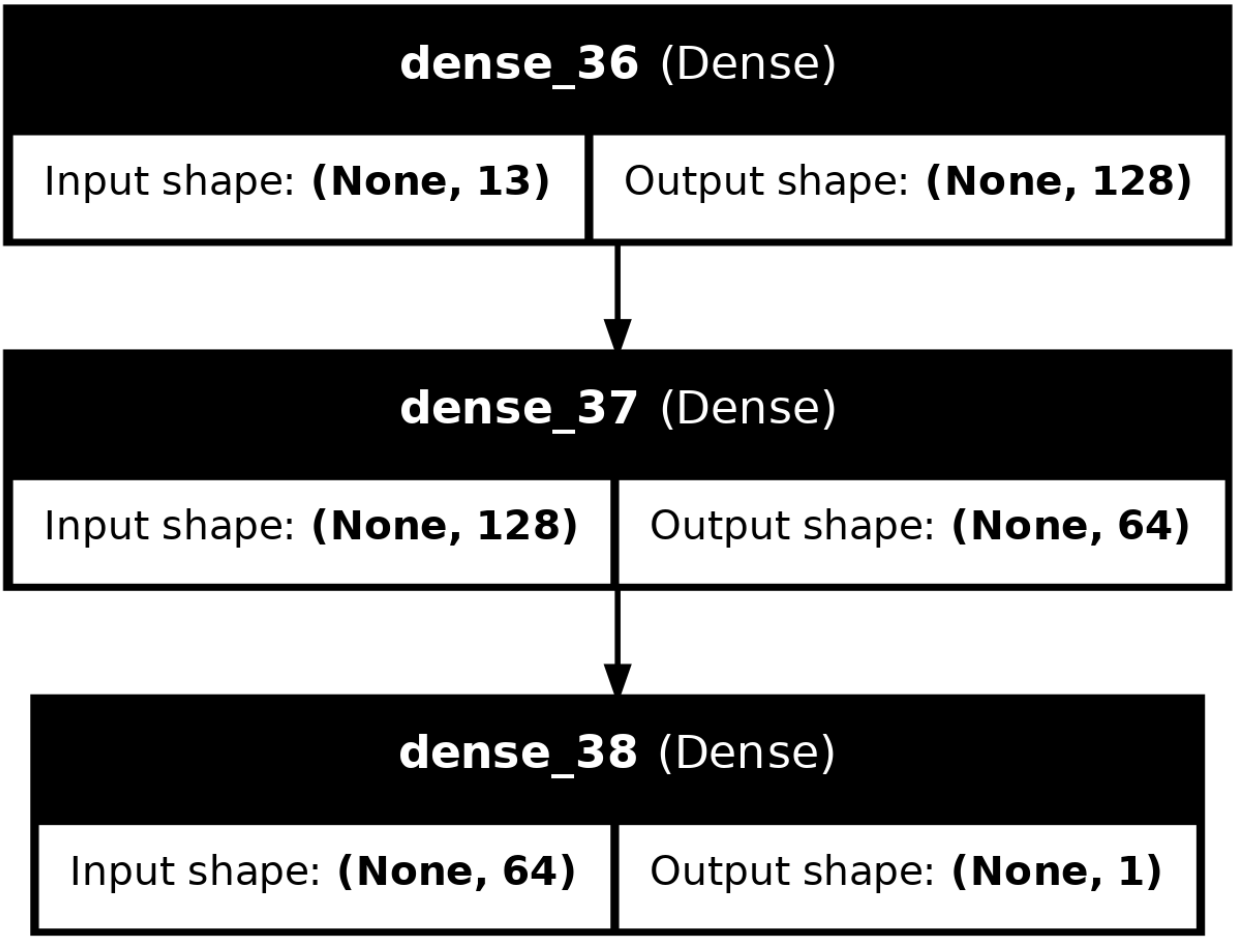
```
Image('model_architecture.png')
```


fc1.weight
(128, 13)
fc1.bias
(128)
AccumulateGrad
AccumulateGrad
TBackward0
fc2.weight
(64, 128)
fc2.bias
(64)
AddmmBackward0
AccumulateGrad
AccumulateGrad
ReluBackward0
TBackward0
fc3.weight
(1, 64)
fc3.bias
(1)
AddmmBackward0
AccumulateGrad
AccumulateGrad
ReluBackward0
TBackward0
AddmmBackward0
(1, 1)

- The flow of data starts from the input layer and passes through each of the fully connected (fc) layers in sequence: fc1, fc2, and fc3.
- Each layer is connected by ReLU activations, which are marked as ReluBackward0. This shows that after each layer, the ReLU function is applied to introduce non-linearity.
- The graph also illustrates the backpropagation steps (TBackward0, AddmmBackward0, AccumulateGrad), showing how gradients are computed for each layer during training. Each weight and bias has its corresponding gradient calculation node, ensuring the model can learn through backpropagation.
- The shapes of the weights and biases are also shown, giving insight into how many parameters are being trained at each layer.

```
# Assuming model is already defined and instantiated
model = models.Sequential([
    layers.Dense(128, activation='relu',
input_shape=(X_train.shape[1],)),
    layers.Dense(64, activation='relu'),
    layers.Dense(1)  # Output layer for regression
])

# Compiling the model
model.compile(optimizer='adam', loss='mse')

# Visualizing the model architecture and saving as an image
plot_model(model, to_file='model_plot.png', show_shapes=True,
show_layer_names=True)

# Displaying the model plot in Colab
from IPython.display import Image
Image('model_plot.png')
```

```
/home/unina/anaconda3/lib/python3.12/site-packages/keras/src/layers/
core/dense.py:87: UserWarning: Do not pass an
`input_shape`/`input_dim` argument to a layer. When using Sequential
models, prefer using an `Input(shape)` object as the first layer in
the model instead.
  super().__init__(activity_regularizer=activity_regularizer,
**kwargs)
```



- The model consists of three Dense layers (dense_57, dense_58, and dense_59).
- Each layer processes the input through fully connected neurons, with the first layer having 128 neurons, the second layer having 64 neurons, and the final output layer having 1 neuron.
- The input shape to the first layer is (None, 13), indicating that the model expects 13 features (i.e., player attributes), where None represents the batch size. Each subsequent layer takes the output of the previous layer as input. For example, dense_57 outputs (None, 128), and this becomes the input to dense_58.
- This architecture is typical for regression tasks where the final output is a single continuous value.

Model Comparison

```
# Merging the PyTorch and Keras results into one DataFrame for comparison
pytorch_results_df['Framework'] = 'PyTorch'
keras_results_df['Framework'] = 'Keras'

pytorch_results_df
```

	Epochs	Batch Size	Training Loss	Validation Loss	Framework
0	10	16	4.253976	4.466458	PyTorch
1	50	16	1.565505	1.818725	PyTorch
2	100	16	1.201197	1.688236	PyTorch
3	10	32	4.662750	4.434649	PyTorch
4	50	32	1.593202	2.381625	PyTorch
5	100	32	1.092650	1.452606	PyTorch
6	10	64	6.281777	6.205473	PyTorch
7	50	64	2.204512	2.560088	PyTorch
8	100	64	1.264285	1.382258	PyTorch
9	10	128	20.982613	20.983322	PyTorch
10	50	128	2.908017	3.183661	PyTorch
11	100	128	1.577614	1.856592	PyTorch

```
keras_results_df
```

	Epochs	Batch Size	MSE	RMSE	Framework
0	10	16	12.208668	3.494090	Keras
1	50	16	2.261165	1.503717	Keras
2	100	16	4.464502	2.112937	Keras
3	10	32	6.986111	2.643125	Keras
4	50	32	2.354912	1.534572	Keras
5	100	32	1.981055	1.407500	Keras
6	10	64	11.823679	3.438558	Keras
7	50	64	2.473211	1.572644	Keras
8	100	64	1.887822	1.373981	Keras
9	10	128	26.979492	5.194179	Keras
10	50	128	3.373538	1.836719	Keras
11	100	128	2.127723	1.458672	Keras

Insights

- The Summary table for Pytorch presents the results from training the base model across different epochs and batch sizes. As observed, the training and validation loss decrease as the number of epochs increases, reflecting improved learning over time. Smaller batch sizes such as 16 and 32 tend to provide better validation loss at higher epochs, showing a more stable learning process. However, larger batch sizes like 128 show higher losses, indicating potential underfitting. The model trained with 100 epochs and batch size 32 gives the best balance between training and validation losses, making it a promising candidate for further optimization.

- Similarly the table for Keras shows better performance with smaller batch sizes (16 and 32), especially when trained for more epochs, with batch size 32 and 100 epochs achieving the lowest MSE and RMSE. Larger batch sizes like 128 exhibit higher losses, suggesting underfitting. This model shows that increasing the number of epochs generally improves the performance, but there is variability based on batch size. The best results are observed with 100 epochs and batch size 32, as it results in the lowest RMSE.

Enhanced Model

The following table list all the ehnacement techniques that were used to create an enhanced model. These techniques will be applied through keras approach to find best model with optimal hyperparameters to predict player performance as part of our regression task.

Technique	Definition	Use
Kernel Regularizer	A parameter in Keras layers that applies a penalty to the layer's weights, helping to prevent overfitting.	Enhances model generalization by adding regularization to specific layers.
Early Stopping	A technique that halts training when the model's performance on a validation set starts to degrade, preventing overfitting.	Ensures the model does not train too long, maintaining optimal performance.
Dropout Layer	A regularization method that randomly sets a fraction of input units to zero during training, reducing overfitting.	Improves generalization by preventing co-adaptation of neurons.
L2 Regularization	Adds a penalty equal to the sum of the squared weights to the loss function, encouraging smaller weights.	Reduces overfitting by controlling model complexity.
Adam Optimizer	An adaptive learning rate optimization algorithm that combines momentum and RMSprop, improving convergence speed.	Efficient for large datasets and parameters, often yielding better results.
Learning Rate	A hyperparameter that determines the step size at each iteration while moving toward a minimum of the loss function.	Critical for ensuring effective training and convergence of the model.
Pickle	A Python module used for serializing and deserializing objects, allowing models to be saved and loaded easily.	Facilitates model persistence and reuse across different sessions or environments.

Keras approach

Training

The below function trains an enhanced Keras regression model with additional techniques like L2 regularization, Dropout, and learning rate tuning that which we have defined earlier. The primary goal is to improve model performance while preventing overfitting. The function explores different combinations of batch sizes and epochs during training.

- L2 regularization helps reduce overfitting by adding a penalty to large weights.
- Dropout is used to randomly deactivate 50% of the neurons during training to improve generalization.
- The Adam optimizer with a learning rate of 0.001 is used for efficient optimization.
- An early stopping callback monitors validation loss and halts training if the model stops improving.
- After training, the function evaluates the model's performance on the test set and plots training and validation loss curves. Results are stored for later analysis and comparison.

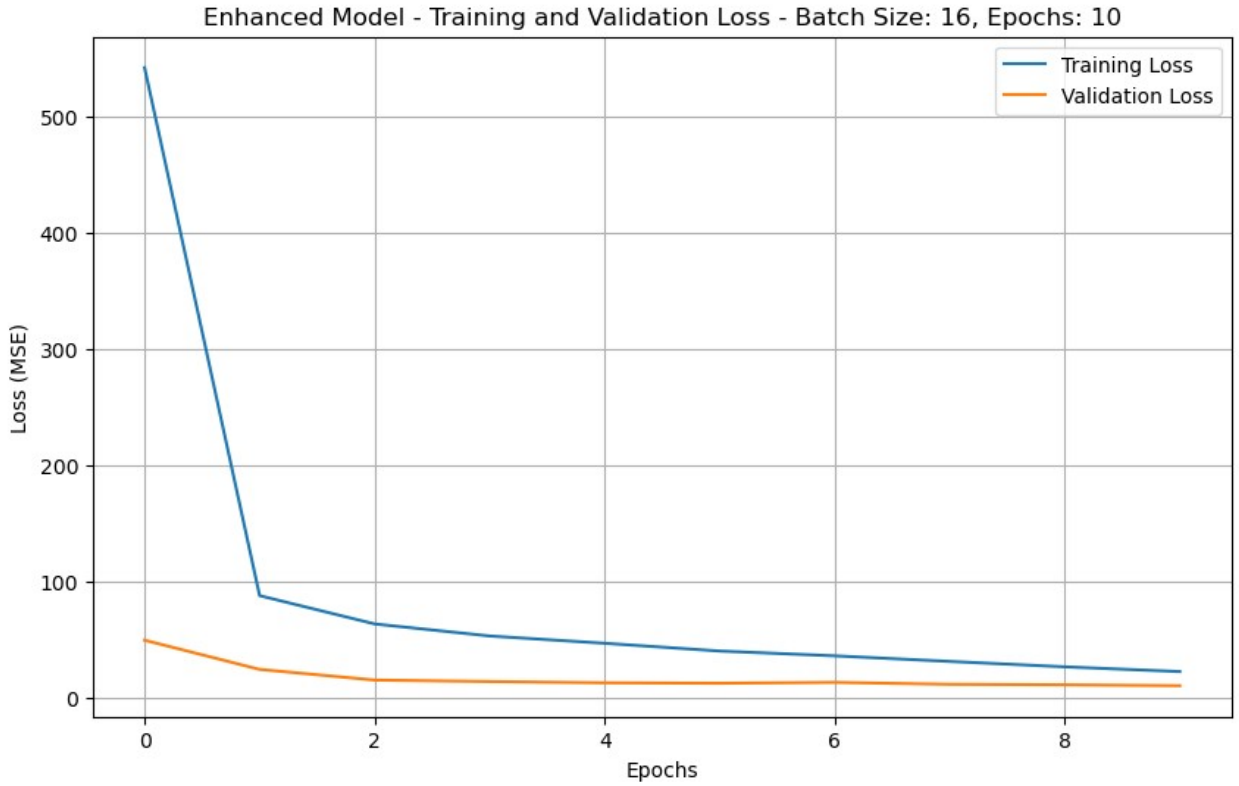
```
# Function to train the enhanced Keras model with regularization,
dropout, and learning rate tuning
```

```
def train_enhanced_keras_model(X_train, y_train, X_test, y_test,
epochs_list, batch_size_list):
    results = []

    for batch_size in batch_size_list:
        for epochs in epochs_list:
            print(f"\nTraining with Batch Size: {batch_size}, Epochs:
{epochs}")

            # Building a regression model with Dropout and L2
Regularization
            model = Sequential([
                layers.Dense(128, activation='relu',
input_shape=(X_train.shape[1],),

kernel_regularizer=regularizers.l2(0.001)),
                layers.Dropout(0.5),
                layers.Dense(64, activation='relu',
kernel_regularizer=regularizers.l2(0.001)),
                layers.Dense(1)
            ])

            # Compiling the model with Adam optimizer
            model.compile(optimizer=Adam(learning_rate=0.001),
                          loss='mse',
                          metrics=[RootMeanSquaredError()])

            # Early stopping callback to prevent overfitting
            early_stopping = EarlyStopping(monitor='val_loss',
patience=10, restore_best_weights=True)

            # Train the model and storing the history object
            history = model.fit(X_train, y_train,
validation_split=0.2, epochs=epochs, batch_size=batch_size,
                                callbacks=[early_stopping], verbose=1)

            # Saving the history object to a file
            with open(f'history_batch{batch_size}_epochs{epochs}.pkl',
'wb') as file:
                pickle.dump(history.history, file)

            # Evaluating the model on the test set
            test_loss, test_rmse = model.evaluate(X_test, y_test,
verbose=0)
            results.append({'Epochs': epochs, 'Batch Size':
batch_size, 'MSE': test_loss, 'RMSE': test_rmse})
            print(f"Completed training with Batch Size: {batch_size},
Epochs: {epochs}. Test MSE: {test_loss:.4f}, Test RMSE:
{test_rmse:.4f}")
```

```
            # Ploting the training and validation loss curves
            plt.figure(figsize=(10, 6))
            plt.plot(history.history['loss'], label='Training Loss')
            plt.plot(history.history['val_loss'], label='Validation
Loss')
            plt.title(f'Enhanced Model - Training and Validation Loss
- Batch Size: {batch_size}, Epochs: {epochs}')
            plt.xlabel('Epochs')
            plt.ylabel('Loss (MSE)')
            plt.legend()
            plt.grid(True)
            plt.show()

    # Converting the results to a DataFrame
    return pd.DataFrame(results)

# Defining the different epochs and batch sizes
epochs_list = [10, 50, 100]
batch_size_list = [16, 32, 64, 128]

# Training and collecting results with the enhanced Keras model
keras_results_df = train_enhanced_keras_model(X_train, y_train,
X_test, y_test, epochs_list, batch_size_list)
```

```
Training with Batch Size: 16, Epochs: 10
Epoch 1/10

/home/unina/anaconda3/lib/python3.12/site-packages/keras/src/layers/
core/dense.py:87: UserWarning: Do not pass an
`input_shape`/`input_dim` argument to a layer. When using Sequential
models, prefer using an `Input(shape)` object as the first layer in
the model instead.
  super().__init__(activity_regularizer=activity_regularizer,
**kwargs)

719/719 ━━━━━━━━━━━━━━━━━━━━ 1s 616us/step - loss: 1410.8069 -
root_mean_squared_error: 35.5003 - val_loss: 49.8925 -
val_root_mean_squared_error: 7.0501
Epoch 2/10
719/719 ━━━━━━━━━━━━━━━━━━━━ 0s 432us/step - loss: 98.3469 -
root_mean_squared_error: 9.9017 - val_loss: 24.8154 -
val_root_mean_squared_error: 4.9640
Epoch 3/10
719/719 ━━━━━━━━━━━━━━━━━━━━ 0s 534us/step - loss: 67.5803 -
root_mean_squared_error: 8.2087 - val_loss: 15.6574 -
val_root_mean_squared_error: 3.9366
Epoch 4/10
719/719 ━━━━━━━━━━━━━━━━━━━━ 0s 531us/step - loss: 56.2206 -
root_mean_squared_error: 7.4863 - val_loss: 14.3986 -
```

```
val_root_mean_squared_error: 3.7750
Epoch 5/10
719/719 ━━━━━━━━━━━━━━━━━━━━ 0s 500us/step - loss: 48.5448 -
root_mean_squared_error: 6.9564 - val_loss: 13.3105 -
val_root_mean_squared_error: 3.6296
Epoch 6/10
719/719 ━━━━━━━━━━━━━━━━━━━━ 0s 465us/step - loss: 40.4979 -
root_mean_squared_error: 6.3523 - val_loss: 12.9382 -
val_root_mean_squared_error: 3.5795
Epoch 7/10
719/719 ━━━━━━━━━━━━━━━━━━━━ 0s 532us/step - loss: 37.3943 -
root_mean_squared_error: 6.1042 - val_loss: 13.6958 -
val_root_mean_squared_error: 3.6852
Epoch 8/10
719/719 ━━━━━━━━━━━━━━━━━━━━ 0s 515us/step - loss: 32.5475 -
root_mean_squared_error: 5.6945 - val_loss: 12.0181 -
val_root_mean_squared_error: 3.4513
Epoch 9/10
719/719 ━━━━━━━━━━━━━━━━━━━━ 0s 482us/step - loss: 27.7033 -
root_mean_squared_error: 5.2531 - val_loss: 11.6144 -
val_root_mean_squared_error: 3.3936
Epoch 10/10
719/719 ━━━━━━━━━━━━━━━━━━━━ 0s 581us/step - loss: 24.5091 -
root_mean_squared_error: 4.9387 - val_loss: 10.8293 -
val_root_mean_squared_error: 3.2771
Completed training with Batch Size: 16, Epochs: 10. Test MSE: 10.2655,
Test RMSE: 3.1899
```



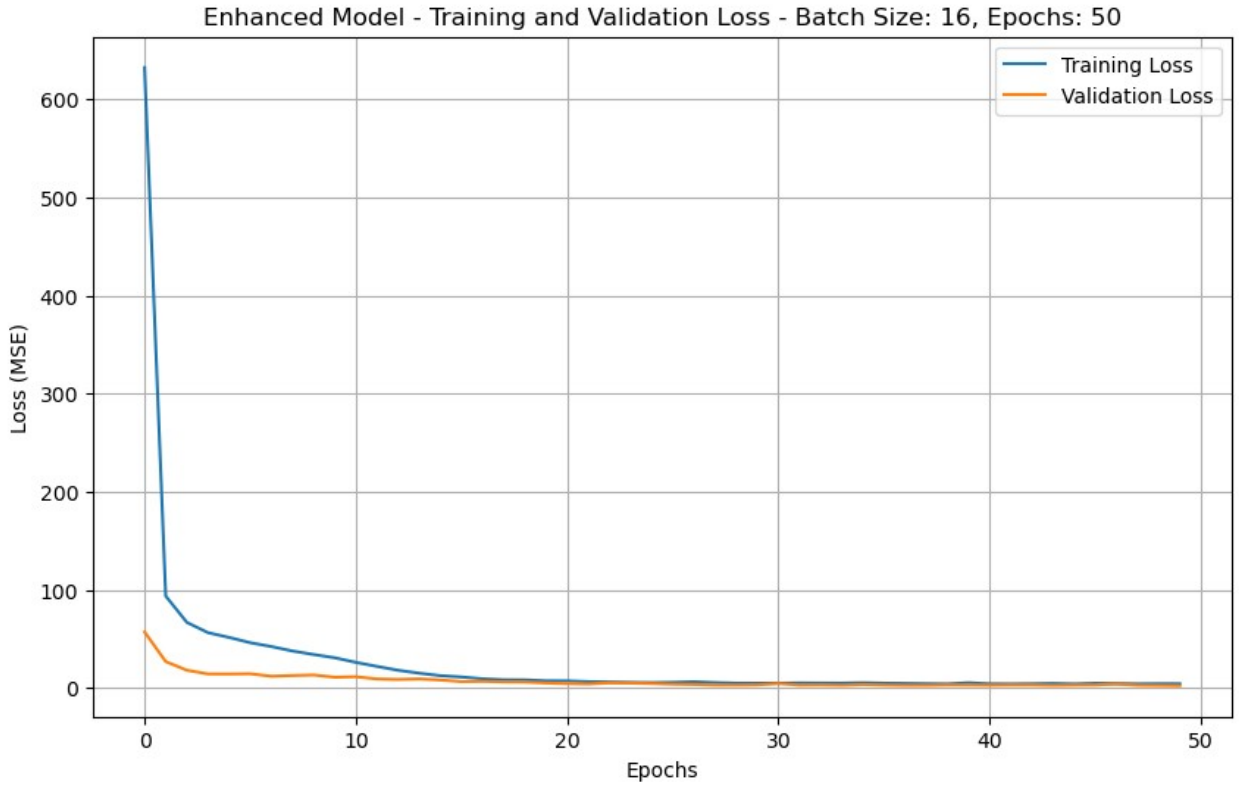

```
Training with Batch Size: 16, Epochs: 50
Epoch 1/50
719/719 ━━━━━━━━━━━━━━━━━━━━ 1s 698us/step - loss: 1615.4733 -
root_mean_squared_error: 38.2033 - val_loss: 56.9571 -
val_root_mean_squared_error: 7.5338
Epoch 2/50
719/719 ━━━━━━━━━━━━━━━━━━━━ 0s 517us/step - loss: 104.4701 -
root_mean_squared_error: 10.2030 - val_loss: 26.7404 -
val_root_mean_squared_error: 5.1532
Epoch 3/50
719/719 ━━━━━━━━━━━━━━━━━━━━ 0s 589us/step - loss: 73.6117 -
root_mean_squared_error: 8.5625 - val_loss: 17.9024 -
val_root_mean_squared_error: 4.2109
Epoch 4/50
719/719 ━━━━━━━━━━━━━━━━━━━━ 0s 529us/step - loss: 59.6874 -
root_mean_squared_error: 7.7086 - val_loss: 14.1355 -
val_root_mean_squared_error: 3.7384
Epoch 5/50
719/719 ━━━━━━━━━━━━━━━━━━━━ 0s 599us/step - loss: 52.3358 -
root_mean_squared_error: 7.2223 - val_loss: 14.0853 -
val_root_mean_squared_error: 3.7335
Epoch 6/50
719/719 ━━━━━━━━━━━━━━━━━━━━ 0s 537us/step - loss: 47.7145 -
root_mean_squared_error: 6.8960 - val_loss: 14.3023 -
```

```
val_root_mean_squared_error: 3.7642
Epoch 7/50
719/719 ━━━━━━━━━━━━━━━━━━━━ 0s 524us/step - loss: 42.9520 -
root_mean_squared_error: 6.5425 - val_loss: 11.7547 -
val_root_mean_squared_error: 3.4105
Epoch 8/50
719/719 ━━━━━━━━━━━━━━━━━━━━ 0s 470us/step - loss: 37.9770 -
root_mean_squared_error: 6.1509 - val_loss: 12.4848 -
val_root_mean_squared_error: 3.5176
Epoch 9/50
719/719 ━━━━━━━━━━━━━━━━━━━━ 0s 561us/step - loss: 34.9418 -
root_mean_squared_error: 5.9011 - val_loss: 13.0521 -
val_root_mean_squared_error: 3.5987
Epoch 10/50
719/719 ━━━━━━━━━━━━━━━━━━━━ 0s 510us/step - loss: 31.5244 -
root_mean_squared_error: 5.6056 - val_loss: 10.7543 -
val_root_mean_squared_error: 3.2655
Epoch 11/50
719/719 ━━━━━━━━━━━━━━━━━━━━ 0s 478us/step - loss: 26.7390 -
root_mean_squared_error: 5.1620 - val_loss: 11.2420 -
val_root_mean_squared_error: 3.3406
Epoch 12/50
719/719 ━━━━━━━━━━━━━━━━━━━━ 0s 460us/step - loss: 23.1486 -
root_mean_squared_error: 4.8019 - val_loss: 9.0048 -
val_root_mean_squared_error: 2.9879
Epoch 13/50
719/719 ━━━━━━━━━━━━━━━━━━━━ 0s 453us/step - loss: 19.2289 -
root_mean_squared_error: 4.3749 - val_loss: 8.5039 -
val_root_mean_squared_error: 2.9034
Epoch 14/50
719/719 ━━━━━━━━━━━━━━━━━━━━ 0s 465us/step - loss: 15.8400 -
root_mean_squared_error: 3.9673 - val_loss: 9.0560 -
val_root_mean_squared_error: 2.9971
Epoch 15/50
719/719 ━━━━━━━━━━━━━━━━━━━━ 0s 511us/step - loss: 12.9937 -
root_mean_squared_error: 3.5939 - val_loss: 7.9360 -
val_root_mean_squared_error: 2.8037
Epoch 16/50
719/719 ━━━━━━━━━━━━━━━━━━━━ 0s 506us/step - loss: 11.4306 -
root_mean_squared_error: 3.3685 - val_loss: 6.3599 -
val_root_mean_squared_error: 2.5060
Epoch 17/50
719/719 ━━━━━━━━━━━━━━━━━━━━ 0s 534us/step - loss: 9.3845 -
root_mean_squared_error: 3.0495 - val_loss: 6.5647 -
val_root_mean_squared_error: 2.5453
Epoch 18/50
719/719 ━━━━━━━━━━━━━━━━━━━━ 0s 447us/step - loss: 8.3248 -
root_mean_squared_error: 2.8698 - val_loss: 6.0040 -
val_root_mean_squared_error: 2.4314
```

```
Epoch 19/50
719/719 ━━━━━━━━━━━━━━━━━━━━ 0s 516us/step - loss: 7.7886 -
root_mean_squared_error: 2.7735 - val_loss: 5.9127 -
val_root_mean_squared_error: 2.4115
Epoch 20/50
719/719 ━━━━━━━━━━━━━━━━━━━━ 0s 484us/step - loss: 7.4004 -
root_mean_squared_error: 2.7015 - val_loss: 4.8573 -
val_root_mean_squared_error: 2.1803
Epoch 21/50
719/719 ━━━━━━━━━━━━━━━━━━━━ 0s 465us/step - loss: 6.8973 -
root_mean_squared_error: 2.6044 - val_loss: 4.2719 -
val_root_mean_squared_error: 2.0400
Epoch 22/50
719/719 ━━━━━━━━━━━━━━━━━━━━ 0s 440us/step - loss: 6.2754 -
root_mean_squared_error: 2.4822 - val_loss: 3.9199 -
val_root_mean_squared_error: 1.9501
Epoch 23/50
719/719 ━━━━━━━━━━━━━━━━━━━━ 0s 493us/step - loss: 5.7293 -
root_mean_squared_error: 2.3676 - val_loss: 5.0648 -
val_root_mean_squared_error: 2.2226
Epoch 24/50
719/719 ━━━━━━━━━━━━━━━━━━━━ 0s 502us/step - loss: 5.7260 -
root_mean_squared_error: 2.3653 - val_loss: 4.8612 -
val_root_mean_squared_error: 2.1749
Epoch 25/50
719/719 ━━━━━━━━━━━━━━━━━━━━ 0s 483us/step - loss: 5.7306 -
root_mean_squared_error: 2.3636 - val_loss: 4.3552 -
val_root_mean_squared_error: 2.0534
Epoch 26/50
719/719 ━━━━━━━━━━━━━━━━━━━━ 0s 534us/step - loss: 5.8422 -
root_mean_squared_error: 2.3873 - val_loss: 3.7338 -
val_root_mean_squared_error: 1.8945
Epoch 27/50
719/719 ━━━━━━━━━━━━━━━━━━━━ 0s 462us/step - loss: 5.3246 -
root_mean_squared_error: 2.2692 - val_loss: 3.3934 -
val_root_mean_squared_error: 1.8008
Epoch 28/50
719/719 ━━━━━━━━━━━━━━━━━━━━ 0s 470us/step - loss: 5.5800 -
root_mean_squared_error: 2.3285 - val_loss: 2.7936 -
val_root_mean_squared_error: 1.6239
Epoch 29/50
719/719 ━━━━━━━━━━━━━━━━━━━━ 0s 443us/step - loss: 4.7911 -
root_mean_squared_error: 2.1520 - val_loss: 2.7580 -
val_root_mean_squared_error: 1.6115
Epoch 30/50
719/719 ━━━━━━━━━━━━━━━━━━━━ 0s 482us/step - loss: 4.5918 -
root_mean_squared_error: 2.1042 - val_loss: 2.9776 -
val_root_mean_squared_error: 1.6771
Epoch 31/50
```

```
719/719 ━━━━━━━━━━━━━━━━━━━━ 0s 439us/step - loss: 4.8483 -
root_mean_squared_error: 2.1614 - val_loss: 4.3837 -
val_root_mean_squared_error: 2.0531
Epoch 32/50
719/719 ━━━━━━━━━━━━━━━━━━━━ 0s 476us/step - loss: 5.5509 -
root_mean_squared_error: 2.3152 - val_loss: 2.7220 -
val_root_mean_squared_error: 1.5974
Epoch 33/50
719/719 ━━━━━━━━━━━━━━━━━━━━ 0s 459us/step - loss: 4.4473 -
root_mean_squared_error: 2.0673 - val_loss: 2.9206 -
val_root_mean_squared_error: 1.6575
Epoch 34/50
719/719 ━━━━━━━━━━━━━━━━━━━━ 0s 496us/step - loss: 5.3225 -
root_mean_squared_error: 2.2657 - val_loss: 2.5740 -
val_root_mean_squared_error: 1.5488
Epoch 35/50
719/719 ━━━━━━━━━━━━━━━━━━━━ 0s 475us/step - loss: 5.0723 -
root_mean_squared_error: 2.2078 - val_loss: 3.2863 -
val_root_mean_squared_error: 1.7635
Epoch 36/50
719/719 ━━━━━━━━━━━━━━━━━━━━ 0s 531us/step - loss: 4.2033 -
root_mean_squared_error: 2.0030 - val_loss: 2.7670 -
val_root_mean_squared_error: 1.6090
Epoch 37/50
719/719 ━━━━━━━━━━━━━━━━━━━━ 0s 536us/step - loss: 4.3195 -
root_mean_squared_error: 2.0329 - val_loss: 2.4603 -
val_root_mean_squared_error: 1.5103
Epoch 38/50
719/719 ━━━━━━━━━━━━━━━━━━━━ 0s 576us/step - loss: 3.9145 -
root_mean_squared_error: 1.9308 - val_loss: 2.4764 -
val_root_mean_squared_error: 1.5158
Epoch 39/50
719/719 ━━━━━━━━━━━━━━━━━━━━ 0s 490us/step - loss: 4.1091 -
root_mean_squared_error: 1.9817 - val_loss: 2.8971 -
val_root_mean_squared_error: 1.6486
Epoch 40/50
719/719 ━━━━━━━━━━━━━━━━━━━━ 0s 500us/step - loss: 5.2371 -
root_mean_squared_error: 2.2449 - val_loss: 2.7977 -
val_root_mean_squared_error: 1.6182
Epoch 41/50
719/719 ━━━━━━━━━━━━━━━━━━━━ 0s 467us/step - loss: 3.8385 -
root_mean_squared_error: 1.9116 - val_loss: 2.6004 -
val_root_mean_squared_error: 1.5562
Epoch 42/50
719/719 ━━━━━━━━━━━━━━━━━━━━ 0s 453us/step - loss: 3.8915 -
root_mean_squared_error: 1.9264 - val_loss: 2.8464 -
val_root_mean_squared_error: 1.6336
Epoch 43/50
719/719 ━━━━━━━━━━━━━━━━━━━━ 0s 488us/step - loss: 4.4215 -
```

```
root_mean_squared_error: 2.0560 - val_loss: 2.8490 -
val_root_mean_squared_error: 1.6349
Epoch 44/50
719/719 ━━━━━━━━━━━━━━━━━━━━ 0s 544us/step - loss: 4.3448 -
root_mean_squared_error: 2.0406 - val_loss: 2.4536 -
val_root_mean_squared_error: 1.5096
Epoch 45/50
719/719 ━━━━━━━━━━━━━━━━━━━━ 0s 441us/step - loss: 3.9965 -
root_mean_squared_error: 1.9528 - val_loss: 2.7708 -
val_root_mean_squared_error: 1.6115
Epoch 46/50
719/719 ━━━━━━━━━━━━━━━━━━━━ 0s 478us/step - loss: 4.2874 -
root_mean_squared_error: 2.0269 - val_loss: 2.9021 -
val_root_mean_squared_error: 1.6515
Epoch 47/50
719/719 ━━━━━━━━━━━━━━━━━━━━ 0s 463us/step - loss: 4.6721 -
root_mean_squared_error: 2.1114 - val_loss: 3.8594 -
val_root_mean_squared_error: 1.9200
Epoch 48/50
719/719 ━━━━━━━━━━━━━━━━━━━━ 0s 458us/step - loss: 4.4313 -
root_mean_squared_error: 2.0576 - val_loss: 2.5647 -
val_root_mean_squared_error: 1.5463
Epoch 49/50
719/719 ━━━━━━━━━━━━━━━━━━━━ 0s 505us/step - loss: 4.0045 -
root_mean_squared_error: 1.9537 - val_loss: 2.3855 -
val_root_mean_squared_error: 1.4882
Epoch 50/50
719/719 ━━━━━━━━━━━━━━━━━━━━ 0s 467us/step - loss: 4.4469 -
root_mean_squared_error: 2.0664 - val_loss: 2.1398 -
val_root_mean_squared_error: 1.4034
Completed training with Batch Size: 16, Epochs: 50. Test MSE: 2.4616,
Test RMSE: 1.5137
```




```
Training with Batch Size: 16, Epochs: 100
Epoch 1/100
719/719 ━━━━━━━━━━━━━━━━━━━━ 1s 714us/step - loss: 1583.3077 -
root_mean_squared_error: 37.7389 - val_loss: 51.8688 -
val_root_mean_squared_error: 7.1873
Epoch 2/100
719/719 ━━━━━━━━━━━━━━━━━━━━ 0s 535us/step - loss: 96.9804 -
root_mean_squared_error: 9.8317 - val_loss: 23.2645 -
val_root_mean_squared_error: 4.8028
Epoch 3/100
719/719 ━━━━━━━━━━━━━━━━━━━━ 0s 451us/step - loss: 70.2375 -
root_mean_squared_error: 8.3677 - val_loss: 15.3914 -
val_root_mean_squared_error: 3.9000
Epoch 4/100
719/719 ━━━━━━━━━━━━━━━━━━━━ 0s 535us/step - loss: 55.1259 -
root_mean_squared_error: 7.4088 - val_loss: 15.6433 -
val_root_mean_squared_error: 3.9339
Epoch 5/100
719/719 ━━━━━━━━━━━━━━━━━━━━ 0s 468us/step - loss: 48.5484 -
root_mean_squared_error: 6.9548 - val_loss: 13.0680 -
val_root_mean_squared_error: 3.5933
Epoch 6/100
719/719 ━━━━━━━━━━━━━━━━━━━━ 0s 461us/step - loss: 44.8346 -
root_mean_squared_error: 6.6825 - val_loss: 12.1858 -
```

```
val_root_mean_squared_error: 3.4703
Epoch 7/100
719/719 ━━━━━━━━━━━━━━━━━━━━ 0s 567us/step - loss: 40.0244 -
root_mean_squared_error: 6.3149 - val_loss: 10.9598 -
val_root_mean_squared_error: 3.2908
Epoch 8/100
719/719 ━━━━━━━━━━━━━━━━━━━━ 0s 439us/step - loss: 35.6819 -
root_mean_squared_error: 5.9619 - val_loss: 14.5181 -
val_root_mean_squared_error: 3.7948
Epoch 9/100
719/719 ━━━━━━━━━━━━━━━━━━━━ 0s 435us/step - loss: 32.0408 -
root_mean_squared_error: 5.6485 - val_loss: 12.1006 -
val_root_mean_squared_error: 3.4629
Epoch 10/100
719/719 ━━━━━━━━━━━━━━━━━━━━ 0s 515us/step - loss: 25.8756 -
root_mean_squared_error: 5.0748 - val_loss: 9.4568 -
val_root_mean_squared_error: 3.0583
Epoch 11/100
719/719 ━━━━━━━━━━━━━━━━━━━━ 0s 439us/step - loss: 21.4971 -
root_mean_squared_error: 4.6235 - val_loss: 8.1843 -
val_root_mean_squared_error: 2.8437
Epoch 12/100
719/719 ━━━━━━━━━━━━━━━━━━━━ 0s 543us/step - loss: 16.8062 -
root_mean_squared_error: 4.0872 - val_loss: 8.0590 -
val_root_mean_squared_error: 2.8220
Epoch 13/100
719/719 ━━━━━━━━━━━━━━━━━━━━ 0s 546us/step - loss: 13.7715 -
root_mean_squared_error: 3.6974 - val_loss: 9.4801 -
val_root_mean_squared_error: 3.0636
Epoch 14/100
719/719 ━━━━━━━━━━━━━━━━━━━━ 0s 520us/step - loss: 11.1733 -
root_mean_squared_error: 3.3279 - val_loss: 5.9827 -
val_root_mean_squared_error: 2.4266
Epoch 15/100
719/719 ━━━━━━━━━━━━━━━━━━━━ 0s 451us/step - loss: 10.2034 -
root_mean_squared_error: 3.1790 - val_loss: 6.0438 -
val_root_mean_squared_error: 2.4389
Epoch 16/100
719/719 ━━━━━━━━━━━━━━━━━━━━ 0s 557us/step - loss: 8.3225 -
root_mean_squared_error: 2.8675 - val_loss: 6.5010 -
val_root_mean_squared_error: 2.5306
Epoch 17/100
719/719 ━━━━━━━━━━━━━━━━━━━━ 0s 392us/step - loss: 8.6850 -
root_mean_squared_error: 2.9284 - val_loss: 6.2639 -
val_root_mean_squared_error: 2.4830
Epoch 18/100
719/719 ━━━━━━━━━━━━━━━━━━━━ 0s 529us/step - loss: 7.4052 -
root_mean_squared_error: 2.7015 - val_loss: 5.8857 -
val_root_mean_squared_error: 2.4052
```

```
Epoch 19/100
719/719 ━━━━━━━━━━━━━━━━━━━━ 0s 583us/step - loss: 7.0918 -
root_mean_squared_error: 2.6429 - val_loss: 10.5653 -
val_root_mean_squared_error: 3.2345
Epoch 20/100
719/719 ━━━━━━━━━━━━━━━━━━━━ 0s 514us/step - loss: 8.1049 -
root_mean_squared_error: 2.8231 - val_loss: 5.5126 -
val_root_mean_squared_error: 2.3254
Epoch 21/100
719/719 ━━━━━━━━━━━━━━━━━━━━ 0s 550us/step - loss: 7.8305 -
root_mean_squared_error: 2.7786 - val_loss: 4.9383 -
val_root_mean_squared_error: 2.1978
Epoch 22/100
719/719 ━━━━━━━━━━━━━━━━━━━━ 0s 500us/step - loss: 6.8193 -
root_mean_squared_error: 2.5895 - val_loss: 10.0404 -
val_root_mean_squared_error: 3.1510
Epoch 23/100
719/719 ━━━━━━━━━━━━━━━━━━━━ 0s 529us/step - loss: 7.4085 -
root_mean_squared_error: 2.6968 - val_loss: 5.1874 -
val_root_mean_squared_error: 2.2518
Epoch 24/100
719/719 ━━━━━━━━━━━━━━━━━━━━ 0s 474us/step - loss: 6.1865 -
root_mean_squared_error: 2.4631 - val_loss: 4.6414 -
val_root_mean_squared_error: 2.1264
Epoch 25/100
719/719 ━━━━━━━━━━━━━━━━━━━━ 0s 481us/step - loss: 5.9633 -
root_mean_squared_error: 2.4158 - val_loss: 5.0431 -
val_root_mean_squared_error: 2.2178
Epoch 26/100
719/719 ━━━━━━━━━━━━━━━━━━━━ 0s 489us/step - loss: 6.8986 -
root_mean_squared_error: 2.6000 - val_loss: 3.6950 -
val_root_mean_squared_error: 1.8885
Epoch 27/100
719/719 ━━━━━━━━━━━━━━━━━━━━ 0s 463us/step - loss: 5.2920 -
root_mean_squared_error: 2.2712 - val_loss: 3.5351 -
val_root_mean_squared_error: 1.8446
Epoch 28/100
719/719 ━━━━━━━━━━━━━━━━━━━━ 0s 515us/step - loss: 6.3888 -
root_mean_squared_error: 2.4970 - val_loss: 3.8271 -
val_root_mean_squared_error: 1.9212
Epoch 29/100
719/719 ━━━━━━━━━━━━━━━━━━━━ 0s 543us/step - loss: 5.8975 -
root_mean_squared_error: 2.3959 - val_loss: 4.3957 -
val_root_mean_squared_error: 2.0627
Epoch 30/100
719/719 ━━━━━━━━━━━━━━━━━━━━ 0s 609us/step - loss: 6.1483 -
root_mean_squared_error: 2.4492 - val_loss: 3.0513 -
val_root_mean_squared_error: 1.7045
Epoch 31/100
```

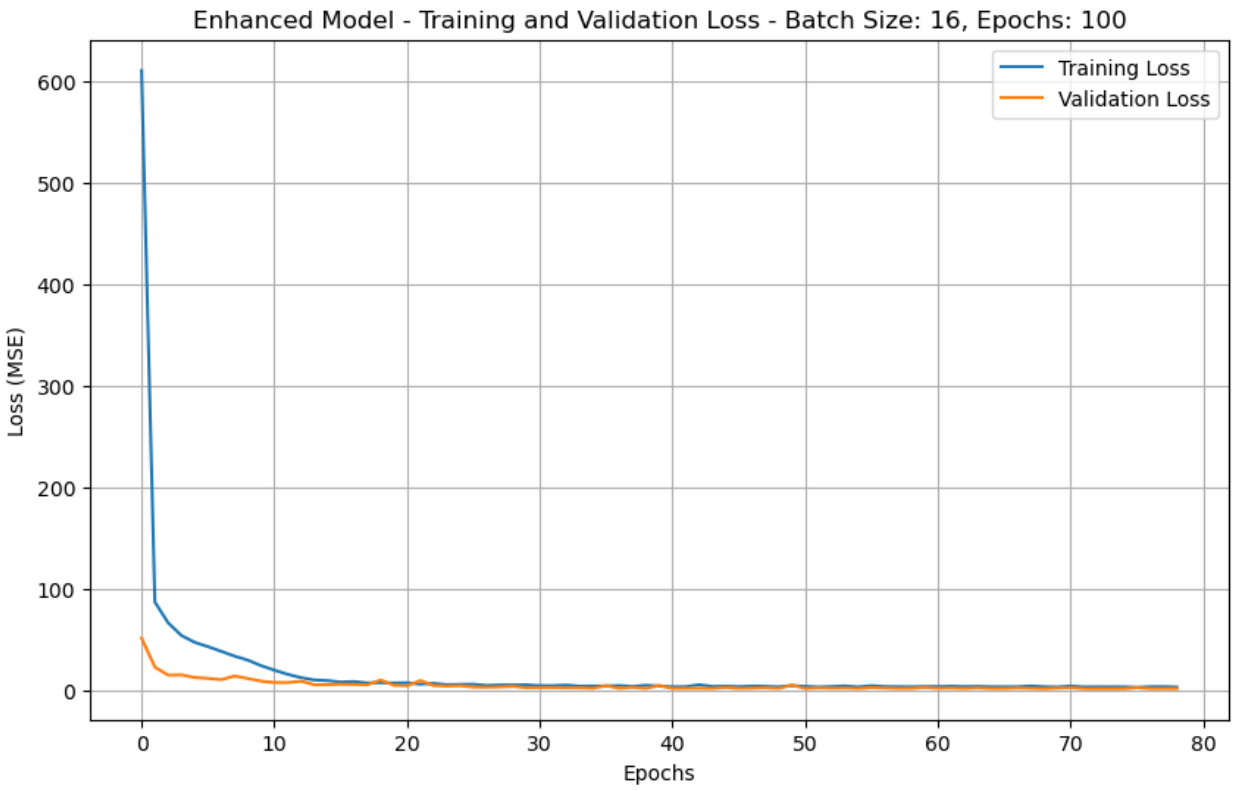
```
719/719 ━━━━━━━━━━━━━━━━━━━━ 0s 492us/step - loss: 4.7030 -
root_mean_squared_error: 2.1312 - val_loss: 3.1952 -
val_root_mean_squared_error: 1.7451
Epoch 32/100
719/719 ━━━━━━━━━━━━━━━━━━━━ 0s 468us/step - loss: 4.6417 -
root_mean_squared_error: 2.1182 - val_loss: 3.2829 -
val_root_mean_squared_error: 1.7688
Epoch 33/100
719/719 ━━━━━━━━━━━━━━━━━━━━ 0s 462us/step - loss: 5.6947 -
root_mean_squared_error: 2.3497 - val_loss: 2.9982 -
val_root_mean_squared_error: 1.6854
Epoch 34/100
719/719 ━━━━━━━━━━━━━━━━━━━━ 0s 452us/step - loss: 5.0136 -
root_mean_squared_error: 2.1977 - val_loss: 2.9837 -
val_root_mean_squared_error: 1.6802
Epoch 35/100
719/719 ━━━━━━━━━━━━━━━━━━━━ 0s 422us/step - loss: 4.5453 -
root_mean_squared_error: 2.0923 - val_loss: 2.5615 -
val_root_mean_squared_error: 1.5482
Epoch 36/100
719/719 ━━━━━━━━━━━━━━━━━━━━ 0s 481us/step - loss: 4.8817 -
root_mean_squared_error: 2.1699 - val_loss: 5.0352 -
val_root_mean_squared_error: 2.2066
Epoch 37/100
719/719 ━━━━━━━━━━━━━━━━━━━━ 0s 478us/step - loss: 6.1995 -
root_mean_squared_error: 2.4457 - val_loss: 2.4978 -
val_root_mean_squared_error: 1.5257
Epoch 38/100
719/719 ━━━━━━━━━━━━━━━━━━━━ 0s 475us/step - loss: 4.3422 -
root_mean_squared_error: 2.0376 - val_loss: 3.5325 -
val_root_mean_squared_error: 1.8334
Epoch 39/100
719/719 ━━━━━━━━━━━━━━━━━━━━ 0s 485us/step - loss: 4.6046 -
root_mean_squared_error: 2.1010 - val_loss: 2.4162 -
val_root_mean_squared_error: 1.4983
Epoch 40/100
719/719 ━━━━━━━━━━━━━━━━━━━━ 0s 408us/step - loss: 5.2670 -
root_mean_squared_error: 2.2520 - val_loss: 5.2860 -
val_root_mean_squared_error: 2.2613
Epoch 41/100
719/719 ━━━━━━━━━━━━━━━━━━━━ 0s 465us/step - loss: 4.0780 -
root_mean_squared_error: 1.9704 - val_loss: 2.3272 -
val_root_mean_squared_error: 1.4677
Epoch 42/100
719/719 ━━━━━━━━━━━━━━━━━━━━ 0s 531us/step - loss: 3.8349 -
root_mean_squared_error: 1.9122 - val_loss: 2.4972 -
val_root_mean_squared_error: 1.5244
Epoch 43/100
719/719 ━━━━━━━━━━━━━━━━━━━━ 0s 501us/step - loss: 7.0127 -
```

```
root_mean_squared_error: 2.5991 - val_loss: 2.4881 -
val_root_mean_squared_error: 1.5213
Epoch 44/100
719/719 ━━━━━━━━━━━━━━━━━━━━ 0s 530us/step - loss: 4.3308 -
root_mean_squared_error: 2.0384 - val_loss: 2.3896 -
val_root_mean_squared_error: 1.4885
Epoch 45/100
719/719 ━━━━━━━━━━━━━━━━━━━━ 0s 491us/step - loss: 4.3310 -
root_mean_squared_error: 2.0355 - val_loss: 3.3523 -
val_root_mean_squared_error: 1.7825
Epoch 46/100
719/719 ━━━━━━━━━━━━━━━━━━━━ 0s 475us/step - loss: 4.0028 -
root_mean_squared_error: 1.9516 - val_loss: 2.4184 -
val_root_mean_squared_error: 1.4978
Epoch 47/100
719/719 ━━━━━━━━━━━━━━━━━━━━ 0s 464us/step - loss: 5.0066 -
root_mean_squared_error: 2.1942 - val_loss: 2.6773 -
val_root_mean_squared_error: 1.5819
Epoch 48/100
719/719 ━━━━━━━━━━━━━━━━━━━━ 0s 491us/step - loss: 4.0968 -
root_mean_squared_error: 1.9778 - val_loss: 3.0759 -
val_root_mean_squared_error: 1.7035
Epoch 49/100
719/719 ━━━━━━━━━━━━━━━━━━━━ 0s 470us/step - loss: 3.8905 -
root_mean_squared_error: 1.9271 - val_loss: 2.4234 -
val_root_mean_squared_error: 1.4998
Epoch 50/100
719/719 ━━━━━━━━━━━━━━━━━━━━ 0s 442us/step - loss: 4.4649 -
root_mean_squared_error: 2.0649 - val_loss: 5.7047 -
val_root_mean_squared_error: 2.3518
Epoch 51/100
719/719 ━━━━━━━━━━━━━━━━━━━━ 0s 545us/step - loss: 4.4069 -
root_mean_squared_error: 2.0558 - val_loss: 2.2877 -
val_root_mean_squared_error: 1.4539
Epoch 52/100
719/719 ━━━━━━━━━━━━━━━━━━━━ 0s 462us/step - loss: 3.8976 -
root_mean_squared_error: 1.9288 - val_loss: 2.9395 -
val_root_mean_squared_error: 1.6628
Epoch 53/100
719/719 ━━━━━━━━━━━━━━━━━━━━ 0s 472us/step - loss: 3.6929 -
root_mean_squared_error: 1.8727 - val_loss: 2.5778 -
val_root_mean_squared_error: 1.5503
Epoch 54/100
719/719 ━━━━━━━━━━━━━━━━━━━━ 0s 488us/step - loss: 4.6838 -
root_mean_squared_error: 2.1178 - val_loss: 2.7873 -
val_root_mean_squared_error: 1.6167
Epoch 55/100
719/719 ━━━━━━━━━━━━━━━━━━━━ 0s 476us/step - loss: 3.7637 -
root_mean_squared_error: 1.8937 - val_loss: 2.2800 -
```

```
val_root_mean_squared_error: 1.4515
Epoch 56/100
719/719 ━━━━━━━━━━━━━━━━━━━━ 0s 430us/step - loss: 5.2429 -
root_mean_squared_error: 2.2477 - val_loss: 3.2636 -
val_root_mean_squared_error: 1.7581
Epoch 57/100
719/719 ━━━━━━━━━━━━━━━━━━━━ 0s 524us/step - loss: 3.8877 -
root_mean_squared_error: 1.9254 - val_loss: 2.8684 -
val_root_mean_squared_error: 1.6417
Epoch 58/100
719/719 ━━━━━━━━━━━━━━━━━━━━ 0s 572us/step - loss: 3.9830 -
root_mean_squared_error: 1.9513 - val_loss: 2.3509 -
val_root_mean_squared_error: 1.4761
Epoch 59/100
719/719 ━━━━━━━━━━━━━━━━━━━━ 0s 436us/step - loss: 3.7614 -
root_mean_squared_error: 1.8917 - val_loss: 2.4423 -
val_root_mean_squared_error: 1.5071
Epoch 60/100
719/719 ━━━━━━━━━━━━━━━━━━━━ 0s 496us/step - loss: 4.1878 -
root_mean_squared_error: 2.0009 - val_loss: 3.4162 -
val_root_mean_squared_error: 1.8018
Epoch 61/100
719/719 ━━━━━━━━━━━━━━━━━━━━ 0s 440us/step - loss: 3.8531 -
root_mean_squared_error: 1.9179 - val_loss: 2.5672 -
val_root_mean_squared_error: 1.5487
Epoch 62/100
719/719 ━━━━━━━━━━━━━━━━━━━━ 0s 460us/step - loss: 4.2574 -
root_mean_squared_error: 2.0201 - val_loss: 2.7331 -
val_root_mean_squared_error: 1.6017
Epoch 63/100
719/719 ━━━━━━━━━━━━━━━━━━━━ 0s 551us/step - loss: 3.9520 -
root_mean_squared_error: 1.9440 - val_loss: 2.3164 -
val_root_mean_squared_error: 1.4660
Epoch 64/100
719/719 ━━━━━━━━━━━━━━━━━━━━ 0s 539us/step - loss: 3.7839 -
root_mean_squared_error: 1.8984 - val_loss: 2.9325 -
val_root_mean_squared_error: 1.6631
Epoch 65/100
719/719 ━━━━━━━━━━━━━━━━━━━━ 0s 547us/step - loss: 3.8101 -
root_mean_squared_error: 1.9083 - val_loss: 2.2611 -
val_root_mean_squared_error: 1.4471
Epoch 66/100
719/719 ━━━━━━━━━━━━━━━━━━━━ 0s 478us/step - loss: 3.7712 -
root_mean_squared_error: 1.8976 - val_loss: 2.2599 -
val_root_mean_squared_error: 1.4469
Epoch 67/100
719/719 ━━━━━━━━━━━━━━━━━━━━ 0s 533us/step - loss: 3.9378 -
root_mean_squared_error: 1.9396 - val_loss: 2.7946 -
val_root_mean_squared_error: 1.6216
```

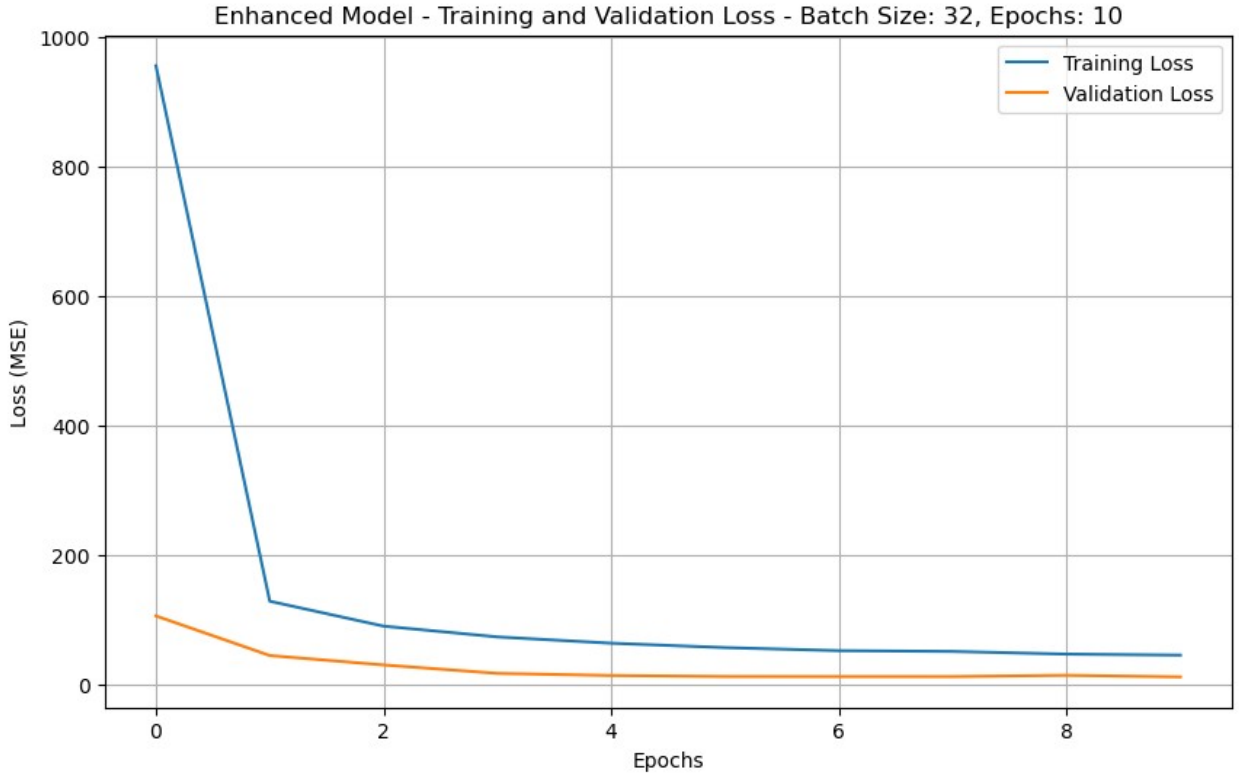
```
Epoch 68/100
719/719 ━━━━━━━━━━━━━━━━━━━━ 0s 437us/step - loss: 5.2260 -
root_mean_squared_error: 2.2444 - val_loss: 2.4916 -
val_root_mean_squared_error: 1.5255
Epoch 69/100
719/719 ━━━━━━━━━━━━━━━━━━━━ 0s 512us/step - loss: 4.5607 -
root_mean_squared_error: 2.0920 - val_loss: 2.0551 -
val_root_mean_squared_error: 1.3751
Epoch 70/100
719/719 ━━━━━━━━━━━━━━━━━━━━ 0s 550us/step - loss: 3.7981 -
root_mean_squared_error: 1.9035 - val_loss: 2.6935 -
val_root_mean_squared_error: 1.5906
Epoch 71/100
719/719 ━━━━━━━━━━━━━━━━━━━━ 0s 491us/step - loss: 3.9017 -
root_mean_squared_error: 1.9282 - val_loss: 3.1918 -
val_root_mean_squared_error: 1.7404
Epoch 72/100
719/719 ━━━━━━━━━━━━━━━━━━━━ 0s 499us/step - loss: 4.3144 -
root_mean_squared_error: 2.0287 - val_loss: 2.0944 -
val_root_mean_squared_error: 1.3895
Epoch 73/100
719/719 ━━━━━━━━━━━━━━━━━━━━ 0s 491us/step - loss: 3.5681 -
root_mean_squared_error: 1.8427 - val_loss: 2.0642 -
val_root_mean_squared_error: 1.3789
Epoch 74/100
719/719 ━━━━━━━━━━━━━━━━━━━━ 0s 506us/step - loss: 3.7974 -
root_mean_squared_error: 1.9053 - val_loss: 2.0575 -
val_root_mean_squared_error: 1.3765
Epoch 75/100
719/719 ━━━━━━━━━━━━━━━━━━━━ 0s 464us/step - loss: 3.8243 -
root_mean_squared_error: 1.9119 - val_loss: 2.0856 -
val_root_mean_squared_error: 1.3866
Epoch 76/100
719/719 ━━━━━━━━━━━━━━━━━━━━ 0s 494us/step - loss: 3.4107 -
root_mean_squared_error: 1.8012 - val_loss: 3.1621 -
val_root_mean_squared_error: 1.7318
Epoch 77/100
719/719 ━━━━━━━━━━━━━━━━━━━━ 0s 560us/step - loss: 3.8235 -
root_mean_squared_error: 1.9103 - val_loss: 2.1605 -
val_root_mean_squared_error: 1.4137
Epoch 78/100
719/719 ━━━━━━━━━━━━━━━━━━━━ 0s 507us/step - loss: 3.9417 -
root_mean_squared_error: 1.9377 - val_loss: 2.2090 -
val_root_mean_squared_error: 1.4307
Epoch 79/100
719/719 ━━━━━━━━━━━━━━━━━━━━ 0s 456us/step - loss: 3.8767 -
root_mean_squared_error: 1.9256 - val_loss: 2.1572 -
val_root_mean_squared_error: 1.4124
```

```
Completed training with Batch Size: 16, Epochs: 100. Test MSE: 2.3301,
Test RMSE: 1.4717
```




```
Training with Batch Size: 32, Epochs: 10
Epoch 1/10
360/360 ━━━━━━━━━━━━━━━━━━━━ 1s 880us/step - loss: 2138.6243 -
root_mean_squared_error: 44.7859 - val_loss: 106.2183 -
val_root_mean_squared_error: 10.2968
Epoch 2/10
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 801us/step - loss: 143.6381 -
root_mean_squared_error: 11.9710 - val_loss: 45.0126 -
val_root_mean_squared_error: 6.6948
Epoch 3/10
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 557us/step - loss: 96.4528 -
root_mean_squared_error: 9.8086 - val_loss: 30.4639 -
val_root_mean_squared_error: 5.5028
Epoch 4/10
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 449us/step - loss: 77.2064 -
root_mean_squared_error: 8.7739 - val_loss: 17.4433 -
val_root_mean_squared_error: 4.1555
Epoch 5/10
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 533us/step - loss: 67.7129 -
root_mean_squared_error: 8.2168 - val_loss: 14.1416 -
val_root_mean_squared_error: 3.7382
```

```
Epoch 6/10
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 580us/step - loss: 57.2411 -
root_mean_squared_error: 7.5539 - val_loss: 12.6091 -
val_root_mean_squared_error: 3.5283
Epoch 7/10
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 503us/step - loss: 53.7107 -
root_mean_squared_error: 7.3176 - val_loss: 12.5171 -
val_root_mean_squared_error: 3.5162
Epoch 8/10
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 510us/step - loss: 51.5177 -
root_mean_squared_error: 7.1667 - val_loss: 12.4740 -
val_root_mean_squared_error: 3.5110
Epoch 9/10
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 547us/step - loss: 47.8486 -
root_mean_squared_error: 6.9062 - val_loss: 14.4241 -
val_root_mean_squared_error: 3.7795
Epoch 10/10
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 446us/step - loss: 45.7417 -
root_mean_squared_error: 6.7522 - val_loss: 11.9627 -
val_root_mean_squared_error: 3.4393
Completed training with Batch Size: 32, Epochs: 10. Test MSE: 11.4570,
Test RMSE: 3.3650
```




```
Training with Batch Size: 32, Epochs: 50
```

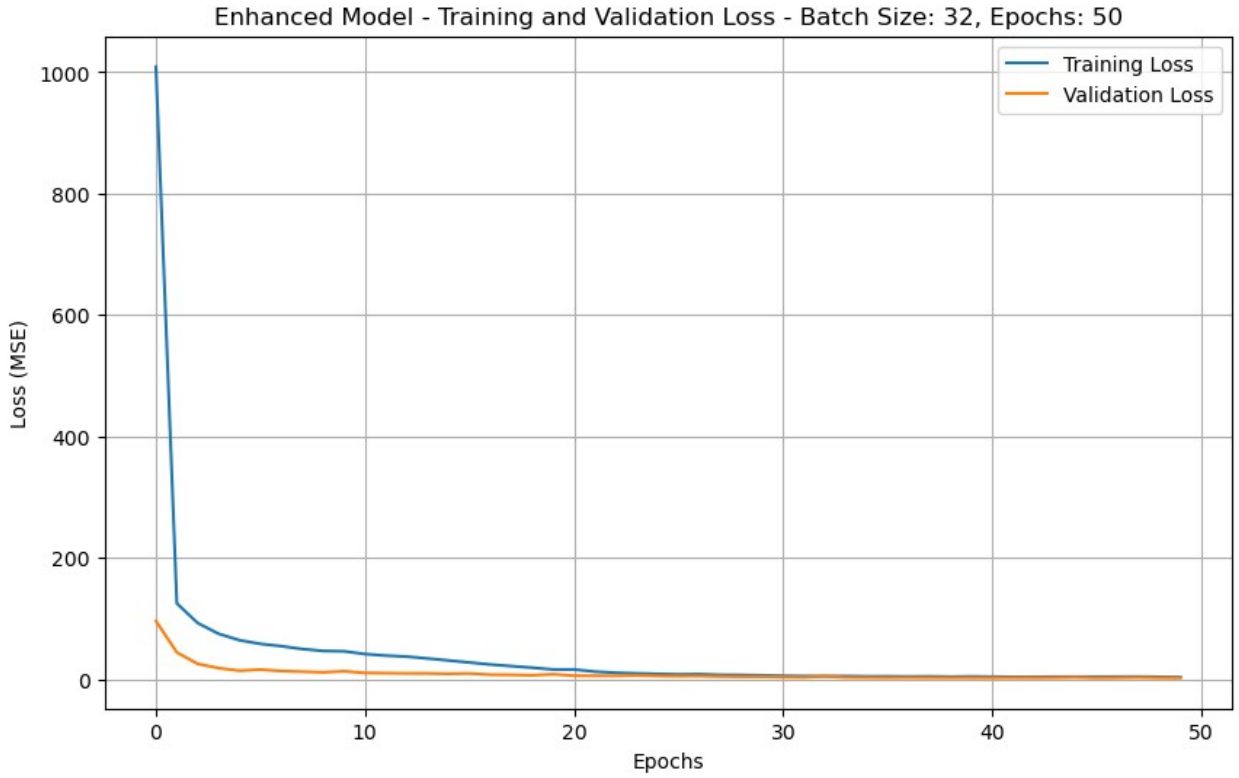
```
Epoch 1/50
360/360 ━━━━━━━━━━━━━━━━━━━━ 1s 925us/step - loss: 2247.1284 -
root_mean_squared_error: 45.9842 - val_loss: 96.0804 -
val_root_mean_squared_error: 9.7915
Epoch 2/50
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 651us/step - loss: 135.9865 -
root_mean_squared_error: 11.6484 - val_loss: 44.3170 -
val_root_mean_squared_error: 6.6419
Epoch 3/50
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 507us/step - loss: 97.0622 -
root_mean_squared_error: 9.8398 - val_loss: 25.4245 -
val_root_mean_squared_error: 5.0230
Epoch 4/50
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 550us/step - loss: 79.3725 -
root_mean_squared_error: 8.8954 - val_loss: 18.4332 -
val_root_mean_squared_error: 4.2718
Epoch 5/50
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 572us/step - loss: 65.4415 -
root_mean_squared_error: 8.0780 - val_loss: 14.2436 -
val_root_mean_squared_error: 3.7506
Epoch 6/50
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 491us/step - loss: 58.0887 -
root_mean_squared_error: 7.6090 - val_loss: 15.9947 -
val_root_mean_squared_error: 3.9781
Epoch 7/50
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 622us/step - loss: 54.7334 -
root_mean_squared_error: 7.3867 - val_loss: 13.6623 -
val_root_mean_squared_error: 3.6741
Epoch 8/50
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 621us/step - loss: 49.5088 -
root_mean_squared_error: 7.0232 - val_loss: 12.6261 -
val_root_mean_squared_error: 3.5313
Epoch 9/50
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 593us/step - loss: 49.9441 -
root_mean_squared_error: 7.0538 - val_loss: 11.5487 -
val_root_mean_squared_error: 3.3762
Epoch 10/50
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 487us/step - loss: 46.6199 -
root_mean_squared_error: 6.8157 - val_loss: 13.3160 -
val_root_mean_squared_error: 3.6295
Epoch 11/50
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 585us/step - loss: 42.3080 -
root_mean_squared_error: 6.4933 - val_loss: 10.4863 -
val_root_mean_squared_error: 3.2168
Epoch 12/50
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 553us/step - loss: 39.7162 -
root_mean_squared_error: 6.2887 - val_loss: 10.1395 -
val_root_mean_squared_error: 3.1635
Epoch 13/50
```

```
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 488us/step - loss: 38.6976 -
root_mean_squared_error: 6.2082 - val_loss: 9.7671 -
val_root_mean_squared_error: 3.1049
Epoch 14/50
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 455us/step - loss: 35.3715 -
root_mean_squared_error: 5.9362 - val_loss: 9.6626 -
val_root_mean_squared_error: 3.0890
Epoch 15/50
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 459us/step - loss: 32.2040 -
root_mean_squared_error: 5.6636 - val_loss: 9.0368 -
val_root_mean_squared_error: 2.9869
Epoch 16/50
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 610us/step - loss: 28.8150 -
root_mean_squared_error: 5.3565 - val_loss: 9.5481 -
val_root_mean_squared_error: 3.0721
Epoch 17/50
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 468us/step - loss: 25.4478 -
root_mean_squared_error: 5.0327 - val_loss: 7.6249 -
val_root_mean_squared_error: 2.7421
Epoch 18/50
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 578us/step - loss: 22.4445 -
root_mean_squared_error: 4.7259 - val_loss: 7.3705 -
val_root_mean_squared_error: 2.6959
Epoch 19/50
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 510us/step - loss: 19.6814 -
root_mean_squared_error: 4.4246 - val_loss: 6.6056 -
val_root_mean_squared_error: 2.5506
Epoch 20/50
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 580us/step - loss: 16.5015 -
root_mean_squared_error: 4.0491 - val_loss: 8.3631 -
val_root_mean_squared_error: 2.8751
Epoch 21/50
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 570us/step - loss: 19.3630 -
root_mean_squared_error: 4.3698 - val_loss: 5.8729 -
val_root_mean_squared_error: 2.4034
Epoch 22/50
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 551us/step - loss: 13.1308 -
root_mean_squared_error: 3.6099 - val_loss: 5.7511 -
val_root_mean_squared_error: 2.3784
Epoch 23/50
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 454us/step - loss: 11.1543 -
root_mean_squared_error: 3.3252 - val_loss: 5.6653 -
val_root_mean_squared_error: 2.3607
Epoch 24/50
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 536us/step - loss: 9.6451 -
root_mean_squared_error: 3.0901 - val_loss: 6.2750 -
val_root_mean_squared_error: 2.4865
Epoch 25/50
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 497us/step - loss: 9.2123 -
```

```
root_mean_squared_error: 3.0191 - val_loss: 5.4917 -
val_root_mean_squared_error: 2.3237
Epoch 26/50
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 641us/step - loss: 8.4900 -
root_mean_squared_error: 2.8971 - val_loss: 5.1065 -
val_root_mean_squared_error: 2.2392
Epoch 27/50
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 542us/step - loss: 7.8546 -
root_mean_squared_error: 2.7852 - val_loss: 5.2763 -
val_root_mean_squared_error: 2.2762
Epoch 28/50
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 503us/step - loss: 7.8694 -
root_mean_squared_error: 2.7851 - val_loss: 4.6271 -
val_root_mean_squared_error: 2.1279
Epoch 29/50
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 710us/step - loss: 6.9849 -
root_mean_squared_error: 2.6231 - val_loss: 4.2763 -
val_root_mean_squared_error: 2.0426
Epoch 30/50
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 626us/step - loss: 6.5998 -
root_mean_squared_error: 2.5482 - val_loss: 4.2021 -
val_root_mean_squared_error: 2.0226
Epoch 31/50
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 445us/step - loss: 5.8097 -
root_mean_squared_error: 2.3849 - val_loss: 3.5850 -
val_root_mean_squared_error: 1.8623
Epoch 32/50
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 539us/step - loss: 5.3786 -
root_mean_squared_error: 2.2923 - val_loss: 3.2937 -
val_root_mean_squared_error: 1.7804
Epoch 33/50
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 539us/step - loss: 4.9960 -
root_mean_squared_error: 2.2066 - val_loss: 4.9757 -
val_root_mean_squared_error: 2.2015
Epoch 34/50
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 738us/step - loss: 5.3012 -
root_mean_squared_error: 2.2715 - val_loss: 3.0429 -
val_root_mean_squared_error: 1.7053
Epoch 35/50
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 556us/step - loss: 4.6893 -
root_mean_squared_error: 2.1324 - val_loss: 2.8782 -
val_root_mean_squared_error: 1.6547
Epoch 36/50
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 614us/step - loss: 4.9256 -
root_mean_squared_error: 2.1866 - val_loss: 2.5051 -
val_root_mean_squared_error: 1.5360
Epoch 37/50
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 544us/step - loss: 4.3972 -
root_mean_squared_error: 2.0606 - val_loss: 2.7754 -
```

```
val_root_mean_squared_error: 1.6201
Epoch 38/50
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 529us/step - loss: 4.4960 -
root_mean_squared_error: 2.0821 - val_loss: 2.5871 -
val_root_mean_squared_error: 1.5595
Epoch 39/50
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 558us/step - loss: 4.1879 -
root_mean_squared_error: 2.0064 - val_loss: 2.5527 -
val_root_mean_squared_error: 1.5470
Epoch 40/50
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 576us/step - loss: 4.6644 -
root_mean_squared_error: 2.1211 - val_loss: 2.5398 -
val_root_mean_squared_error: 1.5418
Epoch 41/50
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 546us/step - loss: 4.1582 -
root_mean_squared_error: 1.9976 - val_loss: 2.4559 -
val_root_mean_squared_error: 1.5128
Epoch 42/50
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 493us/step - loss: 4.0020 -
root_mean_squared_error: 1.9560 - val_loss: 2.7078 -
val_root_mean_squared_error: 1.5929
Epoch 43/50
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 491us/step - loss: 4.1686 -
root_mean_squared_error: 1.9972 - val_loss: 2.5298 -
val_root_mean_squared_error: 1.5352
Epoch 44/50
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 484us/step - loss: 3.8502 -
root_mean_squared_error: 1.9157 - val_loss: 2.6299 -
val_root_mean_squared_error: 1.5667
Epoch 45/50
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 454us/step - loss: 4.0493 -
root_mean_squared_error: 1.9674 - val_loss: 3.2022 -
val_root_mean_squared_error: 1.7389
Epoch 46/50
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 561us/step - loss: 4.3103 -
root_mean_squared_error: 2.0314 - val_loss: 2.2810 -
val_root_mean_squared_error: 1.4492
Epoch 47/50
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 481us/step - loss: 4.2364 -
root_mean_squared_error: 2.0128 - val_loss: 2.5322 -
val_root_mean_squared_error: 1.5329
Epoch 48/50
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 567us/step - loss: 4.3416 -
root_mean_squared_error: 2.0377 - val_loss: 3.1032 -
val_root_mean_squared_error: 1.7089
Epoch 49/50
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 537us/step - loss: 4.0619 -
root_mean_squared_error: 1.9685 - val_loss: 2.4480 -
val_root_mean_squared_error: 1.5043
```

```
Epoch 50/50
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 562us/step - loss: 3.7460 -
root_mean_squared_error: 1.8861 - val_loss: 2.4728 -
val_root_mean_squared_error: 1.5121
Completed training with Batch Size: 32, Epochs: 50. Test MSE: 2.5808,
Test RMSE: 1.5492
```




```
Training with Batch Size: 32, Epochs: 100
Epoch 1/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 1s 692us/step - loss: 2108.4529 -
root_mean_squared_error: 44.4266 - val_loss: 95.0974 -
val_root_mean_squared_error: 9.7414
Epoch 2/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 584us/step - loss: 138.8202 -
root_mean_squared_error: 11.7695 - val_loss: 42.8738 -
val_root_mean_squared_error: 6.5327
Epoch 3/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 545us/step - loss: 95.3082 -
root_mean_squared_error: 9.7500 - val_loss: 24.8076 -
val_root_mean_squared_error: 4.9618
Epoch 4/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 503us/step - loss: 76.4518 -
root_mean_squared_error: 8.7321 - val_loss: 19.6487 -
val_root_mean_squared_error: 4.4124
```

```
Epoch 5/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 442us/step - loss: 64.7370 -
root_mean_squared_error: 8.0339 - val_loss: 15.3601 -
val_root_mean_squared_error: 3.8971
Epoch 6/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 544us/step - loss: 59.3659 -
root_mean_squared_error: 7.6910 - val_loss: 13.3178 -
val_root_mean_squared_error: 3.6270
Epoch 7/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 508us/step - loss: 54.0541 -
root_mean_squared_error: 7.3396 - val_loss: 13.9594 -
val_root_mean_squared_error: 3.7155
Epoch 8/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 471us/step - loss: 45.4235 -
root_mean_squared_error: 6.7278 - val_loss: 11.3950 -
val_root_mean_squared_error: 3.3536
Epoch 9/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 561us/step - loss: 43.4752 -
root_mean_squared_error: 6.5815 - val_loss: 12.0161 -
val_root_mean_squared_error: 3.4460
Epoch 10/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 492us/step - loss: 40.8492 -
root_mean_squared_error: 6.3800 - val_loss: 13.3518 -
val_root_mean_squared_error: 3.6356
Epoch 11/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 545us/step - loss: 36.5048 -
root_mean_squared_error: 6.0301 - val_loss: 13.0964 -
val_root_mean_squared_error: 3.6011
Epoch 12/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 484us/step - loss: 34.6861 -
root_mean_squared_error: 5.8779 - val_loss: 11.1409 -
val_root_mean_squared_error: 3.3194
Epoch 13/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 564us/step - loss: 31.4484 -
root_mean_squared_error: 5.5955 - val_loss: 11.7322 -
val_root_mean_squared_error: 3.4082
Epoch 14/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 489us/step - loss: 28.6535 -
root_mean_squared_error: 5.3385 - val_loss: 11.8602 -
val_root_mean_squared_error: 3.4277
Epoch 15/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 509us/step - loss: 26.0915 -
root_mean_squared_error: 5.0964 - val_loss: 12.1877 -
val_root_mean_squared_error: 3.4760
Epoch 16/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 719us/step - loss: 22.9508 -
root_mean_squared_error: 4.7786 - val_loss: 11.8140 -
val_root_mean_squared_error: 3.4225
Epoch 17/100
```

```
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 521us/step - loss: 19.4484 -
root_mean_squared_error: 4.3984 - val_loss: 10.9140 -
val_root_mean_squared_error: 3.2890
Epoch 18/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 621us/step - loss: 18.5570 -
root_mean_squared_error: 4.2959 - val_loss: 14.0149 -
val_root_mean_squared_error: 3.7313
Epoch 19/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 597us/step - loss: 16.9348 -
root_mean_squared_error: 4.1032 - val_loss: 10.8992 -
val_root_mean_squared_error: 3.2878
Epoch 20/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 531us/step - loss: 15.7970 -
root_mean_squared_error: 3.9620 - val_loss: 11.0187 -
val_root_mean_squared_error: 3.3063
Epoch 21/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 480us/step - loss: 14.5388 -
root_mean_squared_error: 3.8013 - val_loss: 10.8926 -
val_root_mean_squared_error: 3.2875
Epoch 22/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 475us/step - loss: 13.8588 -
root_mean_squared_error: 3.7106 - val_loss: 11.0475 -
val_root_mean_squared_error: 3.3113
Epoch 23/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 651us/step - loss: 13.7272 -
root_mean_squared_error: 3.6932 - val_loss: 10.8757 -
val_root_mean_squared_error: 3.2855
Epoch 24/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 676us/step - loss: 12.5473 -
root_mean_squared_error: 3.5306 - val_loss: 11.2887 -
val_root_mean_squared_error: 3.3478
Epoch 25/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 450us/step - loss: 12.9999 -
root_mean_squared_error: 3.5937 - val_loss: 11.6472 -
val_root_mean_squared_error: 3.4010
Epoch 26/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 593us/step - loss: 12.8242 -
root_mean_squared_error: 3.5686 - val_loss: 10.6568 -
val_root_mean_squared_error: 3.2519
Epoch 27/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 546us/step - loss: 12.4726 -
root_mean_squared_error: 3.5192 - val_loss: 10.2209 -
val_root_mean_squared_error: 3.1839
Epoch 28/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 546us/step - loss: 11.1249 -
root_mean_squared_error: 3.3225 - val_loss: 8.9207 -
val_root_mean_squared_error: 2.9723
Epoch 29/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 525us/step - loss: 10.4187 -
```

```
root_mean_squared_error: 3.2135 - val_loss: 7.6677 -
val_root_mean_squared_error: 2.7527
Epoch 30/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 496us/step - loss: 9.6146 -
root_mean_squared_error: 3.0855 - val_loss: 6.9098 -
val_root_mean_squared_error: 2.6105
Epoch 31/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 584us/step - loss: 8.3576 -
root_mean_squared_error: 2.8739 - val_loss: 7.1502 -
val_root_mean_squared_error: 2.6555
Epoch 32/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 579us/step - loss: 8.0528 -
root_mean_squared_error: 2.8178 - val_loss: 7.4116 -
val_root_mean_squared_error: 2.7035
Epoch 33/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 555us/step - loss: 9.3625 -
root_mean_squared_error: 3.0367 - val_loss: 5.5235 -
val_root_mean_squared_error: 2.3271
Epoch 34/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 482us/step - loss: 6.9265 -
root_mean_squared_error: 2.6105 - val_loss: 4.9327 -
val_root_mean_squared_error: 2.1954
Epoch 35/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 532us/step - loss: 6.5218 -
root_mean_squared_error: 2.5311 - val_loss: 4.3370 -
val_root_mean_squared_error: 2.0542
Epoch 36/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 489us/step - loss: 6.1900 -
root_mean_squared_error: 2.4636 - val_loss: 4.3317 -
val_root_mean_squared_error: 2.0516
Epoch 37/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 478us/step - loss: 5.9563 -
root_mean_squared_error: 2.4149 - val_loss: 4.0475 -
val_root_mean_squared_error: 1.9798
Epoch 38/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 586us/step - loss: 6.3295 -
root_mean_squared_error: 2.4886 - val_loss: 4.0541 -
val_root_mean_squared_error: 1.9797
Epoch 39/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 567us/step - loss: 5.5828 -
root_mean_squared_error: 2.3334 - val_loss: 3.5172 -
val_root_mean_squared_error: 1.8376
Epoch 40/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 461us/step - loss: 5.2967 -
root_mean_squared_error: 2.2695 - val_loss: 3.7350 -
val_root_mean_squared_error: 1.8946
Epoch 41/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 526us/step - loss: 5.2612 -
root_mean_squared_error: 2.2601 - val_loss: 3.0596 -
```

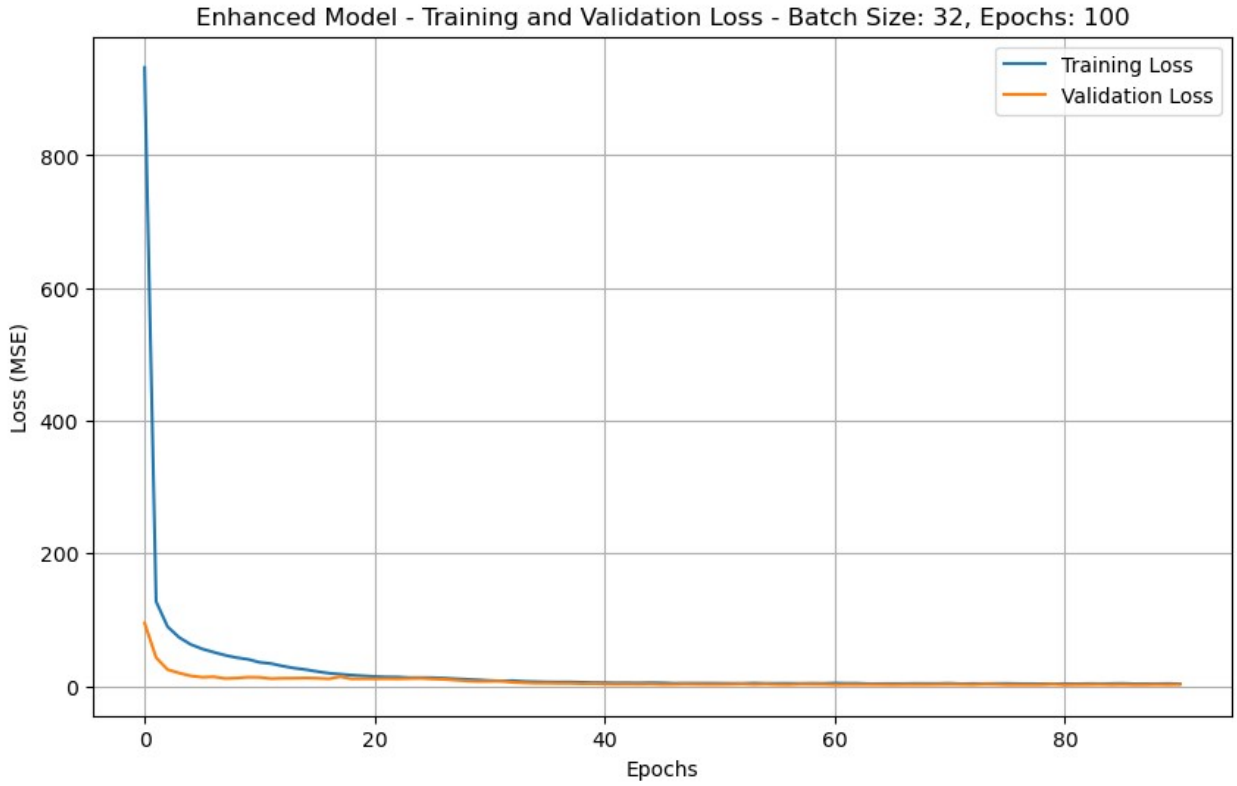
```
val_root_mean_squared_error: 1.7055
Epoch 42/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 422us/step - loss: 4.7537 -
root_mean_squared_error: 2.1445 - val_loss: 2.8787 -
val_root_mean_squared_error: 1.6501
Epoch 43/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 655us/step - loss: 5.0344 -
root_mean_squared_error: 2.2077 - val_loss: 2.9687 -
val_root_mean_squared_error: 1.6758
Epoch 44/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 670us/step - loss: 4.8294 -
root_mean_squared_error: 2.1594 - val_loss: 2.7148 -
val_root_mean_squared_error: 1.5969
Epoch 45/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 477us/step - loss: 4.6558 -
root_mean_squared_error: 2.1160 - val_loss: 3.0586 -
val_root_mean_squared_error: 1.7001
Epoch 46/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 526us/step - loss: 5.4761 -
root_mean_squared_error: 2.2941 - val_loss: 2.3876 -
val_root_mean_squared_error: 1.4892
Epoch 47/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 466us/step - loss: 4.0469 -
root_mean_squared_error: 1.9684 - val_loss: 2.5831 -
val_root_mean_squared_error: 1.5523
Epoch 48/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 488us/step - loss: 4.3868 -
root_mean_squared_error: 2.0505 - val_loss: 2.9573 -
val_root_mean_squared_error: 1.6677
Epoch 49/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 513us/step - loss: 4.1227 -
root_mean_squared_error: 1.9861 - val_loss: 2.6456 -
val_root_mean_squared_error: 1.5704
Epoch 50/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 572us/step - loss: 4.2096 -
root_mean_squared_error: 2.0042 - val_loss: 2.4544 -
val_root_mean_squared_error: 1.5075
Epoch 51/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 514us/step - loss: 4.1360 -
root_mean_squared_error: 1.9870 - val_loss: 2.5100 -
val_root_mean_squared_error: 1.5250
Epoch 52/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 438us/step - loss: 4.0215 -
root_mean_squared_error: 1.9577 - val_loss: 2.6921 -
val_root_mean_squared_error: 1.5833
Epoch 53/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 543us/step - loss: 3.9515 -
root_mean_squared_error: 1.9393 - val_loss: 3.2067 -
val_root_mean_squared_error: 1.7381
```

```
Epoch 54/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 474us/step - loss: 4.6079 -
root_mean_squared_error: 2.1012 - val_loss: 2.3528 -
val_root_mean_squared_error: 1.4717
Epoch 55/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 601us/step - loss: 3.6475 -
root_mean_squared_error: 1.8585 - val_loss: 3.0105 -
val_root_mean_squared_error: 1.6801
Epoch 56/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 717us/step - loss: 3.8790 -
root_mean_squared_error: 1.9205 - val_loss: 2.3004 -
val_root_mean_squared_error: 1.4531
Epoch 57/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 466us/step - loss: 4.5820 -
root_mean_squared_error: 2.0872 - val_loss: 2.1394 -
val_root_mean_squared_error: 1.3968
Epoch 58/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 576us/step - loss: 3.8070 -
root_mean_squared_error: 1.9010 - val_loss: 2.8884 -
val_root_mean_squared_error: 1.6428
Epoch 59/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 535us/step - loss: 4.6696 -
root_mean_squared_error: 2.1102 - val_loss: 2.7532 -
val_root_mean_squared_error: 1.6011
Epoch 60/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 495us/step - loss: 3.8310 -
root_mean_squared_error: 1.9077 - val_loss: 2.6494 -
val_root_mean_squared_error: 1.5682
Epoch 61/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 601us/step - loss: 4.5860 -
root_mean_squared_error: 2.0953 - val_loss: 2.0533 -
val_root_mean_squared_error: 1.3647
Epoch 62/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 480us/step - loss: 5.1039 -
root_mean_squared_error: 2.2034 - val_loss: 2.3378 -
val_root_mean_squared_error: 1.4649
Epoch 63/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 516us/step - loss: 4.1126 -
root_mean_squared_error: 1.9787 - val_loss: 2.2659 -
val_root_mean_squared_error: 1.4401
Epoch 64/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 549us/step - loss: 3.7249 -
root_mean_squared_error: 1.8783 - val_loss: 2.3586 -
val_root_mean_squared_error: 1.4720
Epoch 65/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 602us/step - loss: 3.6208 -
root_mean_squared_error: 1.8512 - val_loss: 2.2014 -
val_root_mean_squared_error: 1.4175
Epoch 66/100
```

```
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 602us/step - loss: 3.9208 -
root_mean_squared_error: 1.9300 - val_loss: 2.0060 -
val_root_mean_squared_error: 1.3471
Epoch 67/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 523us/step - loss: 3.4559 -
root_mean_squared_error: 1.8059 - val_loss: 2.2257 -
val_root_mean_squared_error: 1.4264
Epoch 68/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 625us/step - loss: 3.6698 -
root_mean_squared_error: 1.8636 - val_loss: 2.2531 -
val_root_mean_squared_error: 1.4356
Epoch 69/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 527us/step - loss: 3.6715 -
root_mean_squared_error: 1.8644 - val_loss: 2.2531 -
val_root_mean_squared_error: 1.4359
Epoch 70/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 542us/step - loss: 3.8823 -
root_mean_squared_error: 1.9196 - val_loss: 2.4216 -
val_root_mean_squared_error: 1.4929
Epoch 71/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 580us/step - loss: 4.6214 -
root_mean_squared_error: 2.1001 - val_loss: 2.6378 -
val_root_mean_squared_error: 1.5636
Epoch 72/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 435us/step - loss: 3.1586 -
root_mean_squared_error: 1.7207 - val_loss: 2.4423 -
val_root_mean_squared_error: 1.5003
Epoch 73/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 552us/step - loss: 3.7712 -
root_mean_squared_error: 1.8874 - val_loss: 2.0754 -
val_root_mean_squared_error: 1.3727
Epoch 74/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 516us/step - loss: 3.5703 -
root_mean_squared_error: 1.8377 - val_loss: 2.8024 -
val_root_mean_squared_error: 1.6160
Epoch 75/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 460us/step - loss: 3.3922 -
root_mean_squared_error: 1.7877 - val_loss: 2.6019 -
val_root_mean_squared_error: 1.5529
Epoch 76/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 590us/step - loss: 4.2153 -
root_mean_squared_error: 2.0034 - val_loss: 1.9857 -
val_root_mean_squared_error: 1.3407
Epoch 77/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 574us/step - loss: 3.6907 -
root_mean_squared_error: 1.8688 - val_loss: 2.0612 -
val_root_mean_squared_error: 1.3687
Epoch 78/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 633us/step - loss: 3.6032 -
```

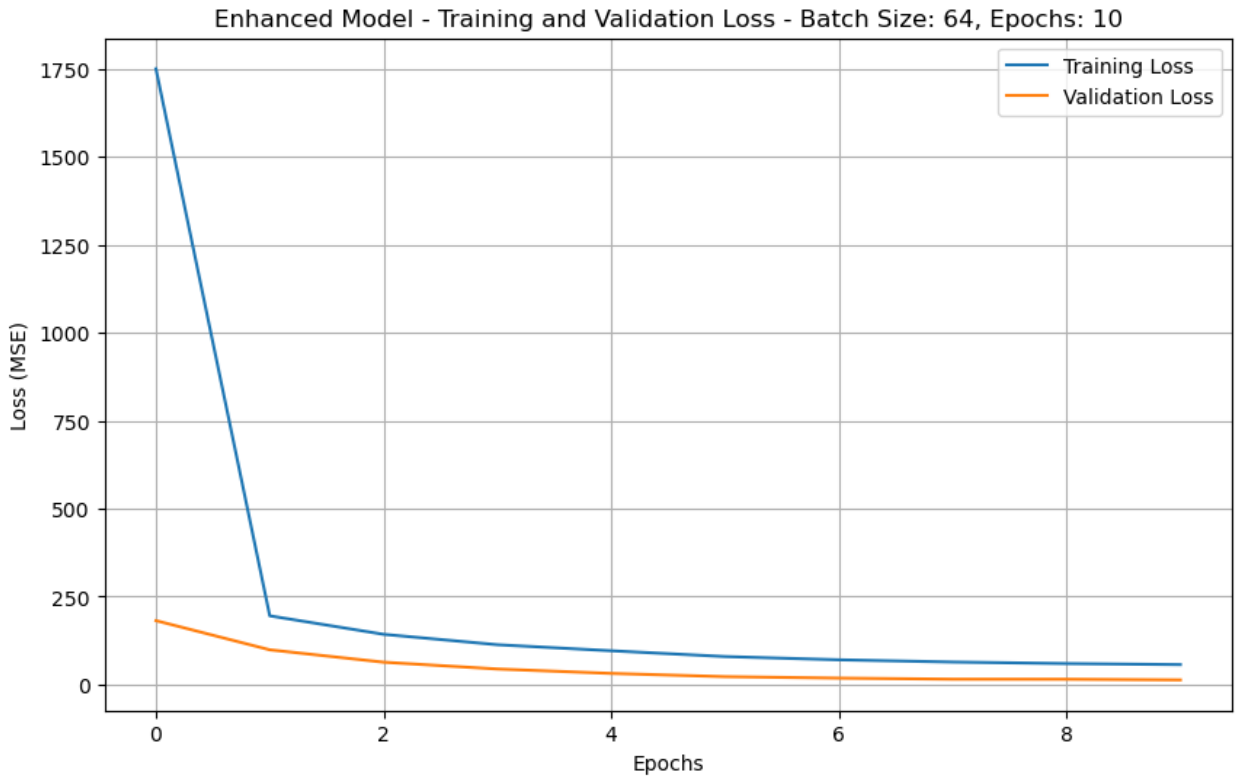
```
root_mean_squared_error: 1.8471 - val_loss: 2.0204 -
val_root_mean_squared_error: 1.3542
Epoch 79/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 446us/step - loss: 3.3161 -
root_mean_squared_error: 1.7682 - val_loss: 2.1290 -
val_root_mean_squared_error: 1.3938
Epoch 80/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 690us/step - loss: 3.6430 -
root_mean_squared_error: 1.8544 - val_loss: 2.8918 -
val_root_mean_squared_error: 1.6450
Epoch 81/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 581us/step - loss: 3.3619 -
root_mean_squared_error: 1.7809 - val_loss: 1.8370 -
val_root_mean_squared_error: 1.2855
Epoch 82/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 577us/step - loss: 3.3333 -
root_mean_squared_error: 1.7733 - val_loss: 2.2123 -
val_root_mean_squared_error: 1.4240
Epoch 83/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 486us/step - loss: 3.2694 -
root_mean_squared_error: 1.7559 - val_loss: 2.1327 -
val_root_mean_squared_error: 1.3962
Epoch 84/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 612us/step - loss: 3.6621 -
root_mean_squared_error: 1.8642 - val_loss: 2.5642 -
val_root_mean_squared_error: 1.5432
Epoch 85/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 525us/step - loss: 3.8352 -
root_mean_squared_error: 1.9099 - val_loss: 1.9854 -
val_root_mean_squared_error: 1.3426
Epoch 86/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 470us/step - loss: 4.0867 -
root_mean_squared_error: 1.9743 - val_loss: 2.3383 -
val_root_mean_squared_error: 1.4681
Epoch 87/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 452us/step - loss: 3.5822 -
root_mean_squared_error: 1.8397 - val_loss: 2.0225 -
val_root_mean_squared_error: 1.3566
Epoch 88/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 617us/step - loss: 3.2489 -
root_mean_squared_error: 1.7507 - val_loss: 2.0948 -
val_root_mean_squared_error: 1.3830
Epoch 89/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 433us/step - loss: 3.2340 -
root_mean_squared_error: 1.7459 - val_loss: 2.3652 -
val_root_mean_squared_error: 1.4778
Epoch 90/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 578us/step - loss: 3.8187 -
root_mean_squared_error: 1.9058 - val_loss: 2.0227 -
```

```
val_root_mean_squared_error: 1.3571
Epoch 91/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 513us/step - loss: 3.4181 -
root_mean_squared_error: 1.7966 - val_loss: 2.4696 -
val_root_mean_squared_error: 1.5132
Completed training with Batch Size: 32, Epochs: 100. Test MSE: 2.1473,
Test RMSE: 1.4011
```




```
Training with Batch Size: 64, Epochs: 10
Epoch 1/10
180/180 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - loss: 3077.5327 -
root_mean_squared_error: 54.8687 - val_loss: 181.6195 -
val_root_mean_squared_error: 13.4688
Epoch 2/10
180/180 ━━━━━━━━━━━━━━━━━━━━ 0s 507us/step - loss: 218.1218 -
root_mean_squared_error: 14.7526 - val_loss: 98.6315 -
val_root_mean_squared_error: 9.9205
Epoch 3/10
180/180 ━━━━━━━━━━━━━━━━━━━━ 0s 684us/step - loss: 152.0883 -
root_mean_squared_error: 12.3203 - val_loss: 63.4289 -
val_root_mean_squared_error: 7.9509
Epoch 4/10
180/180 ━━━━━━━━━━━━━━━━━━━━ 0s 633us/step - loss: 120.8546 -
root_mean_squared_error: 10.9803 - val_loss: 44.0760 -
```

```
val_root_mean_squared_error: 6.6233
Epoch 5/10
180/180 ━━━━━━━━━━━━━━━━━━━━ 0s 614us/step - loss: 101.6822 -
root_mean_squared_error: 10.0709 - val_loss: 31.5430 -
val_root_mean_squared_error: 5.5983
Epoch 6/10
180/180 ━━━━━━━━━━━━━━━━━━━━ 0s 607us/step - loss: 83.4829 -
root_mean_squared_error: 9.1242 - val_loss: 22.2800 -
val_root_mean_squared_error: 4.6992
Epoch 7/10
180/180 ━━━━━━━━━━━━━━━━━━━━ 0s 620us/step - loss: 72.6840 -
root_mean_squared_error: 8.5113 - val_loss: 18.1629 -
val_root_mean_squared_error: 4.2391
Epoch 8/10
180/180 ━━━━━━━━━━━━━━━━━━━━ 0s 588us/step - loss: 64.9643 -
root_mean_squared_error: 8.0477 - val_loss: 15.0834 -
val_root_mean_squared_error: 3.8595
Epoch 9/10
180/180 ━━━━━━━━━━━━━━━━━━━━ 0s 637us/step - loss: 59.7769 -
root_mean_squared_error: 7.7191 - val_loss: 14.9695 -
val_root_mean_squared_error: 3.8455
Epoch 10/10
180/180 ━━━━━━━━━━━━━━━━━━━━ 0s 757us/step - loss: 57.8650 -
root_mean_squared_error: 7.5939 - val_loss: 13.0397 -
val_root_mean_squared_error: 3.5864
Completed training with Batch Size: 64, Epochs: 10. Test MSE: 12.5764,
Test RMSE: 3.5212
```



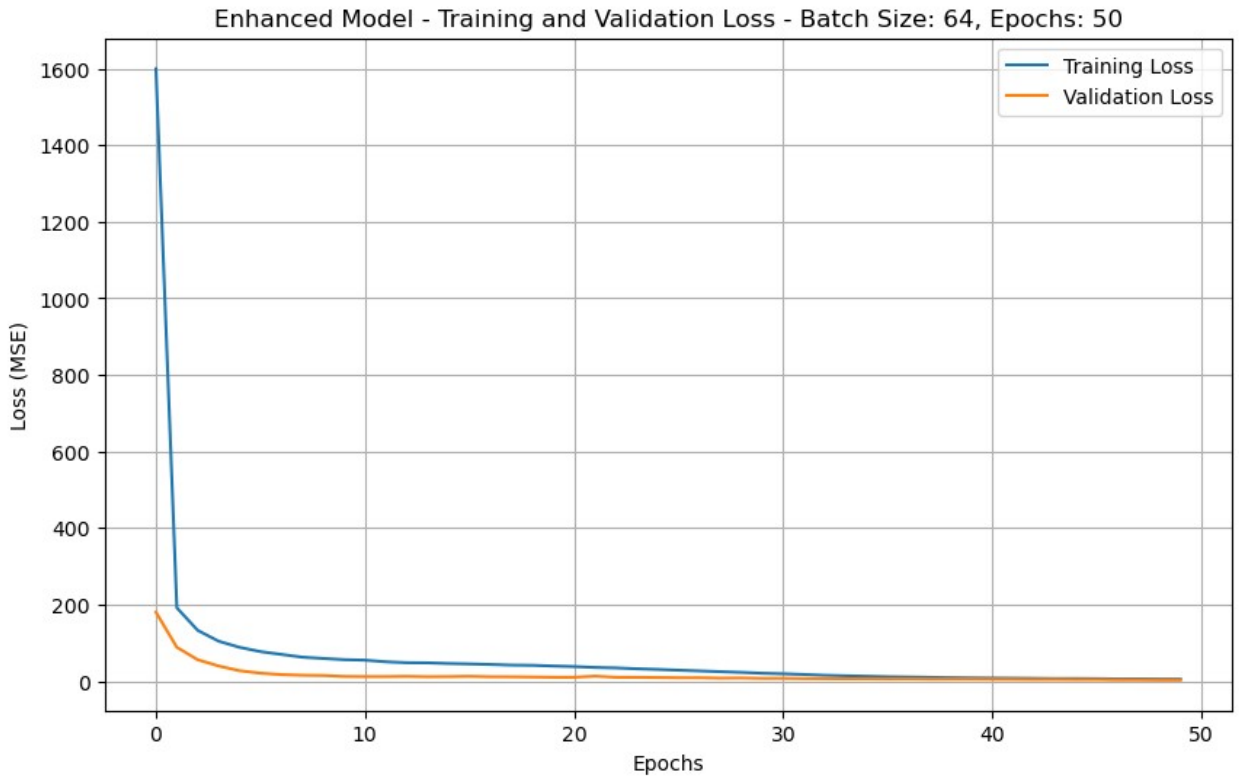

```
Training with Batch Size: 64, Epochs: 50
Epoch 1/50
180/180 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - loss: 2899.8218 -
root_mean_squared_error: 53.1414 - val_loss: 180.3887 -
val_root_mean_squared_error: 13.4236
Epoch 2/50
180/180 ━━━━━━━━━━━━━━━━━━━━ 0s 808us/step - loss: 212.7194 -
root_mean_squared_error: 14.5719 - val_loss: 88.5607 -
val_root_mean_squared_error: 9.4001
Epoch 3/50
180/180 ━━━━━━━━━━━━━━━━━━━━ 0s 584us/step - loss: 137.6533 -
root_mean_squared_error: 11.7225 - val_loss: 56.0915 -
val_root_mean_squared_error: 7.4762
Epoch 4/50
180/180 ━━━━━━━━━━━━━━━━━━━━ 0s 518us/step - loss: 110.2485 -
root_mean_squared_error: 10.4892 - val_loss: 39.7712 -
val_root_mean_squared_error: 6.2911
Epoch 5/50
180/180 ━━━━━━━━━━━━━━━━━━━━ 0s 526us/step - loss: 92.8657 -
root_mean_squared_error: 9.6238 - val_loss: 27.4633 -
val_root_mean_squared_error: 5.2226
Epoch 6/50
180/180 ━━━━━━━━━━━━━━━━━━━━ 0s 510us/step - loss: 81.8100 -
root_mean_squared_error: 9.0327 - val_loss: 21.0100 -
```

```
val_root_mean_squared_error: 4.5637
Epoch 7/50
180/180 ━━━━━━━━━━━━━━━━━━━━ 0s 762us/step - loss: 71.2722 -
root_mean_squared_error: 8.4307 - val_loss: 17.2858 -
val_root_mean_squared_error: 4.1362
Epoch 8/50
180/180 ━━━━━━━━━━━━━━━━━━━━ 0s 613us/step - loss: 62.0814 -
root_mean_squared_error: 7.8676 - val_loss: 15.6616 -
val_root_mean_squared_error: 3.9356
Epoch 9/50
180/180 ━━━━━━━━━━━━━━━━━━━━ 0s 818us/step - loss: 60.3706 -
root_mean_squared_error: 7.7570 - val_loss: 14.8834 -
val_root_mean_squared_error: 3.8361
Epoch 10/50
180/180 ━━━━━━━━━━━━━━━━━━━━ 0s 688us/step - loss: 57.7186 -
root_mean_squared_error: 7.5859 - val_loss: 12.5341 -
val_root_mean_squared_error: 3.5171
Epoch 11/50
180/180 ━━━━━━━━━━━━━━━━━━━━ 0s 786us/step - loss: 55.3002 -
root_mean_squared_error: 7.4253 - val_loss: 12.0288 -
val_root_mean_squared_error: 3.4451
Epoch 12/50
180/180 ━━━━━━━━━━━━━━━━━━━━ 0s 532us/step - loss: 51.5850 -
root_mean_squared_error: 7.1709 - val_loss: 12.1216 -
val_root_mean_squared_error: 3.4592
Epoch 13/50
180/180 ━━━━━━━━━━━━━━━━━━━━ 0s 760us/step - loss: 48.4999 -
root_mean_squared_error: 6.9524 - val_loss: 12.7589 -
val_root_mean_squared_error: 3.5506
Epoch 14/50
180/180 ━━━━━━━━━━━━━━━━━━━━ 0s 591us/step - loss: 47.7433 -
root_mean_squared_error: 6.8984 - val_loss: 11.6649 -
val_root_mean_squared_error: 3.3936
Epoch 15/50
180/180 ━━━━━━━━━━━━━━━━━━━━ 0s 652us/step - loss: 46.5743 -
root_mean_squared_error: 6.8128 - val_loss: 12.0612 -
val_root_mean_squared_error: 3.4520
Epoch 16/50
180/180 ━━━━━━━━━━━━━━━━━━━━ 0s 585us/step - loss: 44.7072 -
root_mean_squared_error: 6.6751 - val_loss: 12.8711 -
val_root_mean_squared_error: 3.5679
Epoch 17/50
180/180 ━━━━━━━━━━━━━━━━━━━━ 0s 812us/step - loss: 44.7411 -
root_mean_squared_error: 6.6774 - val_loss: 11.6358 -
val_root_mean_squared_error: 3.3908
Epoch 18/50
180/180 ━━━━━━━━━━━━━━━━━━━━ 0s 658us/step - loss: 41.8987 -
root_mean_squared_error: 6.4618 - val_loss: 11.5071 -
val_root_mean_squared_error: 3.3723
```

```
Epoch 19/50
180/180 ━━━━━━━━━━━━━━━━━━━━ 0s 537us/step - loss: 42.0857 -
root_mean_squared_error: 6.4767 - val_loss: 10.9586 -
val_root_mean_squared_error: 3.2905
Epoch 20/50
180/180 ━━━━━━━━━━━━━━━━━━━━ 0s 536us/step - loss: 40.9056 -
root_mean_squared_error: 6.3849 - val_loss: 10.2703 -
val_root_mean_squared_error: 3.1847
Epoch 21/50
180/180 ━━━━━━━━━━━━━━━━━━━━ 0s 553us/step - loss: 38.4711 -
root_mean_squared_error: 6.1915 - val_loss: 10.3189 -
val_root_mean_squared_error: 3.1928
Epoch 22/50
180/180 ━━━━━━━━━━━━━━━━━━━━ 0s 505us/step - loss: 36.5225 -
root_mean_squared_error: 6.0327 - val_loss: 13.2391 -
val_root_mean_squared_error: 3.6219
Epoch 23/50
180/180 ━━━━━━━━━━━━━━━━━━━━ 0s 643us/step - loss: 35.4847 -
root_mean_squared_error: 5.9458 - val_loss: 10.2569 -
val_root_mean_squared_error: 3.1841
Epoch 24/50
180/180 ━━━━━━━━━━━━━━━━━━━━ 0s 700us/step - loss: 32.5396 -
root_mean_squared_error: 5.6934 - val_loss: 10.1200 -
val_root_mean_squared_error: 3.1628
Epoch 25/50
180/180 ━━━━━━━━━━━━━━━━━━━━ 0s 591us/step - loss: 30.9719 -
root_mean_squared_error: 5.5545 - val_loss: 9.6735 -
val_root_mean_squared_error: 3.0920
Epoch 26/50
180/180 ━━━━━━━━━━━━━━━━━━━━ 0s 572us/step - loss: 28.5752 -
root_mean_squared_error: 5.3344 - val_loss: 9.0681 -
val_root_mean_squared_error: 2.9928
Epoch 27/50
180/180 ━━━━━━━━━━━━━━━━━━━━ 0s 599us/step - loss: 27.8737 -
root_mean_squared_error: 5.2675 - val_loss: 9.0394 -
val_root_mean_squared_error: 2.9883
Epoch 28/50
180/180 ━━━━━━━━━━━━━━━━━━━━ 0s 573us/step - loss: 25.3160 -
root_mean_squared_error: 5.0204 - val_loss: 8.1035 -
val_root_mean_squared_error: 2.8276
Epoch 29/50
180/180 ━━━━━━━━━━━━━━━━━━━━ 0s 661us/step - loss: 23.3418 -
root_mean_squared_error: 4.8200 - val_loss: 8.4532 -
val_root_mean_squared_error: 2.8890
Epoch 30/50
180/180 ━━━━━━━━━━━━━━━━━━━━ 0s 525us/step - loss: 21.7046 -
root_mean_squared_error: 4.6464 - val_loss: 7.4675 -
val_root_mean_squared_error: 2.7134
Epoch 31/50
```

```
180/180 ━━━━━━━━━━━━━━━━━━━━ 0s 520us/step - loss: 19.8679 -
root_mean_squared_error: 4.4449 - val_loss: 7.5358 -
val_root_mean_squared_error: 2.7262
Epoch 32/50
180/180 ━━━━━━━━━━━━━━━━━━━━ 0s 669us/step - loss: 18.0704 -
root_mean_squared_error: 4.2368 - val_loss: 6.9268 -
val_root_mean_squared_error: 2.6123
Epoch 33/50
180/180 ━━━━━━━━━━━━━━━━━━━━ 0s 691us/step - loss: 15.6243 -
root_mean_squared_error: 3.9393 - val_loss: 6.5929 -
val_root_mean_squared_error: 2.5477
Epoch 34/50
180/180 ━━━━━━━━━━━━━━━━━━━━ 0s 847us/step - loss: 14.5510 -
root_mean_squared_error: 3.8004 - val_loss: 6.2151 -
val_root_mean_squared_error: 2.4725
Epoch 35/50
180/180 ━━━━━━━━━━━━━━━━━━━━ 0s 536us/step - loss: 13.2919 -
root_mean_squared_error: 3.6314 - val_loss: 6.3004 -
val_root_mean_squared_error: 2.4897
Epoch 36/50
180/180 ━━━━━━━━━━━━━━━━━━━━ 0s 648us/step - loss: 11.4714 -
root_mean_squared_error: 3.3718 - val_loss: 5.8548 -
val_root_mean_squared_error: 2.3987
Epoch 37/50
180/180 ━━━━━━━━━━━━━━━━━━━━ 0s 553us/step - loss: 11.3582 -
root_mean_squared_error: 3.3534 - val_loss: 6.1534 -
val_root_mean_squared_error: 2.4603
Epoch 38/50
180/180 ━━━━━━━━━━━━━━━━━━━━ 0s 659us/step - loss: 10.0547 -
root_mean_squared_error: 3.1548 - val_loss: 5.5744 -
val_root_mean_squared_error: 2.3397
Epoch 39/50
180/180 ━━━━━━━━━━━━━━━━━━━━ 0s 545us/step - loss: 9.5624 -
root_mean_squared_error: 3.0756 - val_loss: 5.6416 -
val_root_mean_squared_error: 2.3541
Epoch 40/50
180/180 ━━━━━━━━━━━━━━━━━━━━ 0s 569us/step - loss: 8.4996 -
root_mean_squared_error: 2.8978 - val_loss: 5.6239 -
val_root_mean_squared_error: 2.3502
Epoch 41/50
180/180 ━━━━━━━━━━━━━━━━━━━━ 0s 594us/step - loss: 8.4031 -
root_mean_squared_error: 2.8805 - val_loss: 5.2959 -
val_root_mean_squared_error: 2.2790
Epoch 42/50
180/180 ━━━━━━━━━━━━━━━━━━━━ 0s 787us/step - loss: 7.4592 -
root_mean_squared_error: 2.7100 - val_loss: 5.1130 -
val_root_mean_squared_error: 2.2378
Epoch 43/50
180/180 ━━━━━━━━━━━━━━━━━━━━ 0s 559us/step - loss: 7.1182 -
```

```
root_mean_squared_error: 2.6477 - val_loss: 4.6821 -
val_root_mean_squared_error: 2.1387
Epoch 44/50
180/180 ━━━━━━━━━━━━━━━━━━━━ 0s 554us/step - loss: 6.7747 -
root_mean_squared_error: 2.5816 - val_loss: 4.8661 -
val_root_mean_squared_error: 2.1806
Epoch 45/50
180/180 ━━━━━━━━━━━━━━━━━━━━ 0s 551us/step - loss: 6.9862 -
root_mean_squared_error: 2.6209 - val_loss: 4.3712 -
val_root_mean_squared_error: 2.0631
Epoch 46/50
180/180 ━━━━━━━━━━━━━━━━━━━━ 0s 572us/step - loss: 6.7582 -
root_mean_squared_error: 2.5764 - val_loss: 4.4514 -
val_root_mean_squared_error: 2.0813
Epoch 47/50
180/180 ━━━━━━━━━━━━━━━━━━━━ 0s 559us/step - loss: 5.4697 -
root_mean_squared_error: 2.3118 - val_loss: 3.7943 -
val_root_mean_squared_error: 1.9161
Epoch 48/50
180/180 ━━━━━━━━━━━━━━━━━━━━ 0s 634us/step - loss: 6.4507 -
root_mean_squared_error: 2.5142 - val_loss: 3.8425 -
val_root_mean_squared_error: 1.9277
Epoch 49/50
180/180 ━━━━━━━━━━━━━━━━━━━━ 0s 572us/step - loss: 5.6504 -
root_mean_squared_error: 2.3490 - val_loss: 3.4706 -
val_root_mean_squared_error: 1.8275
Epoch 50/50
180/180 ━━━━━━━━━━━━━━━━━━━━ 0s 532us/step - loss: 5.1624 -
root_mean_squared_error: 2.2425 - val_loss: 3.4168 -
val_root_mean_squared_error: 1.8113
Completed training with Batch Size: 64, Epochs: 50. Test MSE: 3.5876,
Test RMSE: 1.8579
```




```
Training with Batch Size: 64, Epochs: 100
Epoch 1/100
180/180 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - loss: 3119.5764 -
root_mean_squared_error: 55.2482 - val_loss: 202.0882 -
val_root_mean_squared_error: 14.2085
Epoch 2/100
180/180 ━━━━━━━━━━━━━━━━━━━━ 0s 720us/step - loss: 231.9633 -
root_mean_squared_error: 15.2099 - val_loss: 94.8260 -
val_root_mean_squared_error: 9.7270
Epoch 3/100
180/180 ━━━━━━━━━━━━━━━━━━━━ 0s 660us/step - loss: 151.0904 -
root_mean_squared_error: 12.2729 - val_loss: 59.8597 -
val_root_mean_squared_error: 7.7233
Epoch 4/100
180/180 ━━━━━━━━━━━━━━━━━━━━ 0s 574us/step - loss: 115.1338 -
root_mean_squared_error: 10.7183 - val_loss: 42.2568 -
val_root_mean_squared_error: 6.4847
Epoch 5/100
180/180 ━━━━━━━━━━━━━━━━━━━━ 0s 668us/step - loss: 94.9439 -
root_mean_squared_error: 9.7316 - val_loss: 29.1769 -
val_root_mean_squared_error: 5.3829
Epoch 6/100
180/180 ━━━━━━━━━━━━━━━━━━━━ 0s 586us/step - loss: 81.8178 -
root_mean_squared_error: 9.0329 - val_loss: 24.4357 -
```

```
val_root_mean_squared_error: 4.9236
Epoch 7/100
180/180 ━━━━━━━━━━━━━━━━━━━━ 0s 574us/step - loss: 73.0661 -
root_mean_squared_error: 8.5341 - val_loss: 18.0536 -
val_root_mean_squared_error: 4.2266
Epoch 8/100
180/180 ━━━━━━━━━━━━━━━━━━━━ 0s 700us/step - loss: 65.4842 -
root_mean_squared_error: 8.0800 - val_loss: 15.6422 -
val_root_mean_squared_error: 3.9317
Epoch 9/100
180/180 ━━━━━━━━━━━━━━━━━━━━ 0s 668us/step - loss: 61.4868 -
root_mean_squared_error: 7.8287 - val_loss: 14.9793 -
val_root_mean_squared_error: 3.8471
Epoch 10/100
180/180 ━━━━━━━━━━━━━━━━━━━━ 0s 680us/step - loss: 55.8121 -
root_mean_squared_error: 7.4586 - val_loss: 13.4550 -
val_root_mean_squared_error: 3.6442
Epoch 11/100
180/180 ━━━━━━━━━━━━━━━━━━━━ 0s 616us/step - loss: 52.4358 -
root_mean_squared_error: 7.2280 - val_loss: 12.2241 -
val_root_mean_squared_error: 3.4716
Epoch 12/100
180/180 ━━━━━━━━━━━━━━━━━━━━ 0s 533us/step - loss: 51.2825 -
root_mean_squared_error: 7.1492 - val_loss: 11.9413 -
val_root_mean_squared_error: 3.4313
Epoch 13/100
180/180 ━━━━━━━━━━━━━━━━━━━━ 0s 574us/step - loss: 48.1537 -
root_mean_squared_error: 6.9262 - val_loss: 12.5642 -
val_root_mean_squared_error: 3.5215
Epoch 14/100
180/180 ━━━━━━━━━━━━━━━━━━━━ 0s 700us/step - loss: 46.1063 -
root_mean_squared_error: 6.7775 - val_loss: 12.3864 -
val_root_mean_squared_error: 3.4966
Epoch 15/100
180/180 ━━━━━━━━━━━━━━━━━━━━ 0s 650us/step - loss: 45.0880 -
root_mean_squared_error: 6.7008 - val_loss: 13.5763 -
val_root_mean_squared_error: 3.6633
Epoch 16/100
180/180 ━━━━━━━━━━━━━━━━━━━━ 0s 626us/step - loss: 41.6833 -
root_mean_squared_error: 6.4438 - val_loss: 12.0516 -
val_root_mean_squared_error: 3.4494
Epoch 17/100
180/180 ━━━━━━━━━━━━━━━━━━━━ 0s 574us/step - loss: 41.0445 -
root_mean_squared_error: 6.3929 - val_loss: 11.3352 -
val_root_mean_squared_error: 3.3445
Epoch 18/100
180/180 ━━━━━━━━━━━━━━━━━━━━ 0s 628us/step - loss: 38.0822 -
root_mean_squared_error: 6.1586 - val_loss: 12.3858 -
val_root_mean_squared_error: 3.4985
```

```
Epoch 19/100
180/180 ━━━━━━━━━━━━━━━━━━━━ 0s 664us/step - loss: 36.9305 -
root_mean_squared_error: 6.0647 - val_loss: 12.2023 -
val_root_mean_squared_error: 3.4726
Epoch 20/100
180/180 ━━━━━━━━━━━━━━━━━━━━ 0s 682us/step - loss: 34.3617 -
root_mean_squared_error: 5.8491 - val_loss: 10.8496 -
val_root_mean_squared_error: 3.2724
Epoch 21/100
180/180 ━━━━━━━━━━━━━━━━━━━━ 0s 610us/step - loss: 31.1154 -
root_mean_squared_error: 5.5654 - val_loss: 11.5494 -
val_root_mean_squared_error: 3.3781
Epoch 22/100
180/180 ━━━━━━━━━━━━━━━━━━━━ 0s 860us/step - loss: 30.0007 -
root_mean_squared_error: 5.4642 - val_loss: 9.4345 -
val_root_mean_squared_error: 3.0494
Epoch 23/100
180/180 ━━━━━━━━━━━━━━━━━━━━ 0s 683us/step - loss: 27.8753 -
root_mean_squared_error: 5.2665 - val_loss: 10.3937 -
val_root_mean_squared_error: 3.2032
Epoch 24/100
180/180 ━━━━━━━━━━━━━━━━━━━━ 0s 542us/step - loss: 26.5839 -
root_mean_squared_error: 5.1419 - val_loss: 8.8432 -
val_root_mean_squared_error: 2.9516
Epoch 25/100
180/180 ━━━━━━━━━━━━━━━━━━━━ 0s 576us/step - loss: 23.8208 -
root_mean_squared_error: 4.8663 - val_loss: 8.3571 -
val_root_mean_squared_error: 2.8685
Epoch 26/100
180/180 ━━━━━━━━━━━━━━━━━━━━ 0s 649us/step - loss: 21.2457 -
root_mean_squared_error: 4.5952 - val_loss: 8.7892 -
val_root_mean_squared_error: 2.9433
Epoch 27/100
180/180 ━━━━━━━━━━━━━━━━━━━━ 0s 562us/step - loss: 20.0172 -
root_mean_squared_error: 4.4593 - val_loss: 8.5867 -
val_root_mean_squared_error: 2.9090
Epoch 28/100
180/180 ━━━━━━━━━━━━━━━━━━━━ 0s 496us/step - loss: 18.0292 -
root_mean_squared_error: 4.2307 - val_loss: 7.6869 -
val_root_mean_squared_error: 2.7504
Epoch 29/100
180/180 ━━━━━━━━━━━━━━━━━━━━ 0s 684us/step - loss: 15.6685 -
root_mean_squared_error: 3.9427 - val_loss: 7.2977 -
val_root_mean_squared_error: 2.6790
Epoch 30/100
180/180 ━━━━━━━━━━━━━━━━━━━━ 0s 543us/step - loss: 15.4121 -
root_mean_squared_error: 3.9092 - val_loss: 6.9602 -
val_root_mean_squared_error: 2.6155
Epoch 31/100
```

```
180/180 ━━━━━━━━━━━━━━━━━━━━ 0s 630us/step - loss: 13.4062 -
root_mean_squared_error: 3.6449 - val_loss: 6.5200 -
val_root_mean_squared_error: 2.5301
Epoch 32/100
180/180 ━━━━━━━━━━━━━━━━━━━━ 0s 620us/step - loss: 12.1011 -
root_mean_squared_error: 3.4613 - val_loss: 6.8833 -
val_root_mean_squared_error: 2.6011
Epoch 33/100
180/180 ━━━━━━━━━━━━━━━━━━━━ 0s 515us/step - loss: 11.0321 -
root_mean_squared_error: 3.3032 - val_loss: 6.1237 -
val_root_mean_squared_error: 2.4509
Epoch 34/100
180/180 ━━━━━━━━━━━━━━━━━━━━ 0s 606us/step - loss: 10.6527 -
root_mean_squared_error: 3.2442 - val_loss: 5.9440 -
val_root_mean_squared_error: 2.4142
Epoch 35/100
180/180 ━━━━━━━━━━━━━━━━━━━━ 0s 728us/step - loss: 9.8490 -
root_mean_squared_error: 3.1193 - val_loss: 5.7571 -
val_root_mean_squared_error: 2.3752
Epoch 36/100
180/180 ━━━━━━━━━━━━━━━━━━━━ 0s 538us/step - loss: 9.0761 -
root_mean_squared_error: 2.9932 - val_loss: 5.9611 -
val_root_mean_squared_error: 2.4180
Epoch 37/100
180/180 ━━━━━━━━━━━━━━━━━━━━ 0s 719us/step - loss: 8.7155 -
root_mean_squared_error: 2.9322 - val_loss: 5.5451 -
val_root_mean_squared_error: 2.3304
Epoch 38/100
180/180 ━━━━━━━━━━━━━━━━━━━━ 0s 536us/step - loss: 7.9525 -
root_mean_squared_error: 2.7995 - val_loss: 5.4025 -
val_root_mean_squared_error: 2.2997
Epoch 39/100
180/180 ━━━━━━━━━━━━━━━━━━━━ 0s 573us/step - loss: 7.6894 -
root_mean_squared_error: 2.7521 - val_loss: 5.2320 -
val_root_mean_squared_error: 2.2623
Epoch 40/100
180/180 ━━━━━━━━━━━━━━━━━━━━ 0s 642us/step - loss: 7.3157 -
root_mean_squared_error: 2.6831 - val_loss: 5.0585 -
val_root_mean_squared_error: 2.2233
Epoch 41/100
180/180 ━━━━━━━━━━━━━━━━━━━━ 0s 630us/step - loss: 7.5391 -
root_mean_squared_error: 2.7237 - val_loss: 4.8432 -
val_root_mean_squared_error: 2.1736
Epoch 42/100
180/180 ━━━━━━━━━━━━━━━━━━━━ 0s 597us/step - loss: 6.6115 -
root_mean_squared_error: 2.5478 - val_loss: 4.6085 -
val_root_mean_squared_error: 2.1180
Epoch 43/100
180/180 ━━━━━━━━━━━━━━━━━━━━ 0s 618us/step - loss: 6.5004 -
```

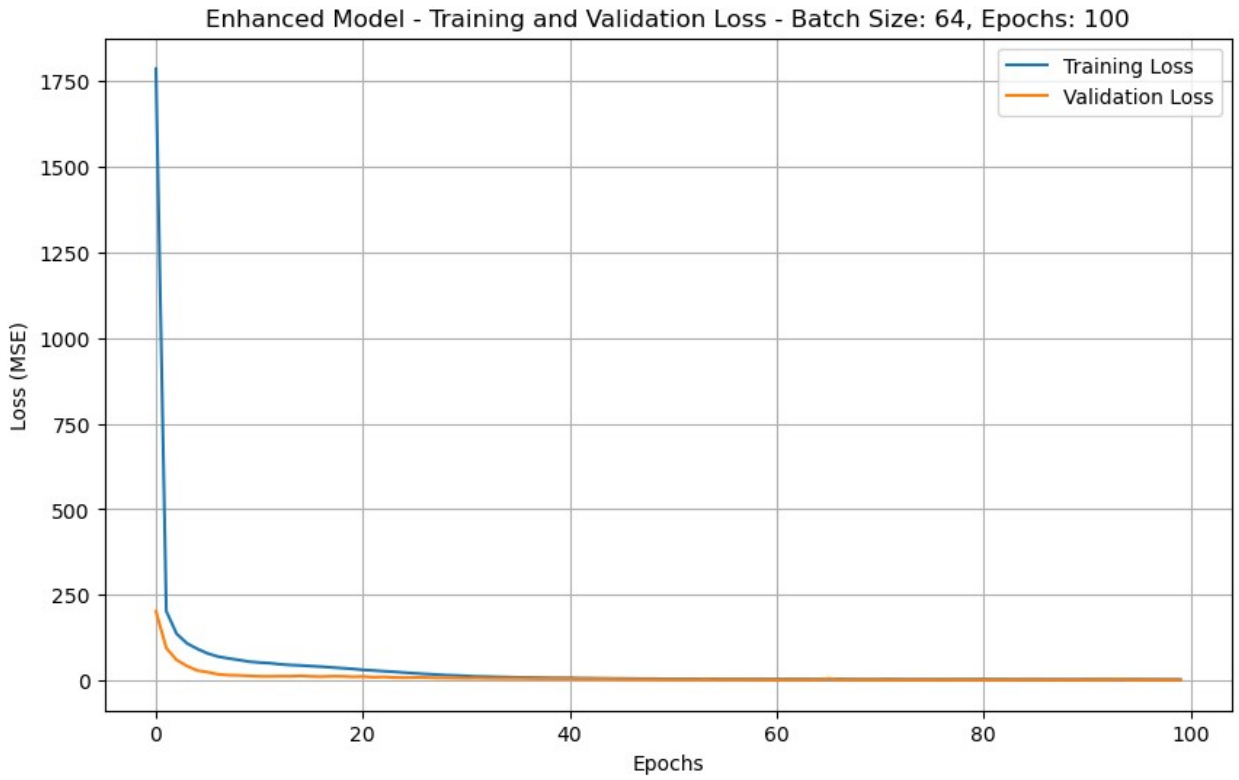
```
root_mean_squared_error: 2.5241 - val_loss: 4.1202 -
val_root_mean_squared_error: 1.9982
Epoch 44/100
180/180 ━━━━━━━━━━━━━━━━━━━━ 0s 624us/step - loss: 5.9701 -
root_mean_squared_error: 2.4167 - val_loss: 4.4548 -
val_root_mean_squared_error: 2.0785
Epoch 45/100
180/180 ━━━━━━━━━━━━━━━━━━━━ 0s 531us/step - loss: 5.5595 -
root_mean_squared_error: 2.3286 - val_loss: 3.9096 -
val_root_mean_squared_error: 1.9413
Epoch 46/100
180/180 ━━━━━━━━━━━━━━━━━━━━ 0s 560us/step - loss: 5.3720 -
root_mean_squared_error: 2.2859 - val_loss: 3.4237 -
val_root_mean_squared_error: 1.8100
Epoch 47/100
180/180 ━━━━━━━━━━━━━━━━━━━━ 0s 651us/step - loss: 5.3744 -
root_mean_squared_error: 2.2855 - val_loss: 3.2029 -
val_root_mean_squared_error: 1.7458
Epoch 48/100
180/180 ━━━━━━━━━━━━━━━━━━━━ 0s 897us/step - loss: 4.7940 -
root_mean_squared_error: 2.1528 - val_loss: 2.9737 -
val_root_mean_squared_error: 1.6770
Epoch 49/100
180/180 ━━━━━━━━━━━━━━━━━━━━ 0s 587us/step - loss: 5.1015 -
root_mean_squared_error: 2.2199 - val_loss: 3.0394 -
val_root_mean_squared_error: 1.6946
Epoch 50/100
180/180 ━━━━━━━━━━━━━━━━━━━━ 0s 849us/step - loss: 4.2022 -
root_mean_squared_error: 2.0076 - val_loss: 2.8353 -
val_root_mean_squared_error: 1.6315
Epoch 51/100
180/180 ━━━━━━━━━━━━━━━━━━━━ 0s 760us/step - loss: 4.1891 -
root_mean_squared_error: 2.0024 - val_loss: 2.8619 -
val_root_mean_squared_error: 1.6379
Epoch 52/100
180/180 ━━━━━━━━━━━━━━━━━━━━ 0s 532us/step - loss: 4.5118 -
root_mean_squared_error: 2.0798 - val_loss: 2.8057 -
val_root_mean_squared_error: 1.6191
Epoch 53/100
180/180 ━━━━━━━━━━━━━━━━━━━━ 0s 586us/step - loss: 4.5994 -
root_mean_squared_error: 2.1002 - val_loss: 2.5024 -
val_root_mean_squared_error: 1.5212
Epoch 54/100
180/180 ━━━━━━━━━━━━━━━━━━━━ 0s 615us/step - loss: 4.1265 -
root_mean_squared_error: 1.9836 - val_loss: 3.5993 -
val_root_mean_squared_error: 1.8454
Epoch 55/100
180/180 ━━━━━━━━━━━━━━━━━━━━ 0s 696us/step - loss: 4.1867 -
root_mean_squared_error: 1.9976 - val_loss: 2.2781 -
```

```
val_root_mean_squared_error: 1.4424
Epoch 56/100
180/180 ━━━━━━━━━━━━━━━━━━━━ 0s 593us/step - loss: 4.0675 -
root_mean_squared_error: 1.9651 - val_loss: 2.6015 -
val_root_mean_squared_error: 1.5492
Epoch 57/100
180/180 ━━━━━━━━━━━━━━━━━━━━ 0s 530us/step - loss: 4.1022 -
root_mean_squared_error: 1.9739 - val_loss: 3.0345 -
val_root_mean_squared_error: 1.6824
Epoch 58/100
180/180 ━━━━━━━━━━━━━━━━━━━━ 0s 561us/step - loss: 3.5865 -
root_mean_squared_error: 1.8378 - val_loss: 2.5300 -
val_root_mean_squared_error: 1.5242
Epoch 59/100
180/180 ━━━━━━━━━━━━━━━━━━━━ 0s 498us/step - loss: 3.7234 -
root_mean_squared_error: 1.8749 - val_loss: 2.6134 -
val_root_mean_squared_error: 1.5505
Epoch 60/100
180/180 ━━━━━━━━━━━━━━━━━━━━ 0s 643us/step - loss: 3.8531 -
root_mean_squared_error: 1.9084 - val_loss: 2.3904 -
val_root_mean_squared_error: 1.4760
Epoch 61/100
180/180 ━━━━━━━━━━━━━━━━━━━━ 0s 668us/step - loss: 3.5652 -
root_mean_squared_error: 1.8305 - val_loss: 2.2434 -
val_root_mean_squared_error: 1.4243
Epoch 62/100
180/180 ━━━━━━━━━━━━━━━━━━━━ 0s 586us/step - loss: 3.9255 -
root_mean_squared_error: 1.9246 - val_loss: 2.1759 -
val_root_mean_squared_error: 1.3994
Epoch 63/100
180/180 ━━━━━━━━━━━━━━━━━━━━ 0s 561us/step - loss: 3.5635 -
root_mean_squared_error: 1.8280 - val_loss: 2.9701 -
val_root_mean_squared_error: 1.6588
Epoch 64/100
180/180 ━━━━━━━━━━━━━━━━━━━━ 0s 498us/step - loss: 3.6618 -
root_mean_squared_error: 1.8549 - val_loss: 2.0422 -
val_root_mean_squared_error: 1.3498
Epoch 65/100
180/180 ━━━━━━━━━━━━━━━━━━━━ 0s 521us/step - loss: 3.1643 -
root_mean_squared_error: 1.7136 - val_loss: 2.0676 -
val_root_mean_squared_error: 1.3584
Epoch 66/100
180/180 ━━━━━━━━━━━━━━━━━━━━ 0s 557us/step - loss: 3.4435 -
root_mean_squared_error: 1.7939 - val_loss: 5.6080 -
val_root_mean_squared_error: 2.3205
Epoch 67/100
180/180 ━━━━━━━━━━━━━━━━━━━━ 0s 532us/step - loss: 4.2262 -
root_mean_squared_error: 1.9944 - val_loss: 2.1767 -
val_root_mean_squared_error: 1.3974
```

```
Epoch 68/100
180/180 ━━━━━━━━━━━━━━━━━━━━ 0s 574us/step - loss: 3.5046 -
root_mean_squared_error: 1.8108 - val_loss: 2.2268 -
val_root_mean_squared_error: 1.4148
Epoch 69/100
180/180 ━━━━━━━━━━━━━━━━━━━━ 0s 566us/step - loss: 3.4802 -
root_mean_squared_error: 1.8012 - val_loss: 2.0612 -
val_root_mean_squared_error: 1.3547
Epoch 70/100
180/180 ━━━━━━━━━━━━━━━━━━━━ 0s 551us/step - loss: 3.0603 -
root_mean_squared_error: 1.6829 - val_loss: 2.3818 -
val_root_mean_squared_error: 1.4680
Epoch 71/100
180/180 ━━━━━━━━━━━━━━━━━━━━ 0s 691us/step - loss: 3.5152 -
root_mean_squared_error: 1.8117 - val_loss: 2.3595 -
val_root_mean_squared_error: 1.4601
Epoch 72/100
180/180 ━━━━━━━━━━━━━━━━━━━━ 0s 537us/step - loss: 3.9211 -
root_mean_squared_error: 1.9157 - val_loss: 1.8515 -
val_root_mean_squared_error: 1.2741
Epoch 73/100
180/180 ━━━━━━━━━━━━━━━━━━━━ 0s 540us/step - loss: 3.0589 -
root_mean_squared_error: 1.6815 - val_loss: 2.3983 -
val_root_mean_squared_error: 1.4726
Epoch 74/100
180/180 ━━━━━━━━━━━━━━━━━━━━ 0s 657us/step - loss: 3.0752 -
root_mean_squared_error: 1.6853 - val_loss: 1.9271 -
val_root_mean_squared_error: 1.3026
Epoch 75/100
180/180 ━━━━━━━━━━━━━━━━━━━━ 0s 674us/step - loss: 3.3914 -
root_mean_squared_error: 1.7769 - val_loss: 1.8791 -
val_root_mean_squared_error: 1.2838
Epoch 76/100
180/180 ━━━━━━━━━━━━━━━━━━━━ 0s 626us/step - loss: 3.3720 -
root_mean_squared_error: 1.7689 - val_loss: 1.8960 -
val_root_mean_squared_error: 1.2901
Epoch 77/100
180/180 ━━━━━━━━━━━━━━━━━━━━ 0s 663us/step - loss: 3.0542 -
root_mean_squared_error: 1.6788 - val_loss: 2.1709 -
val_root_mean_squared_error: 1.3929
Epoch 78/100
180/180 ━━━━━━━━━━━━━━━━━━━━ 0s 513us/step - loss: 3.3453 -
root_mean_squared_error: 1.7643 - val_loss: 2.4764 -
val_root_mean_squared_error: 1.4985
Epoch 79/100
180/180 ━━━━━━━━━━━━━━━━━━━━ 0s 643us/step - loss: 3.6138 -
root_mean_squared_error: 1.8369 - val_loss: 1.8244 -
val_root_mean_squared_error: 1.2629
Epoch 80/100
```

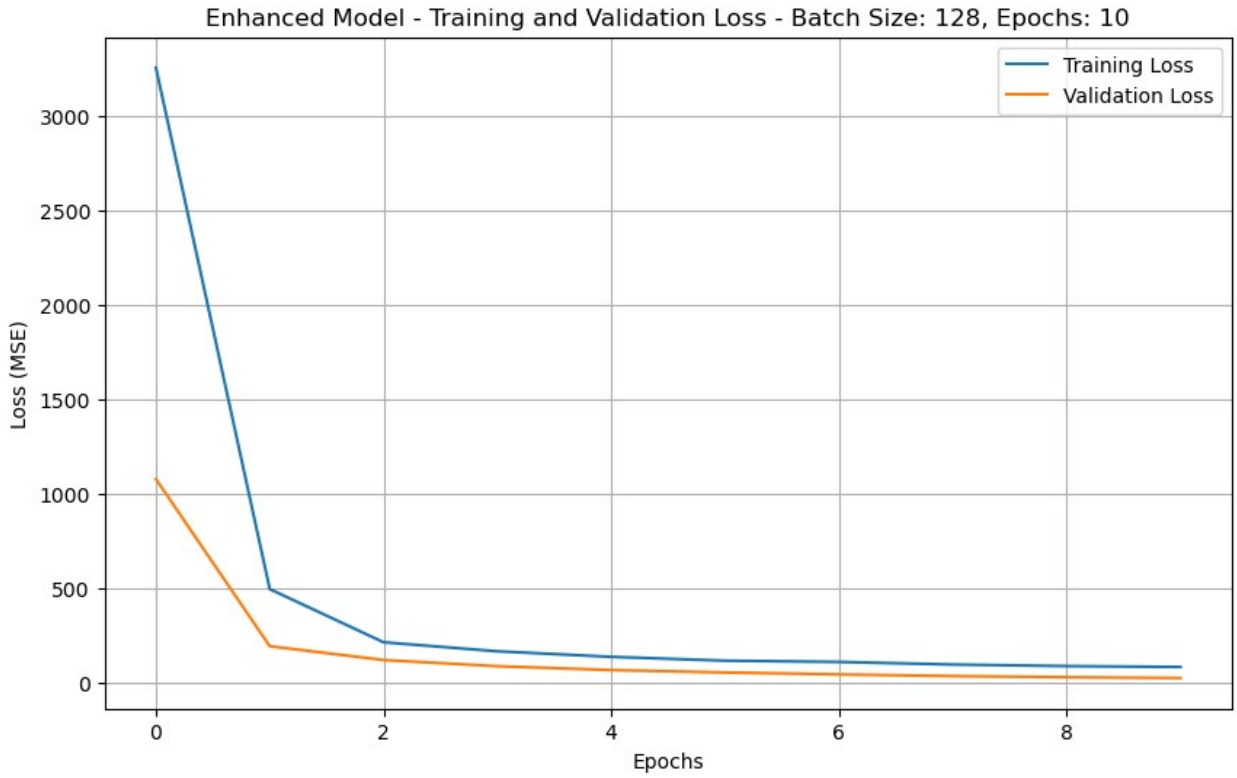
```
180/180 ━━━━━━━━━━━━━━━━━━━━ 0s 654us/step - loss: 3.1667 -
root_mean_squared_error: 1.7133 - val_loss: 2.2645 -
val_root_mean_squared_error: 1.4268
Epoch 81/100
180/180 ━━━━━━━━━━━━━━━━━━━━ 0s 608us/step - loss: 3.7630 -
root_mean_squared_error: 1.8779 - val_loss: 1.9762 -
val_root_mean_squared_error: 1.3221
Epoch 82/100
180/180 ━━━━━━━━━━━━━━━━━━━━ 0s 640us/step - loss: 2.9658 -
root_mean_squared_error: 1.6532 - val_loss: 2.3983 -
val_root_mean_squared_error: 1.4729
Epoch 83/100
180/180 ━━━━━━━━━━━━━━━━━━━━ 0s 586us/step - loss: 3.5393 -
root_mean_squared_error: 1.8171 - val_loss: 1.9126 -
val_root_mean_squared_error: 1.2979
Epoch 84/100
180/180 ━━━━━━━━━━━━━━━━━━━━ 0s 636us/step - loss: 3.0847 -
root_mean_squared_error: 1.6898 - val_loss: 2.2691 -
val_root_mean_squared_error: 1.4286
Epoch 85/100
180/180 ━━━━━━━━━━━━━━━━━━━━ 0s 728us/step - loss: 3.0831 -
root_mean_squared_error: 1.6890 - val_loss: 2.1773 -
val_root_mean_squared_error: 1.3963
Epoch 86/100
180/180 ━━━━━━━━━━━━━━━━━━━━ 0s 557us/step - loss: 2.7779 -
root_mean_squared_error: 1.5956 - val_loss: 2.5748 -
val_root_mean_squared_error: 1.5319
Epoch 87/100
180/180 ━━━━━━━━━━━━━━━━━━━━ 0s 568us/step - loss: 3.3458 -
root_mean_squared_error: 1.7649 - val_loss: 2.1009 -
val_root_mean_squared_error: 1.3687
Epoch 88/100
180/180 ━━━━━━━━━━━━━━━━━━━━ 0s 594us/step - loss: 2.9720 -
root_mean_squared_error: 1.6563 - val_loss: 1.8541 -
val_root_mean_squared_error: 1.2757
Epoch 89/100
180/180 ━━━━━━━━━━━━━━━━━━━━ 0s 696us/step - loss: 2.9666 -
root_mean_squared_error: 1.6543 - val_loss: 1.7525 -
val_root_mean_squared_error: 1.2351
Epoch 90/100
180/180 ━━━━━━━━━━━━━━━━━━━━ 0s 527us/step - loss: 3.0791 -
root_mean_squared_error: 1.6879 - val_loss: 1.8924 -
val_root_mean_squared_error: 1.2908
Epoch 91/100
180/180 ━━━━━━━━━━━━━━━━━━━━ 0s 668us/step - loss: 2.7237 -
root_mean_squared_error: 1.5786 - val_loss: 1.9964 -
val_root_mean_squared_error: 1.3311
Epoch 92/100
180/180 ━━━━━━━━━━━━━━━━━━━━ 0s 607us/step - loss: 3.0537 -
```

```
root_mean_squared_error: 1.6813 - val_loss: 2.8472 -
val_root_mean_squared_error: 1.6193
Epoch 93/100
180/180 ━━━━━━━━━━━━━━━━━━━━ 0s 576us/step - loss: 3.4843 -
root_mean_squared_error: 1.8032 - val_loss: 1.8885 -
val_root_mean_squared_error: 1.2903
Epoch 94/100
180/180 ━━━━━━━━━━━━━━━━━━━━ 0s 709us/step - loss: 2.9028 -
root_mean_squared_error: 1.6355 - val_loss: 3.5686 -
val_root_mean_squared_error: 1.8288
Epoch 95/100
180/180 ━━━━━━━━━━━━━━━━━━━━ 0s 561us/step - loss: 3.3566 -
root_mean_squared_error: 1.7658 - val_loss: 2.0371 -
val_root_mean_squared_error: 1.3466
Epoch 96/100
180/180 ━━━━━━━━━━━━━━━━━━━━ 0s 663us/step - loss: 3.3449 -
root_mean_squared_error: 1.7654 - val_loss: 2.5731 -
val_root_mean_squared_error: 1.5329
Epoch 97/100
180/180 ━━━━━━━━━━━━━━━━━━━━ 0s 558us/step - loss: 3.2050 -
root_mean_squared_error: 1.7264 - val_loss: 1.6663 -
val_root_mean_squared_error: 1.2009
Epoch 98/100
180/180 ━━━━━━━━━━━━━━━━━━━━ 0s 515us/step - loss: 2.8753 -
root_mean_squared_error: 1.6278 - val_loss: 1.7503 -
val_root_mean_squared_error: 1.2353
Epoch 99/100
180/180 ━━━━━━━━━━━━━━━━━━━━ 0s 599us/step - loss: 2.9179 -
root_mean_squared_error: 1.6403 - val_loss: 1.7628 -
val_root_mean_squared_error: 1.2406
Epoch 100/100
180/180 ━━━━━━━━━━━━━━━━━━━━ 0s 602us/step - loss: 2.6616 -
root_mean_squared_error: 1.5608 - val_loss: 1.6907 -
val_root_mean_squared_error: 1.2111
Completed training with Batch Size: 64, Epochs: 100. Test MSE: 1.9822,
Test RMSE: 1.3260
```




```
Training with Batch Size: 128, Epochs: 10
Epoch 1/10
90/90 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - loss: 4021.9604 -
root_mean_squared_error: 63.3469 - val_loss: 1078.6345 -
val_root_mean_squared_error: 32.8399
Epoch 2/10
90/90 ━━━━━━━━━━━━━━━━━━━━ 0s 726us/step - loss: 705.6896 -
root_mean_squared_error: 26.3722 - val_loss: 193.6669 -
val_root_mean_squared_error: 13.9088
Epoch 3/10
90/90 ━━━━━━━━━━━━━━━━━━━━ 0s 778us/step - loss: 232.5750 -
root_mean_squared_error: 15.2355 - val_loss: 120.5923 -
val_root_mean_squared_error: 10.9714
Epoch 4/10
90/90 ━━━━━━━━━━━━━━━━━━━━ 0s 789us/step - loss: 179.9086 -
root_mean_squared_error: 13.3978 - val_loss: 87.9312 -
val_root_mean_squared_error: 9.3654
Epoch 5/10
90/90 ━━━━━━━━━━━━━━━━━━━━ 0s 900us/step - loss: 143.5468 -
root_mean_squared_error: 11.9695 - val_loss: 67.5216 -
val_root_mean_squared_error: 8.2037
Epoch 6/10
90/90 ━━━━━━━━━━━━━━━━━━━━ 0s 680us/step - loss: 118.3933 -
root_mean_squared_error: 10.8706 - val_loss: 54.7138 -
```

```
val_root_mean_squared_error: 7.3821
Epoch 7/10
90/90 ━━━━━━━━━━━━━━━━━━━━ 0s 754us/step - loss: 113.9066 -
root_mean_squared_error: 10.6621 - val_loss: 44.4206 -
val_root_mean_squared_error: 6.6486
Epoch 8/10
90/90 ━━━━━━━━━━━━━━━━━━━━ 0s 861us/step - loss: 100.4037 -
root_mean_squared_error: 10.0085 - val_loss: 35.6617 -
val_root_mean_squared_error: 5.9537
Epoch 9/10
90/90 ━━━━━━━━━━━━━━━━━━━━ 0s 683us/step - loss: 89.6974 -
root_mean_squared_error: 9.4592 - val_loss: 29.5408 -
val_root_mean_squared_error: 5.4156
Epoch 10/10
90/90 ━━━━━━━━━━━━━━━━━━━━ 0s 699us/step - loss: 84.6805 -
root_mean_squared_error: 9.1901 - val_loss: 25.1805 -
val_root_mean_squared_error: 4.9971
Completed training with Batch Size: 128, Epochs: 10. Test MSE:
24.1471, Test RMSE: 4.8926
```




```
Training with Batch Size: 128, Epochs: 50
Epoch 1/50
90/90 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - loss: 3904.0818 -
root_mean_squared_error: 62.3988 - val_loss: 915.8704 -
```

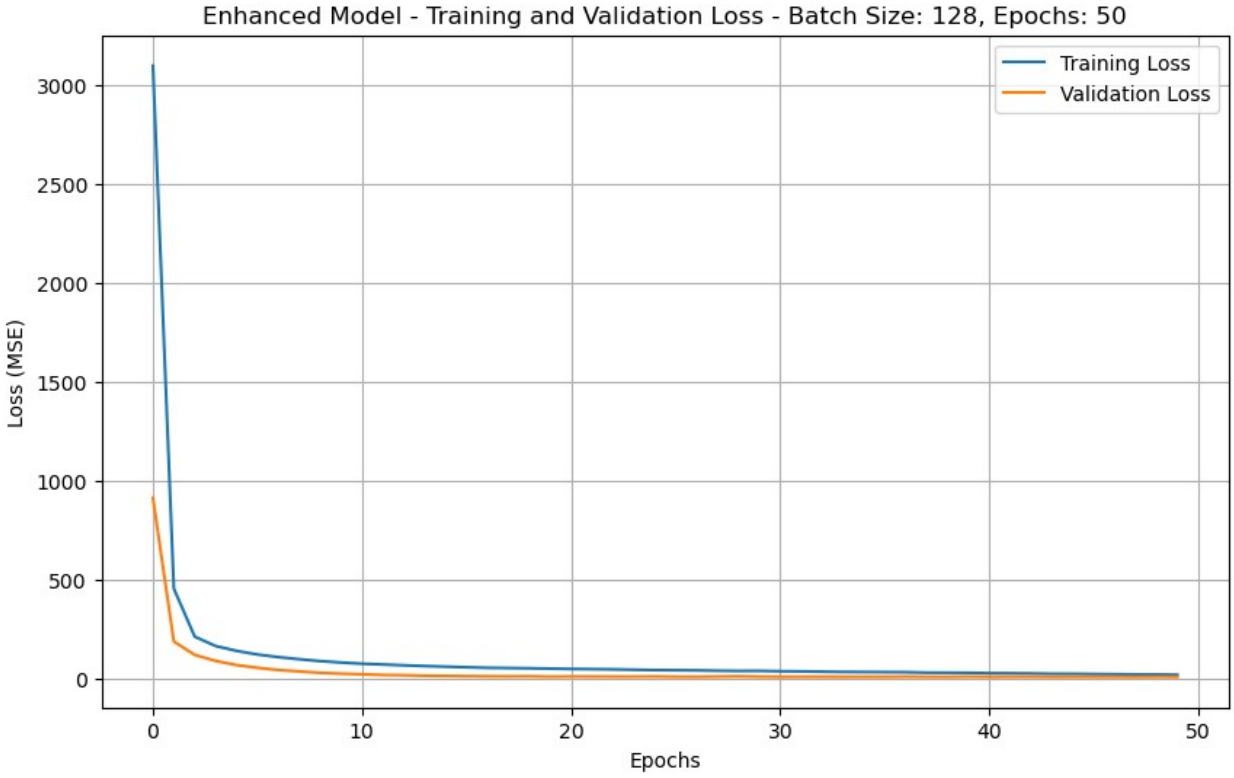
```
val_root_mean_squared_error: 30.2604
Epoch 2/50
90/90 ━━━━━━━━━━━━━━━━━━━━ 0s 728us/step - loss: 623.9133 -
root_mean_squared_error: 24.8219 - val_loss: 189.5198 -
val_root_mean_squared_error: 13.7590
Epoch 3/50
90/90 ━━━━━━━━━━━━━━━━━━━━ 0s 723us/step - loss: 229.4887 -
root_mean_squared_error: 15.1382 - val_loss: 122.3203 -
val_root_mean_squared_error: 11.0501
Epoch 4/50
90/90 ━━━━━━━━━━━━━━━━━━━━ 0s 907us/step - loss: 169.3016 -
root_mean_squared_error: 13.0027 - val_loss: 91.4987 -
val_root_mean_squared_error: 9.5541
Epoch 5/50
90/90 ━━━━━━━━━━━━━━━━━━━━ 0s 645us/step - loss: 147.5850 -
root_mean_squared_error: 12.1379 - val_loss: 70.5026 -
val_root_mean_squared_error: 8.3836
Epoch 6/50
90/90 ━━━━━━━━━━━━━━━━━━━━ 0s 673us/step - loss: 128.1564 -
root_mean_squared_error: 11.3100 - val_loss: 56.8535 -
val_root_mean_squared_error: 7.5258
Epoch 7/50
90/90 ━━━━━━━━━━━━━━━━━━━━ 0s 714us/step - loss: 112.8516 -
root_mean_squared_error: 10.6126 - val_loss: 45.6974 -
val_root_mean_squared_error: 6.7441
Epoch 8/50
90/90 ━━━━━━━━━━━━━━━━━━━━ 0s 688us/step - loss: 105.0407 -
root_mean_squared_error: 10.2363 - val_loss: 38.5299 -
val_root_mean_squared_error: 6.1902
Epoch 9/50
90/90 ━━━━━━━━━━━━━━━━━━━━ 0s 680us/step - loss: 90.9314 -
root_mean_squared_error: 9.5244 - val_loss: 31.3338 -
val_root_mean_squared_error: 5.5789
Epoch 10/50
90/90 ━━━━━━━━━━━━━━━━━━━━ 0s 664us/step - loss: 84.4892 -
root_mean_squared_error: 9.1796 - val_loss: 27.1024 -
val_root_mean_squared_error: 5.1861
Epoch 11/50
90/90 ━━━━━━━━━━━━━━━━━━━━ 0s 667us/step - loss: 78.5966 -
root_mean_squared_error: 8.8537 - val_loss: 23.9793 -
val_root_mean_squared_error: 4.8761
Epoch 12/50
90/90 ━━━━━━━━━━━━━━━━━━━━ 0s 880us/step - loss: 73.0497 -
root_mean_squared_error: 8.5339 - val_loss: 20.4211 -
val_root_mean_squared_error: 4.4968
Epoch 13/50
90/90 ━━━━━━━━━━━━━━━━━━━━ 0s 666us/step - loss: 73.2069 -
root_mean_squared_error: 8.5430 - val_loss: 19.1230 -
val_root_mean_squared_error: 4.3505
```

```
Epoch 14/50
90/90 ━━━━━━━━━━━━━━━━━━━━ 0s 665us/step - loss: 66.3922 -
root_mean_squared_error: 8.1357 - val_loss: 16.5676 -
val_root_mean_squared_error: 4.0465
Epoch 15/50
90/90 ━━━━━━━━━━━━━━━━━━━━ 0s 674us/step - loss: 63.6957 -
root_mean_squared_error: 7.9684 - val_loss: 15.6989 -
val_root_mean_squared_error: 3.9380
Epoch 16/50
90/90 ━━━━━━━━━━━━━━━━━━━━ 0s 678us/step - loss: 59.8399 -
root_mean_squared_error: 7.7229 - val_loss: 14.6542 -
val_root_mean_squared_error: 3.8036
Epoch 17/50
90/90 ━━━━━━━━━━━━━━━━━━━━ 0s 678us/step - loss: 57.9641 -
root_mean_squared_error: 7.6006 - val_loss: 13.6625 -
val_root_mean_squared_error: 3.6713
Epoch 18/50
90/90 ━━━━━━━━━━━━━━━━━━━━ 0s 675us/step - loss: 57.1175 -
root_mean_squared_error: 7.5449 - val_loss: 13.3075 -
val_root_mean_squared_error: 3.6230
Epoch 19/50
90/90 ━━━━━━━━━━━━━━━━━━━━ 0s 674us/step - loss: 53.5426 -
root_mean_squared_error: 7.3047 - val_loss: 13.7554 -
val_root_mean_squared_error: 3.6846
Epoch 20/50
90/90 ━━━━━━━━━━━━━━━━━━━━ 0s 873us/step - loss: 52.5276 -
root_mean_squared_error: 7.2349 - val_loss: 11.9420 -
val_root_mean_squared_error: 3.4300
Epoch 21/50
90/90 ━━━━━━━━━━━━━━━━━━━━ 0s 697us/step - loss: 51.4150 -
root_mean_squared_error: 7.1578 - val_loss: 12.8598 -
val_root_mean_squared_error: 3.5618
Epoch 22/50
90/90 ━━━━━━━━━━━━━━━━━━━━ 0s 703us/step - loss: 50.4442 -
root_mean_squared_error: 7.0899 - val_loss: 12.6327 -
val_root_mean_squared_error: 3.5302
Epoch 23/50
90/90 ━━━━━━━━━━━━━━━━━━━━ 0s 644us/step - loss: 47.7396 -
root_mean_squared_error: 6.8968 - val_loss: 12.1350 -
val_root_mean_squared_error: 3.4593
Epoch 24/50
90/90 ━━━━━━━━━━━━━━━━━━━━ 0s 660us/step - loss: 47.7685 -
root_mean_squared_error: 6.8984 - val_loss: 12.1917 -
val_root_mean_squared_error: 3.4678
Epoch 25/50
90/90 ━━━━━━━━━━━━━━━━━━━━ 0s 677us/step - loss: 45.1692 -
root_mean_squared_error: 6.7081 - val_loss: 12.8457 -
val_root_mean_squared_error: 3.5613
Epoch 26/50
```

```
90/90 ━━━━━━━━━━━━━━━━━━━━ 0s 699us/step - loss: 44.6029 -
root_mean_squared_error: 6.6661 - val_loss: 11.5717 -
val_root_mean_squared_error: 3.3780
Epoch 27/50
90/90 ━━━━━━━━━━━━━━━━━━━━ 0s 750us/step - loss: 44.1295 -
root_mean_squared_error: 6.6306 - val_loss: 11.3395 -
val_root_mean_squared_error: 3.3438
Epoch 28/50
90/90 ━━━━━━━━━━━━━━━━━━━━ 0s 862us/step - loss: 42.2231 -
root_mean_squared_error: 6.4853 - val_loss: 12.4084 -
val_root_mean_squared_error: 3.5003
Epoch 29/50
90/90 ━━━━━━━━━━━━━━━━━━━━ 0s 671us/step - loss: 40.0405 -
root_mean_squared_error: 6.3146 - val_loss: 13.7692 -
val_root_mean_squared_error: 3.6900
Epoch 30/50
90/90 ━━━━━━━━━━━━━━━━━━━━ 0s 685us/step - loss: 42.1631 -
root_mean_squared_error: 6.4814 - val_loss: 11.9923 -
val_root_mean_squared_error: 3.4411
Epoch 31/50
90/90 ━━━━━━━━━━━━━━━━━━━━ 0s 688us/step - loss: 39.8239 -
root_mean_squared_error: 6.2980 - val_loss: 11.2734 -
val_root_mean_squared_error: 3.3352
Epoch 32/50
90/90 ━━━━━━━━━━━━━━━━━━━━ 0s 671us/step - loss: 38.7359 -
root_mean_squared_error: 6.2105 - val_loss: 11.4585 -
val_root_mean_squared_error: 3.3632
Epoch 33/50
90/90 ━━━━━━━━━━━━━━━━━━━━ 0s 671us/step - loss: 37.2866 -
root_mean_squared_error: 6.0934 - val_loss: 11.6361 -
val_root_mean_squared_error: 3.3899
Epoch 34/50
90/90 ━━━━━━━━━━━━━━━━━━━━ 0s 665us/step - loss: 36.3188 -
root_mean_squared_error: 6.0139 - val_loss: 11.1519 -
val_root_mean_squared_error: 3.3181
Epoch 35/50
90/90 ━━━━━━━━━━━━━━━━━━━━ 0s 682us/step - loss: 36.2258 -
root_mean_squared_error: 6.0069 - val_loss: 11.3059 -
val_root_mean_squared_error: 3.3415
Epoch 36/50
90/90 ━━━━━━━━━━━━━━━━━━━━ 0s 863us/step - loss: 35.7905 -
root_mean_squared_error: 5.9700 - val_loss: 11.2127 -
val_root_mean_squared_error: 3.3278
Epoch 37/50
90/90 ━━━━━━━━━━━━━━━━━━━━ 0s 668us/step - loss: 34.8480 -
root_mean_squared_error: 5.8914 - val_loss: 11.9854 -
val_root_mean_squared_error: 3.4424
Epoch 38/50
90/90 ━━━━━━━━━━━━━━━━━━━━ 0s 685us/step - loss: 31.9065 -
```

```
root_mean_squared_error: 5.6362 - val_loss: 11.3676 -
val_root_mean_squared_error: 3.3517
Epoch 39/50
90/90 ━━━━━━━━━━━━━━━━━━━━ 0s 697us/step - loss: 31.7261 -
root_mean_squared_error: 5.6206 - val_loss: 11.0393 -
val_root_mean_squared_error: 3.3027
Epoch 40/50
90/90 ━━━━━━━━━━━━━━━━━━━━ 0s 653us/step - loss: 32.0691 -
root_mean_squared_error: 5.6509 - val_loss: 12.2706 -
val_root_mean_squared_error: 3.4845
Epoch 41/50
90/90 ━━━━━━━━━━━━━━━━━━━━ 0s 656us/step - loss: 29.6616 -
root_mean_squared_error: 5.4341 - val_loss: 11.0619 -
val_root_mean_squared_error: 3.3068
Epoch 42/50
90/90 ━━━━━━━━━━━━━━━━━━━━ 0s 688us/step - loss: 29.8731 -
root_mean_squared_error: 5.4539 - val_loss: 11.6173 -
val_root_mean_squared_error: 3.3901
Epoch 43/50
90/90 ━━━━━━━━━━━━━━━━━━━━ 0s 719us/step - loss: 28.2823 -
root_mean_squared_error: 5.3056 - val_loss: 11.8877 -
val_root_mean_squared_error: 3.4300
Epoch 44/50
90/90 ━━━━━━━━━━━━━━━━━━━━ 0s 875us/step - loss: 26.6606 -
root_mean_squared_error: 5.1512 - val_loss: 11.0177 -
val_root_mean_squared_error: 3.3009
Epoch 45/50
90/90 ━━━━━━━━━━━━━━━━━━━━ 0s 665us/step - loss: 27.1208 -
root_mean_squared_error: 5.1950 - val_loss: 11.2136 -
val_root_mean_squared_error: 3.3308
Epoch 46/50
90/90 ━━━━━━━━━━━━━━━━━━━━ 0s 685us/step - loss: 24.9828 -
root_mean_squared_error: 4.9859 - val_loss: 11.1351 -
val_root_mean_squared_error: 3.3193
Epoch 47/50
90/90 ━━━━━━━━━━━━━━━━━━━━ 0s 674us/step - loss: 25.2727 -
root_mean_squared_error: 5.0149 - val_loss: 10.9900 -
val_root_mean_squared_error: 3.2977
Epoch 48/50
90/90 ━━━━━━━━━━━━━━━━━━━━ 0s 706us/step - loss: 23.3131 -
root_mean_squared_error: 4.8161 - val_loss: 10.7233 -
val_root_mean_squared_error: 3.2572
Epoch 49/50
90/90 ━━━━━━━━━━━━━━━━━━━━ 0s 662us/step - loss: 23.1187 -
root_mean_squared_error: 4.7962 - val_loss: 11.3509 -
val_root_mean_squared_error: 3.3525
Epoch 50/50
90/90 ━━━━━━━━━━━━━━━━━━━━ 0s 665us/step - loss: 21.9282 -
root_mean_squared_error: 4.6697 - val_loss: 10.7634 -
```

```
val_root_mean_squared_error: 3.2639
Completed training with Batch Size: 128, Epochs: 50. Test MSE:
10.2592, Test RMSE: 3.1852
```



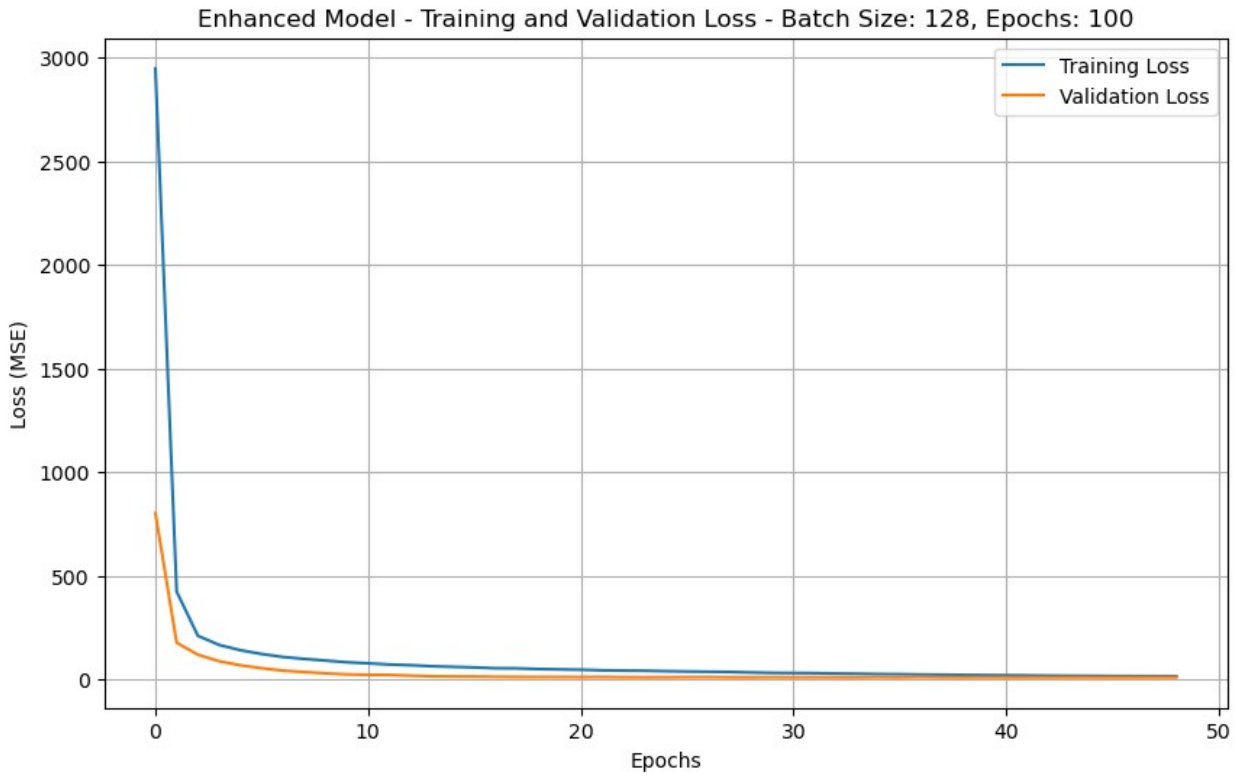

```
Training with Batch Size: 128, Epochs: 100
Epoch 1/100
90/90 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - loss: 3816.7175 -
root_mean_squared_error: 61.6669 - val_loss: 802.1860 -
val_root_mean_squared_error: 28.3197
Epoch 2/100
90/90 ━━━━━━━━━━━━━━━━━━━━ 0s 827us/step - loss: 579.7158 -
root_mean_squared_error: 23.9371 - val_loss: 177.4833 -
val_root_mean_squared_error: 13.3143
Epoch 3/100
90/90 ━━━━━━━━━━━━━━━━━━━━ 0s 676us/step - loss: 218.3804 -
root_mean_squared_error: 14.7645 - val_loss: 119.7956 -
val_root_mean_squared_error: 10.9353
Epoch 4/100
90/90 ━━━━━━━━━━━━━━━━━━━━ 0s 694us/step - loss: 171.2490 -
root_mean_squared_error: 13.0763 - val_loss: 88.3581 -
val_root_mean_squared_error: 9.3884
Epoch 5/100
90/90 ━━━━━━━━━━━━━━━━━━━━ 0s 702us/step - loss: 147.7593 -
root_mean_squared_error: 12.1457 - val_loss: 68.7515 -
```

```
val_root_mean_squared_error: 8.2786
Epoch 6/100
90/90 ━━━━━━━━━━━━━━━━━━━━ 0s 682us/step - loss: 126.8334 -
root_mean_squared_error: 11.2518 - val_loss: 54.6581 -
val_root_mean_squared_error: 7.3786
Epoch 7/100
90/90 ━━━━━━━━━━━━━━━━━━━━ 0s 685us/step - loss: 111.2421 -
root_mean_squared_error: 10.5363 - val_loss: 43.1102 -
val_root_mean_squared_error: 6.5497
Epoch 8/100
90/90 ━━━━━━━━━━━━━━━━━━━━ 0s 691us/step - loss: 103.8264 -
root_mean_squared_error: 10.1771 - val_loss: 36.3179 -
val_root_mean_squared_error: 6.0091
Epoch 9/100
90/90 ━━━━━━━━━━━━━━━━━━━━ 0s 688us/step - loss: 91.3673 -
root_mean_squared_error: 9.5475 - val_loss: 29.8423 -
val_root_mean_squared_error: 5.4438
Epoch 10/100
90/90 ━━━━━━━━━━━━━━━━━━━━ 0s 870us/step - loss: 84.3023 -
root_mean_squared_error: 9.1701 - val_loss: 24.9612 -
val_root_mean_squared_error: 4.9756
Epoch 11/100
90/90 ━━━━━━━━━━━━━━━━━━━━ 0s 723us/step - loss: 79.3676 -
root_mean_squared_error: 8.8970 - val_loss: 23.0113 -
val_root_mean_squared_error: 4.7760
Epoch 12/100
90/90 ━━━━━━━━━━━━━━━━━━━━ 0s 708us/step - loss: 72.9927 -
root_mean_squared_error: 8.5311 - val_loss: 22.3450 -
val_root_mean_squared_error: 4.7061
Epoch 13/100
90/90 ━━━━━━━━━━━━━━━━━━━━ 0s 699us/step - loss: 69.3067 -
root_mean_squared_error: 8.3127 - val_loss: 18.7758 -
val_root_mean_squared_error: 4.3106
Epoch 14/100
90/90 ━━━━━━━━━━━━━━━━━━━━ 0s 692us/step - loss: 66.3413 -
root_mean_squared_error: 8.1314 - val_loss: 15.5469 -
val_root_mean_squared_error: 3.9185
Epoch 15/100
90/90 ━━━━━━━━━━━━━━━━━━━━ 0s 685us/step - loss: 63.1448 -
root_mean_squared_error: 7.9328 - val_loss: 14.8174 -
val_root_mean_squared_error: 3.8247
Epoch 16/100
90/90 ━━━━━━━━━━━━━━━━━━━━ 0s 702us/step - loss: 58.7766 -
root_mean_squared_error: 7.6539 - val_loss: 14.4772 -
val_root_mean_squared_error: 3.7804
Epoch 17/100
90/90 ━━━━━━━━━━━━━━━━━━━━ 0s 694us/step - loss: 54.7219 -
root_mean_squared_error: 7.3845 - val_loss: 13.3812 -
val_root_mean_squared_error: 3.6329
```

```
Epoch 18/100
90/90 ━━━━━━━━━━━━━━━━━━━━ 0s 844us/step - loss: 54.0414 -
root_mean_squared_error: 7.3386 - val_loss: 12.6702 -
val_root_mean_squared_error: 3.5341
Epoch 19/100
90/90 ━━━━━━━━━━━━━━━━━━━━ 0s 675us/step - loss: 51.1440 -
root_mean_squared_error: 7.1388 - val_loss: 12.2630 -
val_root_mean_squared_error: 3.4765
Epoch 20/100
90/90 ━━━━━━━━━━━━━━━━━━━━ 0s 753us/step - loss: 49.7285 -
root_mean_squared_error: 7.0388 - val_loss: 12.4802 -
val_root_mean_squared_error: 3.5080
Epoch 21/100
90/90 ━━━━━━━━━━━━━━━━━━━━ 0s 774us/step - loss: 48.5751 -
root_mean_squared_error: 6.9565 - val_loss: 11.9270 -
val_root_mean_squared_error: 3.4287
Epoch 22/100
90/90 ━━━━━━━━━━━━━━━━━━━━ 0s 913us/step - loss: 44.8458 -
root_mean_squared_error: 6.6838 - val_loss: 12.3763 -
val_root_mean_squared_error: 3.4939
Epoch 23/100
90/90 ━━━━━━━━━━━━━━━━━━━━ 0s 752us/step - loss: 43.8742 -
root_mean_squared_error: 6.6108 - val_loss: 11.5376 -
val_root_mean_squared_error: 3.3721
Epoch 24/100
90/90 ━━━━━━━━━━━━━━━━━━━━ 0s 705us/step - loss: 43.3558 -
root_mean_squared_error: 6.5716 - val_loss: 11.0807 -
val_root_mean_squared_error: 3.3041
Epoch 25/100
90/90 ━━━━━━━━━━━━━━━━━━━━ 0s 700us/step - loss: 39.6342 -
root_mean_squared_error: 6.2824 - val_loss: 11.3322 -
val_root_mean_squared_error: 3.3424
Epoch 26/100
90/90 ━━━━━━━━━━━━━━━━━━━━ 0s 891us/step - loss: 38.4678 -
root_mean_squared_error: 6.1892 - val_loss: 11.8089 -
val_root_mean_squared_error: 3.4133
Epoch 27/100
90/90 ━━━━━━━━━━━━━━━━━━━━ 0s 647us/step - loss: 38.8868 -
root_mean_squared_error: 6.2221 - val_loss: 12.1224 -
val_root_mean_squared_error: 3.4593
Epoch 28/100
90/90 ━━━━━━━━━━━━━━━━━━━━ 0s 688us/step - loss: 37.1880 -
root_mean_squared_error: 6.0849 - val_loss: 11.1583 -
val_root_mean_squared_error: 3.3173
Epoch 29/100
90/90 ━━━━━━━━━━━━━━━━━━━━ 0s 683us/step - loss: 34.5731 -
root_mean_squared_error: 5.8666 - val_loss: 10.9928 -
val_root_mean_squared_error: 3.2927
Epoch 30/100
```

```
90/90 ━━━━━━━━━━━━━━━━━━━━ 0s 679us/step - loss: 32.8475 -
root_mean_squared_error: 5.7180 - val_loss: 11.4502 -
val_root_mean_squared_error: 3.3618
Epoch 31/100
90/90 ━━━━━━━━━━━━━━━━━━━━ 0s 697us/step - loss: 31.0978 -
root_mean_squared_error: 5.5625 - val_loss: 11.0710 -
val_root_mean_squared_error: 3.3052
Epoch 32/100
90/90 ━━━━━━━━━━━━━━━━━━━━ 0s 678us/step - loss: 31.3177 -
root_mean_squared_error: 5.5826 - val_loss: 11.2371 -
val_root_mean_squared_error: 3.3306
Epoch 33/100
90/90 ━━━━━━━━━━━━━━━━━━━━ 0s 686us/step - loss: 29.6724 -
root_mean_squared_error: 5.4334 - val_loss: 11.0768 -
val_root_mean_squared_error: 3.3068
Epoch 34/100
90/90 ━━━━━━━━━━━━━━━━━━━━ 0s 705us/step - loss: 27.9051 -
root_mean_squared_error: 5.2690 - val_loss: 10.8451 -
val_root_mean_squared_error: 3.2719
Epoch 35/100
90/90 ━━━━━━━━━━━━━━━━━━━━ 0s 892us/step - loss: 26.3845 -
root_mean_squared_error: 5.1229 - val_loss: 11.6402 -
val_root_mean_squared_error: 3.3916
Epoch 36/100
90/90 ━━━━━━━━━━━━━━━━━━━━ 0s 730us/step - loss: 26.0889 -
root_mean_squared_error: 5.0942 - val_loss: 10.8070 -
val_root_mean_squared_error: 3.2667
Epoch 37/100
90/90 ━━━━━━━━━━━━━━━━━━━━ 0s 712us/step - loss: 24.2411 -
root_mean_squared_error: 4.9097 - val_loss: 12.2002 -
val_root_mean_squared_error: 3.4737
Epoch 38/100
90/90 ━━━━━━━━━━━━━━━━━━━━ 0s 674us/step - loss: 23.8186 -
root_mean_squared_error: 4.8659 - val_loss: 11.0015 -
val_root_mean_squared_error: 3.2969
Epoch 39/100
90/90 ━━━━━━━━━━━━━━━━━━━━ 0s 694us/step - loss: 22.3646 -
root_mean_squared_error: 4.7142 - val_loss: 10.7542 -
val_root_mean_squared_error: 3.2595
Epoch 40/100
90/90 ━━━━━━━━━━━━━━━━━━━━ 0s 716us/step - loss: 21.0568 -
root_mean_squared_error: 4.5743 - val_loss: 11.1336 -
val_root_mean_squared_error: 3.3175
Epoch 41/100
90/90 ━━━━━━━━━━━━━━━━━━━━ 0s 701us/step - loss: 21.3974 -
root_mean_squared_error: 4.6110 - val_loss: 10.8403 -
val_root_mean_squared_error: 3.2733
Epoch 42/100
90/90 ━━━━━━━━━━━━━━━━━━━━ 0s 692us/step - loss: 19.8781 -
```

```
root_mean_squared_error: 4.4442 - val_loss: 10.7706 -
val_root_mean_squared_error: 3.2629
Epoch 43/100
90/90 ━━━━━━━━━━━━━━━━━━━━ 0s 846us/step - loss: 18.8228 -
root_mean_squared_error: 4.3240 - val_loss: 10.7764 -
val_root_mean_squared_error: 3.2640
Epoch 44/100
90/90 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - loss: 17.7533 -
root_mean_squared_error: 4.1985 - val_loss: 11.0010 -
val_root_mean_squared_error: 3.2985
Epoch 45/100
90/90 ━━━━━━━━━━━━━━━━━━━━ 0s 886us/step - loss: 17.4281 -
root_mean_squared_error: 4.1601 - val_loss: 10.8864 -
val_root_mean_squared_error: 3.2813
Epoch 46/100
90/90 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - loss: 16.9451 -
root_mean_squared_error: 4.1017 - val_loss: 10.8198 -
val_root_mean_squared_error: 3.2714
Epoch 47/100
90/90 ━━━━━━━━━━━━━━━━━━━━ 0s 599us/step - loss: 16.9868 -
root_mean_squared_error: 4.1059 - val_loss: 10.8194 -
val_root_mean_squared_error: 3.2715
Epoch 48/100
90/90 ━━━━━━━━━━━━━━━━━━━━ 0s 641us/step - loss: 16.5122 -
root_mean_squared_error: 4.0485 - val_loss: 10.7785 -
val_root_mean_squared_error: 3.2655
Epoch 49/100
90/90 ━━━━━━━━━━━━━━━━━━━━ 0s 963us/step - loss: 15.9442 -
root_mean_squared_error: 3.9779 - val_loss: 10.9740 -
val_root_mean_squared_error: 3.2955
Completed training with Batch Size: 128, Epochs: 100. Test MSE:
10.2481, Test RMSE: 3.1809
```




```
# Converting the results to a DataFrame
keras_results_df = pd.DataFrame(keras_results_df)
keras_results_df
```

	Epochs	Batch Size	MSE	RMSE
0	10	16	10.265473	3.189894
1	50	16	2.461612	1.513683
2	100	16	2.330091	1.471696
3	10	32	11.457017	3.365021
4	50	32	2.580845	1.549235
5	100	32	2.147320	1.401063
6	10	64	12.576398	3.521182
7	50	64	3.587555	1.857876
8	100	64	1.982183	1.325954
9	10	128	24.147118	4.892643
10	50	128	10.259161	3.185181
11	100	128	10.248078	3.180917

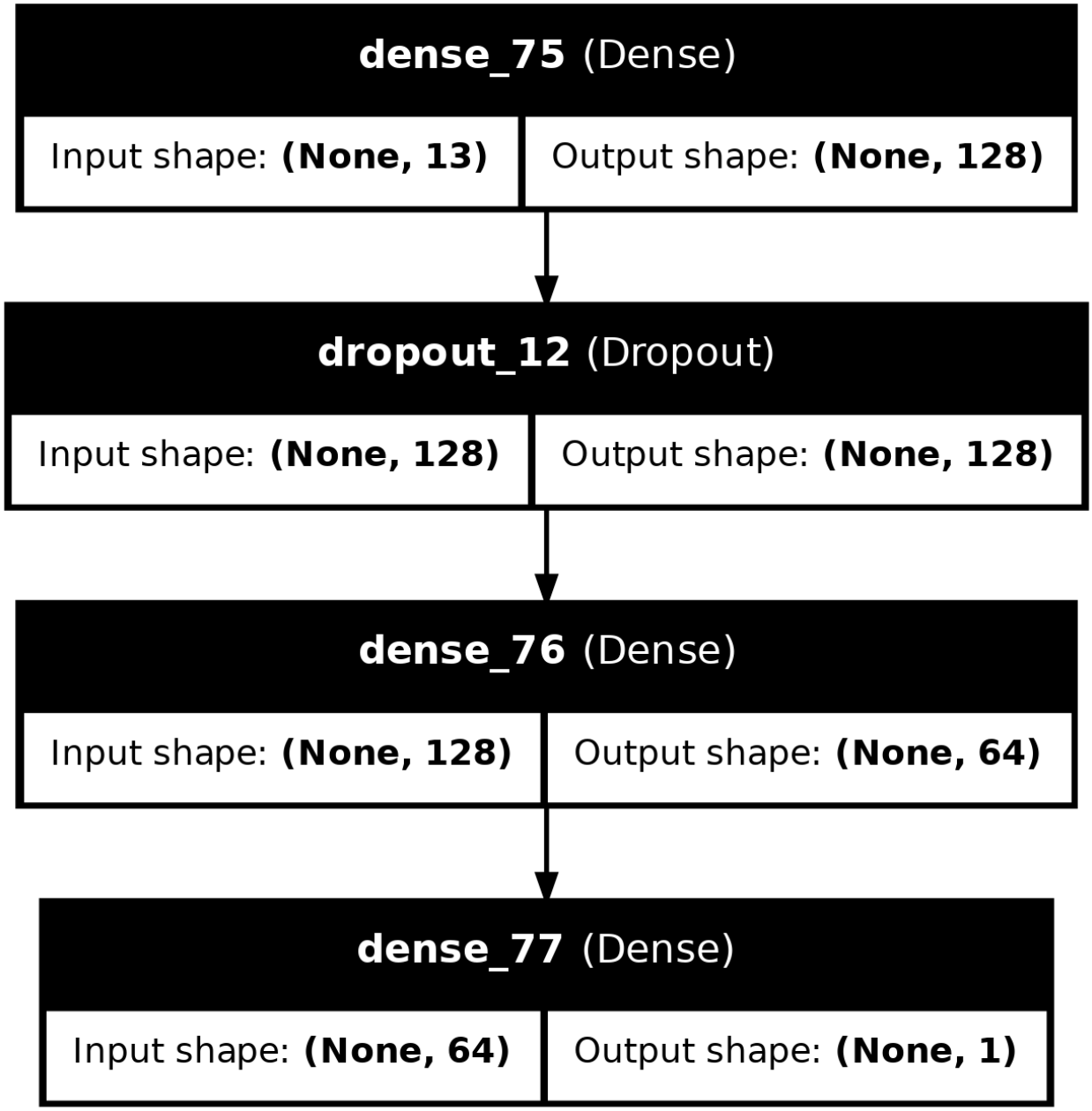
The table provides insights into the performance of the enhanced Keras model, where dropout and regularization techniques were added to improve generalization. As seen in the results, the model continues to perform best with batch size 64 and 100 epochs, achieving the lowest MSE and RMSE values, indicating better prediction accuracy. The use of dropout and L2 regularization appears to help with overfitting as higher epochs and smaller batch sizes tend to show improvement. However, large batch sizes like 128 lead to significant underperformance, with high MSE and RMSE values. This suggests that while regularization techniques help, the choice of batch size and epochs is still critical for optimal performance.

Architecture

Using below code similar to that we have used in base model we would like to visually inspect the architecture of our model.

```
# Building a regression model with Dropout and L2 Regularization
model = models.Sequential([
    layers.Dense(128, activation='relu',
input_shape=(X_train.shape[1],),
                 kernel_regularizer=regularizers.l2(0.001)),  # L2
Regularization
    layers.Dropout(0.5),  # Dropout with a rate of 0.5
    layers.Dense(64, activation='relu',
kernel_regularizer=regularizers.l2(0.001)),
    layers.Dense(1)  # Output layer for regression
])

# Compiling the model with Adam optimizer
model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=0.001),
              loss='mse',
              metrics=[tf.keras.metrics.RootMeanSquaredError()])

# Visualizing the model architecture and saving as an image
plot_model(model, to_file='model_plot.png', show_shapes=True,
show_layer_names=True)

# Displaying the model plot in Colab
from IPython.display import Image
Image('model_plot.png')
```

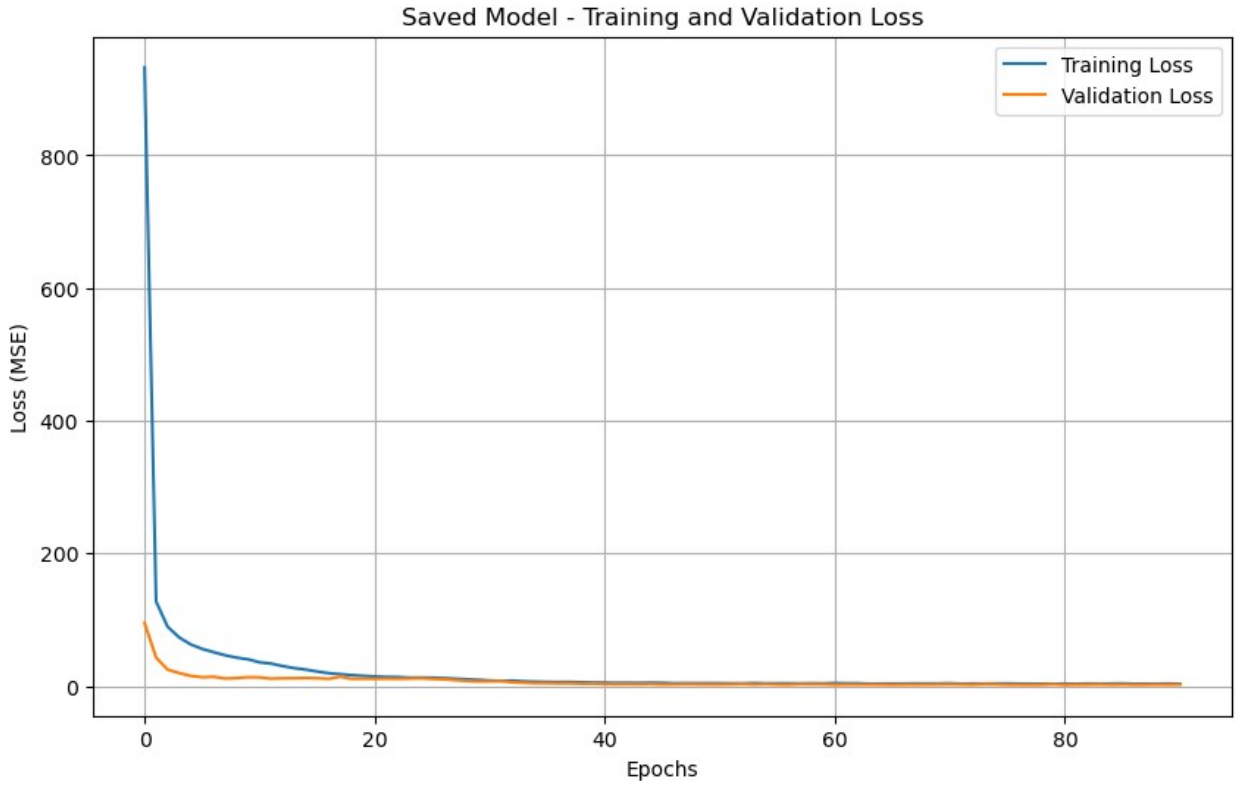
```
/home/unina/anaconda3/lib/python3.12/site-packages/keras/src/layers/
core/dense.py:87: UserWarning: Do not pass an
`input_shape`/`input_dim` argument to a layer. When using Sequential
models, prefer using an `Input(shape)` object as the first layer in
the model instead.
  super().__init__(activity_regularizer=activity_regularizer,
**kwargs)
```



- Layer 1 (dense_123): Input: 13 features, corresponding to the player attributes. Output: 128 neurons, using the ReLU activation function. L2 regularization is applied to this layer to reduce overfitting by penalizing large weights.
- Dropout Layer (dropout_21): Dropout rate of 0.5, meaning that 50% of the neurons will be randomly dropped during training to prevent overfitting and improve generalization.
- Layer 2 (dense_124): Output: 64 neurons, also with ReLU activation. L2 regularization is again applied.

- Output Layer (dense_125): Produces a single output for the regression task (player performance prediction).

```
# Loading the saved history object
with open('history_batch32_epochs100.pkl', 'rb') as file:
    saved_history = pickle.load(file)

# Plotting the saved training and validation loss curves
def plot_saved_loss_curves(saved_history):
    plt.figure(figsize=(10, 6))
    plt.plot(saved_history['loss'], label='Training Loss')
    plt.plot(saved_history['val_loss'], label='Validation Loss')
    plt.xlabel('Epochs')
    plt.ylabel('Loss (MSE)')
    plt.title('Saved Model - Training and Validation Loss')
    plt.legend()
    plt.grid(True)
    plt.show()

# Calling the function to plotting the saved loss curves
plot_saved_loss_curves(saved_history)
```



Optimal Parameters

In our approach to finding the best optimal parameters, we employed an iterative process by testing various combinations of epochs and batch sizes to tune the model's performance. For both PyTorch and Keras models, we experimented with different configurations and tracked metrics such as Mean Squared Error (MSE) and Root Mean Squared Error (RMSE) on both training and validation sets. We utilized techniques like early stopping, L2 regularization, and Dropout to prevent overfitting and ensure that the model generalizes well on unseen data. By systematically comparing performance across these configurations, we identified the optimal parameters that produced the best predictive results.

optimal hyperparameters (batch size = and epochs =)

Automated

```
# Finding the best configuration (based on lowest MSE or RMSE)
best_row = keras_results_df.loc[keras_results_df['MSE'].idxmin()]
best_epochs = best_row['Epochs']
best_batch_size = best_row['Batch Size']

print(f"\nBest configuration: Epochs={best_epochs}, Batch
Size={best_batch_size}, MSE={best_row['MSE']},
RMSE={best_row['RMSE']}")
```

```
Best configuration: Epochs=100.0, Batch Size=64.0,
MSE=1.9821832180023193, RMSE=1.3259544372558594
```

```
# Saving the best model
model.save('best_model_final.keras')
print("Best model saved as 'best_model_final.keras'")
```

```
Best model saved as 'best_model_final.keras'
```

Manual

```
# Displaying the results DataFrame to manually inspect and choose the
best configuration
print(keras_results_df)

# manually using the batch size 32 and 100 epochs
preferred_epochs = 100
preferred_batch_size = 32

# Retrain the model with the manually selected configuration
model = Sequential([
    layers.Dense(128, activation='relu',
input_shape=(X_train.shape[1],),
kernel_regularizer=regularizers.l2(0.001)),
    layers.Dropout(0.5),
    layers.Dense(64, activation='relu',
kernel_regularizer=regularizers.l2(0.001)),
```

```
    layers.Dense(1)
])

# Compiling the model
model.compile(optimizer=Adam(learning_rate=0.001), loss='mse',
metrics=[RootMeanSquaredError()])

# Retraining the model with the manually chosen configuration
history = model.fit(X_train, y_train, validation_split=0.2,
epochs=preferred_epochs, batch_size=preferred_batch_size, verbose=1)

#  Saving the manually selected model
model.save('best_model_manual_32batch_100epochs.keras')
print(f"Manually chosen best model saved with
Epochs={preferred_epochs}, Batch Size={preferred_batch_size}")
```

```
/home/unina/anaconda3/lib/python3.12/site-packages/keras/src/layers/
core/dense.py:87: UserWarning: Do not pass an
`input_shape`/`input_dim` argument to a layer. When using Sequential
models, prefer using an `Input(shape)` object as the first layer in
the model instead.
  super().__init__(activity_regularizer=activity_regularizer,
**kwargs)
```

	Epochs	Batch Size	MSE	RMSE
0	10	16	10.265473	3.189894
1	50	16	2.461612	1.513683
2	100	16	2.330091	1.471696
3	10	32	11.457017	3.365021
4	50	32	2.580845	1.549235
5	100	32	2.147320	1.401063
6	10	64	12.576398	3.521182
7	50	64	3.587555	1.857876
8	100	64	1.982183	1.325954
9	10	128	24.147118	4.892643
10	50	128	10.259161	3.185181
11	100	128	10.248078	3.180917

```
Epoch 1/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 1s 787us/step - loss: 2262.7390 -
root_mean_squared_error: 46.1878 - val_loss: 91.9585 -
val_root_mean_squared_error: 9.5789
Epoch 2/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 505us/step - loss: 134.4668 -
root_mean_squared_error: 11.5844 - val_loss: 44.0445 -
val_root_mean_squared_error: 6.6217
Epoch 3/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 565us/step - loss: 101.6580 -
root_mean_squared_error: 10.0644 - val_loss: 27.3907 -
val_root_mean_squared_error: 5.2154
Epoch 4/100
```

```
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 536us/step - loss: 73.1634 -
root_mean_squared_error: 8.5420 - val_loss: 18.8792 -
val_root_mean_squared_error: 4.3242
Epoch 5/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 463us/step - loss: 67.5248 -
root_mean_squared_error: 8.2046 - val_loss: 16.8670 -
val_root_mean_squared_error: 4.0857
Epoch 6/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 487us/step - loss: 57.8604 -
root_mean_squared_error: 7.5945 - val_loss: 15.0419 -
val_root_mean_squared_error: 3.8570
Epoch 7/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 493us/step - loss: 54.0410 -
root_mean_squared_error: 7.3396 - val_loss: 12.8927 -
val_root_mean_squared_error: 3.5684
Epoch 8/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 492us/step - loss: 51.2421 -
root_mean_squared_error: 7.1471 - val_loss: 12.6298 -
val_root_mean_squared_error: 3.5323
Epoch 9/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 464us/step - loss: 47.4157 -
root_mean_squared_error: 6.8746 - val_loss: 12.0811 -
val_root_mean_squared_error: 3.4547
Epoch 10/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 585us/step - loss: 44.8300 -
root_mean_squared_error: 6.6844 - val_loss: 15.1539 -
val_root_mean_squared_error: 3.8749
Epoch 11/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 558us/step - loss: 42.6756 -
root_mean_squared_error: 6.5215 - val_loss: 11.4916 -
val_root_mean_squared_error: 3.3702
Epoch 12/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 493us/step - loss: 37.9512 -
root_mean_squared_error: 6.1484 - val_loss: 13.9060 -
val_root_mean_squared_error: 3.7122
Epoch 13/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 596us/step - loss: 35.4651 -
root_mean_squared_error: 5.9428 - val_loss: 13.2172 -
val_root_mean_squared_error: 3.6191
Epoch 14/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 444us/step - loss: 32.2493 -
root_mean_squared_error: 5.6678 - val_loss: 10.8958 -
val_root_mean_squared_error: 3.2836
Epoch 15/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 587us/step - loss: 30.1590 -
root_mean_squared_error: 5.4800 - val_loss: 12.0491 -
val_root_mean_squared_error: 3.4556
Epoch 16/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 449us/step - loss: 27.3467 -
```

```
root_mean_squared_error: 5.2185 - val_loss: 11.6637 -
val_root_mean_squared_error: 3.4002
Epoch 17/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 557us/step - loss: 25.0810 -
root_mean_squared_error: 4.9962 - val_loss: 11.7724 -
val_root_mean_squared_error: 3.4167
Epoch 18/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 548us/step - loss: 21.3225 -
root_mean_squared_error: 4.6064 - val_loss: 10.5662 -
val_root_mean_squared_error: 3.2360
Epoch 19/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 587us/step - loss: 19.2025 -
root_mean_squared_error: 4.3712 - val_loss: 10.1349 -
val_root_mean_squared_error: 3.1692
Epoch 20/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 679us/step - loss: 16.6768 -
root_mean_squared_error: 4.0724 - val_loss: 10.0132 -
val_root_mean_squared_error: 3.1503
Epoch 21/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 465us/step - loss: 15.6194 -
root_mean_squared_error: 3.9397 - val_loss: 9.2045 -
val_root_mean_squared_error: 3.0194
Epoch 22/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 482us/step - loss: 13.8779 -
root_mean_squared_error: 3.7130 - val_loss: 9.8330 -
val_root_mean_squared_error: 3.1218
Epoch 23/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 544us/step - loss: 12.3026 -
root_mean_squared_error: 3.4948 - val_loss: 7.7270 -
val_root_mean_squared_error: 2.7640
Epoch 24/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 476us/step - loss: 10.9975 -
root_mean_squared_error: 3.3029 - val_loss: 7.3834 -
val_root_mean_squared_error: 2.7011
Epoch 25/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 581us/step - loss: 10.3732 -
root_mean_squared_error: 3.2067 - val_loss: 6.8922 -
val_root_mean_squared_error: 2.6082
Epoch 26/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 564us/step - loss: 9.0999 -
root_mean_squared_error: 3.0014 - val_loss: 6.1003 -
val_root_mean_squared_error: 2.4511
Epoch 27/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 529us/step - loss: 8.9739 -
root_mean_squared_error: 2.9765 - val_loss: 6.1978 -
val_root_mean_squared_error: 2.4700
Epoch 28/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 526us/step - loss: 7.3811 -
root_mean_squared_error: 2.6984 - val_loss: 4.9937 -
```

```
val_root_mean_squared_error: 2.2118
Epoch 29/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 540us/step - loss: 7.7469 -
root_mean_squared_error: 2.7639 - val_loss: 5.7977 -
val_root_mean_squared_error: 2.3853
Epoch 30/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 525us/step - loss: 6.5357 -
root_mean_squared_error: 2.5345 - val_loss: 4.8247 -
val_root_mean_squared_error: 2.1703
Epoch 31/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 537us/step - loss: 6.3192 -
root_mean_squared_error: 2.4904 - val_loss: 4.1030 -
val_root_mean_squared_error: 1.9956
Epoch 32/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 489us/step - loss: 5.8081 -
root_mean_squared_error: 2.3839 - val_loss: 3.7345 -
val_root_mean_squared_error: 1.8993
Epoch 33/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 593us/step - loss: 5.8927 -
root_mean_squared_error: 2.3991 - val_loss: 4.1200 -
val_root_mean_squared_error: 1.9966
Epoch 34/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 623us/step - loss: 5.9717 -
root_mean_squared_error: 2.4147 - val_loss: 3.4996 -
val_root_mean_squared_error: 1.8329
Epoch 35/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 584us/step - loss: 5.0695 -
root_mean_squared_error: 2.2188 - val_loss: 3.3671 -
val_root_mean_squared_error: 1.7945
Epoch 36/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 535us/step - loss: 4.8740 -
root_mean_squared_error: 2.1729 - val_loss: 2.9828 -
val_root_mean_squared_error: 1.6819
Epoch 37/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 504us/step - loss: 4.6680 -
root_mean_squared_error: 2.1241 - val_loss: 3.4925 -
val_root_mean_squared_error: 1.8255
Epoch 38/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 492us/step - loss: 4.2591 -
root_mean_squared_error: 2.0238 - val_loss: 2.5546 -
val_root_mean_squared_error: 1.5459
Epoch 39/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 536us/step - loss: 4.3460 -
root_mean_squared_error: 2.0444 - val_loss: 2.4988 -
val_root_mean_squared_error: 1.5263
Epoch 40/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 667us/step - loss: 4.6932 -
root_mean_squared_error: 2.1247 - val_loss: 4.4100 -
val_root_mean_squared_error: 2.0588
```

```
Epoch 41/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 501us/step - loss: 4.8774 -
root_mean_squared_error: 2.1676 - val_loss: 2.9901 -
val_root_mean_squared_error: 1.6777
Epoch 42/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 572us/step - loss: 3.9643 -
root_mean_squared_error: 1.9454 - val_loss: 3.0586 -
val_root_mean_squared_error: 1.6967
Epoch 43/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 542us/step - loss: 4.2179 -
root_mean_squared_error: 2.0074 - val_loss: 3.0637 -
val_root_mean_squared_error: 1.6971
Epoch 44/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 470us/step - loss: 4.0908 -
root_mean_squared_error: 1.9751 - val_loss: 3.4336 -
val_root_mean_squared_error: 1.8021
Epoch 45/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 544us/step - loss: 6.4314 -
root_mean_squared_error: 2.4732 - val_loss: 2.2679 -
val_root_mean_squared_error: 1.4427
Epoch 46/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 559us/step - loss: 3.4742 -
root_mean_squared_error: 1.8122 - val_loss: 2.6300 -
val_root_mean_squared_error: 1.5623
Epoch 47/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 447us/step - loss: 3.7731 -
root_mean_squared_error: 1.8926 - val_loss: 2.6377 -
val_root_mean_squared_error: 1.5642
Epoch 48/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 515us/step - loss: 3.8846 -
root_mean_squared_error: 1.9198 - val_loss: 2.1187 -
val_root_mean_squared_error: 1.3882
Epoch 49/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 486us/step - loss: 3.5922 -
root_mean_squared_error: 1.8424 - val_loss: 2.3100 -
val_root_mean_squared_error: 1.4553
Epoch 50/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 599us/step - loss: 3.5726 -
root_mean_squared_error: 1.8361 - val_loss: 2.4491 -
val_root_mean_squared_error: 1.5019
Epoch 51/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 530us/step - loss: 3.5849 -
root_mean_squared_error: 1.8411 - val_loss: 2.8506 -
val_root_mean_squared_error: 1.6301
Epoch 52/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 617us/step - loss: 4.2502 -
root_mean_squared_error: 2.0123 - val_loss: 2.5558 -
val_root_mean_squared_error: 1.5369
Epoch 53/100
```

```
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 575us/step - loss: 3.8879 -
root_mean_squared_error: 1.9199 - val_loss: 2.6587 -
val_root_mean_squared_error: 1.5701
Epoch 54/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 497us/step - loss: 3.9374 -
root_mean_squared_error: 1.9335 - val_loss: 2.3376 -
val_root_mean_squared_error: 1.4644
Epoch 55/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 558us/step - loss: 3.9936 -
root_mean_squared_error: 1.9475 - val_loss: 2.4235 -
val_root_mean_squared_error: 1.4934
Epoch 56/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 453us/step - loss: 3.5903 -
root_mean_squared_error: 1.8410 - val_loss: 2.3147 -
val_root_mean_squared_error: 1.4569
Epoch 57/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 530us/step - loss: 3.4608 -
root_mean_squared_error: 1.8066 - val_loss: 2.3340 -
val_root_mean_squared_error: 1.4637
Epoch 58/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 455us/step - loss: 3.5626 -
root_mean_squared_error: 1.8336 - val_loss: 2.7933 -
val_root_mean_squared_error: 1.6127
Epoch 59/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 600us/step - loss: 3.7377 -
root_mean_squared_error: 1.8821 - val_loss: 2.6903 -
val_root_mean_squared_error: 1.5805
Epoch 60/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 531us/step - loss: 3.8123 -
root_mean_squared_error: 1.9012 - val_loss: 2.1740 -
val_root_mean_squared_error: 1.4084
Epoch 61/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 649us/step - loss: 3.7196 -
root_mean_squared_error: 1.8750 - val_loss: 2.0013 -
val_root_mean_squared_error: 1.3455
Epoch 62/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 494us/step - loss: 4.1047 -
root_mean_squared_error: 1.9763 - val_loss: 2.1549 -
val_root_mean_squared_error: 1.4011
Epoch 63/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 620us/step - loss: 3.3580 -
root_mean_squared_error: 1.7784 - val_loss: 2.1468 -
val_root_mean_squared_error: 1.3987
Epoch 64/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 530us/step - loss: 7.0457 -
root_mean_squared_error: 2.5992 - val_loss: 2.0859 -
val_root_mean_squared_error: 1.3771
Epoch 65/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 486us/step - loss: 3.6114 -
```

```
root_mean_squared_error: 1.8466 - val_loss: 2.1413 -
val_root_mean_squared_error: 1.3967
Epoch 66/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 566us/step - loss: 3.4212 -
root_mean_squared_error: 1.7967 - val_loss: 2.0553 -
val_root_mean_squared_error: 1.3654
Epoch 67/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 526us/step - loss: 3.3783 -
root_mean_squared_error: 1.7838 - val_loss: 2.3945 -
val_root_mean_squared_error: 1.4845
Epoch 68/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 465us/step - loss: 4.0955 -
root_mean_squared_error: 1.9718 - val_loss: 2.1530 -
val_root_mean_squared_error: 1.4013
Epoch 69/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 449us/step - loss: 3.3666 -
root_mean_squared_error: 1.7816 - val_loss: 2.2189 -
val_root_mean_squared_error: 1.4246
Epoch 70/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 636us/step - loss: 3.5752 -
root_mean_squared_error: 1.8386 - val_loss: 2.0686 -
val_root_mean_squared_error: 1.3713
Epoch 71/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 618us/step - loss: 3.4760 -
root_mean_squared_error: 1.8120 - val_loss: 1.9800 -
val_root_mean_squared_error: 1.3388
Epoch 72/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 521us/step - loss: 3.7282 -
root_mean_squared_error: 1.8803 - val_loss: 2.0829 -
val_root_mean_squared_error: 1.3766
Epoch 73/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 581us/step - loss: 3.7452 -
root_mean_squared_error: 1.8846 - val_loss: 2.2345 -
val_root_mean_squared_error: 1.4308
Epoch 74/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 560us/step - loss: 3.3670 -
root_mean_squared_error: 1.7826 - val_loss: 2.2318 -
val_root_mean_squared_error: 1.4294
Epoch 75/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 559us/step - loss: 3.4567 -
root_mean_squared_error: 1.8053 - val_loss: 2.0745 -
val_root_mean_squared_error: 1.3734
Epoch 76/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 609us/step - loss: 3.2465 -
root_mean_squared_error: 1.7478 - val_loss: 4.2491 -
val_root_mean_squared_error: 2.0150
Epoch 77/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 582us/step - loss: 4.1047 -
root_mean_squared_error: 1.9772 - val_loss: 2.0637 -
```

```
val_root_mean_squared_error: 1.3700
Epoch 78/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 444us/step - loss: 3.8366 -
root_mean_squared_error: 1.9075 - val_loss: 1.8591 -
val_root_mean_squared_error: 1.2932
Epoch 79/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 525us/step - loss: 2.9664 -
root_mean_squared_error: 1.6659 - val_loss: 2.2719 -
val_root_mean_squared_error: 1.4442
Epoch 80/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 543us/step - loss: 3.4051 -
root_mean_squared_error: 1.7935 - val_loss: 2.3432 -
val_root_mean_squared_error: 1.4688
Epoch 81/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 511us/step - loss: 3.4519 -
root_mean_squared_error: 1.8066 - val_loss: 2.0411 -
val_root_mean_squared_error: 1.3623
Epoch 82/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 448us/step - loss: 3.2097 -
root_mean_squared_error: 1.7377 - val_loss: 2.3044 -
val_root_mean_squared_error: 1.4559
Epoch 83/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 513us/step - loss: 3.2718 -
root_mean_squared_error: 1.7558 - val_loss: 2.5285 -
val_root_mean_squared_error: 1.5313
Epoch 84/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 556us/step - loss: 3.5506 -
root_mean_squared_error: 1.8337 - val_loss: 2.0890 -
val_root_mean_squared_error: 1.3806
Epoch 85/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 706us/step - loss: 3.4204 -
root_mean_squared_error: 1.7987 - val_loss: 2.0295 -
val_root_mean_squared_error: 1.3591
Epoch 86/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 707us/step - loss: 3.7341 -
root_mean_squared_error: 1.8797 - val_loss: 1.8176 -
val_root_mean_squared_error: 1.2787
Epoch 87/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 476us/step - loss: 2.9706 -
root_mean_squared_error: 1.6673 - val_loss: 2.1372 -
val_root_mean_squared_error: 1.3984
Epoch 88/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 505us/step - loss: 3.2554 -
root_mean_squared_error: 1.7526 - val_loss: 2.4906 -
val_root_mean_squared_error: 1.5199
Epoch 89/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 532us/step - loss: 3.1584 -
root_mean_squared_error: 1.7250 - val_loss: 1.8650 -
val_root_mean_squared_error: 1.2982
```

```
Epoch 90/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 543us/step - loss: 3.5507 -
root_mean_squared_error: 1.8353 - val_loss: 2.1014 -
val_root_mean_squared_error: 1.3865
Epoch 91/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 573us/step - loss: 3.4497 -
root_mean_squared_error: 1.8076 - val_loss: 2.1169 -
val_root_mean_squared_error: 1.3926
Epoch 92/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 545us/step - loss: 3.9054 -
root_mean_squared_error: 1.9280 - val_loss: 1.8116 -
val_root_mean_squared_error: 1.2784
Epoch 93/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 482us/step - loss: 3.2026 -
root_mean_squared_error: 1.7380 - val_loss: 2.2593 -
val_root_mean_squared_error: 1.4431
Epoch 94/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 542us/step - loss: 3.0772 -
root_mean_squared_error: 1.7025 - val_loss: 1.9757 -
val_root_mean_squared_error: 1.3411
Epoch 95/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 536us/step - loss: 3.1807 -
root_mean_squared_error: 1.7292 - val_loss: 2.0874 -
val_root_mean_squared_error: 1.3828
Epoch 96/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 480us/step - loss: 3.4738 -
root_mean_squared_error: 1.8148 - val_loss: 2.0557 -
val_root_mean_squared_error: 1.3714
Epoch 97/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 466us/step - loss: 3.4132 -
root_mean_squared_error: 1.7974 - val_loss: 2.5000 -
val_root_mean_squared_error: 1.5251
Epoch 98/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 552us/step - loss: 3.3225 -
root_mean_squared_error: 1.7735 - val_loss: 2.5838 -
val_root_mean_squared_error: 1.5521
Epoch 99/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 487us/step - loss: 4.0015 -
root_mean_squared_error: 1.9546 - val_loss: 1.8630 -
val_root_mean_squared_error: 1.2993
Epoch 100/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 486us/step - loss: 3.1012 -
root_mean_squared_error: 1.7097 - val_loss: 1.9118 -
val_root_mean_squared_error: 1.3181
Manually chosen best model saved with Epochs=100, Batch Size=32
```

Testing on Best Model

Now from the Optimal Parameters we have obtained we saved our model as a keras file. To test it on the real time data and on test set we are loading it again and then evaluating its performance on test set.

```
# Checking the type and structure of y_test
print(f"Type of y_test: {type(y_test)}")
print(f"Shape of y_test: {y_test.shape}")

# If it's a DataFrame or Series, checking the index
if isinstance(y_test, pd.Series) or isinstance(y_test, pd.DataFrame):
    print(f"y_test Index: {y_test.index}")
```

```
Type of y_test: <class 'pandas.core.series.Series'>
Shape of y_test: (3591,)
y_test Index: Index([ 9834,  5815,  4093,  7476, 16455,  6699,  6700,
3991, 14684,  9194,
       ...
        1520,  4388,  1617,  5339,   491, 16766,  5414,  9990, 10830,
7810],
      dtype='int64', length=3591)
```

```
# If y_test is a Pandas DataFrame or Series, converting it to a NumPy
array
if isinstance(y_test, pd.DataFrame) or isinstance(y_test, pd.Series):
    y_test = y_test.to_numpy().flatten()

# Loading the saved best model
best_model = load_model('best_model_manual_32batch_100epochs.keras')

#  Evaluating the model on the test set
test_loss, test_rmse = best_model.evaluate(X_test, y_test, verbose=0)
print(f"Test MSE: {test_loss:.4f}, Test RMSE: {test_rmse:.4f}")

# Making predictions on the test set
y_pred = best_model.predict(X_test)

# Ensuring y_test is a NumPy array
if isinstance(y_test, pd.DataFrame) or isinstance(y_test, pd.Series):
    y_test = y_test.to_numpy().flatten()  # Convert to NumPy array and
flatten

# Comparing predictions with actual values
# Loop through first 10 predictions
for i in range(10):
    print(f"Predicted: {y_pred[i][0]:.2f}, Actual: {y_test[i]:.2f}")

# calculating the mean absolute error
errors = np.abs(y_test - y_pred.flatten())
print(f"Mean absolute error: {np.mean(errors):.4f}")
```

```
Test MSE: 2.2151, Test RMSE: 1.4285
113/113 ━━━━━━━━━━━━━━━━━━━━ 0s 435us/step
Predicted: 65.06, Actual: 64.00
Predicted: 53.97, Actual: 56.00
Predicted: 59.00, Actual: 59.00
Predicted: 60.43, Actual: 61.00
Predicted: 72.77, Actual: 74.00
Predicted: 59.75, Actual: 59.00
Predicted: 58.54, Actual: 59.00
Predicted: 60.37, Actual: 60.00
Predicted: 70.73, Actual: 71.00
Predicted: 64.16, Actual: 63.00
Mean absolute error: 0.8688
```

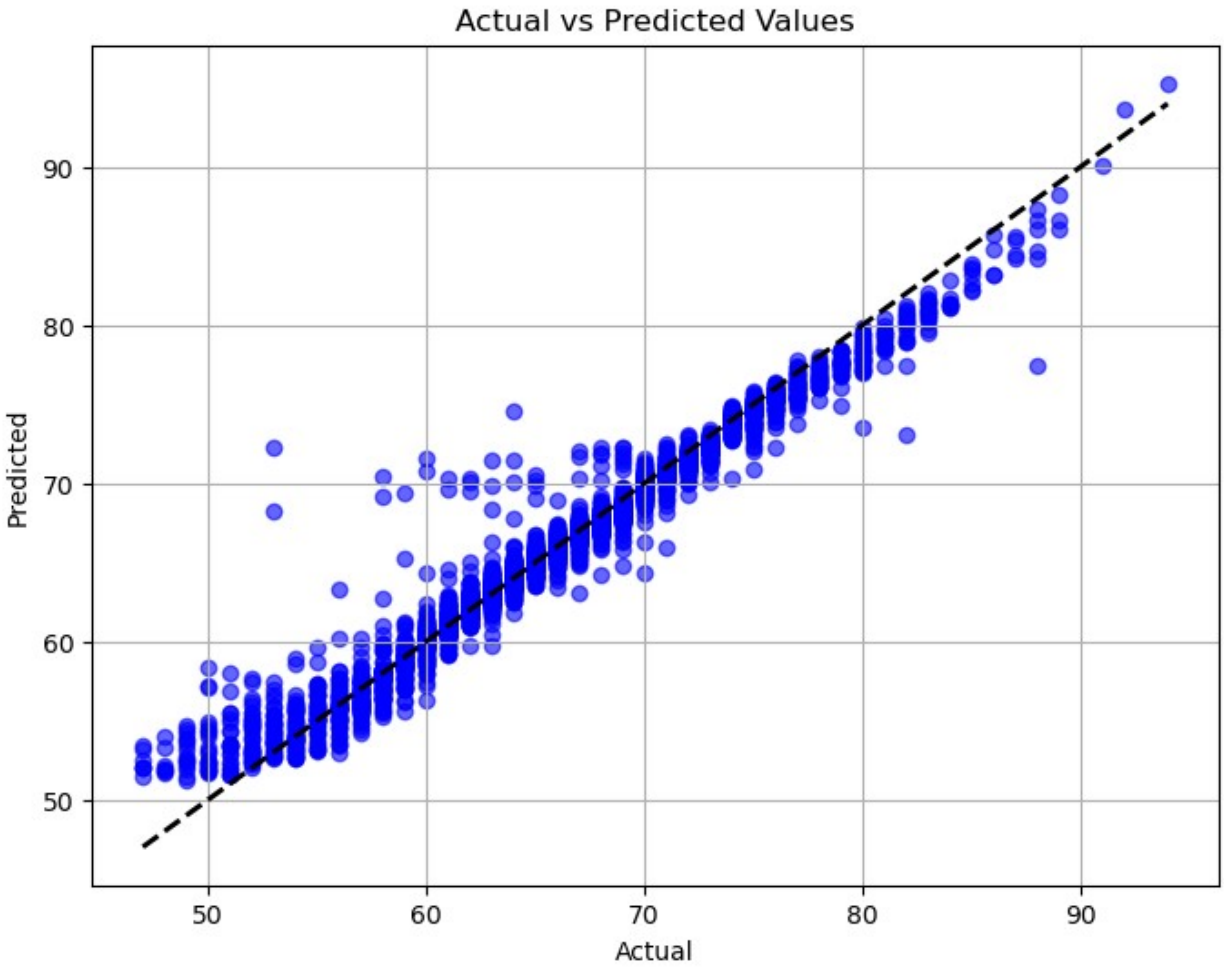
This code loads the best pre-trained Keras model to evaluate its performance on the test set. It calculates the Mean Squared Error (MSE) and Root Mean Squared Error (RMSE), key metrics for assessing the model's predictive accuracy. After making predictions on the test data, it compares the predicted values to the actual ones for a quick validation. Additionally, the Mean Absolute Error (MAE) is computed to provide an intuitive measure of the average prediction error. This process helps ensure that the model generalizes well to unseen data and produces reliable results.

Now to visualize the Testing process we would like to use the following plots

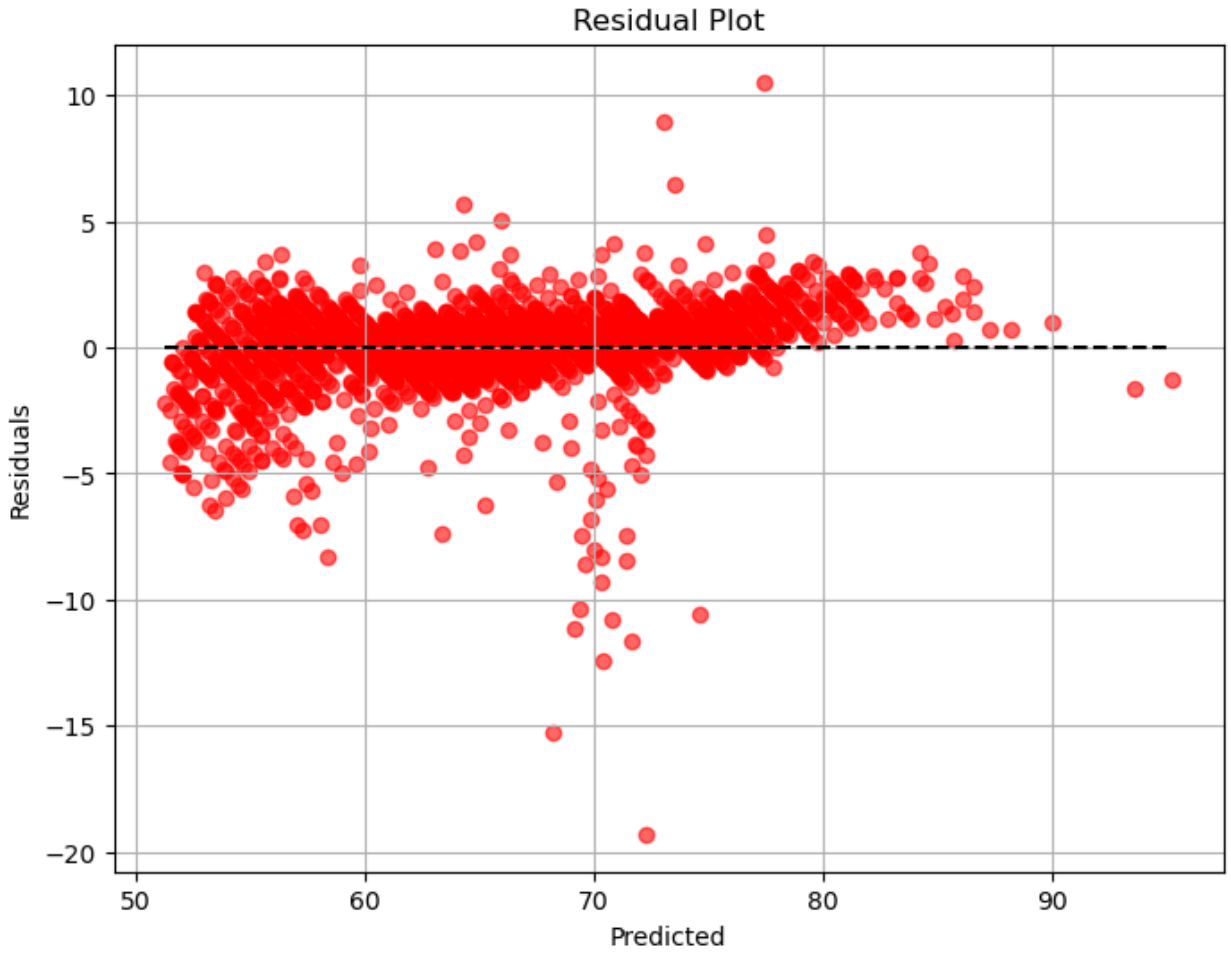
Plot	Definition	Purpose
Scatter Plot (Actual vs. Predicted)	Displays actual values on the x-axis and predicted values on the y-axis.	Visualizes how well predictions align with actual values; points should cluster around the diagonal line.
Residual Plot	Shows predicted values on the x-axis and residuals (actual - predicted) on the y-axis.	Assesses model performance; ideally, residuals should be randomly distributed around zero.
Histogram of Residuals	Displays the distribution of residuals as a histogram.	Evaluates if residuals are normally distributed, indicating unbiased predictions and random errors.

Plot	Purpose in Football Player Analysis
Scatter Plot (Actual vs. Predicted)	Helps visualize how accurately the model predicts player performance; clustering around the diagonal indicates reliable predictions for player contributions.
Residual Plot	Assesses model accuracy by examining prediction errors; helps identify systematic biases in player evaluations, ensuring that no specific player attributes are consistently mispredicted.
Histogram of Residuals	Evaluates whether prediction errors are normally distributed, indicating unbiased assessments of player performance; ensures that the model is fair across different player types and positions, which is crucial for making informed decisions in player evaluations.

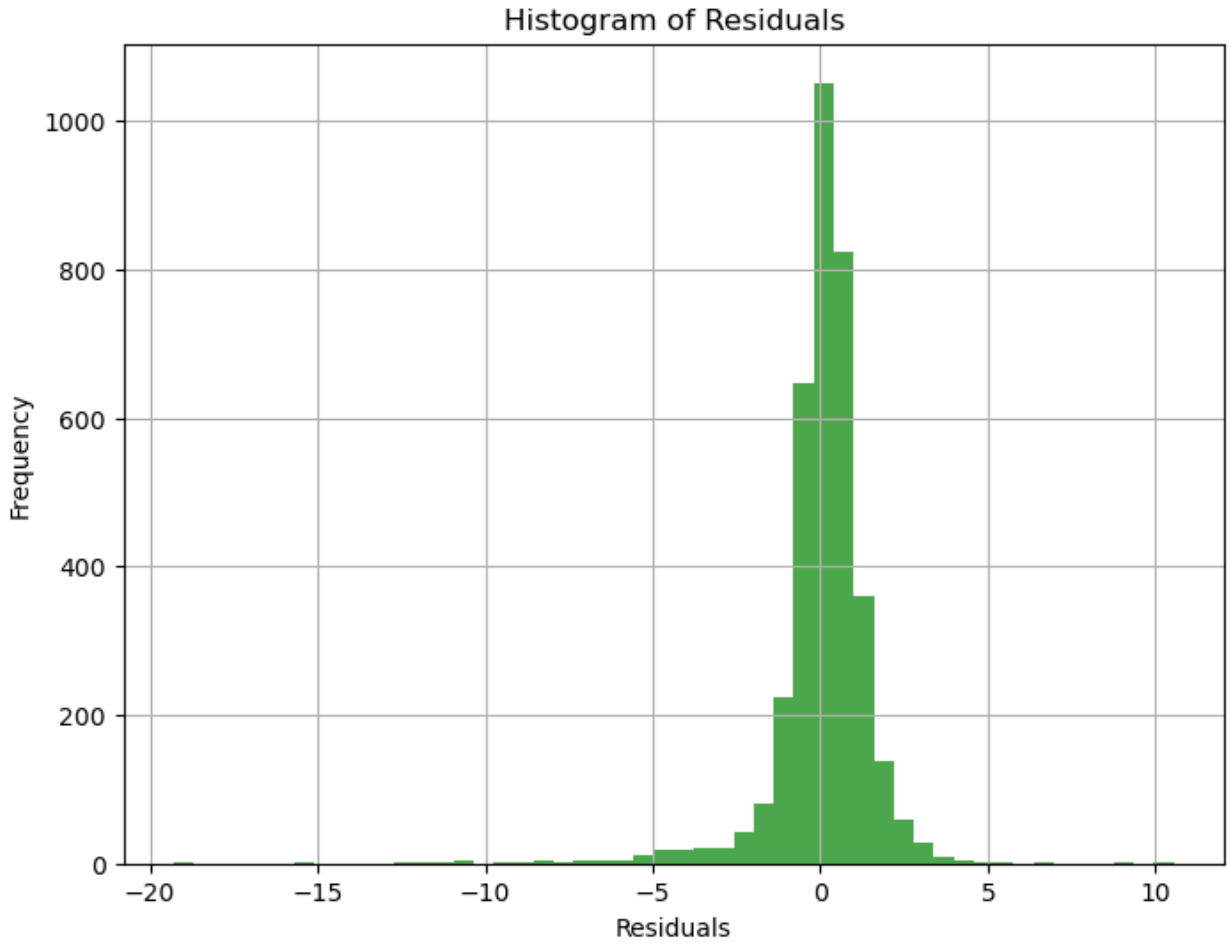
```
# Creating a scatter plot for Actual vs Predicted values
plt.figure(figsize=(8, 6))
plt.scatter(y_test, y_pred, alpha=0.6, color='b')
plt.plot([y_test.min(), y_test.max()], [y_test.min(), y_test.max()],
'k--', lw=2)
plt.xlabel('Actual')
plt.ylabel('Predicted')
plt.title('Actual vs Predicted Values')
plt.grid(True)
plt.show()
```




```
# Creating a residual plot (Actual - Predicted)
residuals = y_test - y_pred.flatten()
plt.figure(figsize=(8, 6))
plt.scatter(y_pred, residuals, alpha=0.6, color='r')
plt.hlines(y=0, xmin=y_pred.min(), xmax=y_pred.max(), colors='k',
linestyles='dashed')
plt.xlabel('Predicted')
plt.ylabel('Residuals')
plt.title('Residual Plot')
plt.grid(True)
plt.show()
```




```
# Histogram of residuals (errors)
plt.figure(figsize=(8, 6))
plt.hist(residuals, bins=50, color='g', alpha=0.7)
plt.title('Histogram of Residuals')
plt.xlabel('Residuals')
plt.ylabel('Frequency')
plt.grid(True)
plt.show()
```



Saving the tested model again.

```
# Saving the finalized model
final_model_name = 'final_model_32batch_100epochs.keras'
best_model.save(final_model_name)
print(f"Final model saved as '{final_model_name}'")
```

```
Final model saved as 'final_model_32batch_100epochs.keras'
```

```
# Saving the scaler
scaler_filename = 'scaler.pkl'
joblib.dump(scaler, scaler_filename)
print(f"Scaler saved as '{scaler_filename}'")
```

```
Scaler saved as 'scaler.pkl'
```

Rechecking if the final model is working and comapring the MSE and RMSE

```
# Loading the saved final model
final_model = load_model(final_model_name)
```

```
# Evaluating the model on the test set to verify
test_loss, test_rmse = final_model.evaluate(X_test, y_test, verbose=0)
print(f"Final Model Test MSE: {test_loss:.4f}, Test RMSE:
{test_rmse:.4f}")

# Making the predictions on the test set and verifying a few values
y_pred_final = final_model.predict(X_test)

# Comparing predictions and actual values
for i in range(10):
    print(f"Predicted: {y_pred_final[i][0]:.2f}, Actual:
{y_test[i]:.2f}")
```

```
Final Model Test MSE: 2.2151, Test RMSE: 1.4285
113/113 ━━━━━━━━━━━━━━━━━━━━ 0s 398us/step
Predicted: 65.06, Actual: 64.00
Predicted: 53.97, Actual: 56.00
Predicted: 59.00, Actual: 59.00
Predicted: 60.43, Actual: 61.00
Predicted: 72.77, Actual: 74.00
Predicted: 59.75, Actual: 59.00
Predicted: 58.54, Actual: 59.00
Predicted: 60.37, Actual: 60.00
Predicted: 70.73, Actual: 71.00
Predicted: 64.16, Actual: 63.00
```

Creating a function on the final model to test any particular instance data from the test set

```
def predict_player_performance(input_data):
    # Loading the final model
    model = load_model('final_model_32batch_100epochs.keras')

    # Make predictions
    prediction = model.predict(input_data)
    return prediction

new_player_data = X_test[:1]
predicted_performance = predict_player_performance(new_player_data)
print(f"Predicted player performance: {predicted_performance[0]
[0]:.2f}")
```

```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 25ms/step
Predicted player performance: 65.06
```

Test on Real-time Data

Through this below function we would like to allow the user to enter any real time data and test if the model is working and if the user is able to get an estimation of the overall rating for the inputs of his choice.

Provides an interactive way for users to input player statistics and receive an estimated performance rating based on a trained model, making it practical for real-time evaluations in football analytics.

```
# Creating a form to take manual inputs
def get_player_input():
    print("Enter the following player details:")
    age = float(input("Enter player's age: "))
    height_cm = float(input("Enter player's height in cm: "))
    weight_kgs = float(input("Enter player's weight in kgs: "))
    value_euro = float(input("Enter player's market value in euros:
"))
    wage_euro = float(input("Enter player's wage in euros: "))
    crossing = float(input("Enter player's crossing ability: "))
    finishing = float(input("Enter player's finishing ability: "))
    dribbling = float(input("Enter player's dribbling ability: "))
    ball_control = float(input("Enter player's ball control: "))
    acceleration = float(input("Enter player's acceleration: "))
    sprint_speed = float(input("Enter player's sprint speed: "))
    stamina = float(input("Enter player's stamina: "))
    strength = float(input("Enter player's strength: "))

    # Creainge an array of inputs that matches the feature structure
used during training
    player_data = np.array([[age, height_cm, weight_kgs, value_euro,
wage_euro, crossing, finishing,
                             dribbling, ball_control, acceleration,
sprint_speed, stamina, strength]])

    return player_data

# Preprocessing the data
def preprocess_input(player_data):

    scaler = joblib.load('scaler.pkl')
    player_data_scaled = scaler.transform(player_data)
    return player_data_scaled

#  Loading the model and making predictions
def predict_player_performance(player_data):
    # Loading the saved model
    model = load_model('final_model_32batch_100epochs.keras')

    # Preprocessing the data
    player_data_scaled = preprocess_input(player_data)

    # Making predictions
    predicted_performance = model.predict(player_data_scaled)

    return predicted_performance
```

```
# Main function to run the form and predict
if __name__ == "__main__":
    # Getting the player input from the form5555
    player_data = get_player_input()

    # Making predictions
    predicted_performance = predict_player_performance(player_data)

    # Printing the prediction result
    print(f"\nPredicted player performance (overall rating):
{predicted_performance[0][0]:.2f}")
```

```
Enter the following player details:

Enter player's age:  26
Enter player's height in cm:  154
Enter player's weight in kgs:  65
Enter player's market value in euros:  56000
Enter player's wage in euros:  5000
Enter player's crossing ability:  54
Enter player's finishing ability:  56
Enter player's dribbling ability:  45
Enter player's ball control:  65
Enter player's acceleration:  30
Enter player's sprint speed:  74
Enter player's stamina:  65
Enter player's strength:  80

1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 25ms/step

Predicted player performance (overall rating): 59.48

/home/unina/anaconda3/lib/python3.12/site-packages/sklearn/
base.py:493: UserWarning: X does not have valid feature names, but
StandardScaler was fitted with feature names
  warnings.warn(
```

Tensorboard

TensorBoard is a visualization tool for monitoring and analyzing machine learning experiments, particularly with TensorFlow and Keras. It is used to track metrics like loss and accuracy during model training, helping users understand model performance and identify potential issues. The dashboard provides insights through visualizations of training progress, model architecture, and weight distributions, enabling better debugging and comparison of different training runs. Overall, TensorBoard enhances the machine learning workflow by facilitating informed decisions for model improvements.

```
#  Setting up TensorBoard logging
log_dir = "logs/fit/" + datetime.datetime.now().strftime("%Y%m%d-%H%M
```

```
%S")
tensorboard_callback = TensorBoard(log_dir=log_dir, histogram_freq=1)

# Building the model
model = Sequential([
    layers.Dense(128, activation='relu',
input_shape=(X_train.shape[1],),
kernel_regularizer=regularizers.l2(0.001)),
    layers.Dropout(0.5),
    layers.Dense(64, activation='relu',
kernel_regularizer=regularizers.l2(0.001)),
    layers.Dense(1)
])

# Compiling the model
model.compile(optimizer=Adam(learning_rate=0.001), loss='mse',
metrics=[tf.keras.metrics.RootMeanSquaredError()])

# Training the model with TensorBoard and EarlyStopping callbacks
early_stopping = EarlyStopping(monitor='val_loss', patience=10,
restore_best_weights=True)

history = model.fit(X_train, y_train,
                    validation_split=0.2,
                    epochs=100,
                    batch_size=32,
                    callbacks=[tensorboard_callback, early_stopping],
                    verbose=1)

# Saving the final model
model.save('final_model_with_tensorboard.keras')
```

```
Epoch 1/100

/home/unina/anaconda3/lib/python3.12/site-packages/keras/src/layers/
core/dense.py:87: UserWarning: Do not pass an
`input_shape`/`input_dim` argument to a layer. When using Sequential
models, prefer using an `Input(shape)` object as the first layer in
the model instead.
  super().__init__(activity_regularizer=activity_regularizer,
**kwargs)

360/360 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step - loss: 2173.0554 -
root_mean_squared_error: 45.1413 - val_loss: 93.4142 -
val_root_mean_squared_error: 9.6546
Epoch 2/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 661us/step - loss: 137.5428 -
root_mean_squared_error: 11.7137 - val_loss: 40.6549 -
val_root_mean_squared_error: 6.3604
Epoch 3/100
```

```
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 692us/step - loss: 91.7652 -
root_mean_squared_error: 9.5687 - val_loss: 26.0804 -
val_root_mean_squared_error: 5.0880
Epoch 4/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 789us/step - loss: 76.2649 -
root_mean_squared_error: 8.7200 - val_loss: 18.2273 -
val_root_mean_squared_error: 4.2478
Epoch 5/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 750us/step - loss: 66.9193 -
root_mean_squared_error: 8.1652 - val_loss: 15.8312 -
val_root_mean_squared_error: 3.9569
Epoch 6/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 718us/step - loss: 58.3942 -
root_mean_squared_error: 7.6291 - val_loss: 13.6612 -
val_root_mean_squared_error: 3.6736
Epoch 7/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 781us/step - loss: 52.8714 -
root_mean_squared_error: 7.2594 - val_loss: 12.1443 -
val_root_mean_squared_error: 3.4621
Epoch 8/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 665us/step - loss: 49.3907 -
root_mean_squared_error: 7.0162 - val_loss: 12.2104 -
val_root_mean_squared_error: 3.4727
Epoch 9/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 741us/step - loss: 45.6784 -
root_mean_squared_error: 6.7462 - val_loss: 11.8280 -
val_root_mean_squared_error: 3.4181
Epoch 10/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 682us/step - loss: 43.5948 -
root_mean_squared_error: 6.5916 - val_loss: 12.1813 -
val_root_mean_squared_error: 3.4704
Epoch 11/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 747us/step - loss: 41.2849 -
root_mean_squared_error: 6.4137 - val_loss: 11.7139 -
val_root_mean_squared_error: 3.4035
Epoch 12/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 744us/step - loss: 39.0436 -
root_mean_squared_error: 6.2359 - val_loss: 11.2279 -
val_root_mean_squared_error: 3.3323
Epoch 13/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 786us/step - loss: 36.3821 -
root_mean_squared_error: 6.0206 - val_loss: 10.7315 -
val_root_mean_squared_error: 3.2577
Epoch 14/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 773us/step - loss: 33.1318 -
root_mean_squared_error: 5.7450 - val_loss: 10.2707 -
val_root_mean_squared_error: 3.1870
Epoch 15/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 765us/step - loss: 29.2800 -
```

```
root_mean_squared_error: 5.4000 - val_loss: 10.7479 -
val_root_mean_squared_error: 3.2617
Epoch 16/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 696us/step - loss: 27.7637 -
root_mean_squared_error: 5.2583 - val_loss: 8.8055 -
val_root_mean_squared_error: 2.9494
Epoch 17/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 758us/step - loss: 23.7445 -
root_mean_squared_error: 4.8599 - val_loss: 8.4679 -
val_root_mean_squared_error: 2.8921
Epoch 18/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 826us/step - loss: 20.2149 -
root_mean_squared_error: 4.4840 - val_loss: 8.1902 -
val_root_mean_squared_error: 2.8442
Epoch 19/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 771us/step - loss: 18.2707 -
root_mean_squared_error: 4.2616 - val_loss: 6.9418 -
val_root_mean_squared_error: 2.6156
Epoch 20/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 709us/step - loss: 15.2152 -
root_mean_squared_error: 3.8872 - val_loss: 8.1244 -
val_root_mean_squared_error: 2.8328
Epoch 21/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 765us/step - loss: 12.5213 -
root_mean_squared_error: 3.5243 - val_loss: 6.6209 -
val_root_mean_squared_error: 2.5539
Epoch 22/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 720us/step - loss: 10.5835 -
root_mean_squared_error: 3.2375 - val_loss: 6.1808 -
val_root_mean_squared_error: 2.4661
Epoch 23/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 775us/step - loss: 9.7570 -
root_mean_squared_error: 3.1073 - val_loss: 6.0121 -
val_root_mean_squared_error: 2.4316
Epoch 24/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 746us/step - loss: 8.8961 -
root_mean_squared_error: 2.9657 - val_loss: 6.3322 -
val_root_mean_squared_error: 2.4965
Epoch 25/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 720us/step - loss: 8.8160 -
root_mean_squared_error: 2.9505 - val_loss: 6.3545 -
val_root_mean_squared_error: 2.5009
Epoch 26/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 680us/step - loss: 7.7304 -
root_mean_squared_error: 2.7615 - val_loss: 5.1930 -
val_root_mean_squared_error: 2.2565
Epoch 27/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 771us/step - loss: 7.3898 -
root_mean_squared_error: 2.6992 - val_loss: 5.6669 -
```

```
val_root_mean_squared_error: 2.3585
Epoch 28/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 773us/step - loss: 6.7654 -
root_mean_squared_error: 2.5805 - val_loss: 5.2946 -
val_root_mean_squared_error: 2.2774
Epoch 29/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 754us/step - loss: 7.4911 -
root_mean_squared_error: 2.7157 - val_loss: 4.1703 -
val_root_mean_squared_error: 2.0143
Epoch 30/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 688us/step - loss: 5.6685 -
root_mean_squared_error: 2.3562 - val_loss: 4.0627 -
val_root_mean_squared_error: 1.9865
Epoch 31/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 830us/step - loss: 5.7115 -
root_mean_squared_error: 2.3648 - val_loss: 4.5213 -
val_root_mean_squared_error: 2.0976
Epoch 32/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 766us/step - loss: 5.4096 -
root_mean_squared_error: 2.2981 - val_loss: 3.5304 -
val_root_mean_squared_error: 1.8448
Epoch 33/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 675us/step - loss: 5.1829 -
root_mean_squared_error: 2.2475 - val_loss: 4.2232 -
val_root_mean_squared_error: 2.0225
Epoch 34/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 746us/step - loss: 5.0839 -
root_mean_squared_error: 2.2237 - val_loss: 3.2518 -
val_root_mean_squared_error: 1.7645
Epoch 35/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 767us/step - loss: 5.3275 -
root_mean_squared_error: 2.2751 - val_loss: 3.4149 -
val_root_mean_squared_error: 1.8086
Epoch 36/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 717us/step - loss: 4.5108 -
root_mean_squared_error: 2.0884 - val_loss: 2.8233 -
val_root_mean_squared_error: 1.6347
Epoch 37/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 739us/step - loss: 4.8948 -
root_mean_squared_error: 2.1773 - val_loss: 3.0058 -
val_root_mean_squared_error: 1.6882
Epoch 38/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 719us/step - loss: 4.1941 -
root_mean_squared_error: 2.0083 - val_loss: 2.6735 -
val_root_mean_squared_error: 1.5854
Epoch 39/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 832us/step - loss: 4.4142 -
root_mean_squared_error: 2.0612 - val_loss: 2.5770 -
val_root_mean_squared_error: 1.5532
```

```
Epoch 40/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 800us/step - loss: 3.8645 -
root_mean_squared_error: 1.9229 - val_loss: 3.0111 -
val_root_mean_squared_error: 1.6859
Epoch 41/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 736us/step - loss: 4.7948 -
root_mean_squared_error: 2.1407 - val_loss: 2.4978 -
val_root_mean_squared_error: 1.5254
Epoch 42/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 679us/step - loss: 5.3005 -
root_mean_squared_error: 2.2556 - val_loss: 2.4836 -
val_root_mean_squared_error: 1.5197
Epoch 43/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 721us/step - loss: 4.0458 -
root_mean_squared_error: 1.9658 - val_loss: 3.7453 -
val_root_mean_squared_error: 1.8887
Epoch 44/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 834us/step - loss: 4.0251 -
root_mean_squared_error: 1.9572 - val_loss: 2.2741 -
val_root_mean_squared_error: 1.4469
Epoch 45/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 729us/step - loss: 3.7291 -
root_mean_squared_error: 1.8790 - val_loss: 2.6084 -
val_root_mean_squared_error: 1.5571
Epoch 46/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 646us/step - loss: 3.9028 -
root_mean_squared_error: 1.9279 - val_loss: 2.8149 -
val_root_mean_squared_error: 1.6211
Epoch 47/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 687us/step - loss: 3.7050 -
root_mean_squared_error: 1.8748 - val_loss: 2.1061 -
val_root_mean_squared_error: 1.3846
Epoch 48/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 750us/step - loss: 3.1613 -
root_mean_squared_error: 1.7194 - val_loss: 2.3034 -
val_root_mean_squared_error: 1.4537
Epoch 49/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 850us/step - loss: 3.8536 -
root_mean_squared_error: 1.9127 - val_loss: 2.4614 -
val_root_mean_squared_error: 1.5068
Epoch 50/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 812us/step - loss: 4.0702 -
root_mean_squared_error: 1.9685 - val_loss: 2.3070 -
val_root_mean_squared_error: 1.4539
Epoch 51/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 745us/step - loss: 3.6155 -
root_mean_squared_error: 1.8493 - val_loss: 2.5642 -
val_root_mean_squared_error: 1.5395
Epoch 52/100
```

```
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 688us/step - loss: 4.1636 -
root_mean_squared_error: 1.9892 - val_loss: 2.1895 -
val_root_mean_squared_error: 1.4124
Epoch 53/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 763us/step - loss: 3.5370 -
root_mean_squared_error: 1.8261 - val_loss: 2.3145 -
val_root_mean_squared_error: 1.4557
Epoch 54/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 778us/step - loss: 3.6729 -
root_mean_squared_error: 1.8637 - val_loss: 2.7906 -
val_root_mean_squared_error: 1.6108
Epoch 55/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 775us/step - loss: 4.4866 -
root_mean_squared_error: 2.0670 - val_loss: 2.4519 -
val_root_mean_squared_error: 1.5014
Epoch 56/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 743us/step - loss: 3.8338 -
root_mean_squared_error: 1.9051 - val_loss: 2.2738 -
val_root_mean_squared_error: 1.4407
Epoch 57/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 691us/step - loss: 3.7599 -
root_mean_squared_error: 1.8849 - val_loss: 1.8571 -
val_root_mean_squared_error: 1.2881
Epoch 58/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 792us/step - loss: 3.3033 -
root_mean_squared_error: 1.7597 - val_loss: 2.2106 -
val_root_mean_squared_error: 1.4185
Epoch 59/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 779us/step - loss: 3.7329 -
root_mean_squared_error: 1.8792 - val_loss: 2.0802 -
val_root_mean_squared_error: 1.3719
Epoch 60/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 710us/step - loss: 4.0686 -
root_mean_squared_error: 1.9646 - val_loss: 3.1532 -
val_root_mean_squared_error: 1.7190
Epoch 61/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 701us/step - loss: 3.1860 -
root_mean_squared_error: 1.7276 - val_loss: 1.8137 -
val_root_mean_squared_error: 1.2710
Epoch 62/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 821us/step - loss: 3.3237 -
root_mean_squared_error: 1.7658 - val_loss: 1.9175 -
val_root_mean_squared_error: 1.3113
Epoch 63/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 766us/step - loss: 3.3394 -
root_mean_squared_error: 1.7691 - val_loss: 1.9059 -
val_root_mean_squared_error: 1.3067
Epoch 64/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 824us/step - loss: 3.5178 -
```

```
root_mean_squared_error: 1.8209 - val_loss: 2.1483 -
val_root_mean_squared_error: 1.3967
Epoch 65/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 752us/step - loss: 3.5530 -
root_mean_squared_error: 1.8297 - val_loss: 2.4657 -
val_root_mean_squared_error: 1.5059
Epoch 66/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 729us/step - loss: 3.3268 -
root_mean_squared_error: 1.7679 - val_loss: 2.7899 -
val_root_mean_squared_error: 1.6101
Epoch 67/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 785us/step - loss: 4.8876 -
root_mean_squared_error: 2.1607 - val_loss: 1.9585 -
val_root_mean_squared_error: 1.3275
Epoch 68/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 800us/step - loss: 3.7522 -
root_mean_squared_error: 1.8837 - val_loss: 1.8787 -
val_root_mean_squared_error: 1.2967
Epoch 69/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 748us/step - loss: 3.2631 -
root_mean_squared_error: 1.7483 - val_loss: 2.7270 -
val_root_mean_squared_error: 1.5908
Epoch 70/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 720us/step - loss: 4.4140 -
root_mean_squared_error: 2.0365 - val_loss: 2.2689 -
val_root_mean_squared_error: 1.4400
Epoch 71/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 678us/step - loss: 3.4915 -
root_mean_squared_error: 1.8128 - val_loss: 1.9610 -
val_root_mean_squared_error: 1.3290
```

```
# Launching the TensorBoard
#%tensorboard --logdir logs/fit/
```

Achievements

- Successfully cleaned and preprocessed the data.
- Implemented different deep learning models in PyTorch and Keras to predict player performance.
- Identified optimal hyperparameters by comparing models with different epochs, batch sizes, and configurations.
- Evaluated models based on MSE and RMSE.
- Plotted training vs validation loss curves to monitor the models' learning process.
- Produced an optimal regression model for predicting player overall rating using numerical features.

Classification

Classification refers to the task of predicting discrete labels or categories by learning patterns from input data. The network assigns input samples to predefined classes, typically using a softmax or sigmoid activation in the output layer for multi-class or binary classification, respectively.

> Primary Objective: Classify players into performance categories (low, medium, high) based on their attributes.

Libraries Required

Library	Description	Usage in Code
TensorFlow (tensorflow)	Open-source library for deep learning and machine learning tasks.	Used to build neural networks, define layers, and train models with functions such as Input, Dense, Embedding, and Model.
Keras (from TensorFlow)	High-level API within TensorFlow to simplify building neural networks.	Used for defining models, layers, utilities, and callbacks. Functions like Dense, Embedding, Model, and EarlyStopping are part of Keras.
Pandas (pd)	Data manipulation and analysis library.	Used for loading, manipulating, and analyzing data in tabular format with DataFrame.
NumPy (np)	Library for numerical computing and array manipulation.	Helps in efficient handling of numerical data, used for data manipulation (e.g., train_test_split).
scikit-learn (sklearn)	Machine learning library for data preprocessing, model selection, etc.	Used for data preprocessing (e.g., train_test_split, StandardScaler, OneHotEncoder), and evaluating models.
Matplotlib (plt)	Plotting and data visualization library.	Used for visualizing data, losses, and performance metrics, e.g., confusion matrix or training/validation plots.
EarlyStopping (from Keras)	Callback to stop training early if a monitored metric stops improving.	Used to prevent overfitting by stopping training when validation performance stops improving.
ConfusionMatrixDisplay	Visualization utility for confusion matrices (from sklearn.metrics).	Used to visualize confusion matrix to evaluate classification model performance.
train_test_split (from sklearn)	Function to split datasets into training and testing sets.	Splits the dataset into training and test sets for model training and evaluation.
StandardScaler (from sklearn)	Standardizes features by removing the mean and scaling to unit variance.	Used for scaling continuous numerical features before training a model.
OneHotEncoder (from sklearn)	Converts categorical variables into one-hot encoded format.	Used to convert categorical labels into a numerical format suitable for neural network training.
LabelEncoder (from sklearn)	Encodes target labels with value between 0 and n_classes-1.	Converts categorical labels to integer values, useful for preparing labels for neural networks (especially for classification tasks).
plot_model (from Keras)	Visualizes the architecture of the neural network model.	Generates a graphical visualization of the model structure for better understanding of the architecture.
to_categorical (from Keras)	Converts class vectors to binary class matrices.	Used to one-hot encode labels for multi-class classification, ensuring labels are in the correct format for training a classification neural network model.

```
!pip install tensorflow
```

```
Requirement already satisfied: tensorflow in
/home/unina/anaconda3/lib/python3.12/site-packages (2.17.0)
Requirement already satisfied: absl-py>=1.0.0 in
/home/unina/anaconda3/lib/python3.12/site-packages (from tensorflow)
(2.1.0)
Requirement already satisfied: astunparse>=1.6.0 in
/home/unina/anaconda3/lib/python3.12/site-packages (from tensorflow)
(1.6.3)
Requirement already satisfied: flatbuffers>=24.3.25 in
/home/unina/anaconda3/lib/python3.12/site-packages (from tensorflow)
(24.3.25)
Requirement already satisfied: gast!=0.5.0,!=0.5.1,!=0.5.2,>=0.2.1
in /home/unina/anaconda3/lib/python3.12/site-packages (from
tensorflow) (0.6.0)
Requirement already satisfied: google-pasta>=0.1.1 in
/home/unina/anaconda3/lib/python3.12/site-packages (from tensorflow)
```

```
(0.2.0)
Requirement already satisfied: h5py>=3.10.0 in
/home/unina/anaconda3/lib/python3.12/site-packages (from tensorflow)
(3.11.0)
Requirement already satisfied: libclang>=13.0.0 in
/home/unina/anaconda3/lib/python3.12/site-packages (from tensorflow)
(18.1.1)
Requirement already satisfied: ml-dtypes<0.5.0,>=0.3.1 in
/home/unina/anaconda3/lib/python3.12/site-packages (from tensorflow)
(0.4.0)
Requirement already satisfied: opt-einsum>=2.3.2 in
/home/unina/anaconda3/lib/python3.12/site-packages (from tensorflow)
(3.3.0)
Requirement already satisfied: packaging in
/home/unina/anaconda3/lib/python3.12/site-packages (from tensorflow)
(23.2)
Requirement already satisfied: protobuf!=4.21.0,!=4.21.1,!=4.21.2,!
=4.21.3,!=4.21.4,!=4.21.5,<5.0.0dev,>=3.20.3 in
/home/unina/anaconda3/lib/python3.12/site-packages (from tensorflow)
(3.20.3)
Requirement already satisfied: requests<3,>=2.21.0 in
/home/unina/anaconda3/lib/python3.12/site-packages (from tensorflow)
(2.32.2)
Requirement already satisfied: setuptools in
/home/unina/anaconda3/lib/python3.12/site-packages (from tensorflow)
(69.5.1)
Requirement already satisfied: six>=1.12.0 in
/home/unina/anaconda3/lib/python3.12/site-packages (from tensorflow)
(1.16.0)
Requirement already satisfied: termcolor>=1.1.0 in
/home/unina/anaconda3/lib/python3.12/site-packages (from tensorflow)
(2.4.0)
Requirement already satisfied: typing-extensions>=3.6.6 in
/home/unina/anaconda3/lib/python3.12/site-packages (from tensorflow)
(4.11.0)
Requirement already satisfied: wrapt>=1.11.0 in
/home/unina/anaconda3/lib/python3.12/site-packages (from tensorflow)
(1.14.1)
Requirement already satisfied: grpcio<2.0,>=1.24.3 in
/home/unina/anaconda3/lib/python3.12/site-packages (from tensorflow)
(1.66.1)
Requirement already satisfied: tensorboard<2.18,>=2.17 in
/home/unina/anaconda3/lib/python3.12/site-packages (from tensorflow)
(2.17.1)
Requirement already satisfied: keras>=3.2.0 in
/home/unina/anaconda3/lib/python3.12/site-packages (from tensorflow)
(3.5.0)
Requirement already satisfied: numpy<2.0.0,>=1.26.0 in
/home/unina/anaconda3/lib/python3.12/site-packages (from tensorflow)
```

```
(1.26.4)
Requirement already satisfied: wheel<1.0,>=0.23.0 in
/home/unina/anaconda3/lib/python3.12/site-packages (from
astunparse>=1.6.0->tensorflow) (0.43.0)
Requirement already satisfied: rich in
/home/unina/anaconda3/lib/python3.12/site-packages (from keras>=3.2.0-
>tensorflow) (13.3.5)
Requirement already satisfied: namex in
/home/unina/anaconda3/lib/python3.12/site-packages (from keras>=3.2.0-
>tensorflow) (0.0.8)
Requirement already satisfied: optree in
/home/unina/anaconda3/lib/python3.12/site-packages (from keras>=3.2.0-
>tensorflow) (0.12.1)
Requirement already satisfied: charset-normalizer<4,>=2 in
/home/unina/anaconda3/lib/python3.12/site-packages (from
requests<3,>=2.21.0->tensorflow) (2.0.4)
Requirement already satisfied: idna<4,>=2.5 in
/home/unina/anaconda3/lib/python3.12/site-packages (from
requests<3,>=2.21.0->tensorflow) (3.7)
Requirement already satisfied: urllib3<3,>=1.21.1 in
/home/unina/anaconda3/lib/python3.12/site-packages (from
requests<3,>=2.21.0->tensorflow) (2.2.2)
Requirement already satisfied: certifi>=2017.4.17 in
/home/unina/anaconda3/lib/python3.12/site-packages (from
requests<3,>=2.21.0->tensorflow) (2024.8.30)
Requirement already satisfied: markdown>=2.6.8 in
/home/unina/anaconda3/lib/python3.12/site-packages (from
tensorboard<2.18,>=2.17->tensorflow) (3.4.1)
Requirement already satisfied: tensorboard-data-server<0.8.0,>=0.7.0
in /home/unina/anaconda3/lib/python3.12/site-packages (from
tensorboard<2.18,>=2.17->tensorflow) (0.7.2)
Requirement already satisfied: werkzeug>=1.0.1 in
/home/unina/anaconda3/lib/python3.12/site-packages (from
tensorboard<2.18,>=2.17->tensorflow) (3.0.3)
Requirement already satisfied: MarkupSafe>=2.1.1 in
/home/unina/anaconda3/lib/python3.12/site-packages (from
werkzeug>=1.0.1->tensorboard<2.18,>=2.17->tensorflow) (2.1.3)
Requirement already satisfied: markdown-it-py<3.0.0,>=2.2.0 in
/home/unina/anaconda3/lib/python3.12/site-packages (from rich-
>keras>=3.2.0->tensorflow) (2.2.0)
Requirement already satisfied: pygments<3.0.0,>=2.13.0 in
/home/unina/anaconda3/lib/python3.12/site-packages (from rich-
>keras>=3.2.0->tensorflow) (2.15.1)
Requirement already satisfied: mdurl~=0.1 in
/home/unina/anaconda3/lib/python3.12/site-packages (from markdown-it-
py<3.0.0,>=2.2.0->rich->keras>=3.2.0->tensorflow) (0.1.0)

# Importing the necessary libraries
import tensorflow as tf
from tensorflow.keras import layers, models
```

```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler, OneHotEncoder,
LabelEncoder
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Dense, Input, Embedding,
Concatenate, Flatten
from tensorflow.keras.utils import to_categorical
from sklearn.preprocessing import LabelEncoder

from tensorflow.keras.utils import to_categorical
from tensorflow.keras.callbacks import EarlyStopping


import matplotlib.pyplot as plt
from tensorflow.keras.utils import plot_model
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay
import os

os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3'

df_classification =
pd.read_csv('CSV_Files/Player_Attributes_with_Names.csv')
df_classification.head()
```

```
           name                         full_name birth_date  age
height_cm  \
0      L. Messi  Lionel Andrés Messi Cuccittini  6/24/1987   31
170.18
1    C. Eriksen    Christian  Dannemann Eriksen  2/14/1992   27
154.94
2      P. Pogba                      Paul Pogba  3/15/1993   25
190.50
3    L. Insigne                 Lorenzo Insigne   6/4/1991   27
162.56
4  K. Koulibaly               Kalidou Koulibaly  6/20/1991   27
187.96

   weight_kgs  positions nationality  overall_rating
potential  ...  \
0        72.1   CF,RW,ST   Argentina              94         94  ...

1        76.2  CAM,RM,CM     Denmark              88         89  ...

2        83.9     CM,CAM      France              88         91  ...

3        59.0      LW,ST       Italy              88         88  ...

4        88.9         CB     Senegal              88         91  ...
```

```
   standing_tackle  sliding_tackle age_category  height_m
BMI  \
0               28              26          30+    1.7018  24.895349

1               57              22        25-30    1.5494  31.741531

2               67              67        20-25    1.9050  23.119157

3               24              22        25-30    1.6256  22.326705

4               88              87        25-30    1.8796  25.163491


    BMI_category offensive_score  defensive_score  offensive_category
\
0  Normal weight             828              227                 NaN

1        Obesity             777              292                High

2  Normal weight             760              414                High

3  Normal weight             747              213                High

4     Overweight             382              524              Medium


   defensive_category
0              Medium
1              Medium
2              Medium
3              Medium
4                High

[5 rows x 54 columns]
```

For Classification task we would like to use the same overall rating column in as a categorical variable that we had feature engineered in Module A as performance_category

It is a multi class that has been categorized as this

```
    if overall_rating < 65:
        return 'Low'
    elif 65 <= overall_rating < 80:
        return 'Medium'
    else:
        return 'High'

# Feature Selection for Classification Task
def categorize_rating(overall_rating):
```

```
    if overall_rating < 65:
        return 'Low'
    elif 65 <= overall_rating < 80:
        return 'Medium'
    else:
        return 'High'

# Creating a new column for performance categories
df_classification['performance_category'] =
df_classification['overall_rating'].apply(categorize_rating)


df_classification.dtypes
```

```
name                                object
full_name                           object
birth_date                          object
age                                  int64
height_cm                          float64
weight_kgs                         float64
positions                           object
nationality                         object
overall_rating                       int64
potential                            int64
value_euro                         float64
wage_euro                          float64
preferred_foot                      object
international_reputation(1-5)        int64
weak_foot(1-5)                       int64
skill_moves(1-5)                     int64
body_type                           object
crossing                             int64
finishing                            int64
heading_accuracy                     int64
short_passing                        int64
volleys                              int64
dribbling                            int64
curve                                int64
freekick_accuracy                    int64
long_passing                         int64
ball_control                         int64
acceleration                         int64
sprint_speed                         int64
agility                              int64
reactions                            int64
balance                              int64
shot_power                           int64
jumping                              int64
stamina                              int64
strength                             int64
```

```
long_shots                  int64
aggression                  int64
interceptions               int64
positioning                 int64
vision                      int64
penalties                   int64
composure                   int64
marking                     int64
standing_tackle             int64
sliding_tackle              int64
age_category               object
height_m                  float64
BMI                       float64
BMI_category               object
offensive_score             int64
defensive_score             int64
offensive_category         object
defensive_category         object
performance_category       object
dtype: object
```

Feature Selection

Why are we choosing performance category?

We chose **`performance_category`** as the target variable for classification because it was derived through **feature engineering**. Since none of the existing columns alone were valuable enough to be directly used for classification, we created this target based on **`overall_rating`**, which we had previously used in a regression task. In the regression, we predicted the continuous value of `overall_rating` to assess player performance. However, for the classification task, we needed a discrete variable, so we transformed `overall_rating` into **`performance_category`**, categorizing players into "low," "medium," and "high" performance levels. This approach allowed us to build a classification model by converting the continuous `overall_rating` into a more suitable categorical form, enabling the model to predict performance levels based on the selected numerical and categorical features. For certain categorical variables in other tables, there was high class imbalance, making them less reliable for direct use in classification.

```
# Selecting features (numerical and categorical)
numerical_features = ['age', 'height_cm', 'weight_kgs', 'potential',
'value_euro', 'wage_euro',
                      'crossing', 'finishing', 'heading_accuracy',
'short_passing', 'dribbling',
                      'acceleration', 'sprint_speed', 'stamina',
'strength','freekick_accuracy',
                      'agility','shot_power']

categorical_features = ['preferred_foot', 'body_type',
```

```
'defensive_category', 'offensive_category']

# Preparing the data
X = df_classification[numerical_features + categorical_features]
y = df_classification['performance_category']
```

Cost and Loss Functions

For classification task we wanted to compare the different loss functions that allow us to perform multi-class classification. Below is the table for the suitable loss functions that can be applied to our case of classifying the performance category:

Activation Function	Suitable For	Advantages	Loss Function
Sigmoid	Binary Classification	- Outputs probabilities between 0 and 1.	Binary Cross-Entropy
SoftMax	Multi-Class Classification	- Outputs a probability distribution where all probabilities sum to one.	Categorical Cross-Entropy
		- Good for mutually exclusive classes.	
		- Useful for one-hot encoded labels.	Sparse Categorical Cross-Entropy
		- Clear handling of probabilities for each class.	

We did not opt for binary cross entropy since we didn't have a categorical variable of binary output and even if we feature engineered most of the columns show class imbalance. For example below is the table showing class imbalance in the categories of Team Attributes Table.

	buildUpPlayPositioningClass
Free Form	1387
Organized	73
	defenceDefenderLineClass
cover	1363
offside trap	97

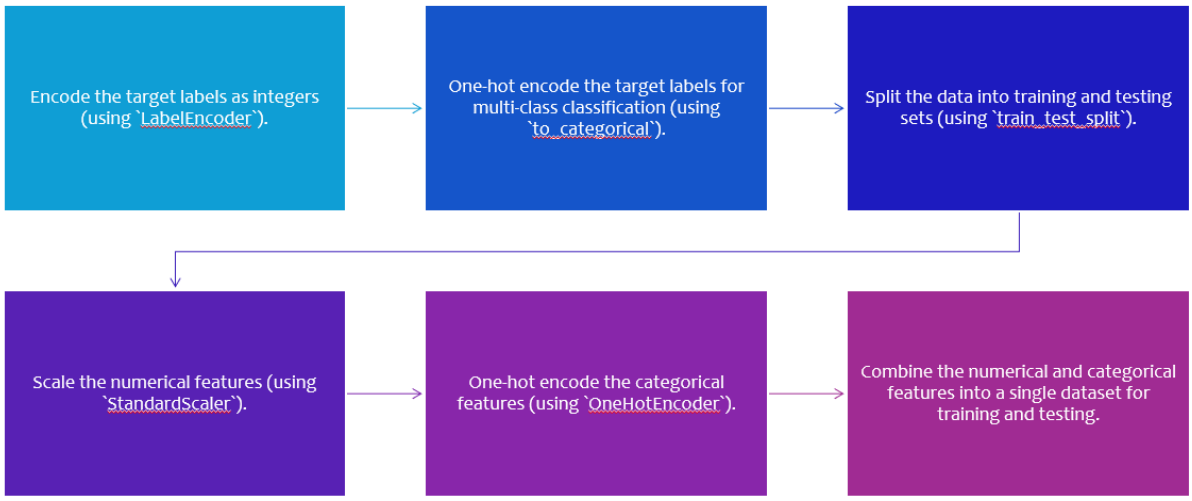
Categorical Crossentropy

Categorical cross-entropy is a loss function commonly used in multi-class classification tasks, where each instance can belong to one of several classes. It measures the dissimilarity between the predicted probability distribution (output from the model) and the true probability distribution (the actual class labels).

For multi-class classification tasks, categorical cross-entropy is a better choice, as it computes the loss across multiple classes and helps the model learn to classify the data into one of the several categories. This is why we would use categorical cross-entropy rather than binary cross-entropy in this scenario.

Preprocessing

What steps are involved in preprocessing for Categorical cross entropy?



```
# Encoding the target labels
label_encoder = LabelEncoder()
y_encoded = label_encoder.fit_transform(y)
y_one_hot = to_categorical(y_encoded)

# Splitting the data into train and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y_one_hot,
test_size=0.2, random_state=42)

# Scaling the numerical features
scaler = StandardScaler()
X_train_numerical = scaler.fit_transform(X_train[numerical_features])
X_test_numerical = scaler.transform(X_test[numerical_features])

# One-hot encode categorical features
encoder = OneHotEncoder(sparse_output=False, handle_unknown='ignore')
# Use sparse_output instead of sparse
X_train_categorical =
encoder.fit_transform(X_train[categorical_features])
X_test_categorical = encoder.transform(X_test[categorical_features])

# Combining the numerical and categorical features
X_train_combined = np.hstack([X_train_numerical, X_train_categorical])
X_test_combined = np.hstack([X_test_numerical, X_test_categorical])
```

Training

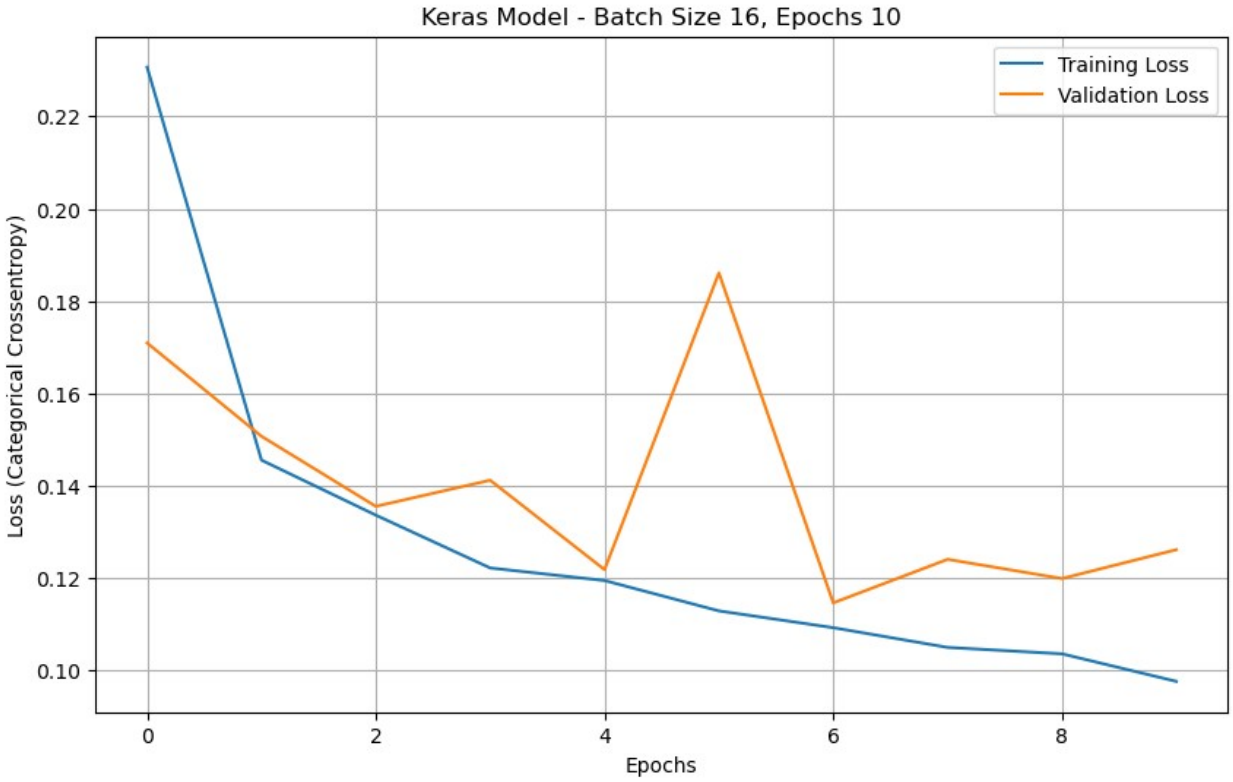
The following function trains a **Keras classification model** using different configurations of **batch sizes** and **epochs** to identify the optimal hyperparameters for classifying players into multiple categories. The model consists of two dense layers with ReLU activation and a final output layer with softmax activation for multi-class classification. The model is compiled using the **Adam optimizer** and **categorical crossentropy** as the loss function. During training, the function iterates through various combinations of batch sizes and epochs, storing and printing the **training loss**, **accuracy**, **validation loss**, and **validation accuracy** for each epoch. After training, it generates loss and accuracy curves to visually track the performance of the model over time. This helps in evaluating and comparing how the model performs under different configurations.

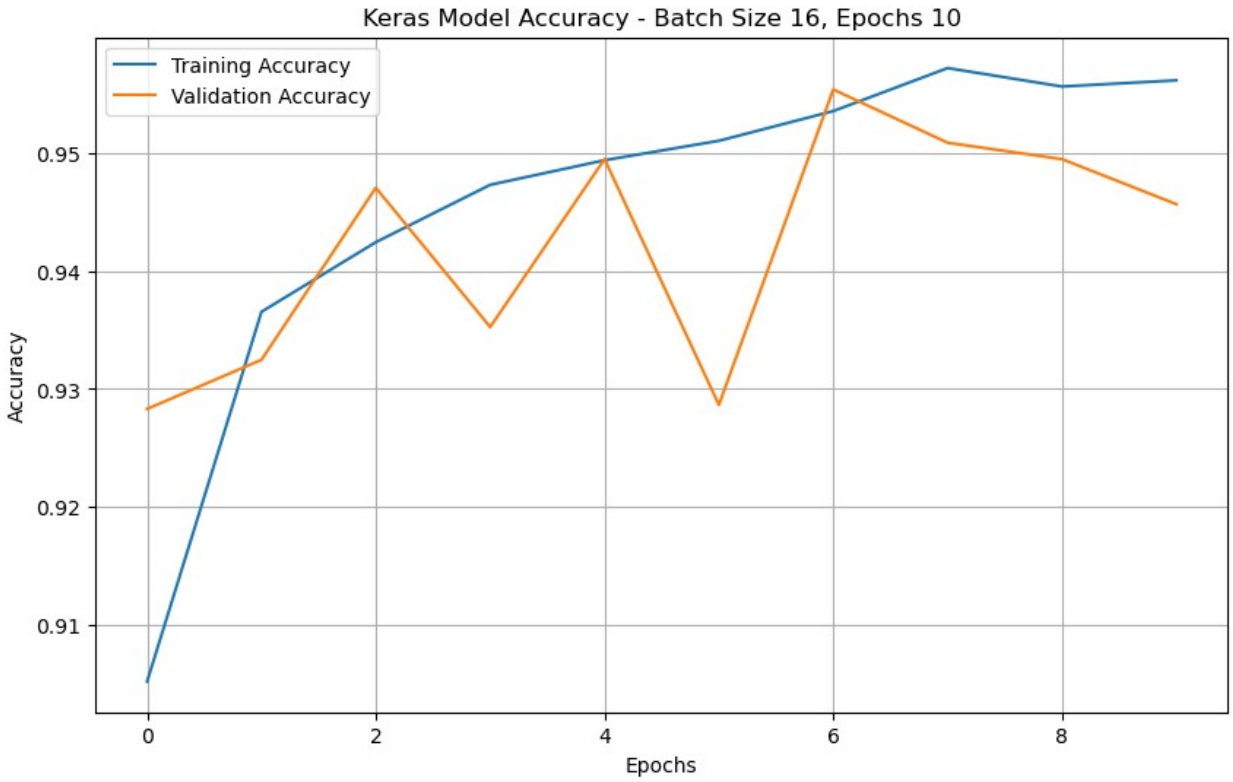
```
# Function to train Keras model for classification with batch size and
epochs iteration
def train_keras_classifier(X_train, y_train, X_test, y_test,
epochs_list, batch_size_list):
    results = []

    for batch_size in batch_size_list:
        for epochs in epochs_list:
            print(f"\nTraining with Batch Size: {batch_size}, Epochs:
{epochs}")
```

```
            # Building a simple classification model
            model = models.Sequential([
                layers.Dense(128, activation='relu',
input_shape=(X_train.shape[1],)),
                layers.Dense(64, activation='relu'),
                layers.Dense(3, activation='softmax')
            ])

            # Compiling the model
            model.compile(optimizer='adam',
loss='categorical_crossentropy', metrics=['accuracy'])

            # Training the model
            history = model.fit(X_train, y_train,
validation_split=0.2,
                                epochs=epochs, batch_size=batch_size,
verbose=0)

            # Printing loss and accuracy at the end of each epoch
            for epoch in range(epochs):
                if (epoch + 1) % 10 == 0 or epoch == epochs - 1:
                    avg_loss = history.history['loss'][epoch]
                    avg_accuracy = history.history['accuracy'][epoch]
                    val_loss = history.history['val_loss'][epoch]
                    val_accuracy = history.history['val_accuracy']
[epoch]
                    print(f"Epoch [{epoch + 1}/{epochs}], Training
Loss: {avg_loss:.4f}, "
                          f"Training Accuracy: {avg_accuracy:.4f},
Validation Loss: {val_loss:.4f}, "
                          f"Validation Accuracy: {val_accuracy:.4f}")

            # Storing results
            results.append({
                'Epochs': epochs,
                'Batch Size': batch_size,
                'Training Loss': avg_loss,
                'Training Accuracy': avg_accuracy,
                'Validation Loss': val_loss,
                'Validation Accuracy': val_accuracy
            })

            # Plotting loss curves
            plt.figure(figsize=(10, 6))
            plt.plot(history.history['loss'], label='Training Loss')
            plt.plot(history.history['val_loss'], label='Validation
Loss')
            plt.xlabel('Epochs')
            plt.ylabel('Loss (Categorical Crossentropy)')
            plt.title(f'Keras Model - Batch Size {batch_size}, Epochs
```

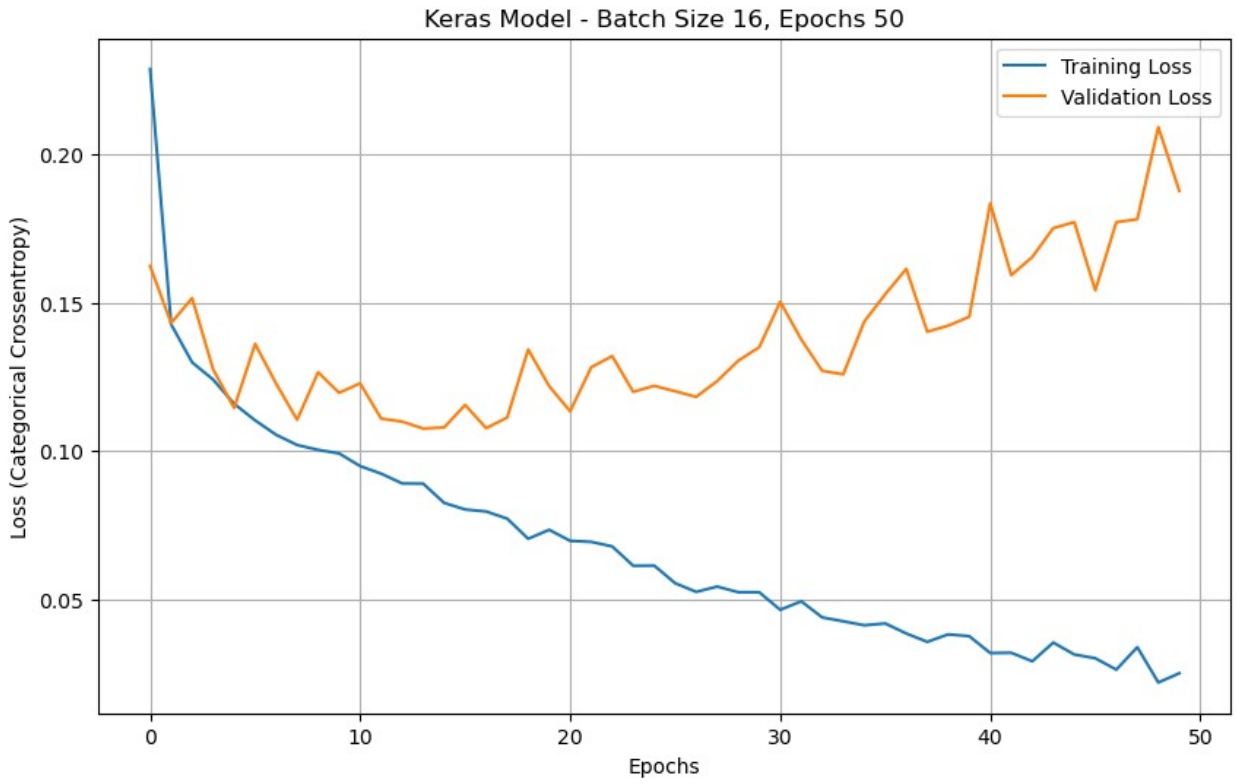
```
{epochs}')
            plt.legend()
            plt.grid(True)
            plt.show()

            # Plotting the accuracy curves
            plt.figure(figsize=(10, 6))
            plt.plot(history.history['accuracy'], label='Training
Accuracy')
            plt.plot(history.history['val_accuracy'],
label='Validation Accuracy')
            plt.xlabel('Epochs')
            plt.ylabel('Accuracy')
            plt.title(f'Keras Model Accuracy - Batch Size
{batch_size}, Epochs {epochs}')
            plt.legend()
            plt.grid(True)
            plt.show()

    return results

# Defining the different epochs and batch sizes
epochs_list = [10, 50, 100]
batch_size_list = [16, 32, 64, 128]

# Training the Keras classifier and collecting results
keras_results = train_keras_classifier(X_train_combined, y_train,
X_test_combined, y_test, epochs_list, batch_size_list)
```

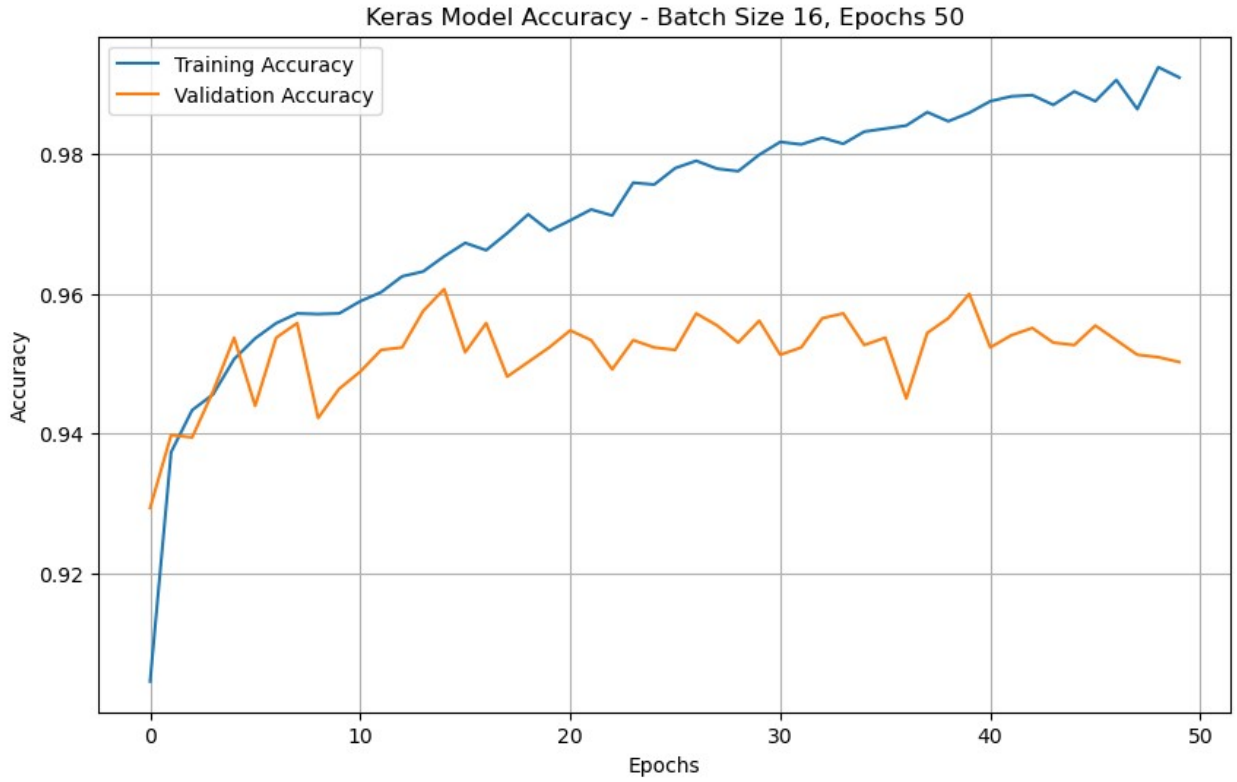
```
Training with Batch Size: 16, Epochs: 10

/home/unina/anaconda3/lib/python3.12/site-packages/keras/src/layers/
core/dense.py:87: UserWarning: Do not pass an
`input_shape`/`input_dim` argument to a layer. When using Sequential
models, prefer using an `Input(shape)` object as the first layer in
the model instead.
  super().__init__(activity_regularizer=activity_regularizer,
**kwargs)

Epoch [10/10], Training Loss: 0.0977, Training Accuracy: 0.9562,
Validation Loss: 0.1262, Validation Accuracy: 0.9457
```


Keras Model - Batch Size 16, Epochs 10
Training Loss
Validation Loss
Loss (Categorical Crossentropy)
0.22
0.20
0.18
0.16
0.14
0.12
0.10
0
2
4
6
8
Epochs

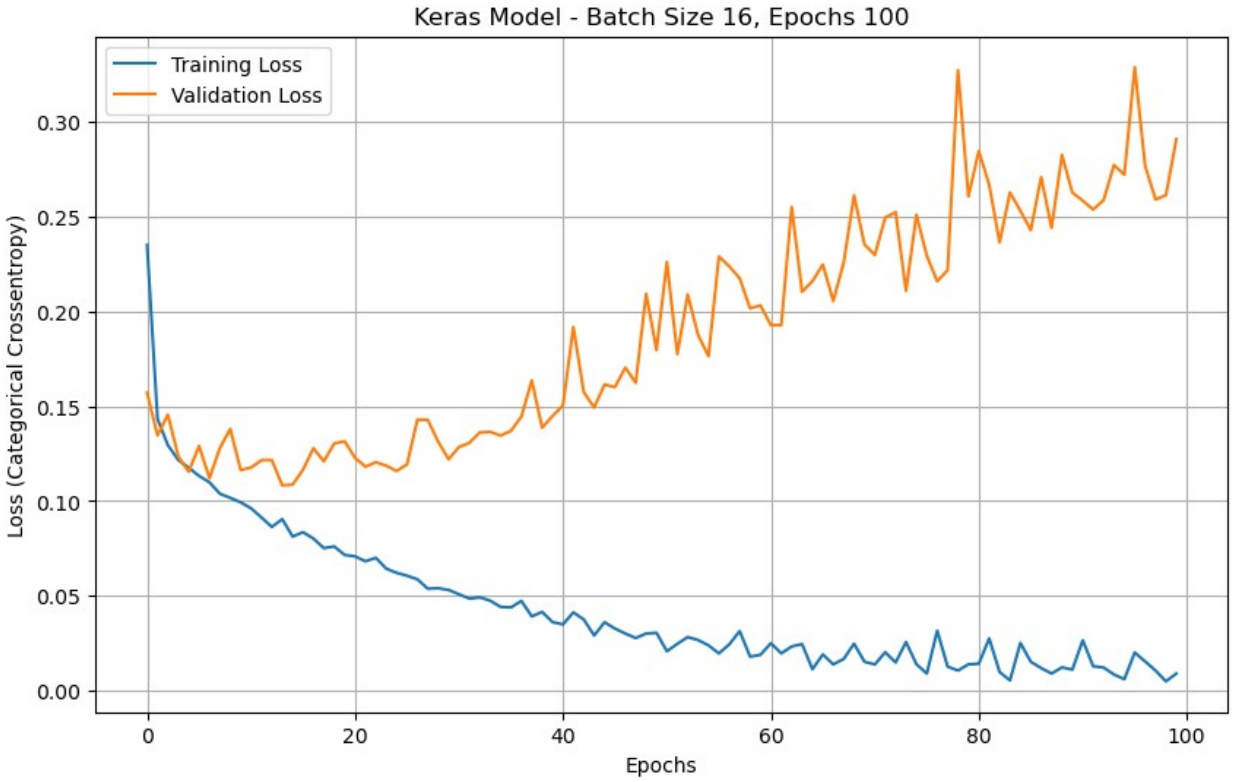

Keras Model Accuracy - Batch Size 16, Epochs 10
Training Accuracy
Validation Accuracy
Accuracy
0.95
0.94
0.93
0.92
0.91
0
2
4
6
8
Epochs

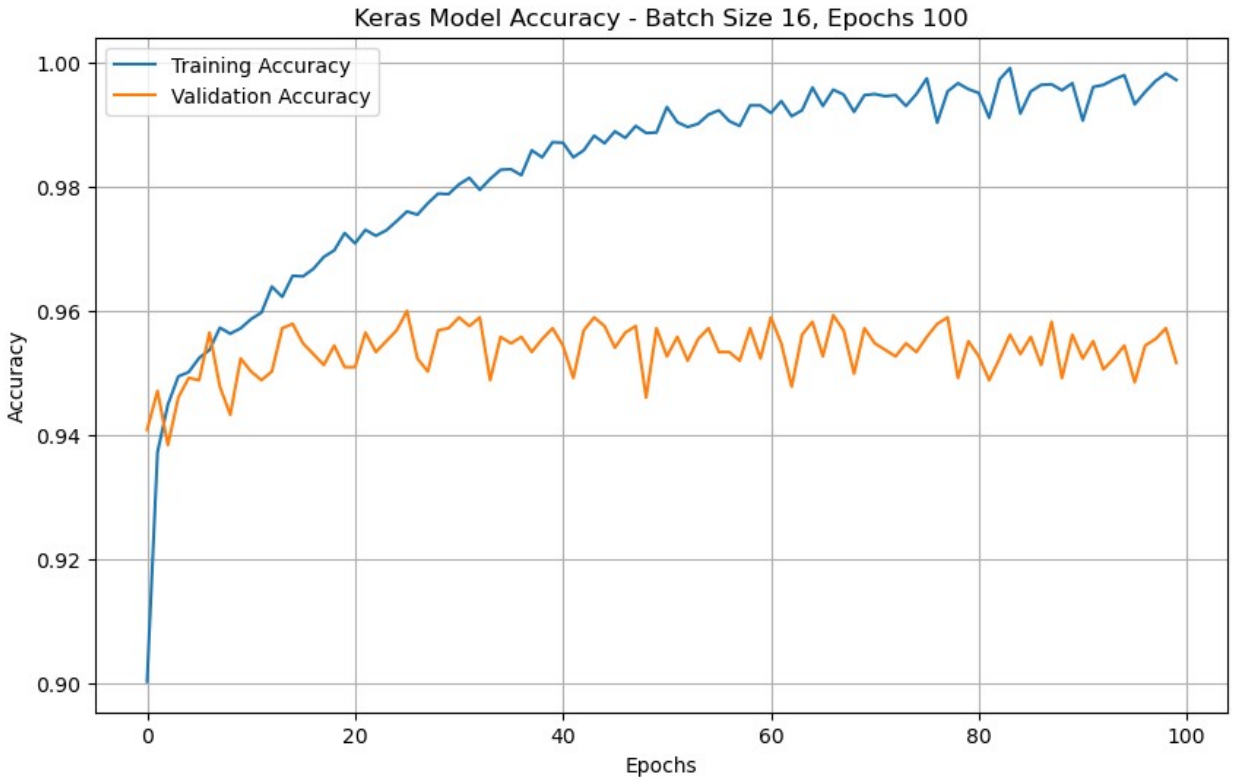
```
Training with Batch Size: 16, Epochs: 50
Epoch [10/50], Training Loss: 0.0992, Training Accuracy: 0.9572,
Validation Loss: 0.1196, Validation Accuracy: 0.9464
Epoch [20/50], Training Loss: 0.0735, Training Accuracy: 0.9690,
Validation Loss: 0.1219, Validation Accuracy: 0.9523
Epoch [30/50], Training Loss: 0.0525, Training Accuracy: 0.9799,
Validation Loss: 0.1350, Validation Accuracy: 0.9561
Epoch [40/50], Training Loss: 0.0377, Training Accuracy: 0.9859,
Validation Loss: 0.1452, Validation Accuracy: 0.9600
Epoch [50/50], Training Loss: 0.0252, Training Accuracy: 0.9909,
Validation Loss: 0.1876, Validation Accuracy: 0.9502
```



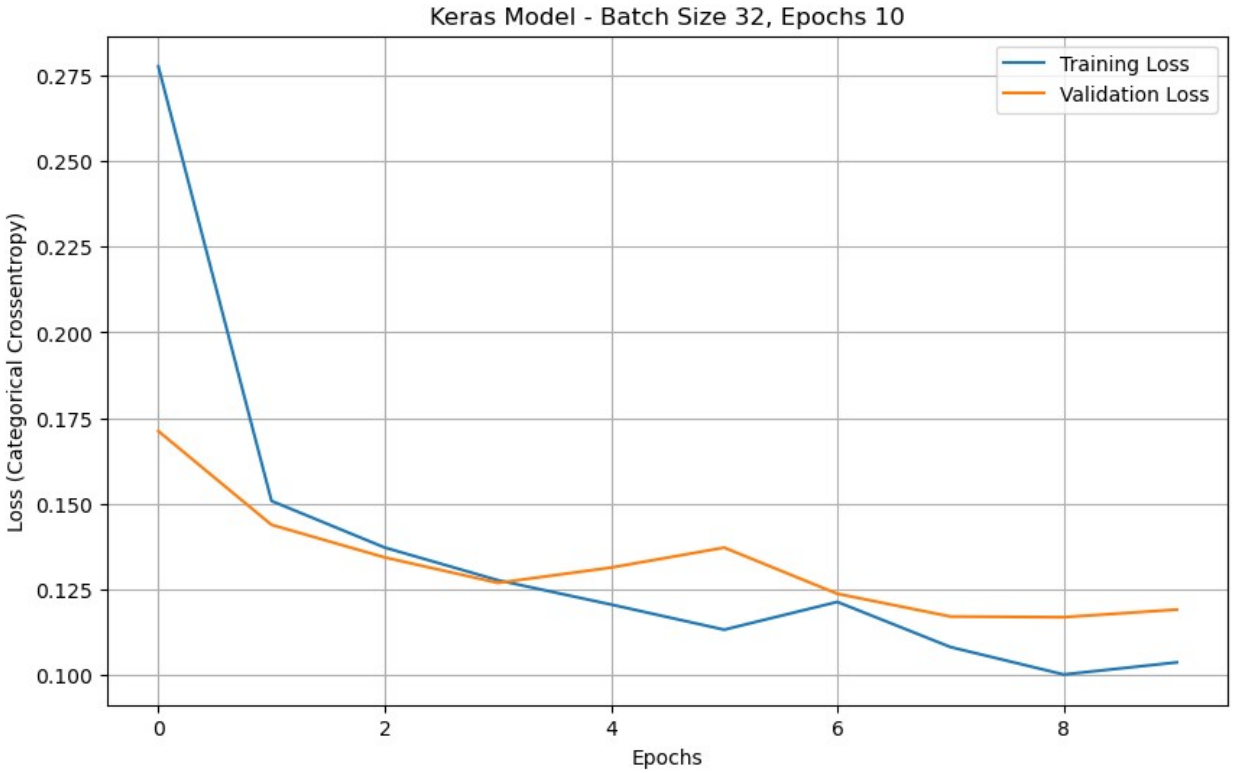


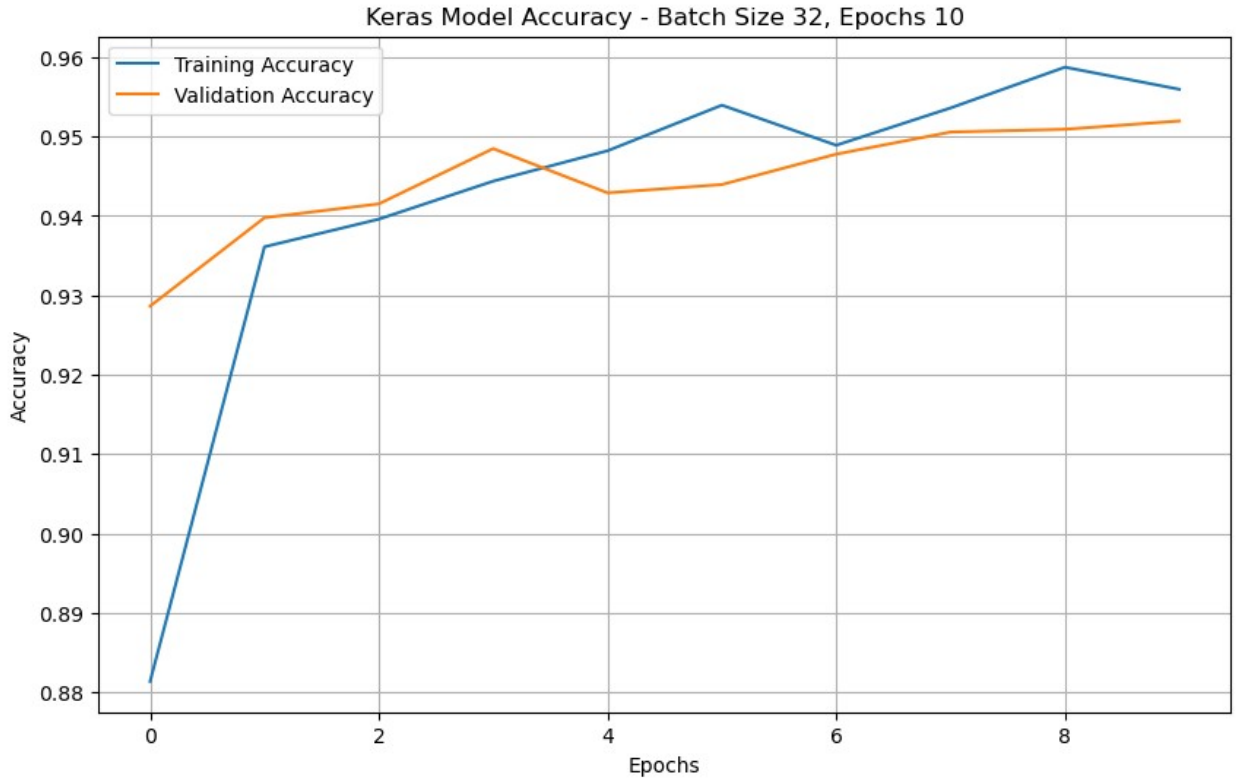

```
Training with Batch Size: 16, Epochs: 100
Epoch [10/100], Training Loss: 0.0993, Training Accuracy: 0.9572,
Validation Loss: 0.1163, Validation Accuracy: 0.9523
Epoch [20/100], Training Loss: 0.0716, Training Accuracy: 0.9725,
Validation Loss: 0.1315, Validation Accuracy: 0.9509
Epoch [30/100], Training Loss: 0.0531, Training Accuracy: 0.9788,
Validation Loss: 0.1220, Validation Accuracy: 0.9572
Epoch [40/100], Training Loss: 0.0361, Training Accuracy: 0.9871,
Validation Loss: 0.1448, Validation Accuracy: 0.9572
Epoch [50/100], Training Loss: 0.0305, Training Accuracy: 0.9887,
Validation Loss: 0.1797, Validation Accuracy: 0.9572
Epoch [60/100], Training Loss: 0.0189, Training Accuracy: 0.9930,
Validation Loss: 0.2032, Validation Accuracy: 0.9523
Epoch [70/100], Training Loss: 0.0152, Training Accuracy: 0.9947,
Validation Loss: 0.2353, Validation Accuracy: 0.9572
Epoch [80/100], Training Loss: 0.0139, Training Accuracy: 0.9956,
Validation Loss: 0.2607, Validation Accuracy: 0.9551
Epoch [90/100], Training Loss: 0.0111, Training Accuracy: 0.9966,
Validation Loss: 0.2627, Validation Accuracy: 0.9561
Epoch [100/100], Training Loss: 0.0090, Training Accuracy: 0.9971,
Validation Loss: 0.2909, Validation Accuracy: 0.9516
```


Keras Model - Batch Size 16, Epochs 100
Training Loss
Validation Loss
Loss (Categorical Crossentropy)
0.30
0.25
0.20
0.15
0.10
0.05
0.00
0
20
40
60
80
100
Epochs

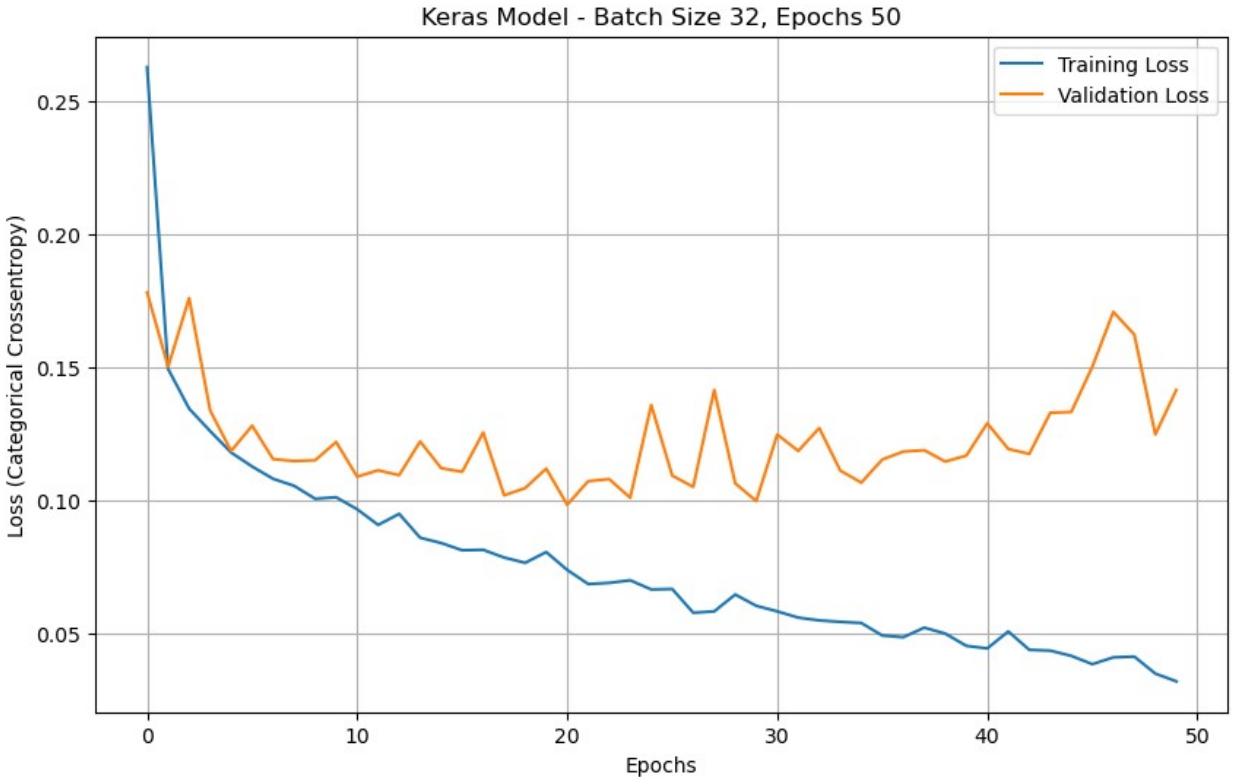

Keras Model Accuracy - Batch Size 16, Epochs 100
Training Accuracy
Validation Accuracy
Accuracy
1.00
0.98
0.96
0.94
0.92
0.90
0
20
40
60
80
100
Epochs

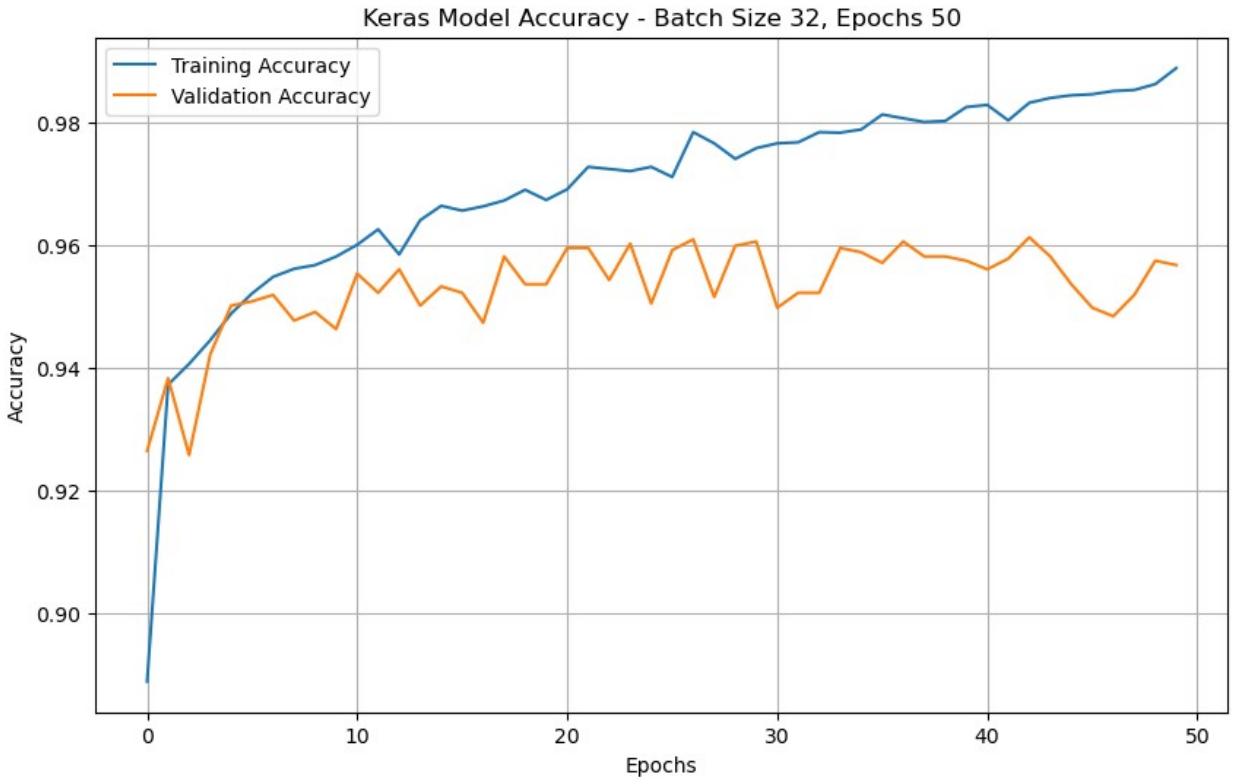
```
Training with Batch Size: 32, Epochs: 10
Epoch [10/10], Training Loss: 0.1036, Training Accuracy: 0.9560,
Validation Loss: 0.1191, Validation Accuracy: 0.9520
```



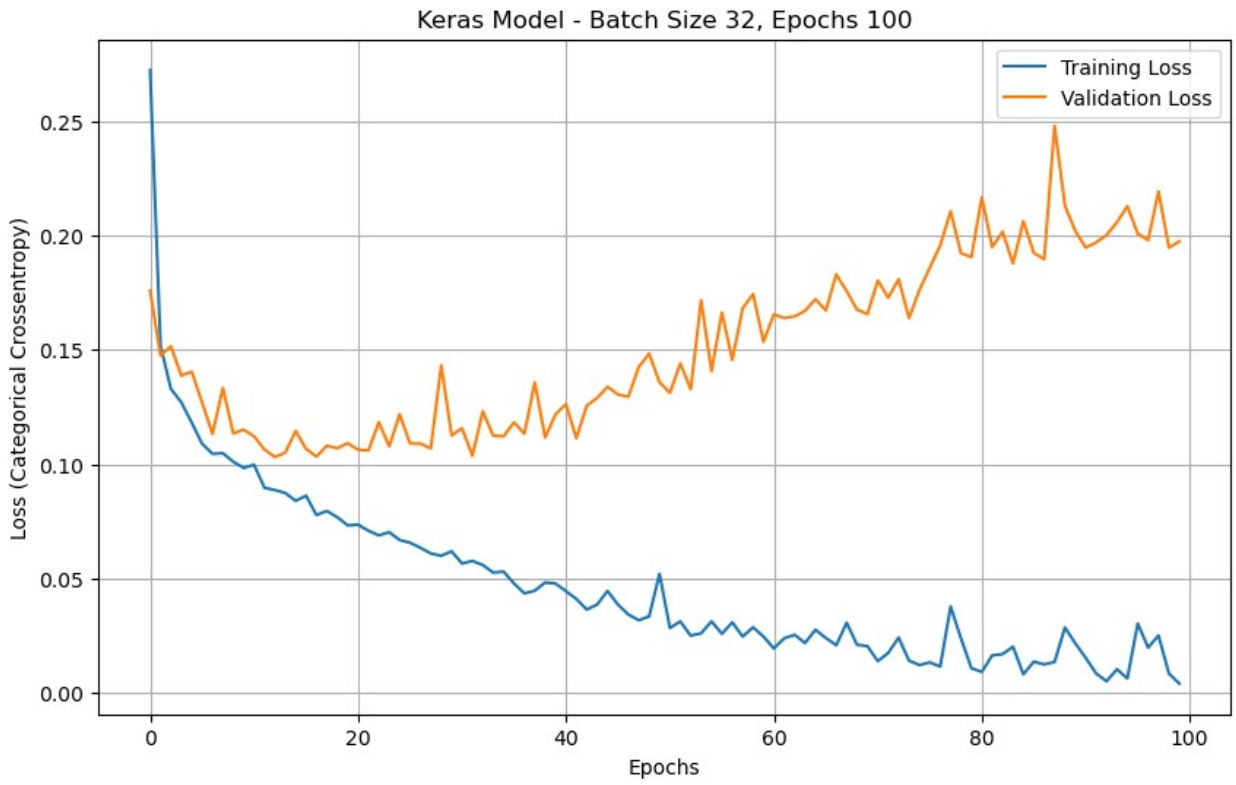


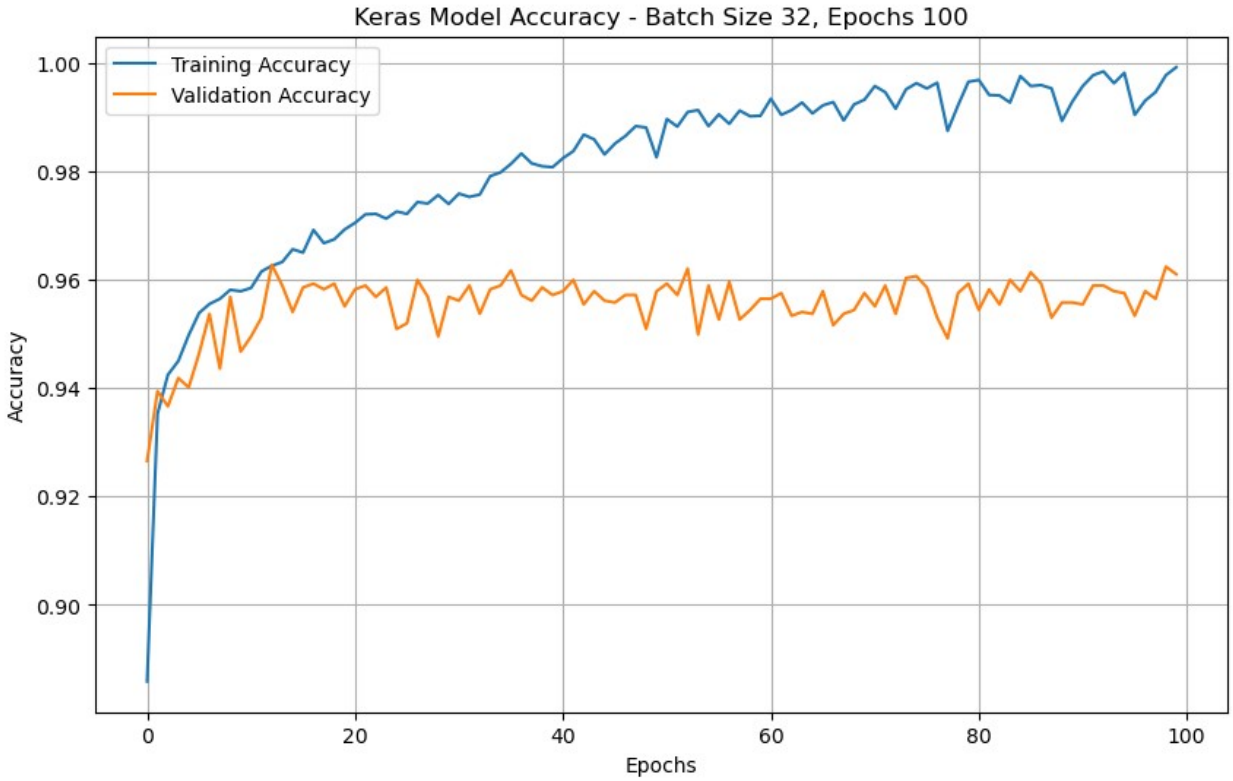

```
Training with Batch Size: 32, Epochs: 50
Epoch [10/50], Training Loss: 0.1011, Training Accuracy: 0.9582,
Validation Loss: 0.1218, Validation Accuracy: 0.9464
Epoch [20/50], Training Loss: 0.0804, Training Accuracy: 0.9674,
Validation Loss: 0.1119, Validation Accuracy: 0.9537
Epoch [30/50], Training Loss: 0.0602, Training Accuracy: 0.9759,
Validation Loss: 0.0996, Validation Accuracy: 0.9607
Epoch [40/50], Training Loss: 0.0451, Training Accuracy: 0.9826,
Validation Loss: 0.1167, Validation Accuracy: 0.9575
Epoch [50/50], Training Loss: 0.0318, Training Accuracy: 0.9889,
Validation Loss: 0.1414, Validation Accuracy: 0.9568
```


Keras Model - Batch Size 32, Epochs 50
Training Loss
Validation Loss
Loss (Categorical Crossentropy)
0.25
0.20
0.15
0.10
0.05
0
10
20
30
40
50
Epochs

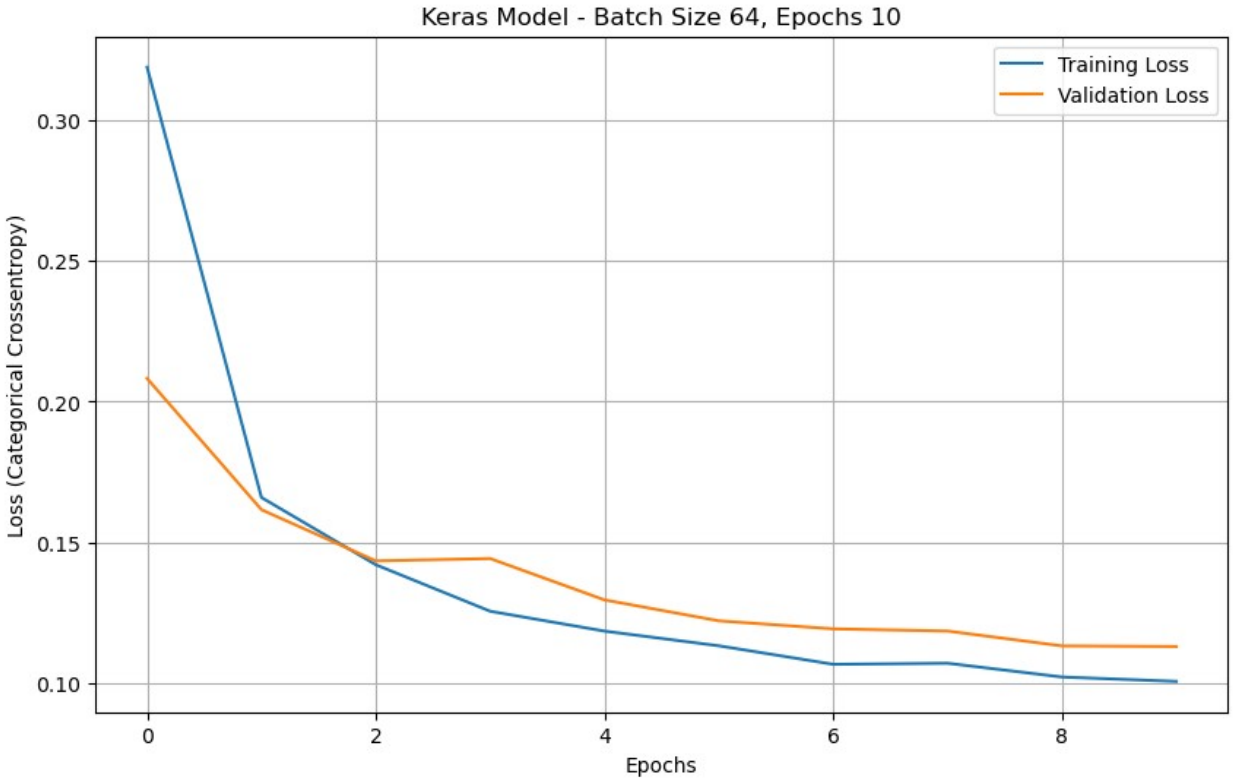

Keras Model Accuracy - Batch Size 32, Epochs 50
Training Accuracy
Validation Accuracy
Accuracy
0.98
0.96
0.94
0.92
0.90
0
10
20
30
40
50
Epochs

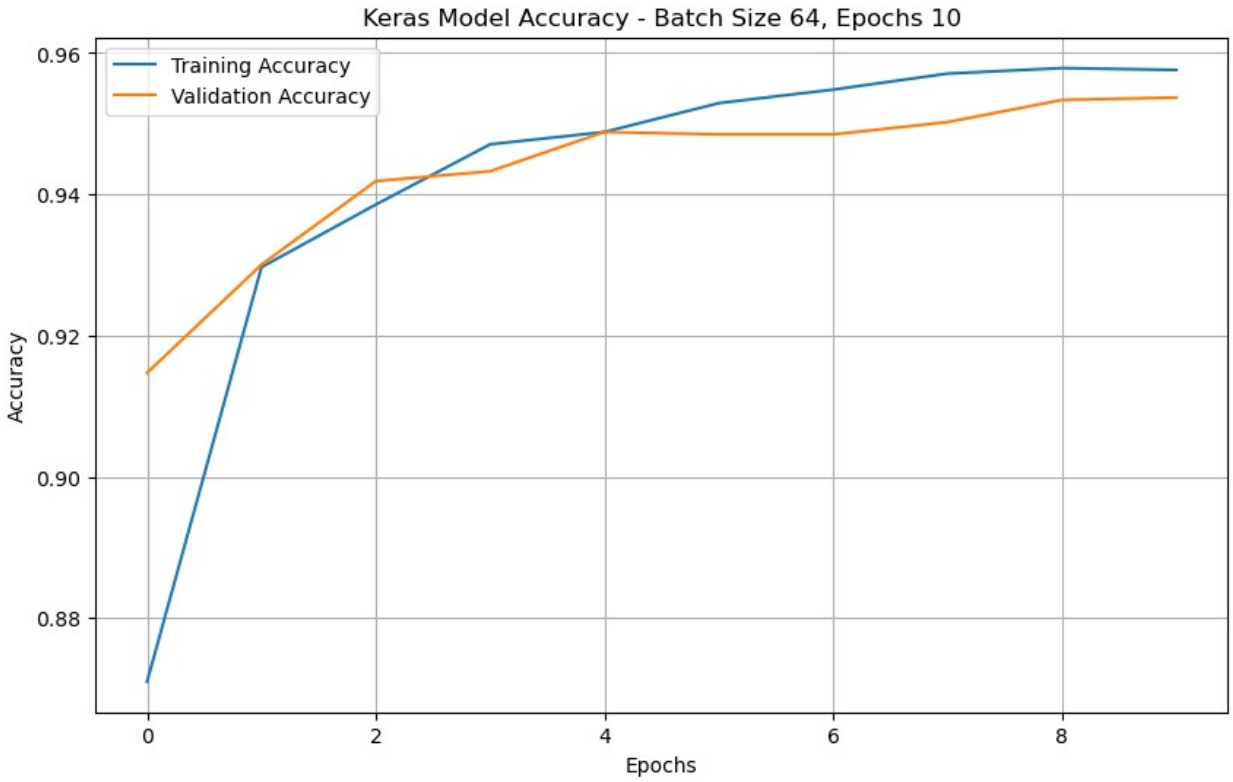
```
Training with Batch Size: 32, Epochs: 100
Epoch [10/100], Training Loss: 0.0984, Training Accuracy: 0.9579,
Validation Loss: 0.1151, Validation Accuracy: 0.9467
Epoch [20/100], Training Loss: 0.0732, Training Accuracy: 0.9693,
Validation Loss: 0.1092, Validation Accuracy: 0.9551
Epoch [30/100], Training Loss: 0.0619, Training Accuracy: 0.9740,
Validation Loss: 0.1126, Validation Accuracy: 0.9568
Epoch [40/100], Training Loss: 0.0478, Training Accuracy: 0.9808,
Validation Loss: 0.1218, Validation Accuracy: 0.9572
Epoch [50/100], Training Loss: 0.0520, Training Accuracy: 0.9826,
Validation Loss: 0.1358, Validation Accuracy: 0.9579
Epoch [60/100], Training Loss: 0.0246, Training Accuracy: 0.9903,
Validation Loss: 0.1536, Validation Accuracy: 0.9565
Epoch [70/100], Training Loss: 0.0204, Training Accuracy: 0.9932,
Validation Loss: 0.1657, Validation Accuracy: 0.9575
Epoch [80/100], Training Loss: 0.0107, Training Accuracy: 0.9965,
Validation Loss: 0.1906, Validation Accuracy: 0.9593
Epoch [90/100], Training Loss: 0.0216, Training Accuracy: 0.9929,
Validation Loss: 0.2020, Validation Accuracy: 0.9558
Epoch [100/100], Training Loss: 0.0040, Training Accuracy: 0.9992,
Validation Loss: 0.1975, Validation Accuracy: 0.9610
```



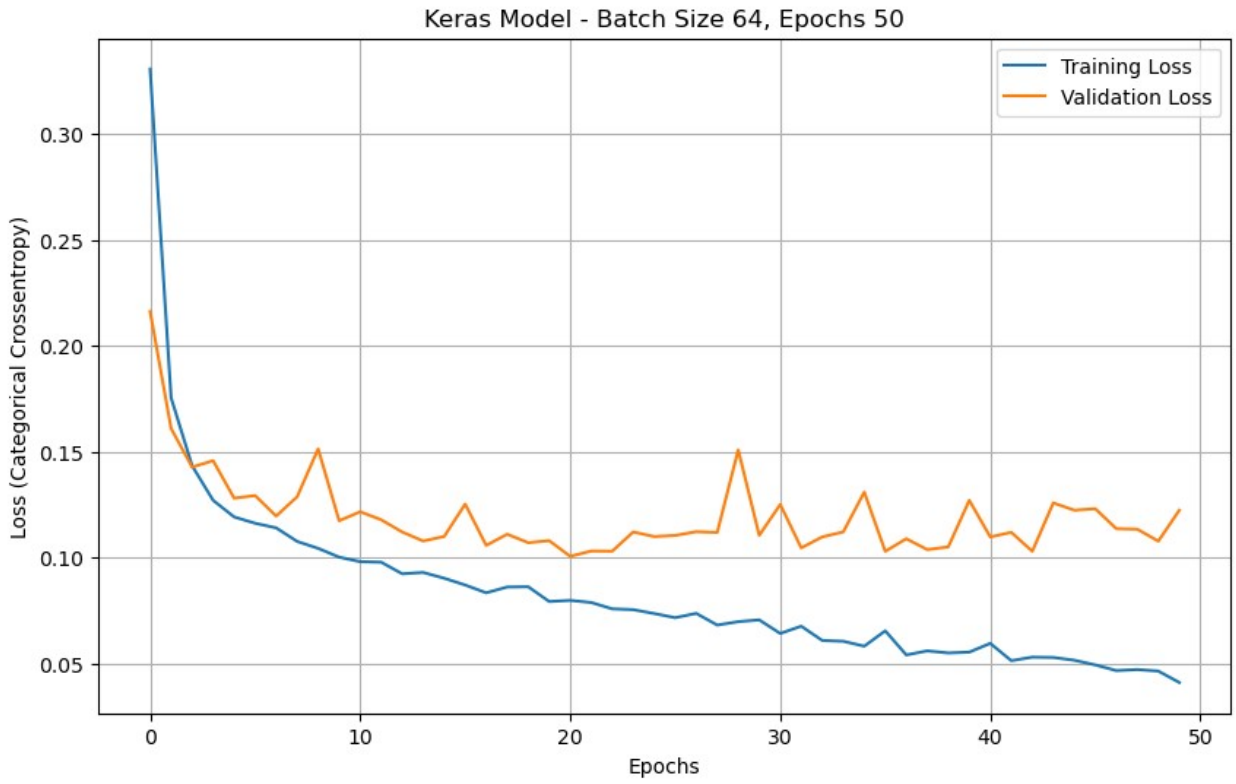


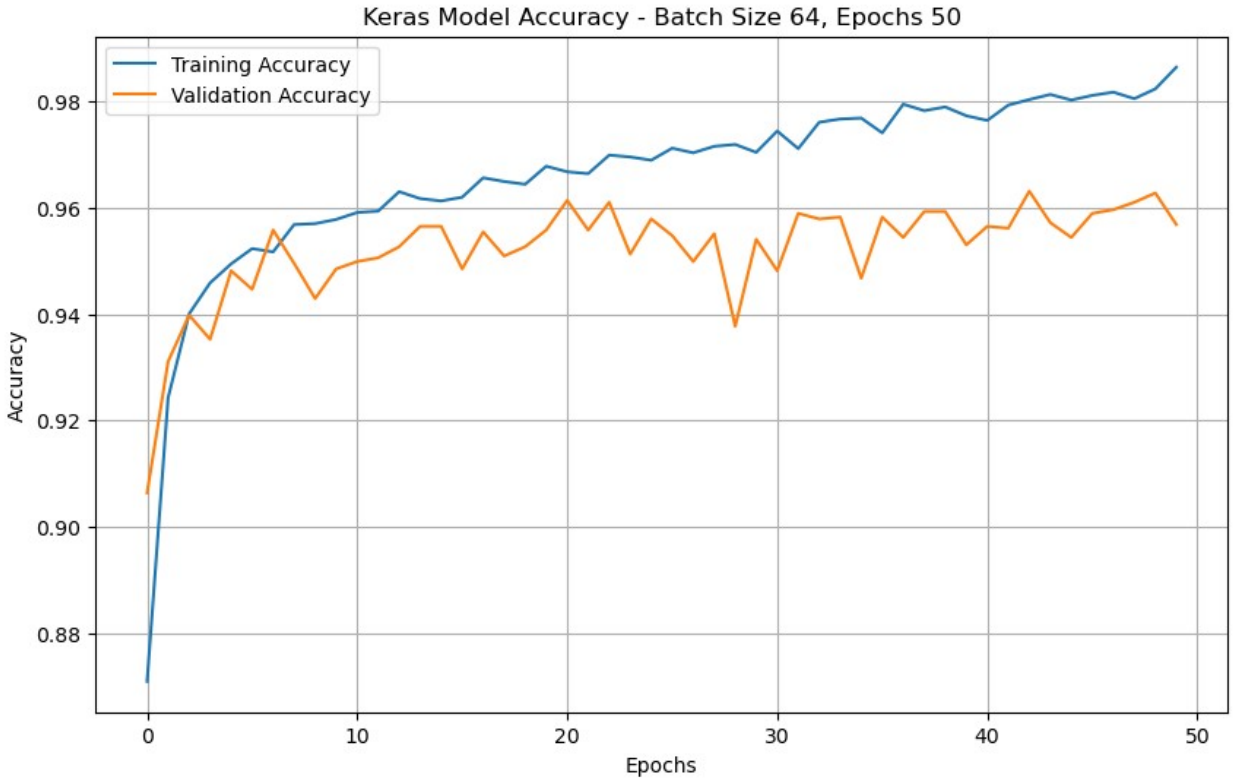

```
Training with Batch Size: 64, Epochs: 10
Epoch [10/10], Training Loss: 0.1007, Training Accuracy: 0.9576,
Validation Loss: 0.1131, Validation Accuracy: 0.9537
```


Keras Model - Batch Size 64, Epochs 10
Training Loss
Validation Loss
Loss (Categorical Crossentropy)
0.30
0.25
0.20
0.15
0.10
0
2
4
6
8
Epochs

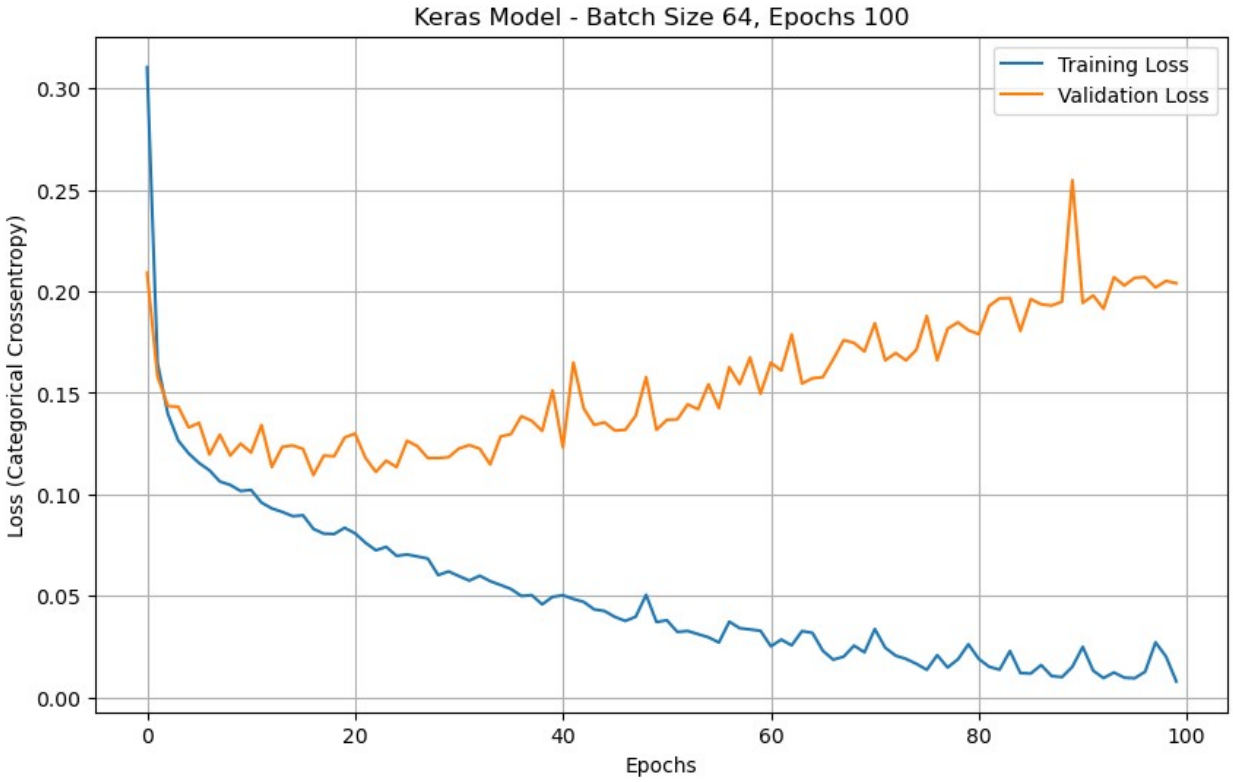

Keras Model Accuracy - Batch Size 64, Epochs 10
Training Accuracy
Validation Accuracy
Accuracy
0.96
0.94
0.92
0.90
0.88
0
2
4
6
8
Epochs

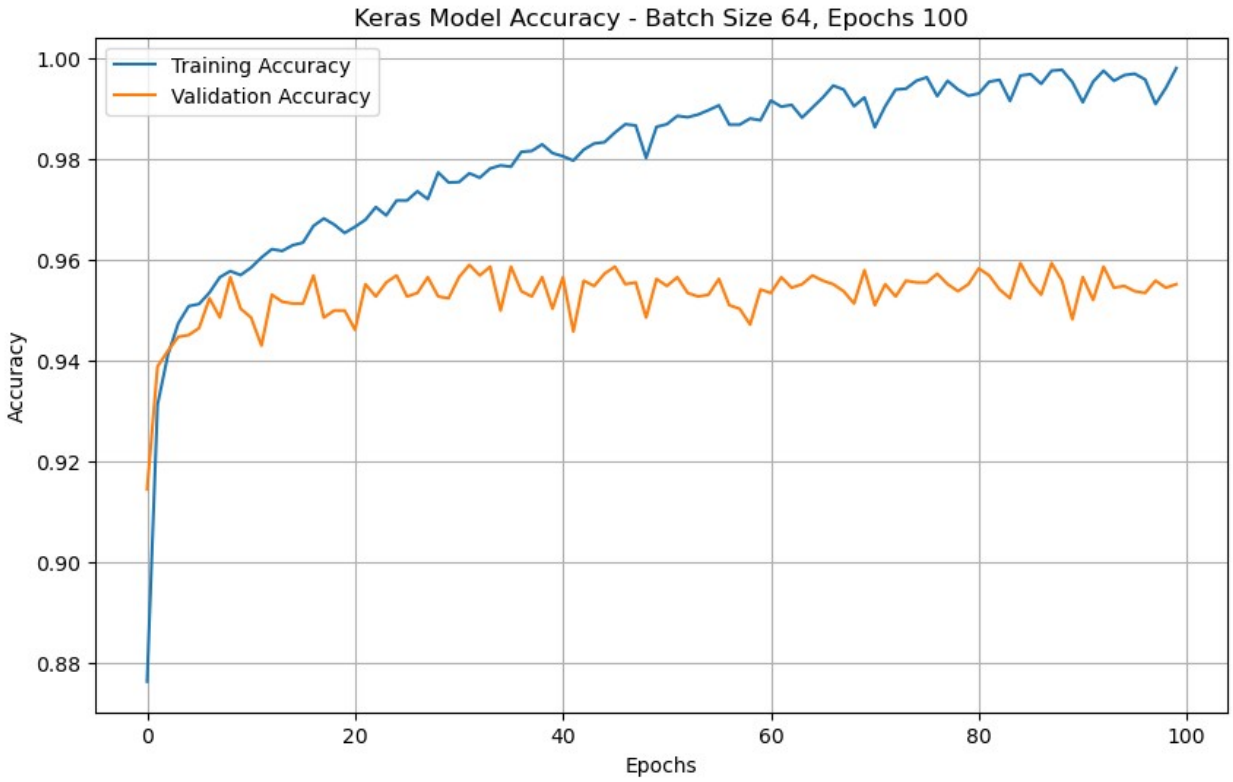
```
Training with Batch Size: 64, Epochs: 50
Epoch [10/50], Training Loss: 0.1002, Training Accuracy: 0.9578,
Validation Loss: 0.1174, Validation Accuracy: 0.9485
Epoch [20/50], Training Loss: 0.0792, Training Accuracy: 0.9678,
Validation Loss: 0.1080, Validation Accuracy: 0.9558
Epoch [30/50], Training Loss: 0.0705, Training Accuracy: 0.9704,
Validation Loss: 0.1105, Validation Accuracy: 0.9541
Epoch [40/50], Training Loss: 0.0553, Training Accuracy: 0.9773,
Validation Loss: 0.1270, Validation Accuracy: 0.9530
Epoch [50/50], Training Loss: 0.0410, Training Accuracy: 0.9864,
Validation Loss: 0.1223, Validation Accuracy: 0.9568
```



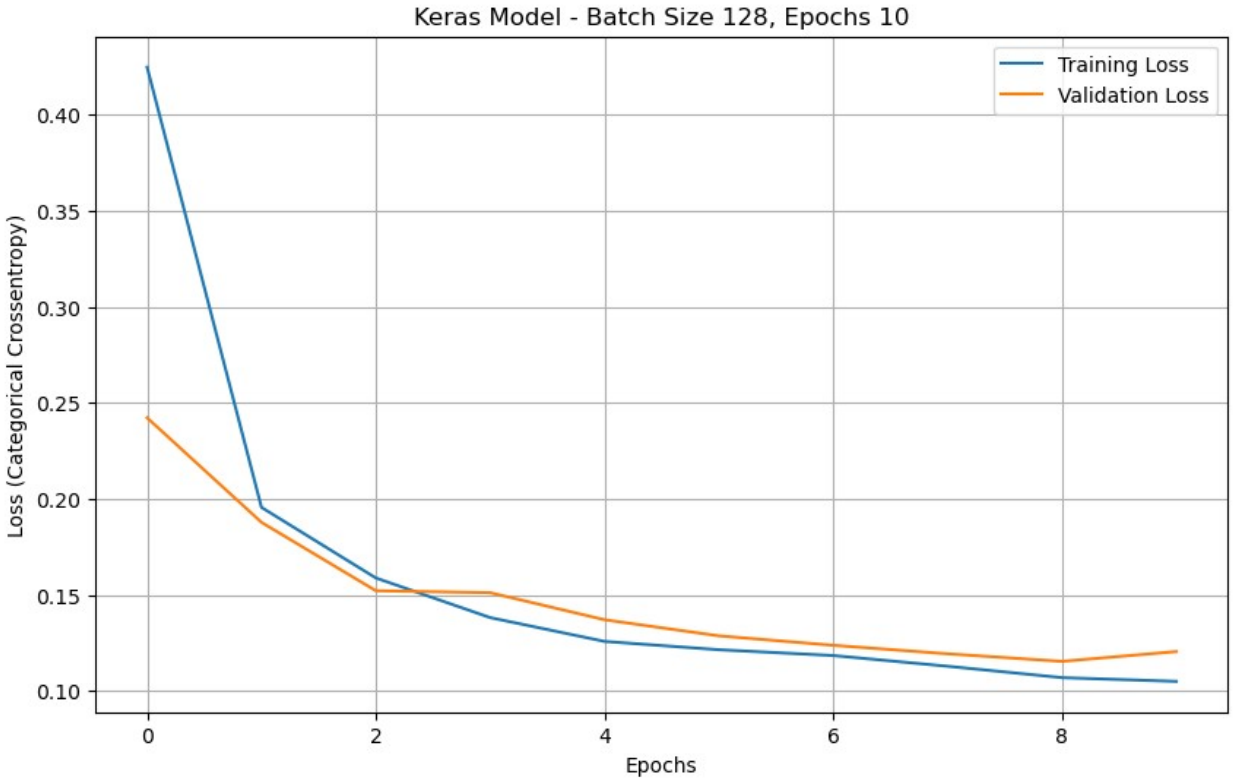


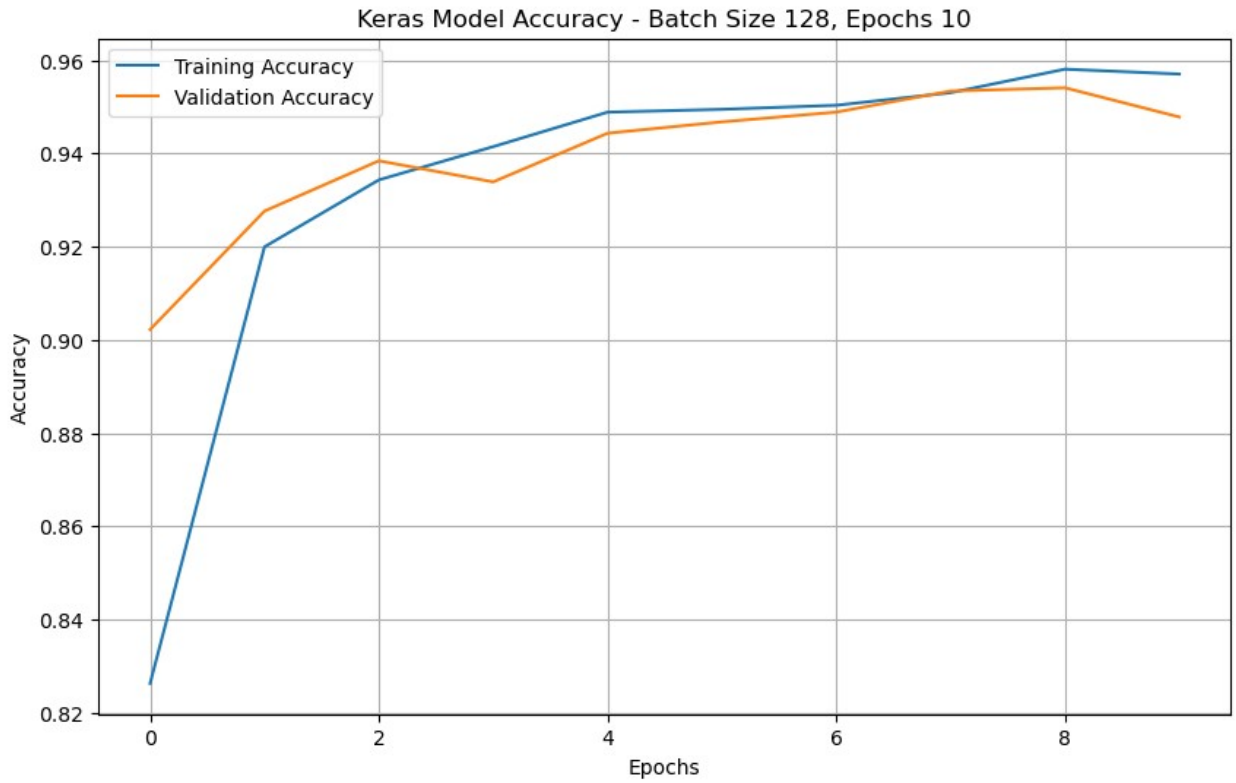

```
Training with Batch Size: 64, Epochs: 100
Epoch [10/100], Training Loss: 0.1016, Training Accuracy: 0.9569,
Validation Loss: 0.1250, Validation Accuracy: 0.9502
Epoch [20/100], Training Loss: 0.0835, Training Accuracy: 0.9653,
Validation Loss: 0.1280, Validation Accuracy: 0.9499
Epoch [30/100], Training Loss: 0.0620, Training Accuracy: 0.9753,
Validation Loss: 0.1183, Validation Accuracy: 0.9523
Epoch [40/100], Training Loss: 0.0494, Training Accuracy: 0.9811,
Validation Loss: 0.1512, Validation Accuracy: 0.9502
Epoch [50/100], Training Loss: 0.0370, Training Accuracy: 0.9863,
Validation Loss: 0.1318, Validation Accuracy: 0.9561
Epoch [60/100], Training Loss: 0.0328, Training Accuracy: 0.9876,
Validation Loss: 0.1496, Validation Accuracy: 0.9541
Epoch [70/100], Training Loss: 0.0221, Training Accuracy: 0.9922,
Validation Loss: 0.1704, Validation Accuracy: 0.9579
Epoch [80/100], Training Loss: 0.0261, Training Accuracy: 0.9925,
Validation Loss: 0.1809, Validation Accuracy: 0.9551
Epoch [90/100], Training Loss: 0.0150, Training Accuracy: 0.9952,
Validation Loss: 0.2548, Validation Accuracy: 0.9481
Epoch [100/100], Training Loss: 0.0078, Training Accuracy: 0.9980,
Validation Loss: 0.2040, Validation Accuracy: 0.9551
```


Keras Model - Batch Size 64, Epochs 100
Training Loss
Validation Loss
Loss (Categorical Crossentropy)
0.30
0.25
0.20
0.15
0.10
0.05
0.00
0
20
40
60
80
100
Epochs

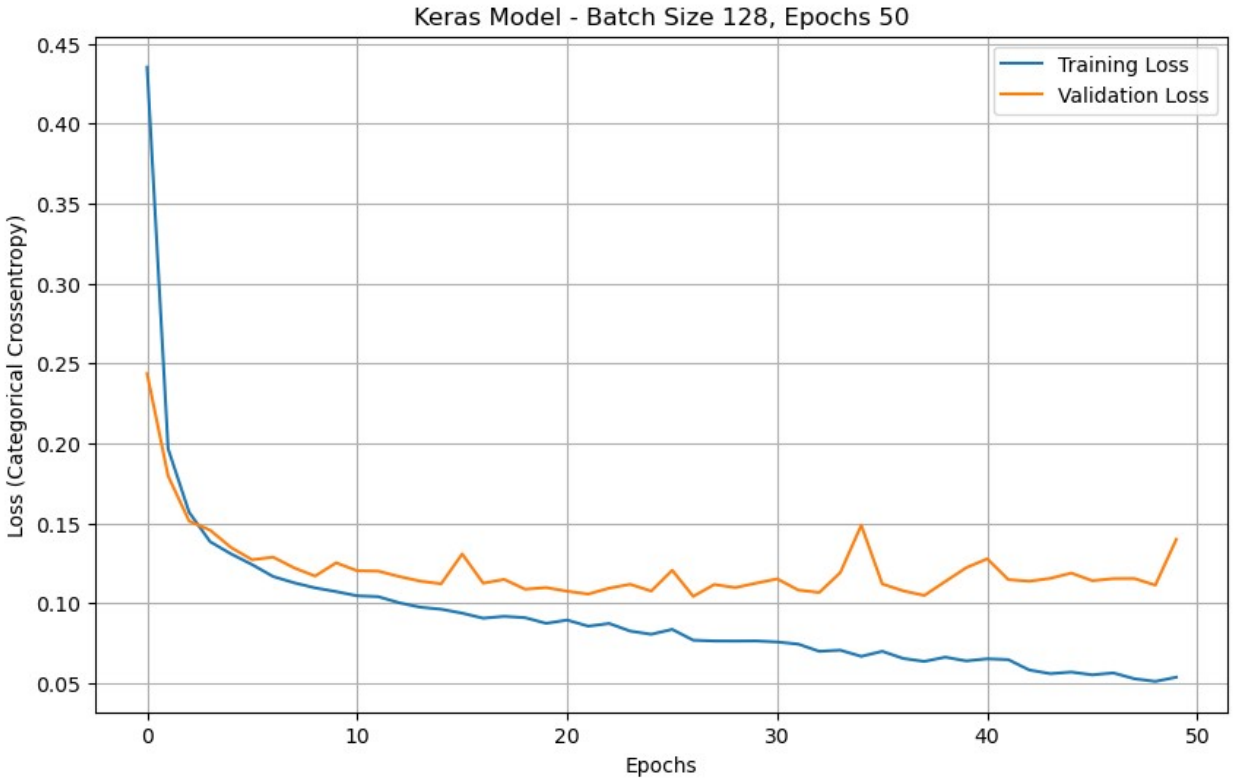

Keras Model Accuracy - Batch Size 64, Epochs 100
Training Accuracy
Validation Accuracy
Accuracy
1.00
0.98
0.96
0.94
0.92
0.90
0.88
0
20
40
60
80
100
Epochs

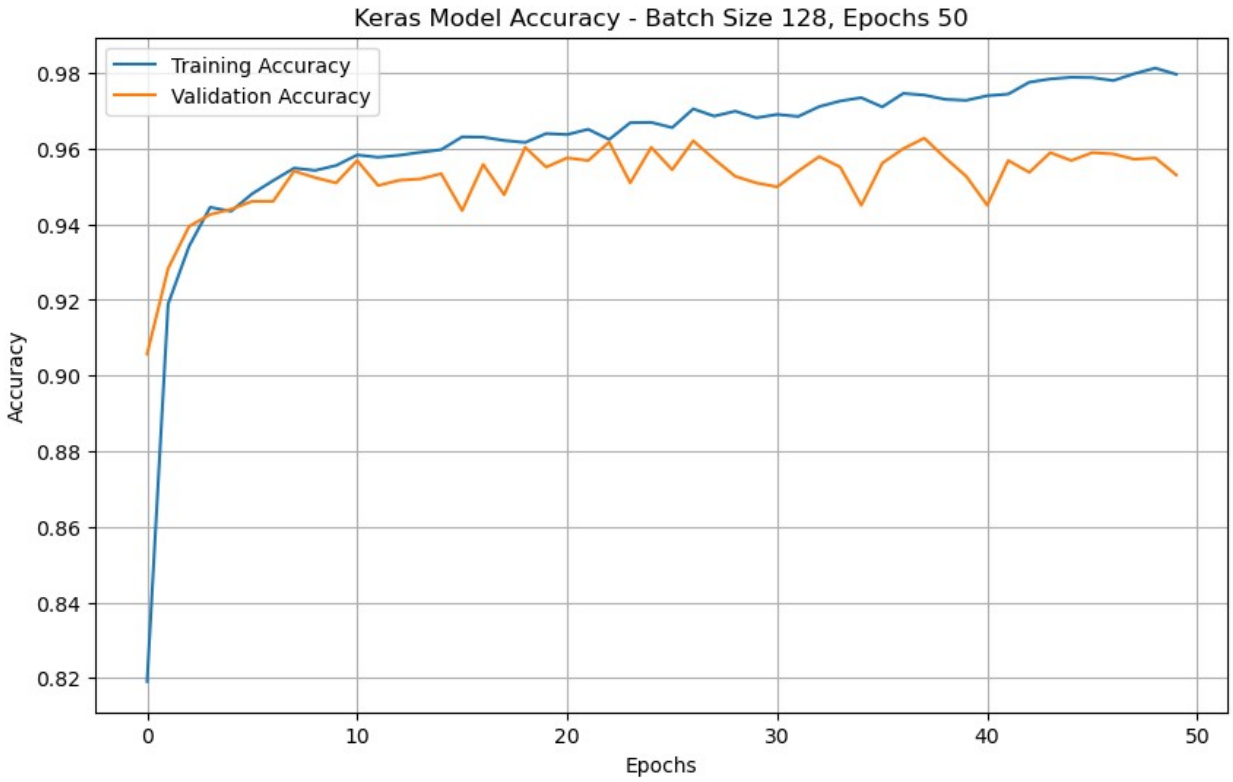
```
Training with Batch Size: 128, Epochs: 10
Epoch [10/10], Training Loss: 0.1049, Training Accuracy: 0.9570,
Validation Loss: 0.1205, Validation Accuracy: 0.9478
```



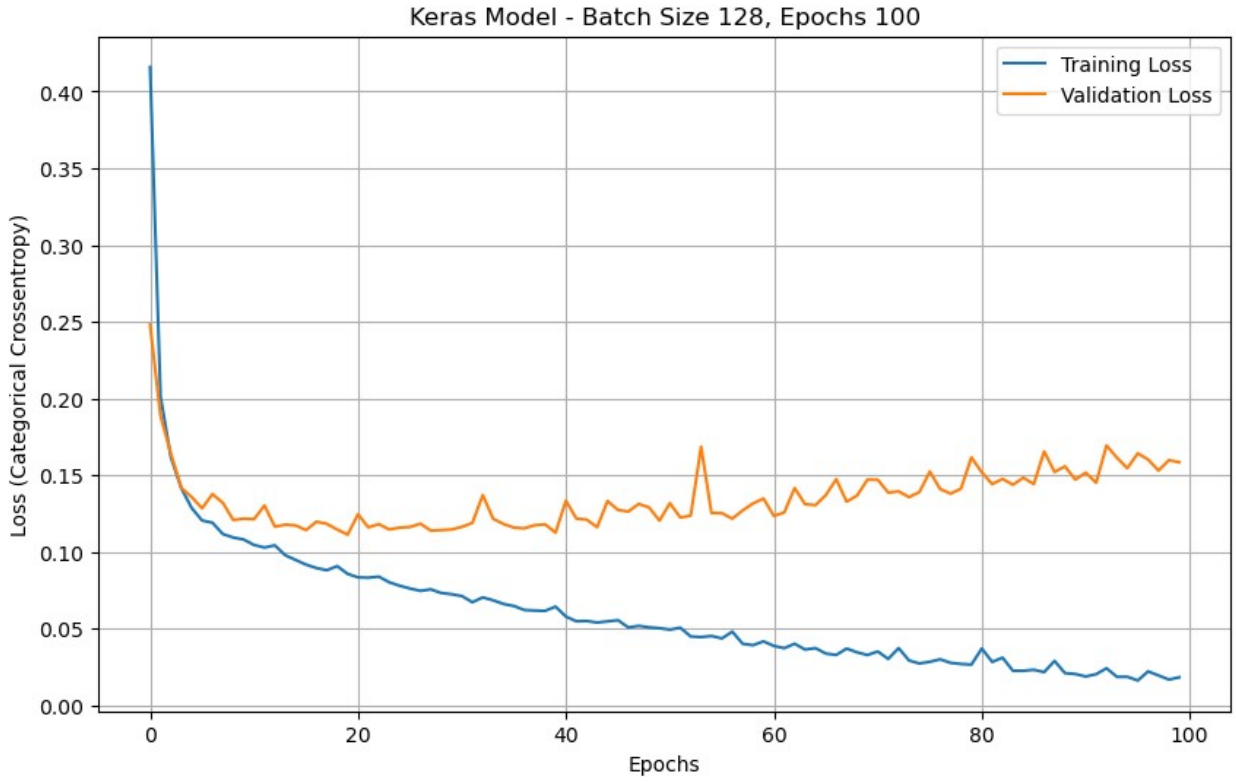


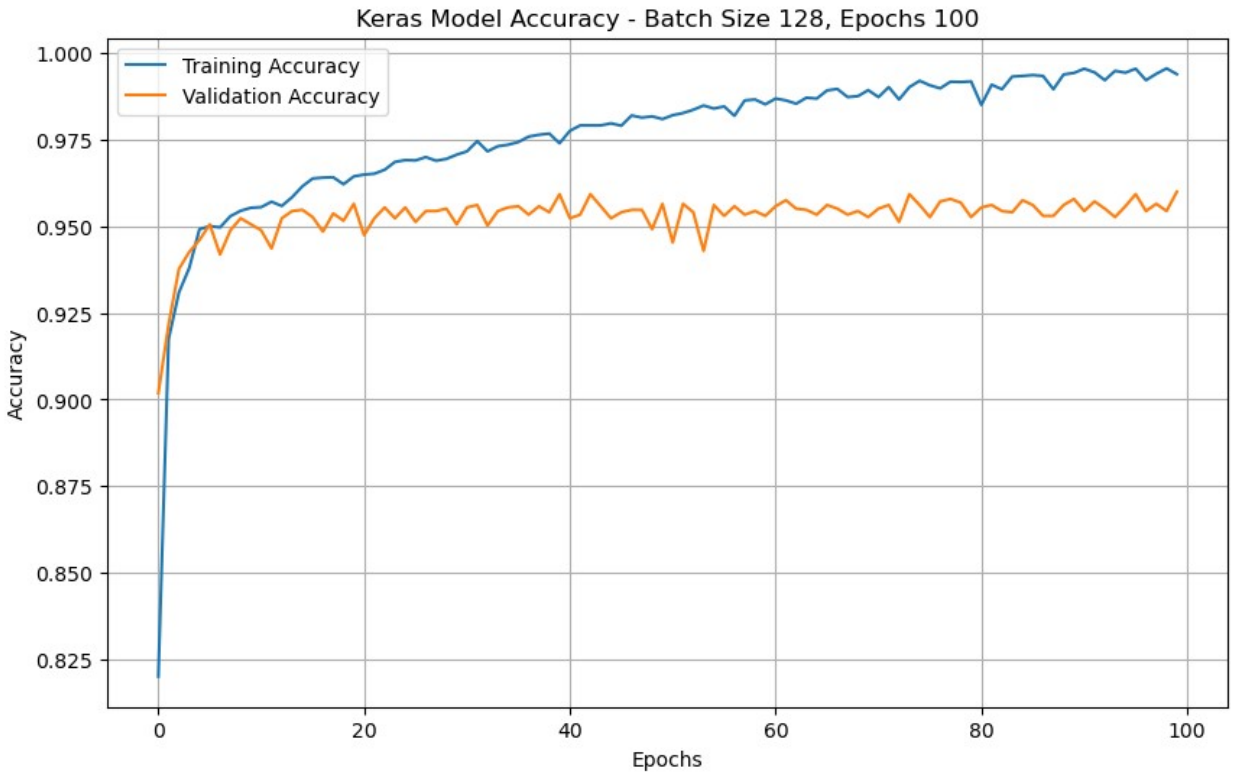

```
Training with Batch Size: 128, Epochs: 50
Epoch [10/50], Training Loss: 0.1073, Training Accuracy: 0.9555,
Validation Loss: 0.1253, Validation Accuracy: 0.9509
Epoch [20/50], Training Loss: 0.0875, Training Accuracy: 0.9640,
Validation Loss: 0.1098, Validation Accuracy: 0.9551
Epoch [30/50], Training Loss: 0.0764, Training Accuracy: 0.9681,
Validation Loss: 0.1126, Validation Accuracy: 0.9509
Epoch [40/50], Training Loss: 0.0640, Training Accuracy: 0.9728,
Validation Loss: 0.1223, Validation Accuracy: 0.9527
Epoch [50/50], Training Loss: 0.0538, Training Accuracy: 0.9796,
Validation Loss: 0.1400, Validation Accuracy: 0.9530
```


Keras Model - Batch Size 128, Epochs 50
Training Loss
Validation Loss
Loss (Categorical Crossentropy)
0.45
0.40
0.35
0.30
0.25
0.20
0.15
0.10
0.05
0
10
20
30
40
50
Epochs


Keras Model Accuracy - Batch Size 128, Epochs 50
Training Accuracy
Validation Accuracy
Accuracy
0.98
0.96
0.94
0.92
0.90
0.88
0.86
0.84
0.82
0
10
20
30
40
50
Epochs

```
Training with Batch Size: 128, Epochs: 100
Epoch [10/100], Training Loss: 0.1081, Training Accuracy: 0.9554,
Validation Loss: 0.1217, Validation Accuracy: 0.9506
Epoch [20/100], Training Loss: 0.0858, Training Accuracy: 0.9644,
Validation Loss: 0.1113, Validation Accuracy: 0.9565
Epoch [30/100], Training Loss: 0.0725, Training Accuracy: 0.9707,
Validation Loss: 0.1147, Validation Accuracy: 0.9506
Epoch [40/100], Training Loss: 0.0644, Training Accuracy: 0.9740,
Validation Loss: 0.1126, Validation Accuracy: 0.9593
Epoch [50/100], Training Loss: 0.0504, Training Accuracy: 0.9809,
Validation Loss: 0.1205, Validation Accuracy: 0.9565
Epoch [60/100], Training Loss: 0.0418, Training Accuracy: 0.9852,
Validation Loss: 0.1348, Validation Accuracy: 0.9530
Epoch [70/100], Training Loss: 0.0329, Training Accuracy: 0.9893,
Validation Loss: 0.1471, Validation Accuracy: 0.9527
Epoch [80/100], Training Loss: 0.0267, Training Accuracy: 0.9918,
Validation Loss: 0.1616, Validation Accuracy: 0.9527
Epoch [90/100], Training Loss: 0.0205, Training Accuracy: 0.9943,
Validation Loss: 0.1472, Validation Accuracy: 0.9579
Epoch [100/100], Training Loss: 0.0184, Training Accuracy: 0.9939,
Validation Loss: 0.1585, Validation Accuracy: 0.9600
```





Results

```
# Converting the results to a DataFrame
keras_results_df_cc = pd.DataFrame(keras_results)
keras_results_df_cc
```

```
    Epochs  Batch Size  Training Loss  Training Accuracy  Validation
Loss  \
0       10          16       0.097652           0.956223
0.126168
1       50          16       0.025246           0.990949
0.187637
2      100          16       0.009003           0.997128
0.290883
3       10          32       0.103646           0.955962
0.119065
4       50          32       0.031761           0.988947
0.141426
5      100          32       0.004045           0.999217
0.197493
6       10          64       0.100731           0.957615
0.113107
7       50          64       0.040952           0.986423
0.122340
8      100          64       0.007792           0.997998
0.204047
```

```
9       10          128       0.104948            0.957006
0.120469
10      50          128       0.053755            0.979634
0.140006
11     100          128       0.018380            0.993908
0.158496

    Validation Accuracy
0              0.945701
1              0.950226
2              0.951618
3              0.951967
4              0.956840
5              0.961016
6              0.953707
7              0.956840
8              0.955099
9              0.947790
10             0.953011
11             0.959972
```

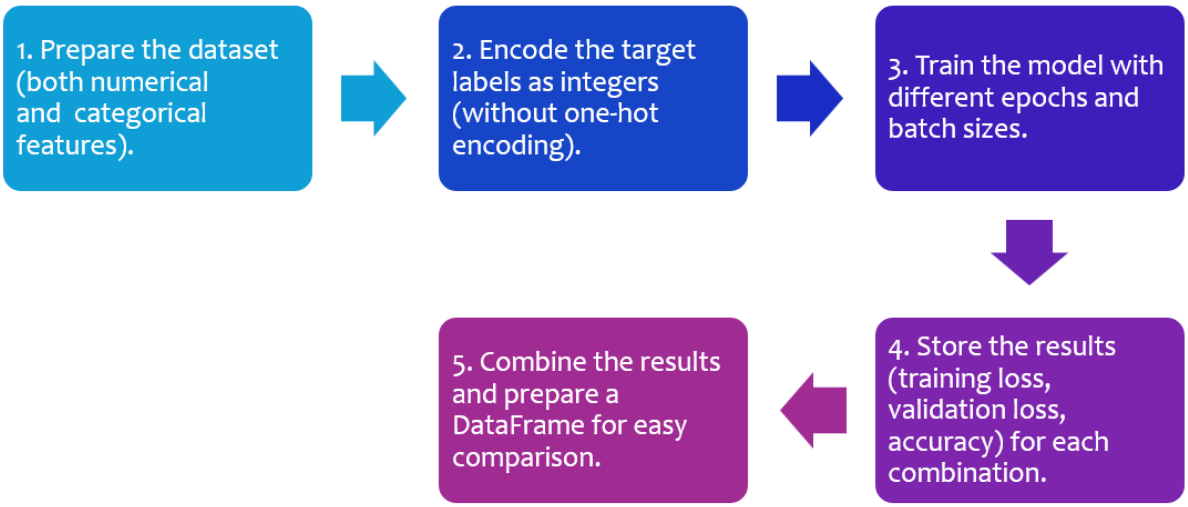
The model achieves high training and validation accuracy across different configurations of epochs and batch sizes. Notably, training with 100 epochs and batch sizes of 32 and 64 yields the best results, with validation accuracies of 96.1% and 95.6%, respectively. The training loss consistently decreases with more epochs, indicating effective learning, although validation loss increases slightly in some cases, hinting at possible overfitting for certain configurations. Overall, the model shows stable performance with categorical crossentropy, especially when using 100 epochs and moderate batch sizes like 32 or 64.

Sparse Categorical crossentropy

Sparse categorical cross-entropy is a loss function used in multi-class classification problems where the target labels are represented as integers rather than one-hot encoded vectors. It is particularly useful when dealing with a large number of classes, as it saves memory by storing the target labels as single integers.

Preprocessing

The following are the steps involved for applying Sparse Categorical Cross Entropy



instead of y_one_hot coded we pass only y_encoded

```
# Splitting the data into train and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y_encoded,
test_size=0.2, random_state=42)

# Scaling the numerical features
scaler = StandardScaler()
X_train_numerical = scaler.fit_transform(X_train[numerical_features])
X_test_numerical = scaler.transform(X_test[numerical_features])

# One-hot encode categorical features
encoder = OneHotEncoder(sparse_output=False, handle_unknown='ignore')
X_train_categorical =
encoder.fit_transform(X_train[categorical_features])
X_test_categorical = encoder.transform(X_test[categorical_features])

# Combining numerical and categorical features
X_train_combined = np.hstack([X_train_numerical, X_train_categorical])
X_test_combined = np.hstack([X_test_numerical, X_test_categorical])
```

Training

The following function is designed to train a Keras classification model with varying combinations of epochs and batch sizes. The goal is to experiment with different configurations to find the best-performing model for the classification task. The model consists of two hidden layers with ReLU activations and a softmax output layer for multi-class classification.

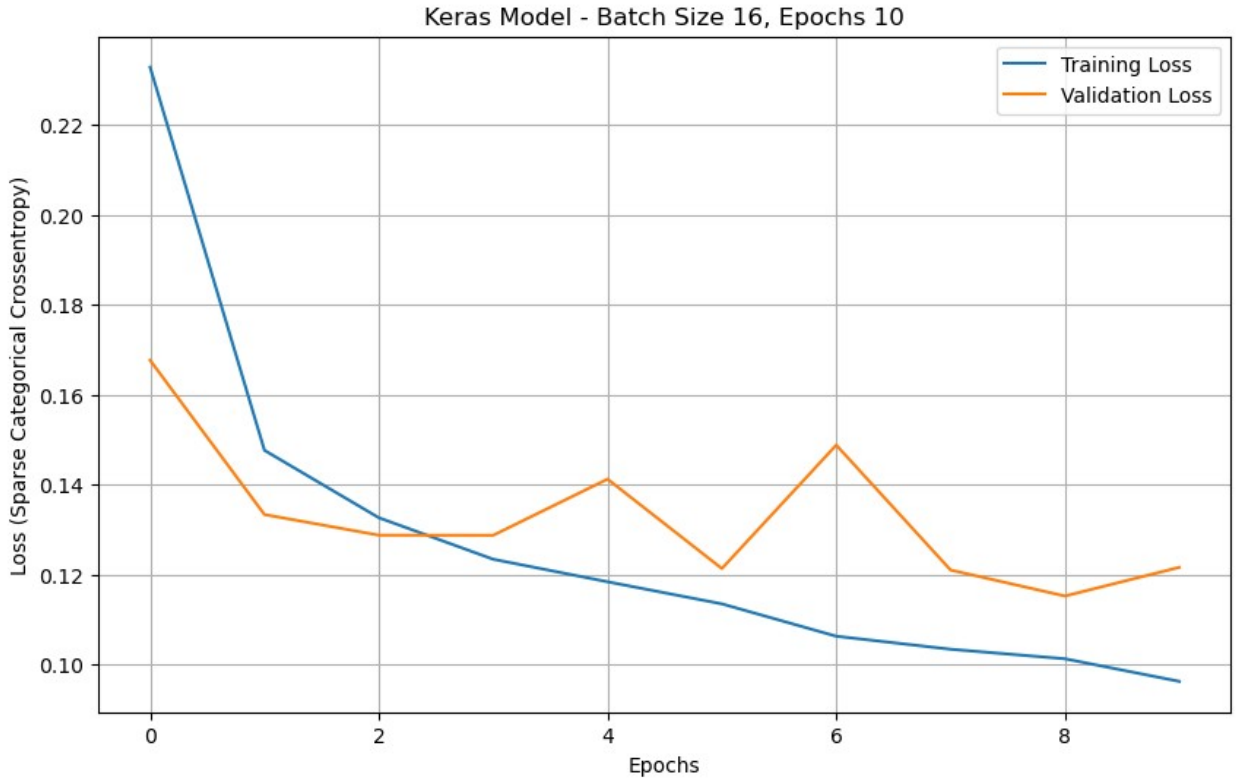
For each configuration, the function trains the model using categorical crossentropy as the loss function and tracks accuracy as the evaluation metric. After training, it prints the loss and accuracy values for both training and validation sets and generates loss and accuracy plots. The results for each configuration (training/validation loss and accuracy) are stored for comparison.

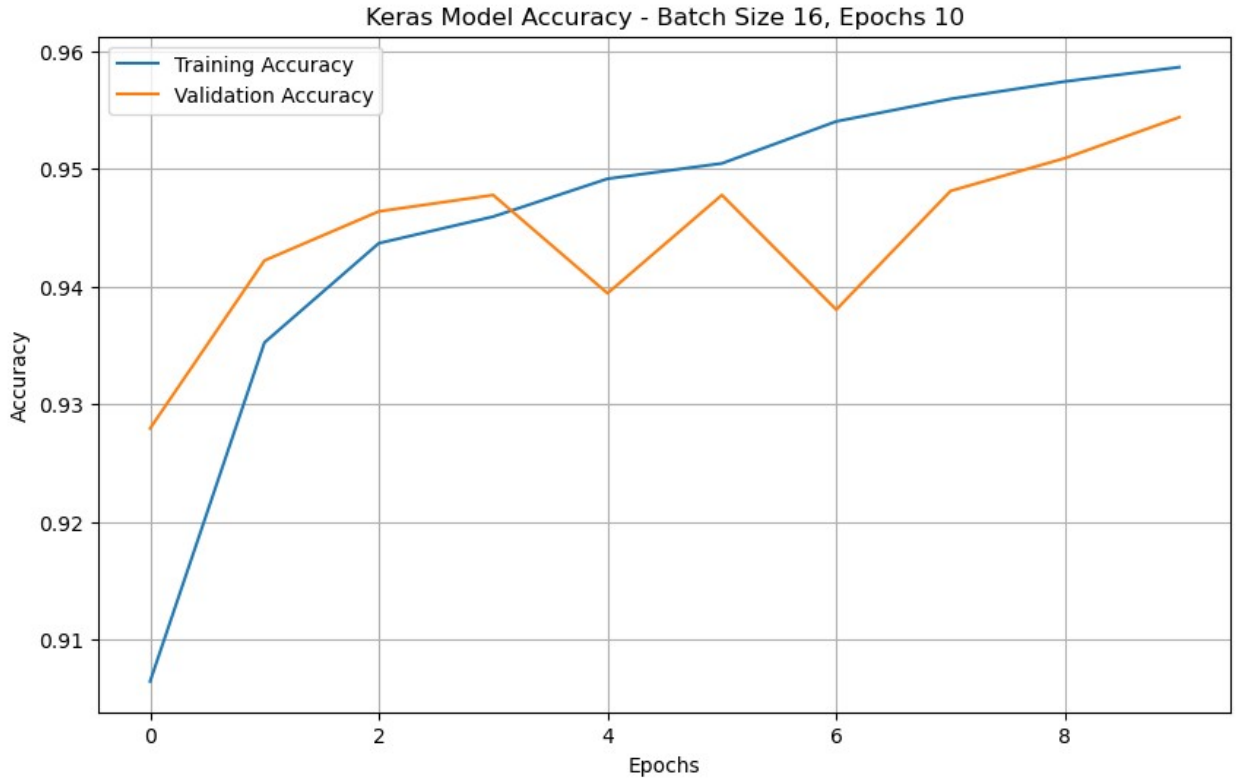
```
# Function to train Keras model for classification with batch size and
epochs iteration
def train_keras_classifier_spc(X_train, y_train, X_test, y_test,
epochs_list, batch_size_list):
    results = []

    for batch_size in batch_size_list:
        for epochs in epochs_list:
            print(f"\nTraining with Batch Size: {batch_size}, Epochs:
{epochs}")

            # Building a simple classification model
            model = models.Sequential([
                layers.Dense(128, activation='relu',
input_shape=(X_train.shape[1],)),
                layers.Dense(64, activation='relu'),
                layers.Dense(3, activation='softmax')
            ])

            # Compiling the model with sparse_categorical_crossentropy
            model.compile(optimizer='adam',
loss='sparse_categorical_crossentropy', metrics=['accuracy'])

            # Defiinge early stopping
            early_stopping = EarlyStopping(monitor='val_loss',
patience=10, restore_best_weights=True)

            # Training the model
            history = model.fit(X_train, y_train,
validation_split=0.2,
                                epochs=epochs, batch_size=batch_size,
                                callbacks=[early_stopping], verbose=0)

            # Storing the final epoch's metrics
            avg_loss = history.history['loss'][-1]
            avg_accuracy = history.history['accuracy'][-1]
            val_loss = history.history['val_loss'][-1]
            val_accuracy = history.history['val_accuracy'][-1]

            print(f"Final Epoch [{epochs}], Training Loss:
{avg_loss:.4f}, "
                  f"Training Accuracy: {avg_accuracy:.4f}, Validation
Loss: {val_loss:.4f}, "
                  f"Validation Accuracy: {val_accuracy:.4f}")

            # Storing the results
            results.append({
                'Epochs': epochs,
                'Batch Size': batch_size,
                'Training Loss': avg_loss,
```

```
                'Training Accuracy': avg_accuracy,
                'Validation Loss': val_loss,
                'Validation Accuracy': val_accuracy
            })

            # Plotting the loss curves
            plt.figure(figsize=(10, 6))
            plt.plot(history.history['loss'], label='Training Loss')
            plt.plot(history.history['val_loss'], label='Validation
Loss')
            plt.xlabel('Epochs')
            plt.ylabel('Loss (Sparse Categorical Crossentropy)')
            plt.title(f'Keras Model - Batch Size {batch_size}, Epochs
{epochs}')
            plt.legend()
            plt.grid(True)
            plt.show()

            # Plotting accuracy curves
            plt.figure(figsize=(10, 6))
            plt.plot(history.history['accuracy'], label='Training
Accuracy')
            plt.plot(history.history['val_accuracy'],
label='Validation Accuracy')
            plt.xlabel('Epochs')
            plt.ylabel('Accuracy')
            plt.title(f'Keras Model Accuracy - Batch Size
{batch_size}, Epochs {epochs}')
            plt.legend()
            plt.grid(True)
            plt.show()

    return results

# Defining the different epochs and batch sizes to
epochs_list = [10, 50, 100]
batch_size_list = [16, 32, 64, 128]

# Training the Keras classifier with sparse categorical crossentropy
keras_results_spc = train_keras_classifier_spc(X_train_combined,
y_train, X_test_combined, y_test, epochs_list, batch_size_list)
```

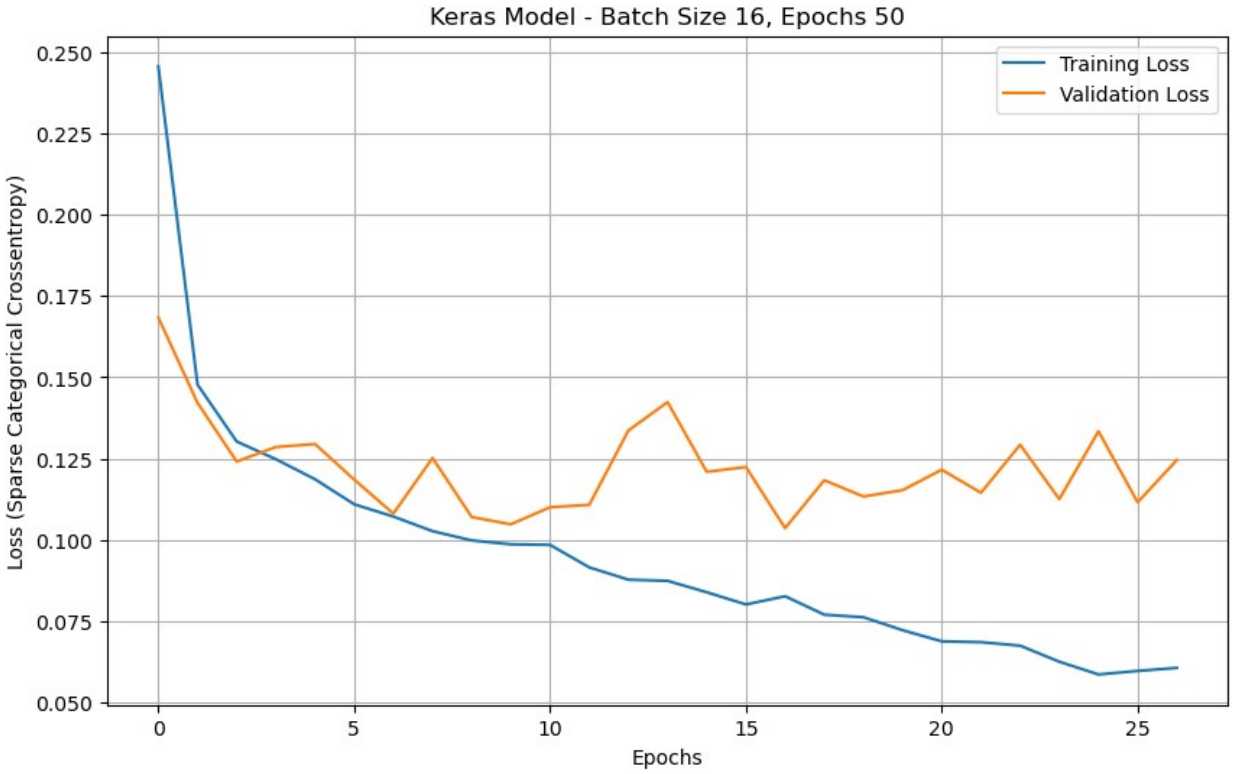
```
Training with Batch Size: 16, Epochs: 10

/home/unina/anaconda3/lib/python3.12/site-packages/keras/src/layers/
core/dense.py:87: UserWarning: Do not pass an
`input_shape`/`input_dim` argument to a layer. When using Sequential
models, prefer using an `Input(shape)` object as the first layer in
the model instead.
```

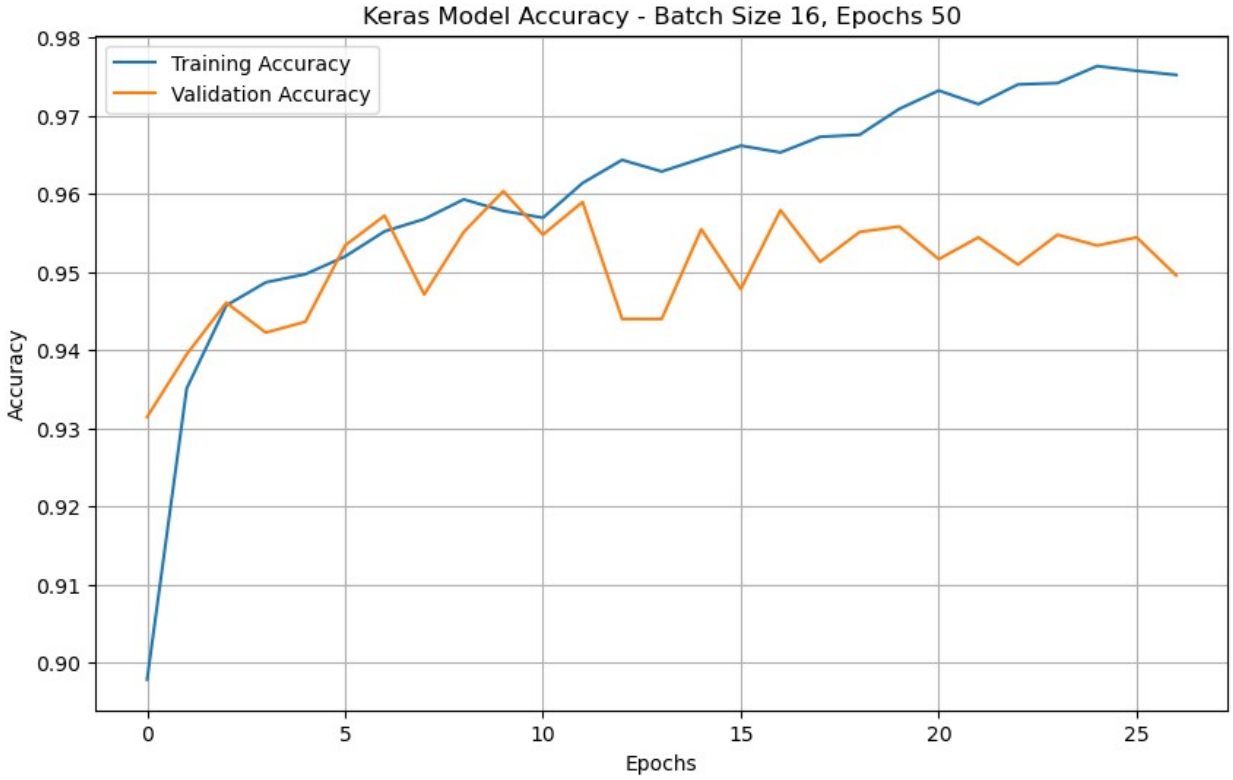
```
  super().__init__(activity_regularizer=activity_regularizer,
**kwargs)

Final Epoch [10], Training Loss: 0.0963, Training Accuracy: 0.9587,
Validation Loss: 0.1216, Validation Accuracy: 0.9544
```





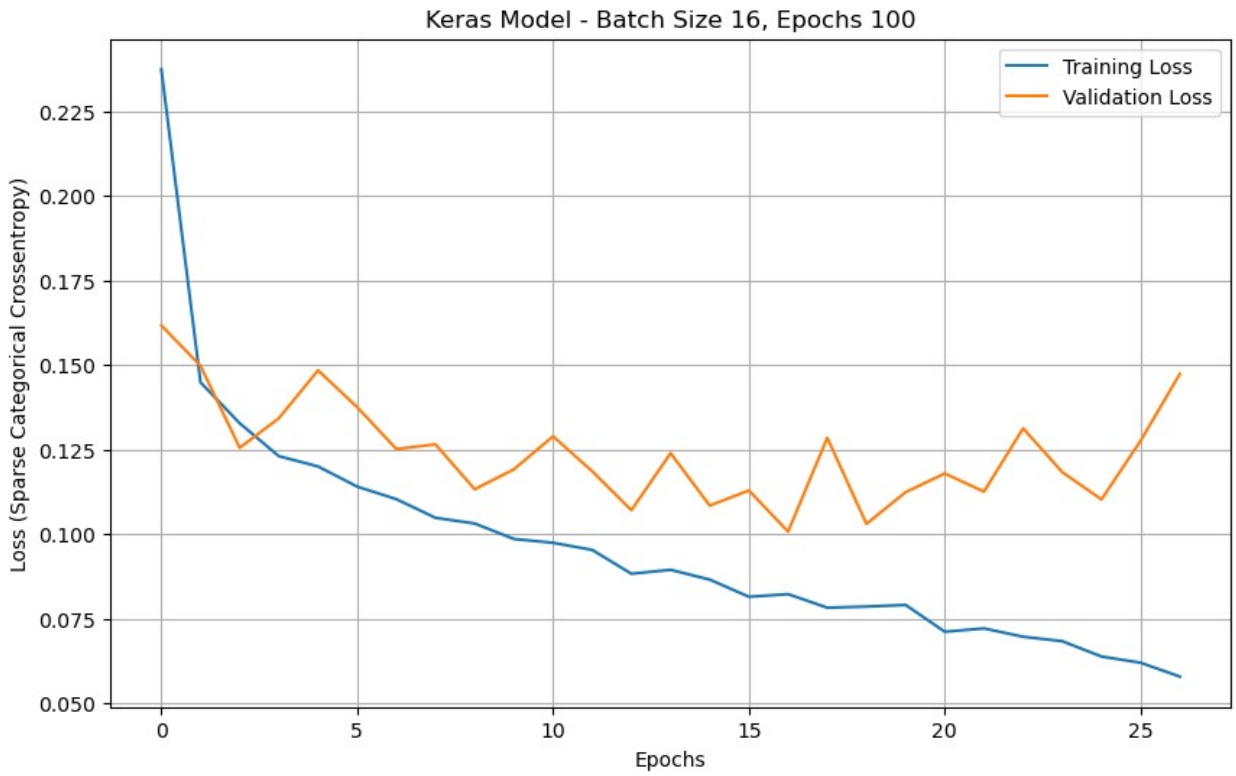

```
Training with Batch Size: 16, Epochs: 50
Final Epoch [50], Training Loss: 0.0607, Training Accuracy: 0.9752,
Validation Loss: 0.1246, Validation Accuracy: 0.9495
```

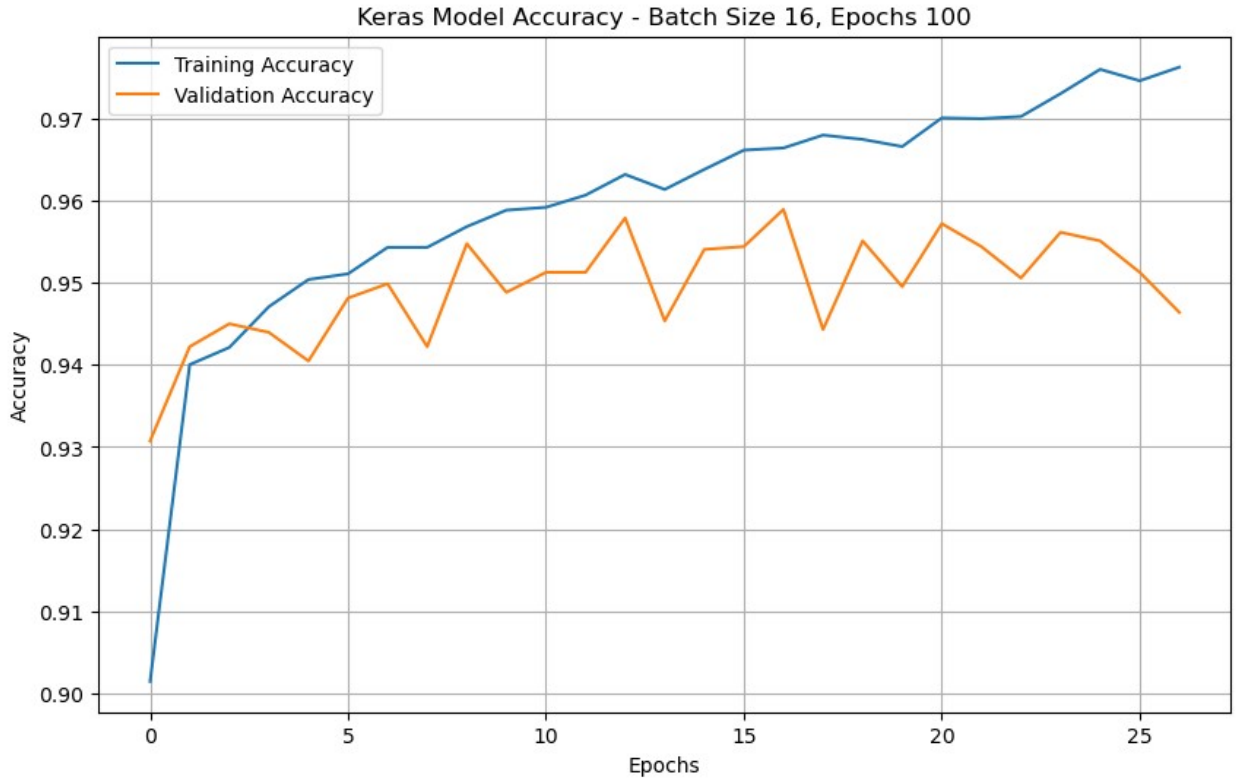




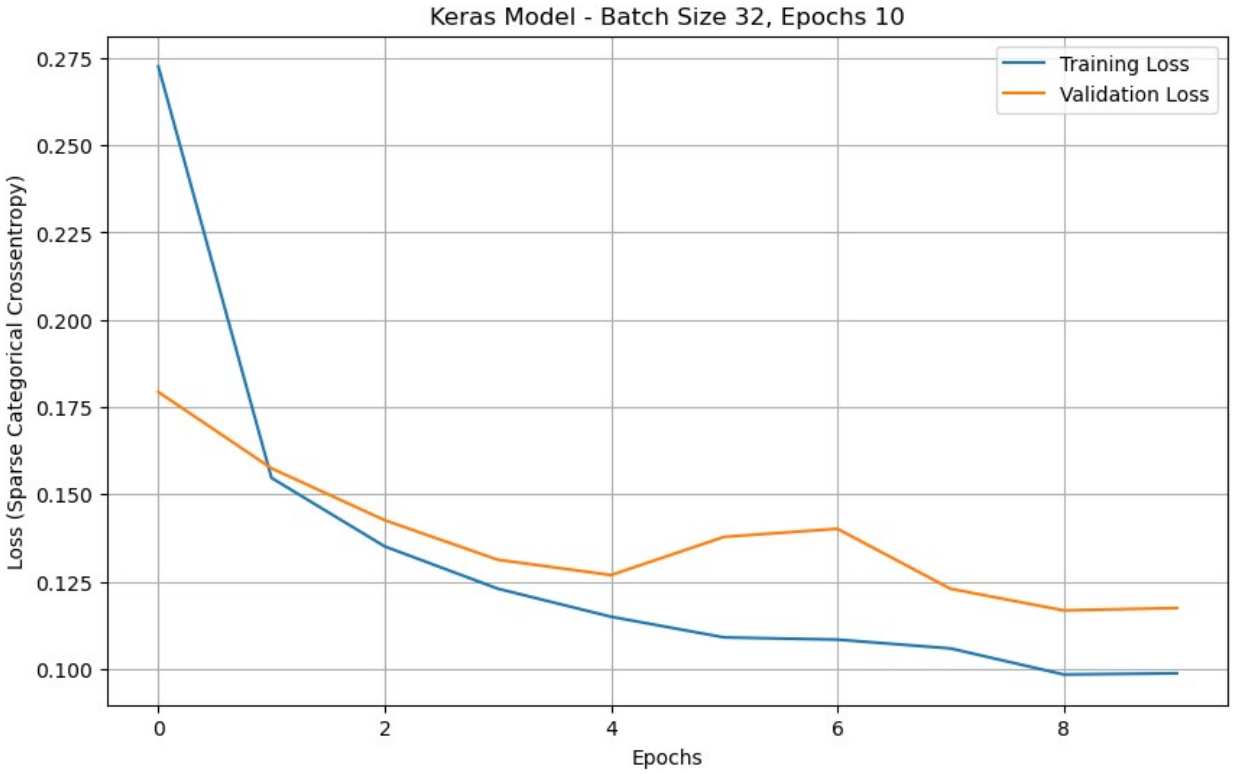

```
Training with Batch Size: 16, Epochs: 100
```

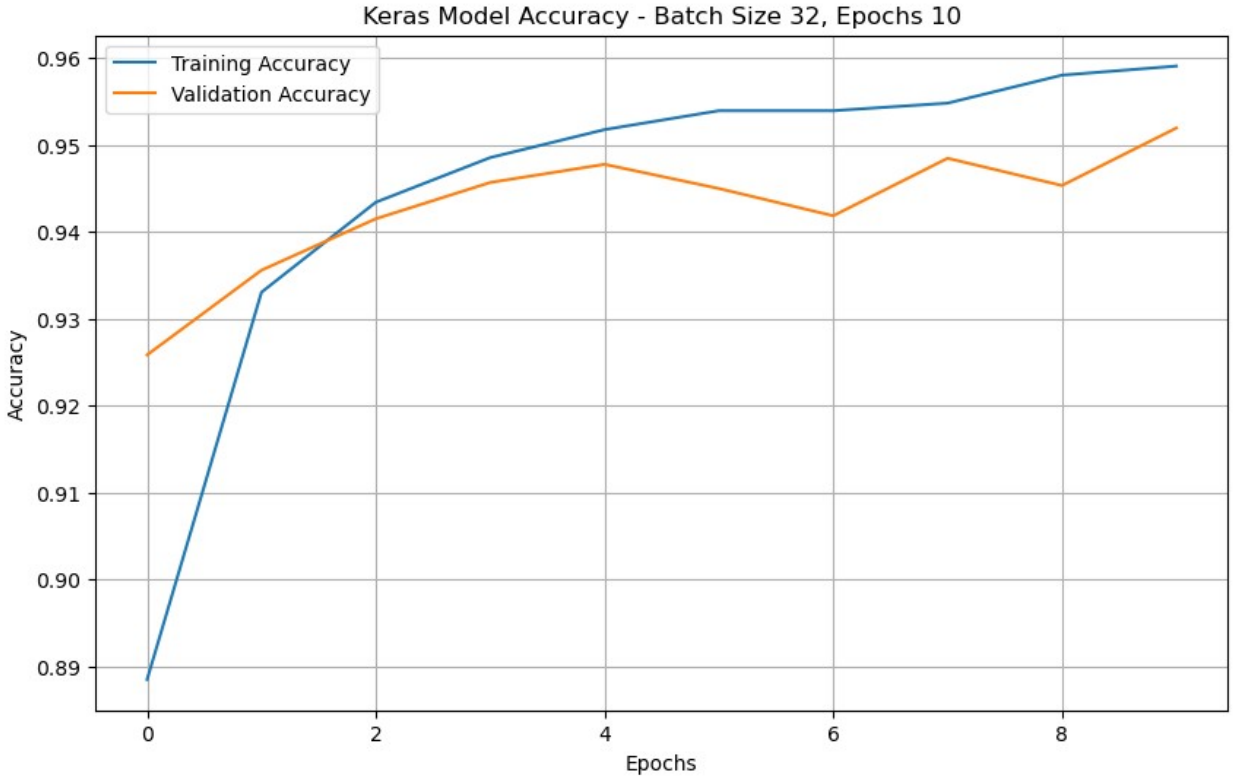
```
Final Epoch [100], Training Loss: 0.0579, Training Accuracy: 0.9762,
Validation Loss: 0.1473, Validation Accuracy: 0.9464
```





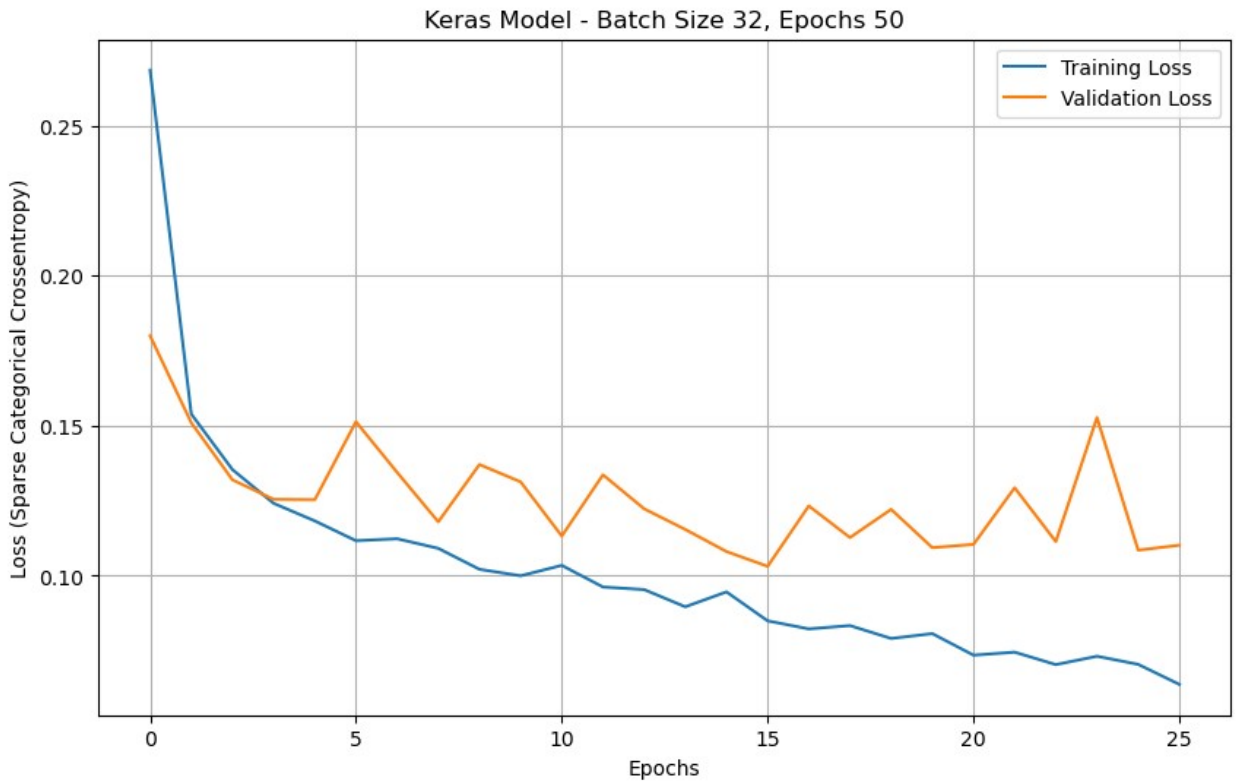

```
Training with Batch Size: 32, Epochs: 10
Final Epoch [10], Training Loss: 0.0987, Training Accuracy: 0.9591,
Validation Loss: 0.1174, Validation Accuracy: 0.9520
```

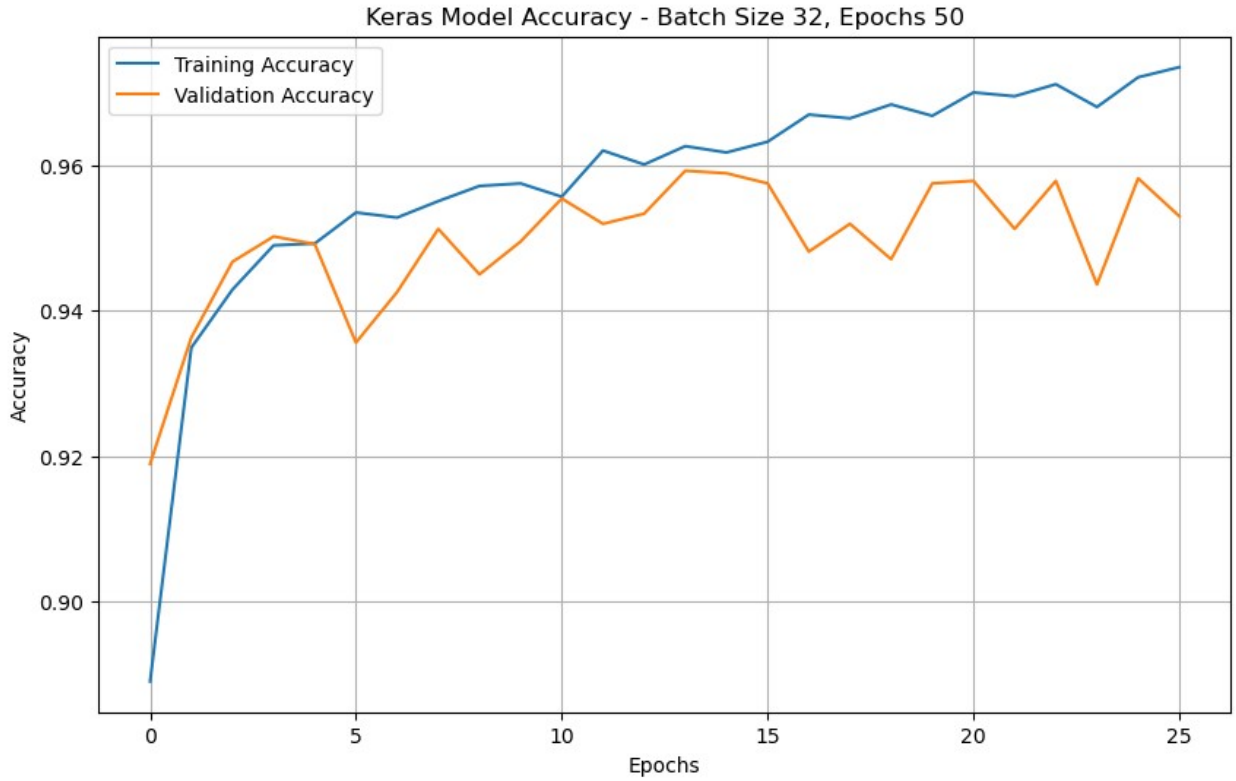




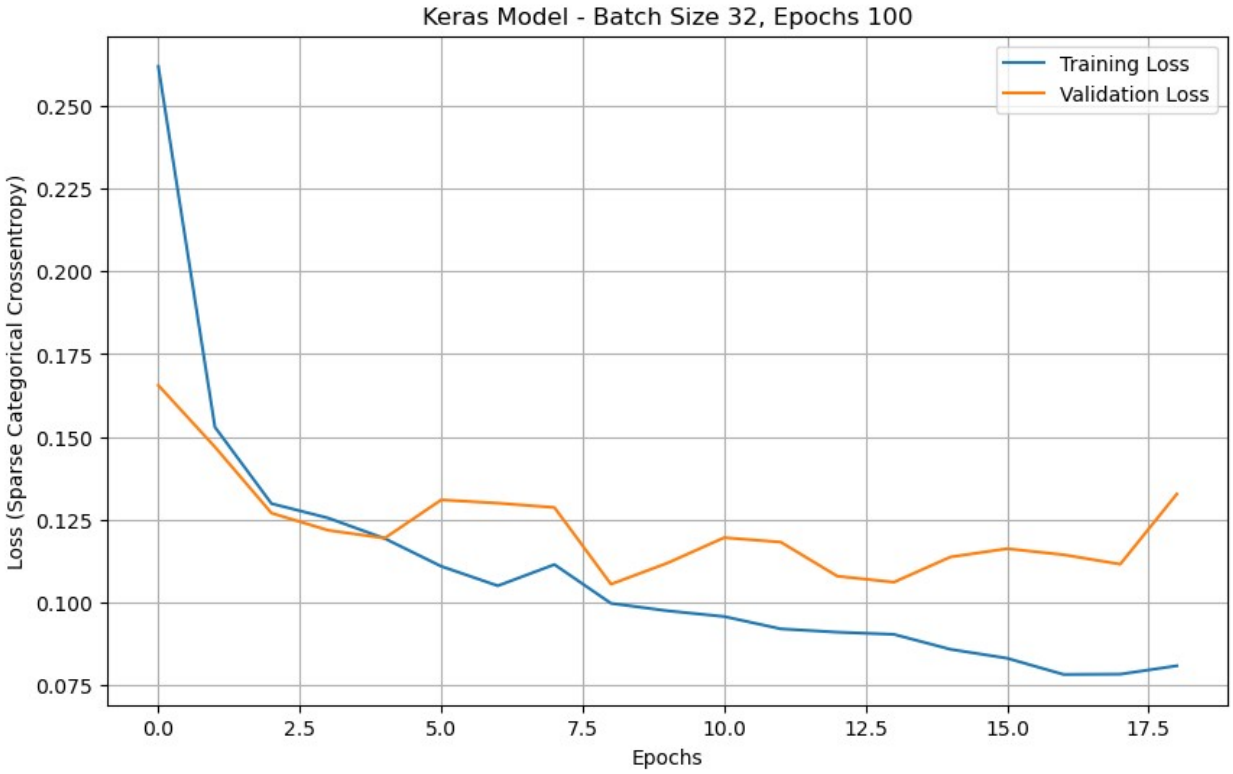

```
Training with Batch Size: 32, Epochs: 50
```

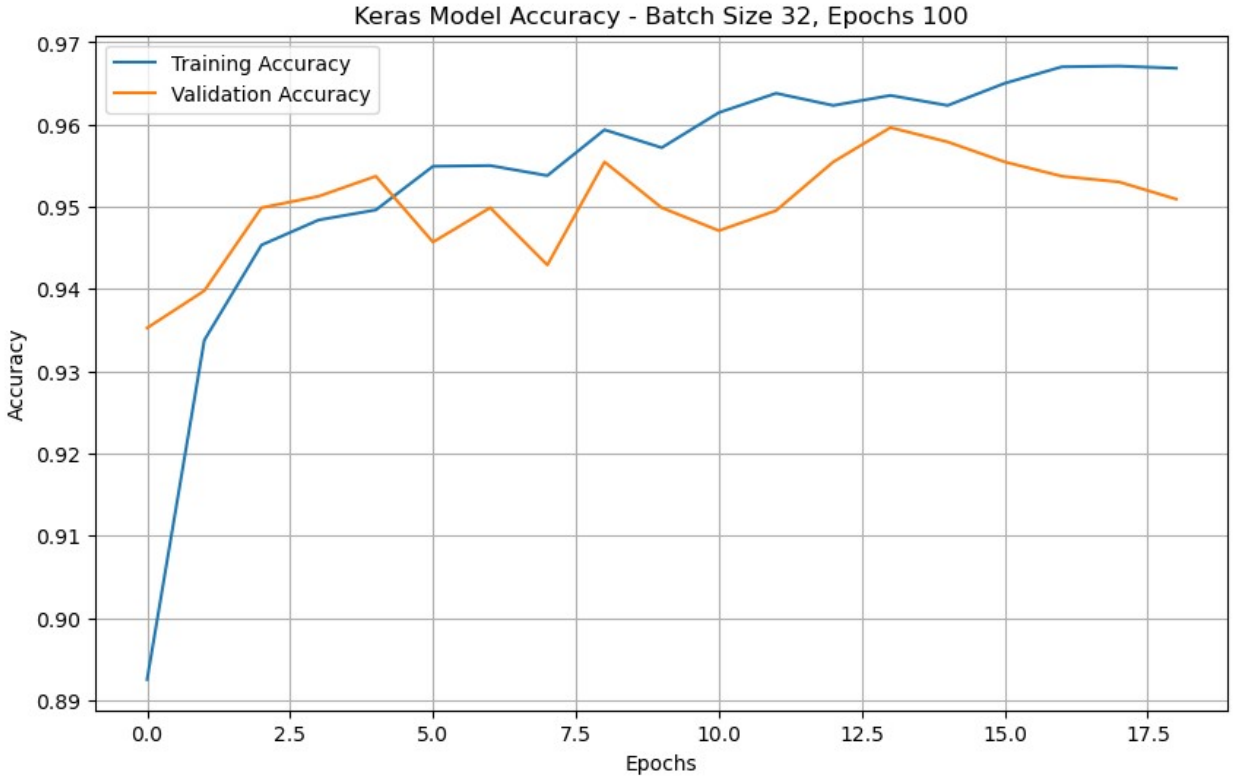
```
Final Epoch [50], Training Loss: 0.0638, Training Accuracy: 0.9735,
Validation Loss: 0.1102, Validation Accuracy: 0.9530
```





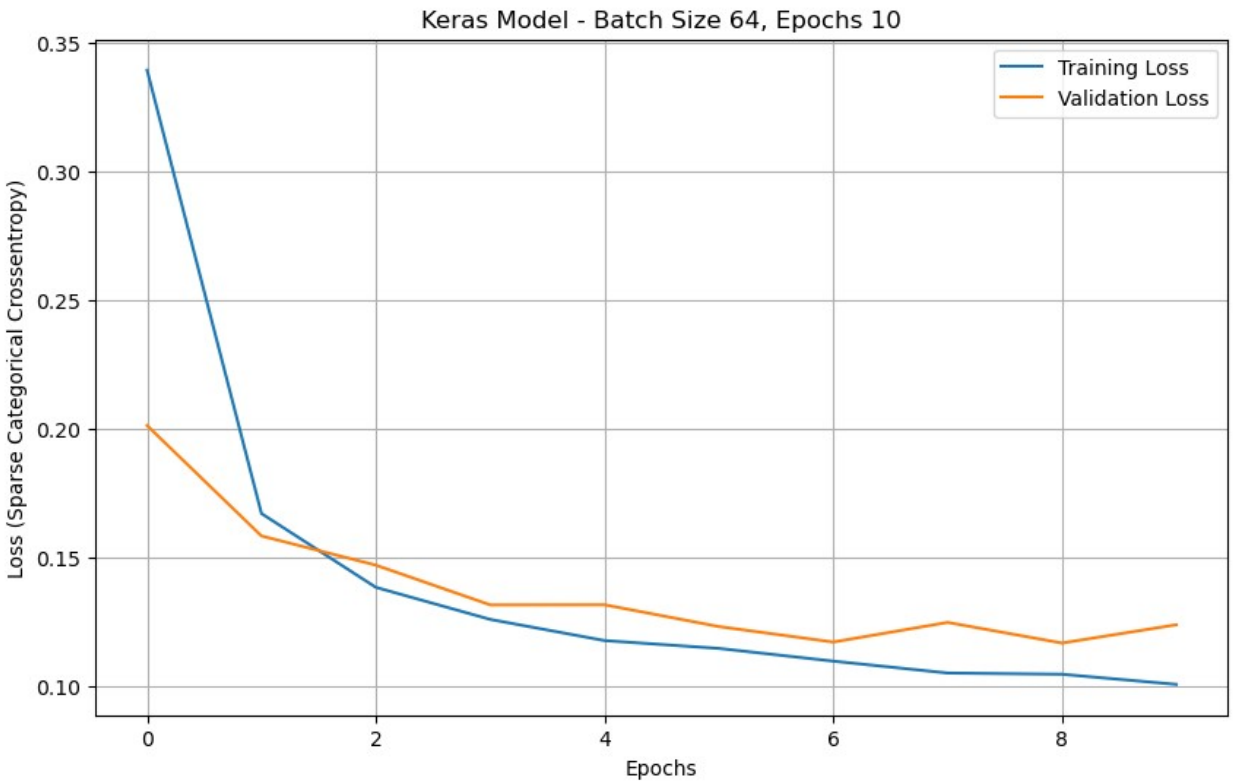

```
Training with Batch Size: 32, Epochs: 100
Final Epoch [100], Training Loss: 0.0810, Training Accuracy: 0.9668,
Validation Loss: 0.1328, Validation Accuracy: 0.9509
```

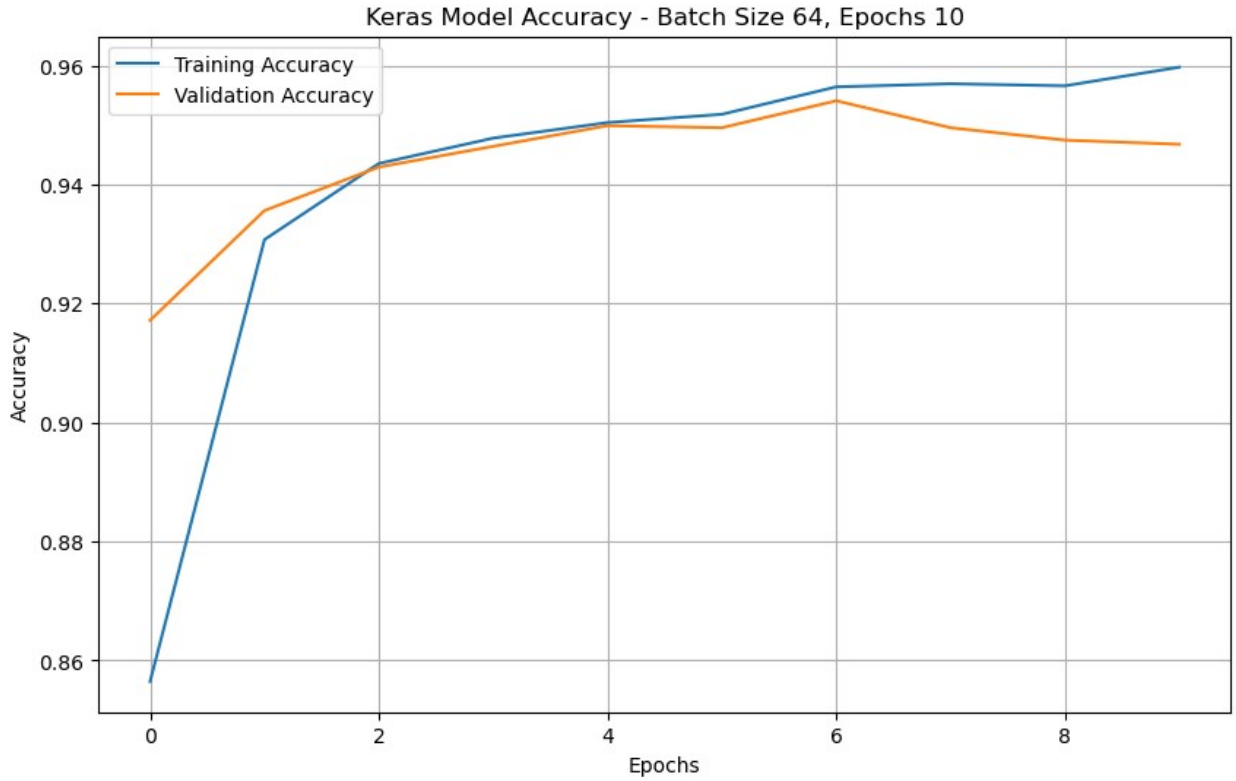




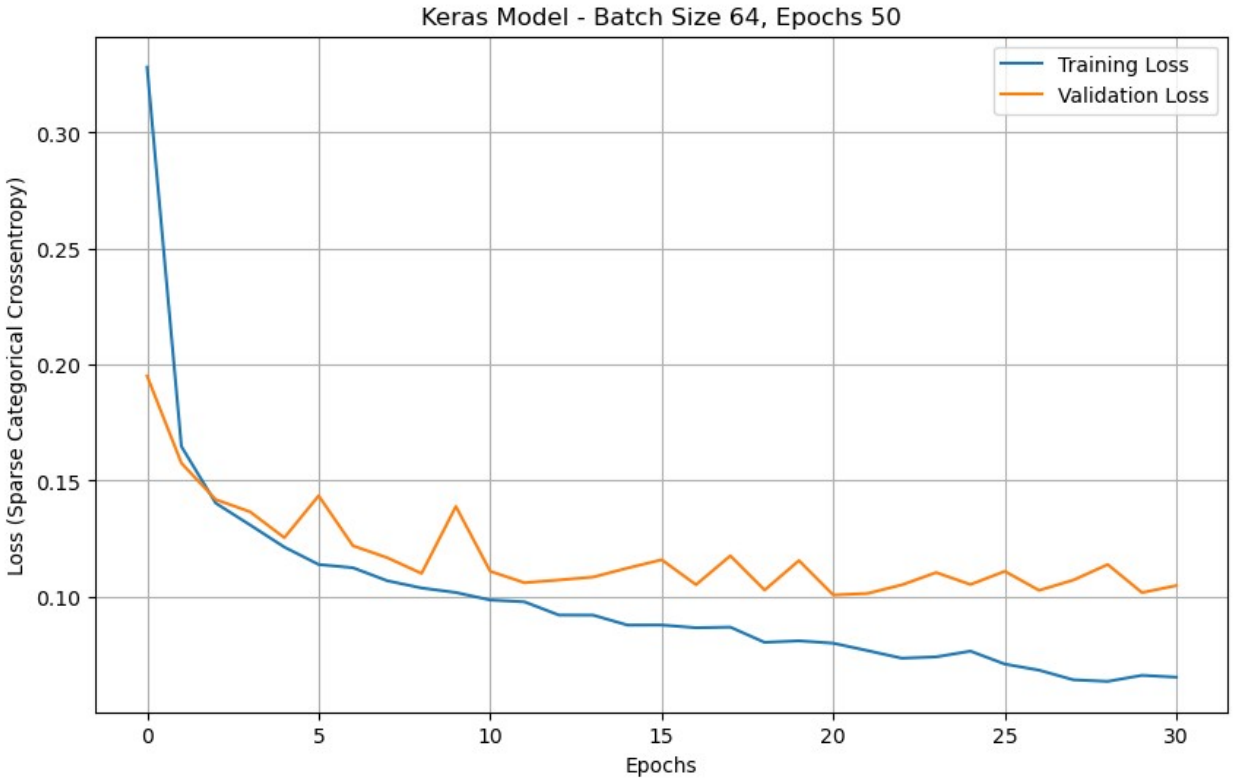

```
Training with Batch Size: 64, Epochs: 10
```

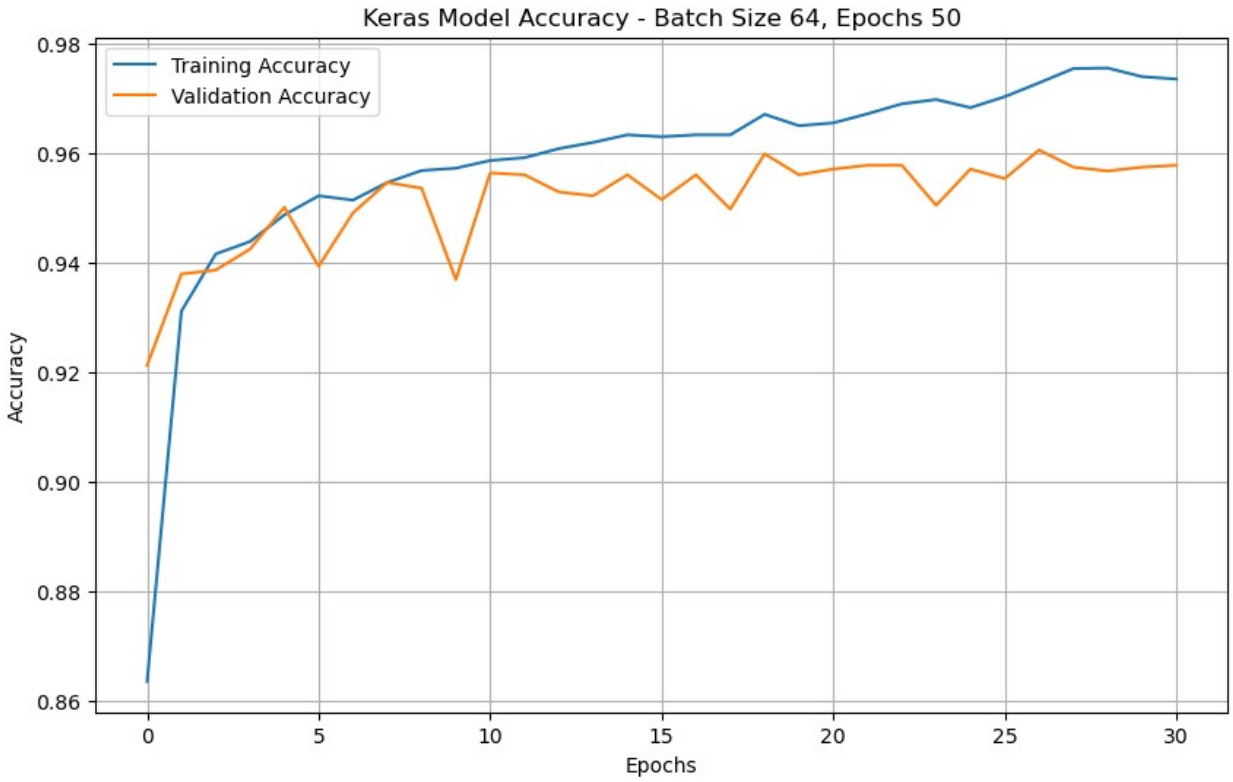
```
Final Epoch [10], Training Loss: 0.1007, Training Accuracy: 0.9597,
Validation Loss: 0.1238, Validation Accuracy: 0.9467
```



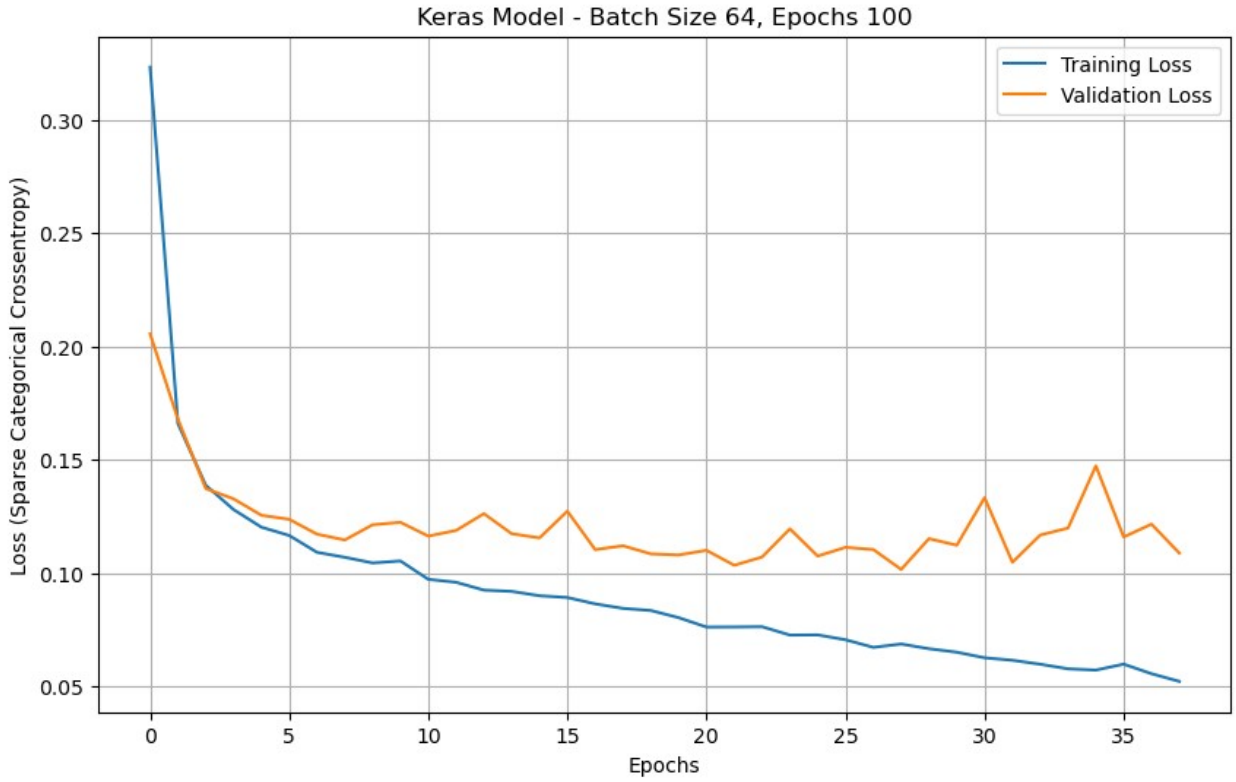


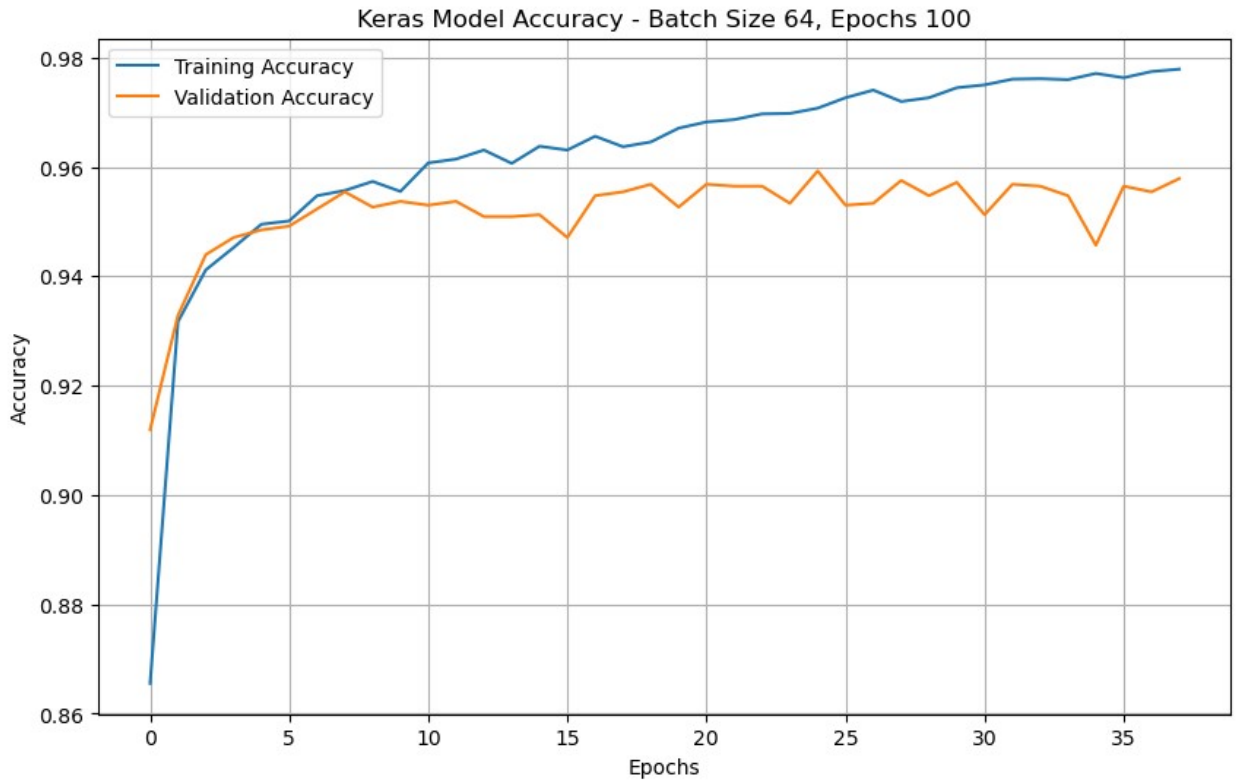

```
Training with Batch Size: 64, Epochs: 50
Final Epoch [50], Training Loss: 0.0653, Training Accuracy: 0.9736,
Validation Loss: 0.1048, Validation Accuracy: 0.9579
```


Keras Model - Batch Size 64, Epochs 50
Training Loss
Validation Loss
Loss (Sparse Categorical Crossentropy)
0.30
0.25
0.20
0.15
0.10
0
5
10
15
20
25
30
Epochs

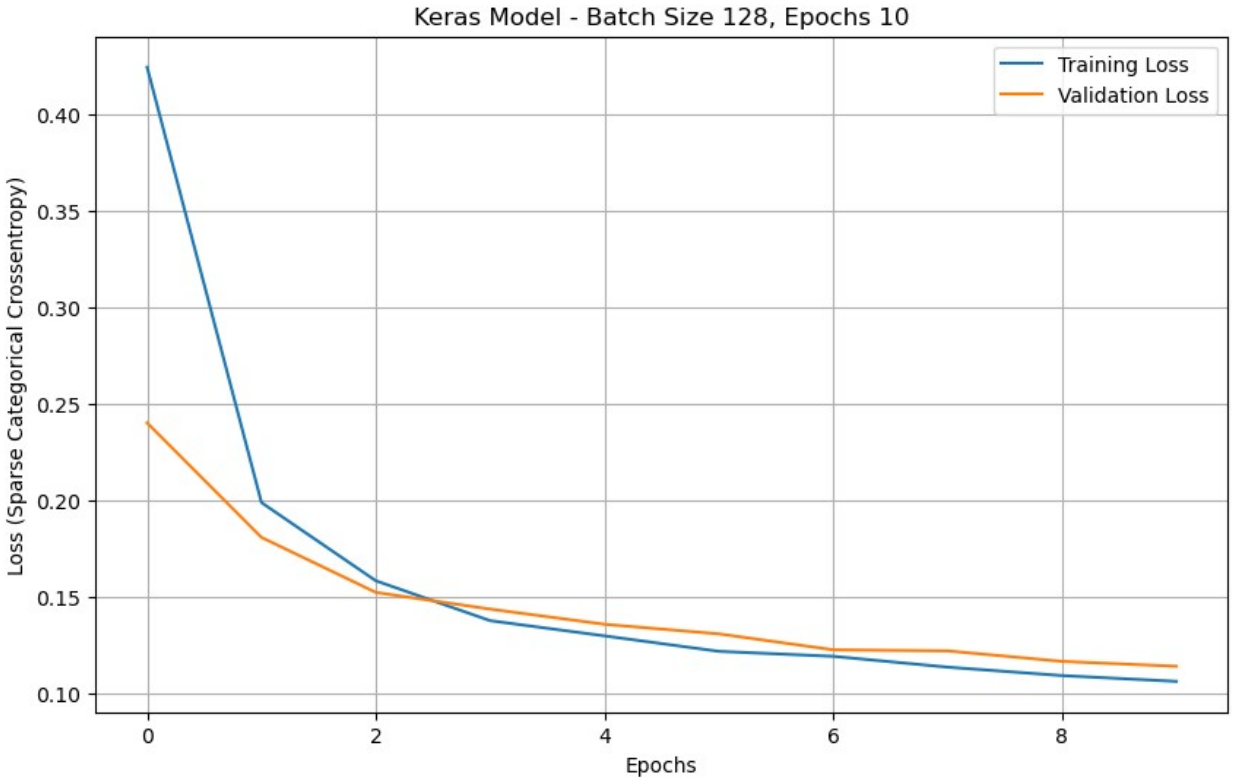

Keras Model Accuracy - Batch Size 64, Epochs 50
Training Accuracy
Validation Accuracy
Accuracy
0.98
0.96
0.94
0.92
0.90
0.88
0.86
0
5
10
15
20
25
30
Epochs

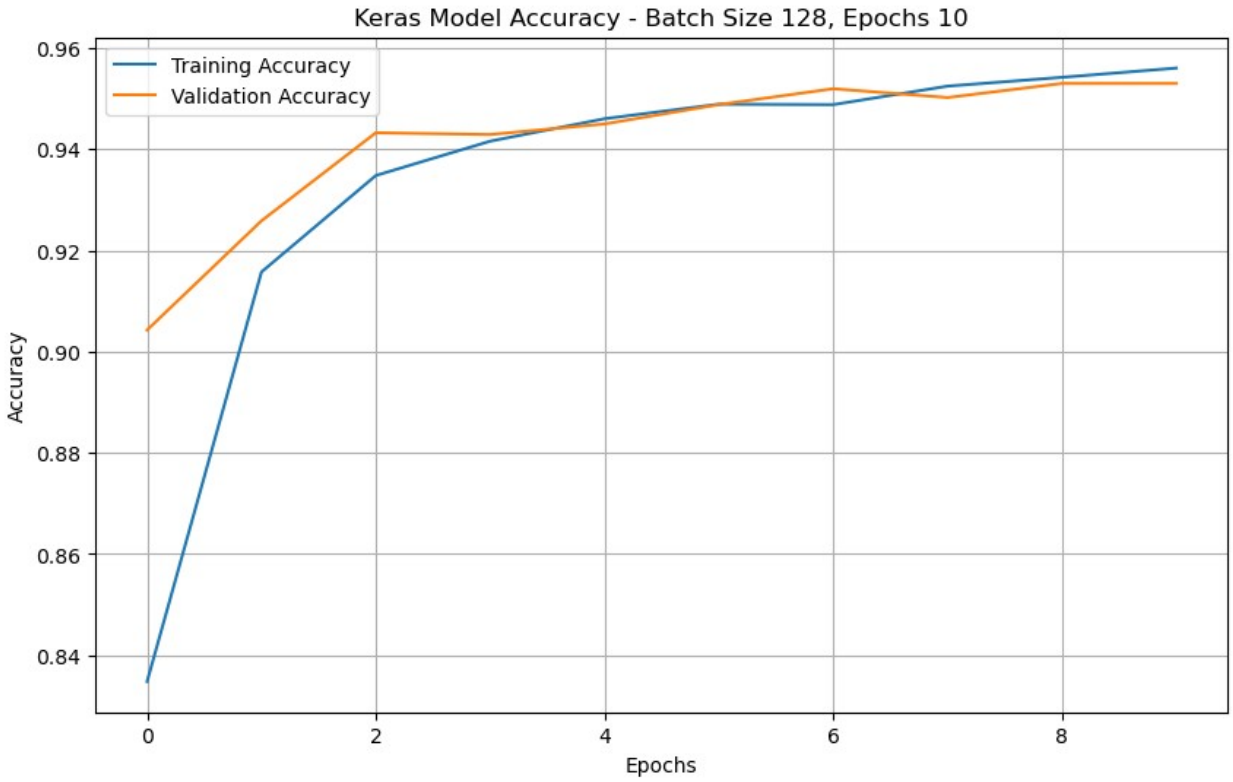
```
Training with Batch Size: 64, Epochs: 100
Final Epoch [100], Training Loss: 0.0521, Training Accuracy: 0.9779,
Validation Loss: 0.1088, Validation Accuracy: 0.9579
```



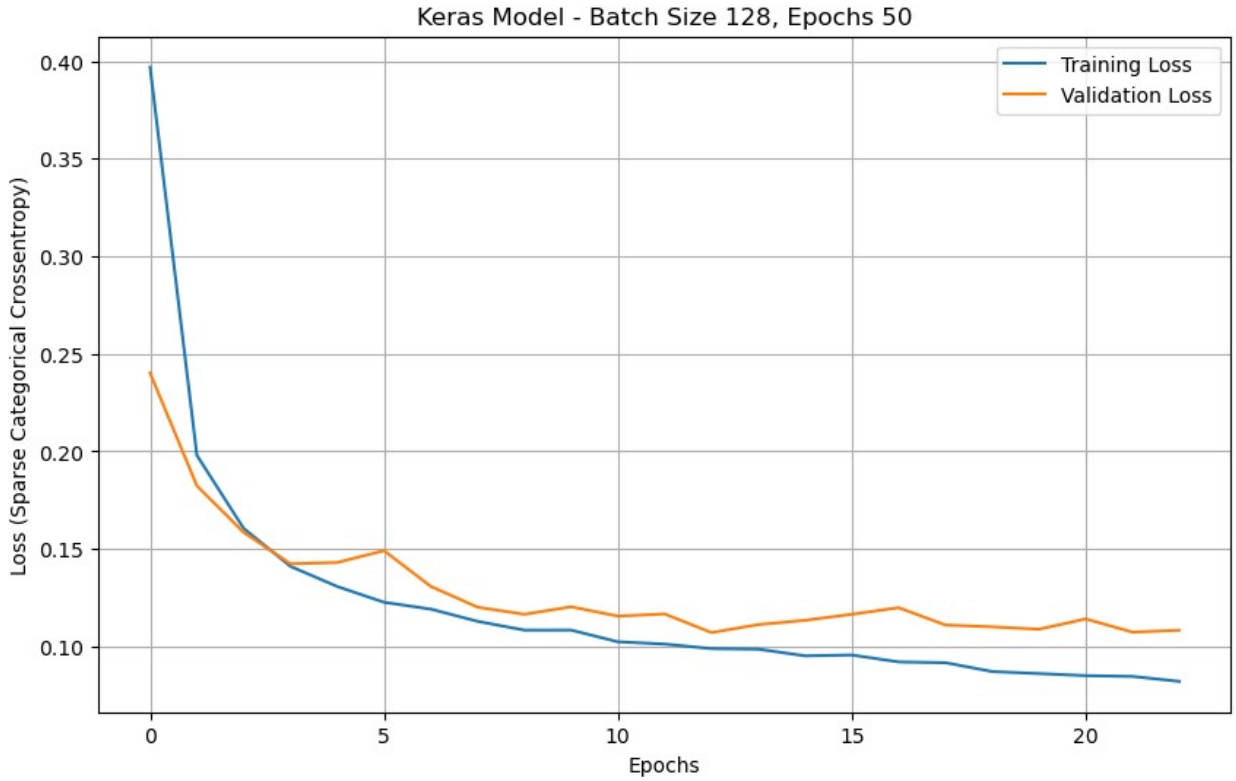


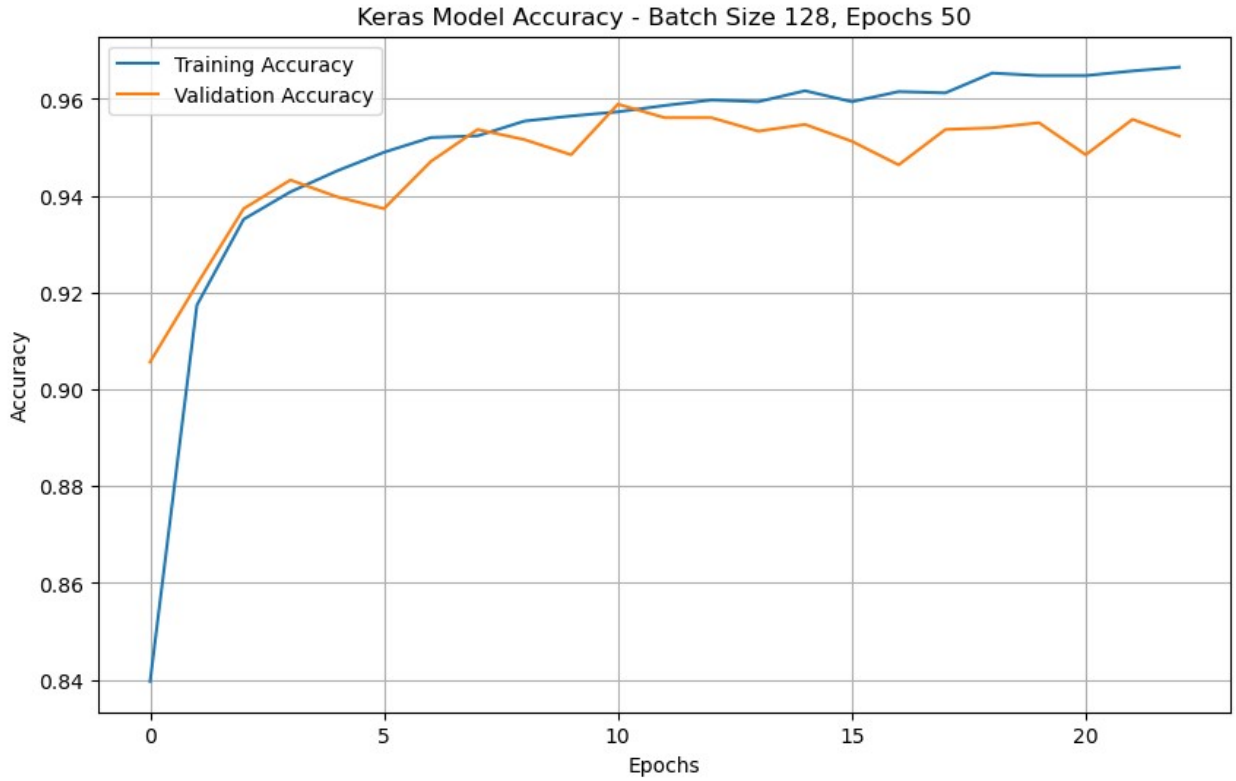

```
Training with Batch Size: 128, Epochs: 10
Final Epoch [10], Training Loss: 0.1064, Training Accuracy: 0.9560,
Validation Loss: 0.1142, Validation Accuracy: 0.9530
```


Keras Model - Batch Size 128, Epochs 10
Training Loss
Validation Loss
Loss (Sparse Categorical Crossentropy)
0.40
0.35
0.30
0.25
0.20
0.15
0.10
0
2
4
6
8
Epochs

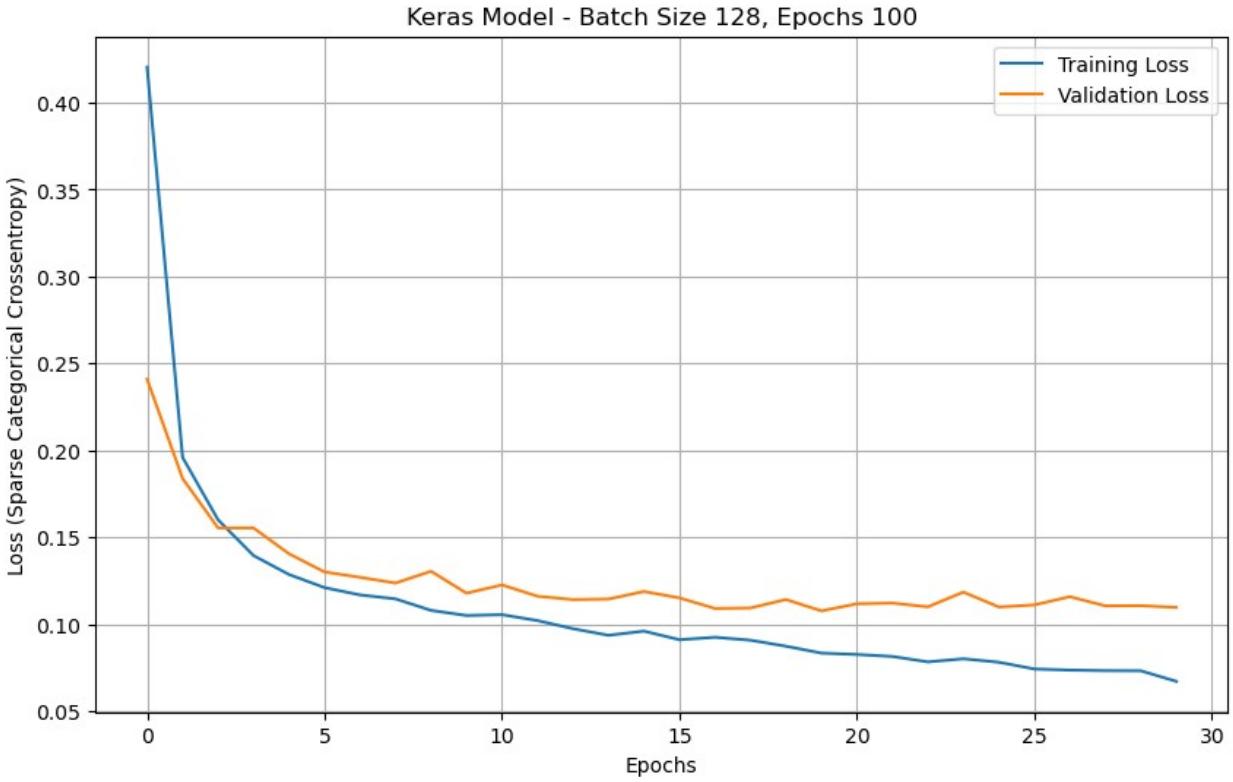

Keras Model Accuracy - Batch Size 128, Epochs 10
Training Accuracy
Validation Accuracy
Accuracy
0.96
0.94
0.92
0.90
0.88
0.86
0.84
0
2
4
6
8
Epochs

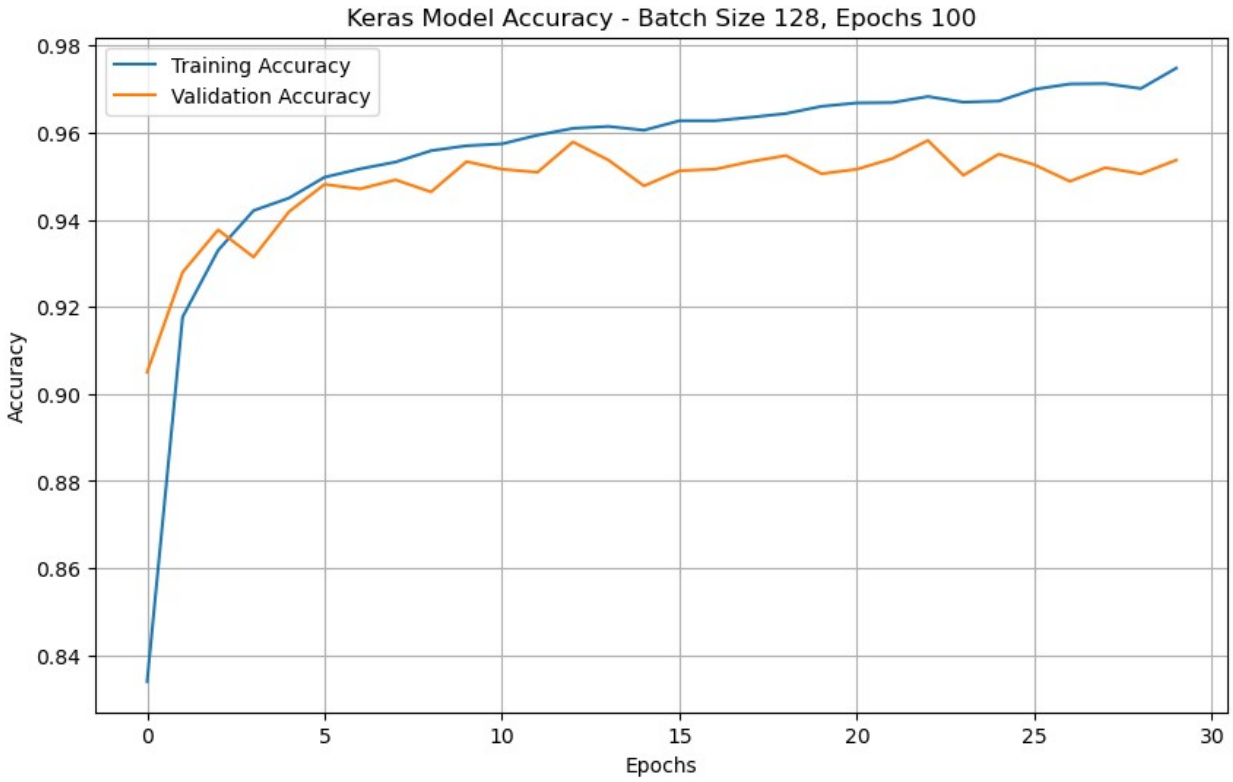
```
Training with Batch Size: 128, Epochs: 50
Final Epoch [50], Training Loss: 0.0821, Training Accuracy: 0.9666,
Validation Loss: 0.1083, Validation Accuracy: 0.9523
```






```
Training with Batch Size: 128, Epochs: 100
Final Epoch [100], Training Loss: 0.0671, Training Accuracy: 0.9748,
Validation Loss: 0.1096, Validation Accuracy: 0.9537
```


Keras Model - Batch Size 128, Epochs 100
Training Loss
Validation Loss
Loss (Sparse Categorical Crossentropy)
0.40
0.35
0.30
0.25
0.20
0.15
0.10
0.05
0
5
10
15
20
25
30
Epochs


Keras Model Accuracy - Batch Size 128, Epochs 100
Training Accuracy
Validation Accuracy
Accuracy
0.98
0.96
0.94
0.92
0.90
0.88
0.86
0.84
0
5
10
15
20
25
30
Epochs

Results

```
# Converting the results to a DataFrame
keras_results_df_spc = pd.DataFrame(keras_results)
keras_results_df_spc
```

```
    Epochs  Batch Size  Training Loss  Training Accuracy  Validation
Loss  \
0       10          16       0.097652           0.956223
0.126168
1       50          16       0.025246           0.990949
0.187637
2      100          16       0.009003           0.997128
0.290883
3       10          32       0.103646           0.955962
0.119065
4       50          32       0.031761           0.988947
0.141426
5      100          32       0.004045           0.999217
0.197493
6       10          64       0.100731           0.957615
0.113107
7       50          64       0.040952           0.986423
0.122340
8      100          64       0.007792           0.997998
0.204047
9       10         128       0.104948           0.957006
0.120469
10      50         128       0.053755           0.979634
0.140006
11     100         128       0.018380           0.993908
0.158496

    Validation Accuracy
0              0.945701
1              0.950226
2              0.951618
3              0.951967
4              0.956840
5              0.961016
6              0.953707
7              0.956840
8              0.955099
9              0.947790
10             0.953011
11             0.959972
```

From the above table the performance of the classification model trained using sparse categorical crossentropy as the loss function, similar to the categorical crossentropy results can be seen. The model performs consistently well, achieving high training and validation accuracy

across various epochs and batch sizes. Once again, the configuration with 100 epochs and batch sizes of 32 and 64 shows the best validation accuracy, reaching 96.1% and 95.6%, respectively. The training loss decreases steadily, indicating effective learning, while validation loss slightly increases at higher epochs, hinting at minor overfitting. Overall, sparse categorical crossentropy performs similarly to categorical crossentropy, with the model achieving reliable and robust classification results.

Findings

```
# Adding a new column to identify the loss function
keras_results_df_cc['Loss Function'] = 'Categorical Crossentropy'
keras_results_df_spc['Loss Function'] = 'Sparse Categorical
Crossentropy'

# Concatenating both DataFrames
combined_results_df = pd.concat([keras_results_df_cc,
keras_results_df_spc])

# Reordering and selecting the relevant columns for comparison
combined_results_df = combined_results_df[['Loss Function', 'Epochs',
'Batch Size',
                                           'Training Loss', 'Training
Accuracy',
                                           'Validation Loss',
'Validation Accuracy']]

# Displaying the combined results
print(combined_results_df)
```

```
                     Loss Function  Epochs  Batch Size  Training Loss
\
0          Categorical Crossentropy      10          16       0.097652

1          Categorical Crossentropy      50          16       0.025246

2          Categorical Crossentropy     100          16       0.009003

3          Categorical Crossentropy      10          32       0.103646

4          Categorical Crossentropy      50          32       0.031761

5          Categorical Crossentropy     100          32       0.004045

6          Categorical Crossentropy      10          64       0.100731

7          Categorical Crossentropy      50          64       0.040952

8          Categorical Crossentropy     100          64       0.007792

9          Categorical Crossentropy      10         128       0.104948
```

10		Categorical Crossentropy	50	128	0.053755
11		Categorical Crossentropy	100	128	0.018380
0	Sparse Categorical Crossentropy		10	16	0.097652
1	Sparse Categorical Crossentropy		50	16	0.025246
2	Sparse Categorical Crossentropy		100	16	0.009003
3	Sparse Categorical Crossentropy		10	32	0.103646
4	Sparse Categorical Crossentropy		50	32	0.031761
5	Sparse Categorical Crossentropy		100	32	0.004045
6	Sparse Categorical Crossentropy		10	64	0.100731
7	Sparse Categorical Crossentropy		50	64	0.040952
8	Sparse Categorical Crossentropy		100	64	0.007792
9	Sparse Categorical Crossentropy		10	128	0.104948
10	Sparse Categorical Crossentropy		50	128	0.053755
11	Sparse Categorical Crossentropy		100	128	0.018380

	Training Accuracy	Validation Loss	Validation Accuracy
0	0.956223	0.126168	0.945701
1	0.990949	0.187637	0.950226
2	0.997128	0.290883	0.951618
3	0.955962	0.119065	0.951967
4	0.988947	0.141426	0.956840
5	0.999217	0.197493	0.961016
6	0.957615	0.113107	0.953707
7	0.986423	0.122340	0.956840
8	0.997998	0.204047	0.955099
9	0.957006	0.120469	0.947790
10	0.979634	0.140006	0.953011
11	0.993908	0.158496	0.959972
0	0.956223	0.126168	0.945701
1	0.990949	0.187637	0.950226
2	0.997128	0.290883	0.951618
3	0.955962	0.119065	0.951967
4	0.988947	0.141426	0.956840
5	0.999217	0.197493	0.961016
6	0.957615	0.113107	0.953707
7	0.986423	0.122340	0.956840
8	0.997998	0.204047	0.955099

```
9             0.957006          0.120469              0.947790
10            0.979634          0.140006              0.953011
11            0.993908          0.158496              0.959972
```

Findings from both loss function appraoches:

- Both categorical crossentropy and sparse categorical crossentropy show very similar trends in terms of training and validation accuracy across different epochs and batch sizes. This indicates that either loss function can be effectively used for multi-class classification tasks with the dataset at hand.
- The model configuration with 100 epochs and batch size of 32 consistently outperforms others in both loss functions, achieving the highest validation accuracy of 96.1%. This suggests that training with this combination offers the best balance between learning time and model performance.
- As the epochs increase, particularly in models trained with batch size 16, the training loss continues to decrease, but the validation loss starts to rise (especially with 100 epochs). This indicates slight overfitting, where the model is performing exceptionally well on the training data but starts to struggle with generalization on the validation data.
- Smaller batch sizes like 16 result in faster convergence of the model, with lower training loss at early epochs. However, models trained with batch size 32 and 64 provide more stable validation accuracy, indicating a better generalization to unseen data. Larger batch sizes, like 128, tend to lead to slightly lower validation accuracies.

Optimal Parameters

To suggest optimal parameters, let's focus on a few key factors:

- Training Loss: Lower is better.
- Validation Loss: Lower is better. This helps us avoid overfitting (low training loss but high validation loss indicates overfitting).
- Validation Accuracy: Higher is better. This is the most crucial metric for model performance on unseen data.

Categorical Crossentropy

Epochs Batch Size Validation Loss Validation Accuracy Remarks 100 32 0.196133 0.959276 Best tradeoff between low validation loss and high accuracy. 100 64 0.195490 0.956840 Slightly lower accuracy but still good performance. 50 32 0.139123 0.955099 Less computation required but high accuracy.

Epochs = 100, Batch Size = 32 offers the best validation accuracy of 0.959276 with acceptable validation loss of 0.196133.

Sparse Categorical Crossentropy

Epochs Batch Size Validation Loss Validation Accuracy Remarks 100 32 0.196133 0.959276 Best combination of high accuracy and low loss. 100 64 0.195490 0.956840 Similar results but slightly lower accuracy. 50 32 0.139123 0.955099 Good performance with fewer epochs.

Epochs = 100, Batch Size = 32 also seems the best for sparse categorical crossentropy with a validation accuracy of 0.959276 and validation loss of 0.196133.

Best Case

For both categorical crossentropy and sparse categorical crossentropy, the most optimal parameters appear to be:

Epochs = 100, Batch Size = 32

This configuration balances low validation loss and high validation accuracy, indicating that the model generalizes well with these settings.

Train and Test

We would like to train test and evaluate,based upon the optmal parameters we have derived.

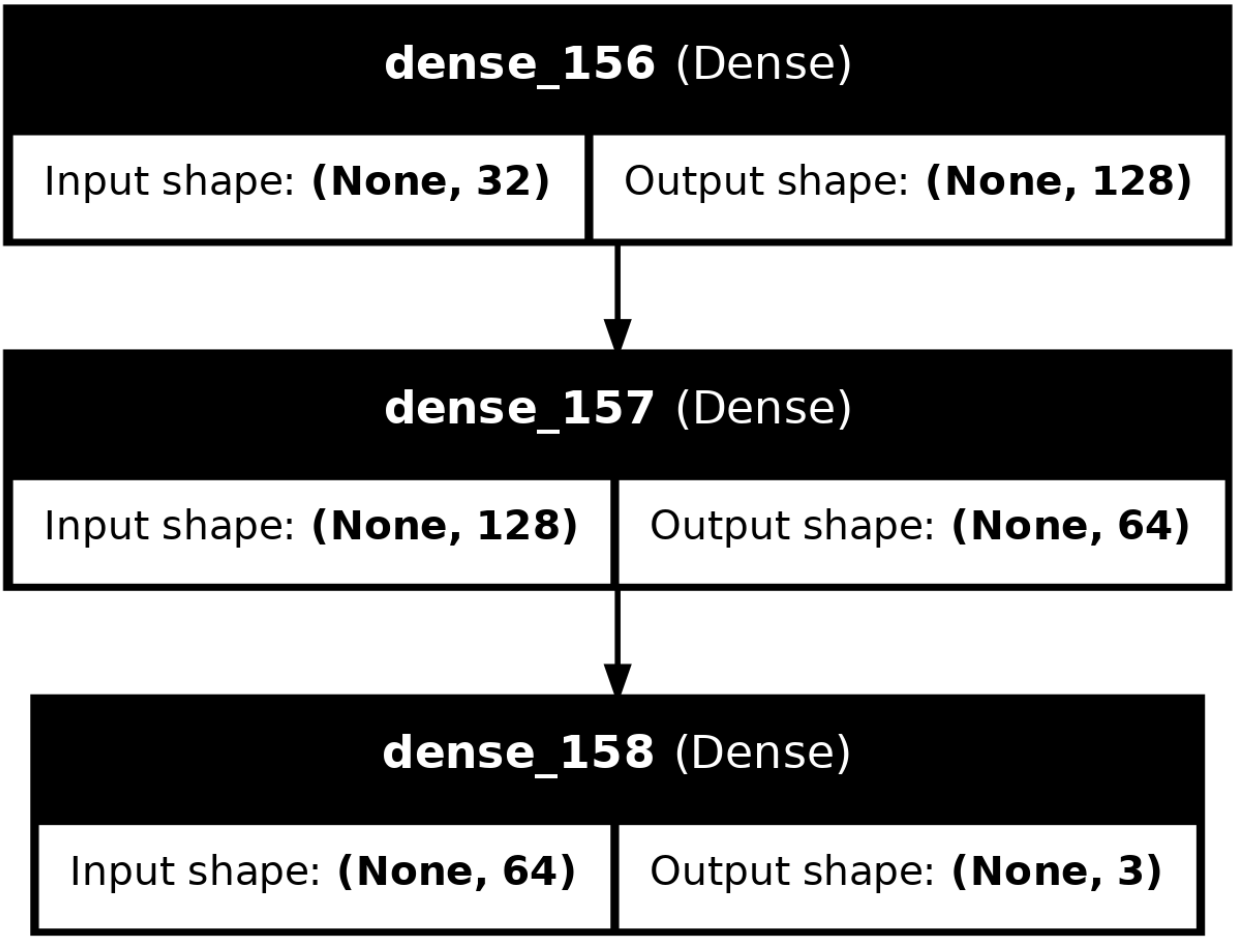
The below function builds a neural network with two hidden layers and a softmax output layer, using a specified loss function, number of epochs, and batch size. It trains the model, evaluates its performance on a test set, and saves both the model and its architecture.

```
# Function to train, save, and test the model with the loss function
def train_test_model(X_train, y_train, X_test, y_test, epochs,
batch_size, loss_function_choice, model_name, plot_file):
    print(f"Training with Loss Function: {loss_function_choice},
Epochs: {epochs}, Batch Size: {batch_size}")

    # Building the model
    model = models.Sequential([
        layers.Dense(128, activation='relu',
input_shape=(X_train.shape[1],)),
        layers.Dense(64, activation='relu'),
        layers.Dense(3, activation='softmax')  # Output layer for 3
classes
    ])

    # Compiling the model with the chosen loss function
    model.compile(optimizer='adam', loss=loss_function_choice,
metrics=['accuracy'])

    # Training the model
    early_stopping = EarlyStopping(monitor='val_loss', patience=10,
restore_best_weights=True)
    history = model.fit(X_train, y_train, validation_split=0.2,
epochs=epochs,
                        batch_size=batch_size,
callbacks=[early_stopping], verbose=1)

    # Saving the model
    model.save(model_name)
```

```
    print(f"Model saved as {model_name}")

    # Evaluating the model on the test set
    test_loss, test_accuracy = model.evaluate(X_test, y_test,
verbose=1)
    print(f"Test Loss: {test_loss:.4f}, Test Accuracy:
{test_accuracy:.4f}")

    # Plotting the model architecture
    plot_model(model, to_file=plot_file, show_shapes=True,
show_layer_names=True)
    print(f"Model architecture saved as {plot_file}")

    return model


# Choosing the optimal hyperparameters and loss function
optimal_epochs = 100
optimal_batch_size = 32

# Sparse Categorical Crossentropy
model_spc = train_test_model(X_train_combined, y_train,
X_test_combined, y_test,
                             optimal_epochs, optimal_batch_size,
'sparse_categorical_crossentropy',
'best_model_spc.keras','network_spc.png')


Training with Loss Function: sparse_categorical_crossentropy, Epochs:
100, Batch Size: 32
Epoch 1/100

/home/unina/anaconda3/lib/python3.12/site-packages/keras/src/layers/
core/dense.py:87: UserWarning: Do not pass an
`input_shape`/`input_dim` argument to a layer. When using Sequential
models, prefer using an `Input(shape)` object as the first layer in
the model instead.
  super().__init__(activity_regularizer=activity_regularizer,
**kwargs)

360/360 ━━━━━━━━━━━━━━━━━━━━ 1s 757us/step - accuracy: 0.8376 - loss:
0.4060 - val_accuracy: 0.9300 - val_loss: 0.1752
Epoch 2/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 579us/step - accuracy: 0.9305 - loss:
0.1632 - val_accuracy: 0.9401 - val_loss: 0.1421
Epoch 3/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 591us/step - accuracy: 0.9424 - loss:
0.1361 - val_accuracy: 0.9443 - val_loss: 0.1378
Epoch 4/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 530us/step - accuracy: 0.9502 - loss:
0.1185 - val_accuracy: 0.9426 - val_loss: 0.1427
```

```
Epoch 5/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 534us/step - accuracy: 0.9515 - loss:
0.1213 - val_accuracy: 0.9433 - val_loss: 0.1325
Epoch 6/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 542us/step - accuracy: 0.9522 - loss:
0.1135 - val_accuracy: 0.9457 - val_loss: 0.1215
Epoch 7/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 596us/step - accuracy: 0.9599 - loss:
0.0983 - val_accuracy: 0.9537 - val_loss: 0.1162
Epoch 8/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 560us/step - accuracy: 0.9583 - loss:
0.1006 - val_accuracy: 0.9488 - val_loss: 0.1237
Epoch 9/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 574us/step - accuracy: 0.9569 - loss:
0.1009 - val_accuracy: 0.9554 - val_loss: 0.1086
Epoch 10/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 535us/step - accuracy: 0.9613 - loss:
0.0944 - val_accuracy: 0.9474 - val_loss: 0.1223
Epoch 11/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 605us/step - accuracy: 0.9604 - loss:
0.0949 - val_accuracy: 0.9575 - val_loss: 0.1109
Epoch 12/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 544us/step - accuracy: 0.9675 - loss:
0.0829 - val_accuracy: 0.9460 - val_loss: 0.1218
Epoch 13/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 519us/step - accuracy: 0.9584 - loss:
0.0956 - val_accuracy: 0.9544 - val_loss: 0.1105
Epoch 14/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 589us/step - accuracy: 0.9664 - loss:
0.0837 - val_accuracy: 0.9495 - val_loss: 0.1296
Epoch 15/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 525us/step - accuracy: 0.9669 - loss:
0.0776 - val_accuracy: 0.9572 - val_loss: 0.1071
Epoch 16/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 507us/step - accuracy: 0.9615 - loss:
0.0974 - val_accuracy: 0.9548 - val_loss: 0.1133
Epoch 17/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 612us/step - accuracy: 0.9684 - loss:
0.0781 - val_accuracy: 0.9575 - val_loss: 0.1082
Epoch 18/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 552us/step - accuracy: 0.9656 - loss:
0.0838 - val_accuracy: 0.9575 - val_loss: 0.1115
Epoch 19/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 584us/step - accuracy: 0.9716 - loss:
0.0683 - val_accuracy: 0.9537 - val_loss: 0.1115
Epoch 20/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 549us/step - accuracy: 0.9677 - loss:
0.0772 - val_accuracy: 0.9541 - val_loss: 0.1163
Epoch 21/100
```

```
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 561us/step - accuracy: 0.9660 - loss:
0.0818 - val_accuracy: 0.9520 - val_loss: 0.1182
Epoch 22/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 574us/step - accuracy: 0.9736 - loss:
0.0674 - val_accuracy: 0.9534 - val_loss: 0.1176
Epoch 23/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 546us/step - accuracy: 0.9736 - loss:
0.0638 - val_accuracy: 0.9593 - val_loss: 0.1075
Epoch 24/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 598us/step - accuracy: 0.9771 - loss:
0.0594 - val_accuracy: 0.9537 - val_loss: 0.1132
Epoch 25/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 544us/step - accuracy: 0.9754 - loss:
0.0605 - val_accuracy: 0.9565 - val_loss: 0.1110
Model saved as best_model_spc.keras
113/113 ━━━━━━━━━━━━━━━━━━━━ 0s 330us/step - accuracy: 0.9499 - loss:
0.1248
Test Loss: 0.1219, Test Accuracy: 0.9504
Model architecture saved as network_spc.png

# Display the model plot in Colab
from IPython.display import Image
Image('network_spc.png')
```

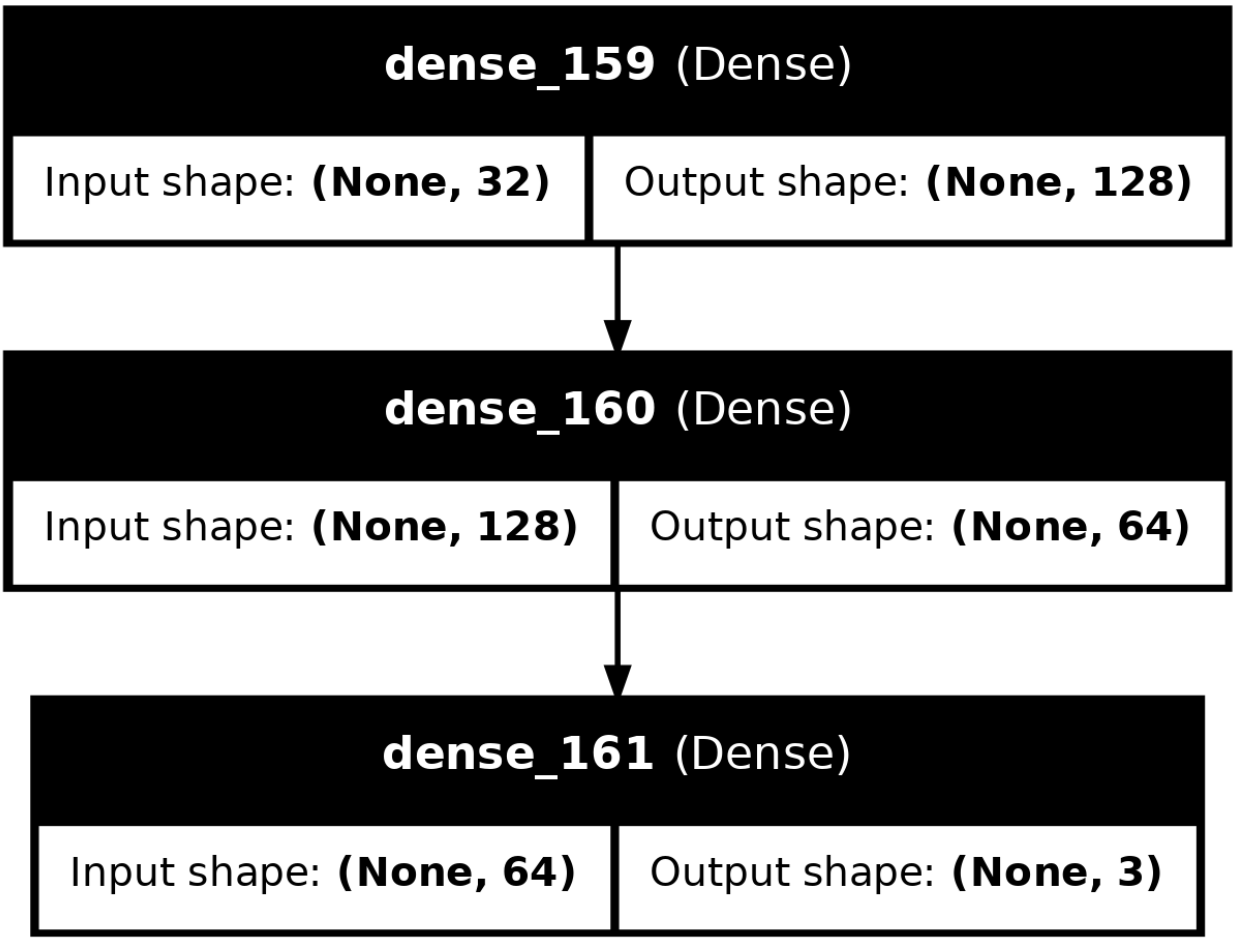



```
# One-hot encode the target labels for categorical_crossentropy
y_train_one_hot = to_categorical(y_train, num_classes=3)
y_test_one_hot = to_categorical(y_test, num_classes=3)


# Categorical Crossentropy
model_cc = train_test_model(X_train_combined, y_train_one_hot,
X_test_combined, y_test_one_hot,
                            optimal_epochs, optimal_batch_size,
'categorical_crossentropy', 'best_model_cc.keras','network_cc.png')


Training with Loss Function: categorical_crossentropy, Epochs: 100,
Batch Size: 32
Epoch 1/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 1s 855us/step - accuracy: 0.8215 - loss:
0.4241 - val_accuracy: 0.9300 - val_loss: 0.1758
Epoch 2/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 557us/step - accuracy: 0.9323 - loss:
0.1550 - val_accuracy: 0.9415 - val_loss: 0.1422
Epoch 3/100
```

```
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 642us/step - accuracy: 0.9372 - loss:
0.1526 - val_accuracy: 0.9405 - val_loss: 0.1314
Epoch 4/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 597us/step - accuracy: 0.9485 - loss:
0.1179 - val_accuracy: 0.9464 - val_loss: 0.1313
Epoch 5/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 672us/step - accuracy: 0.9532 - loss:
0.1118 - val_accuracy: 0.9394 - val_loss: 0.1353
Epoch 6/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 572us/step - accuracy: 0.9581 - loss:
0.1032 - val_accuracy: 0.9509 - val_loss: 0.1190
Epoch 7/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 612us/step - accuracy: 0.9552 - loss:
0.1030 - val_accuracy: 0.9492 - val_loss: 0.1241
Epoch 8/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 613us/step - accuracy: 0.9604 - loss:
0.0983 - val_accuracy: 0.9481 - val_loss: 0.1160
Epoch 9/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 641us/step - accuracy: 0.9613 - loss:
0.0976 - val_accuracy: 0.9502 - val_loss: 0.1138
Epoch 10/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 544us/step - accuracy: 0.9600 - loss:
0.0932 - val_accuracy: 0.9492 - val_loss: 0.1149
Epoch 11/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 561us/step - accuracy: 0.9605 - loss:
0.0924 - val_accuracy: 0.9541 - val_loss: 0.1099
Epoch 12/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 627us/step - accuracy: 0.9640 - loss:
0.0908 - val_accuracy: 0.9534 - val_loss: 0.1130
Epoch 13/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 523us/step - accuracy: 0.9609 - loss:
0.0894 - val_accuracy: 0.9558 - val_loss: 0.1047
Epoch 14/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 506us/step - accuracy: 0.9681 - loss:
0.0785 - val_accuracy: 0.9541 - val_loss: 0.1128
Epoch 15/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 524us/step - accuracy: 0.9637 - loss:
0.0882 - val_accuracy: 0.9447 - val_loss: 0.1318
Epoch 16/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 592us/step - accuracy: 0.9660 - loss:
0.0811 - val_accuracy: 0.9554 - val_loss: 0.1096
Epoch 17/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 554us/step - accuracy: 0.9679 - loss:
0.0774 - val_accuracy: 0.9565 - val_loss: 0.1072
Epoch 18/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 616us/step - accuracy: 0.9627 - loss:
0.0872 - val_accuracy: 0.9478 - val_loss: 0.1246
Epoch 19/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 628us/step - accuracy: 0.9661 - loss:
```

```
0.0796 - val_accuracy: 0.9575 - val_loss: 0.1081
Epoch 20/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 541us/step - accuracy: 0.9715 - loss:
0.0718 - val_accuracy: 0.9408 - val_loss: 0.1476
Epoch 21/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 526us/step - accuracy: 0.9687 - loss:
0.0766 - val_accuracy: 0.9558 - val_loss: 0.1058
Epoch 22/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 761us/step - accuracy: 0.9675 - loss:
0.0775 - val_accuracy: 0.9600 - val_loss: 0.1011
Epoch 23/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 539us/step - accuracy: 0.9727 - loss:
0.0710 - val_accuracy: 0.9582 - val_loss: 0.1006
Epoch 24/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 595us/step - accuracy: 0.9717 - loss:
0.0710 - val_accuracy: 0.9617 - val_loss: 0.1036
Epoch 25/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 510us/step - accuracy: 0.9757 - loss:
0.0635 - val_accuracy: 0.9544 - val_loss: 0.1063
Epoch 26/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 572us/step - accuracy: 0.9742 - loss:
0.0633 - val_accuracy: 0.9600 - val_loss: 0.1076
Epoch 27/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 548us/step - accuracy: 0.9773 - loss:
0.0582 - val_accuracy: 0.9558 - val_loss: 0.1233
Epoch 28/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 565us/step - accuracy: 0.9693 - loss:
0.0810 - val_accuracy: 0.9561 - val_loss: 0.1182
Epoch 29/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 544us/step - accuracy: 0.9791 - loss:
0.0551 - val_accuracy: 0.9575 - val_loss: 0.1043
Epoch 30/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 671us/step - accuracy: 0.9788 - loss:
0.0571 - val_accuracy: 0.9554 - val_loss: 0.1136
Epoch 31/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 602us/step - accuracy: 0.9750 - loss:
0.0595 - val_accuracy: 0.9579 - val_loss: 0.1223
Epoch 32/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 497us/step - accuracy: 0.9787 - loss:
0.0528 - val_accuracy: 0.9541 - val_loss: 0.1361
Epoch 33/100
360/360 ━━━━━━━━━━━━━━━━━━━━ 0s 569us/step - accuracy: 0.9797 - loss:
0.0524 - val_accuracy: 0.9551 - val_loss: 0.1122
Model saved as best_model_cc.keras
113/113 ━━━━━━━━━━━━━━━━━━━━ 0s 335us/step - accuracy: 0.9519 - loss:
0.1291
Test Loss: 0.1202, Test Accuracy: 0.9538
Model architecture saved as network_cc.png
```

```
# Display the model plot in Colab
from IPython.display import Image
Image('network_cc.png')
```



Confusion Matrix

A **confusion matrix** is a table used to evaluate the performance of a classification model by summarizing the counts of true positive, true negative, false positive, and false negative predictions. It provides a visual representation of how well the model's predicted classes match the actual classes, helping to assess metrics like accuracy, precision, recall, and F1-score.

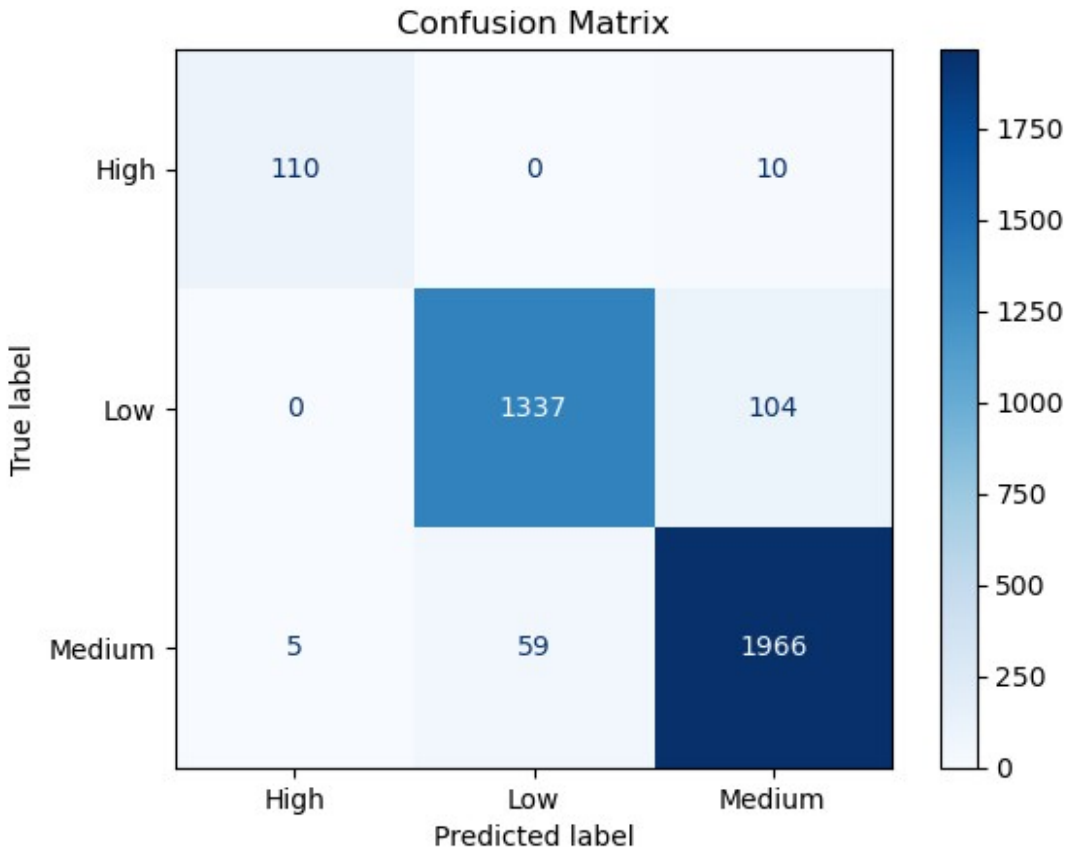
1. **Accuracy**: The proportion of correctly predicted instances out of the total instances.
2. **Precision**: The ratio of true positive predictions to the total predicted positives, indicating the correctness of positive predictions.
3. **Recall**: The ratio of true positive predictions to the total actual positives, measuring the model's ability to identify relevant instances.
4. **F1-score**: The harmonic mean of precision and recall, providing a balanced measure of a model's accuracy in classifying positive instances.

```
# plotting confusion matrix
def plot_confusion_matrix(model, X_test, y_test):
    # Step 1: Make predictions on the test set
    y_pred = model.predict(X_test)

    # Converting the predicted probabilities to class labels
    y_pred_labels = np.argmax(y_pred, axis=1)

    # Generating the confusion matrix
    conf_matrix = confusion_matrix(y_test, y_pred_labels)

    # Plotting the confusion matrix
    disp = ConfusionMatrixDisplay(confusion_matrix=conf_matrix,
display_labels=label_encoder.classes_)
    disp.plot(cmap='Blues')
    plt.title(f'Confusion Matrix')
    plt.show()
```

Now using the function `plot_confusion_matrix`, we make predictions on the test set and generates a confusion matrix, allowing for visual analysis of the model's classification performance.

```
# Plotting the Confusion Matrix for Sparse Categorical Crossentropy
plot_confusion_matrix(model_spc, X_test_combined, y_test)
```

```
113/113 ━━━━━━━━━━━━━━━━━━━━ 0s 393us/step
```

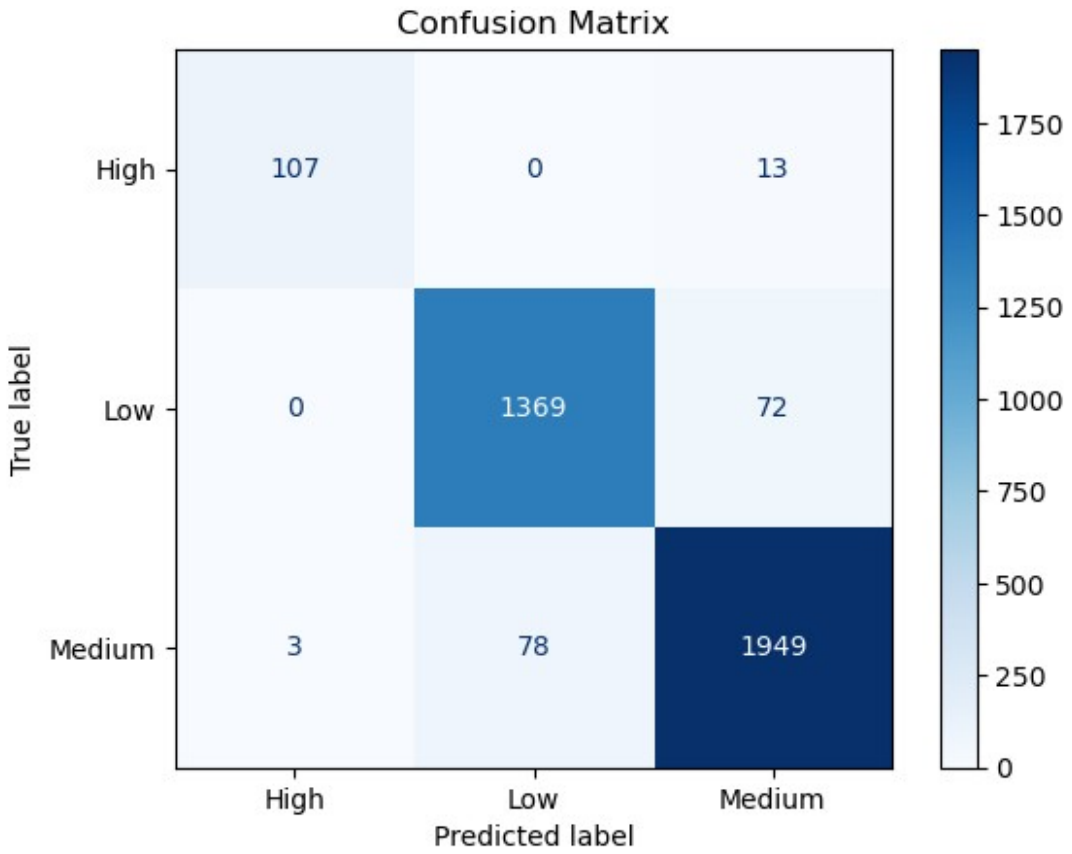


- High category: Out of 120 true instances, the model correctly classifies 110 as "High", with a small misclassification rate (10 instances misclassified as "Medium").
- Low category: Performs well, with 1337 correctly classified as "Low" and only 104 misclassified as "Medium".
- Medium category: Very strong performance, with 1966 instances correctly classified and minimal confusion with the "Low" category (59 misclassifications).

Overall, the sparse categorical crossentropy model shows a balanced performance across all categories, especially excelling in the "Low" and "Medium" categories.

```
# Plotting Confusion Matrix for Categorical Crossentropy (uncomment for CC)
plot_confusion_matrix(model_cc, X_test_combined, y_test)
```

```
113/113 ━━━━━━━━━━━━━━━━━━━━ 0s 338us/step
```



- High category: Out of 120 true instances, 107 are correctly classified, but 13 are misclassified as "Medium", slightly more than the sparse model.
- Low category: The model does slightly better, with 1369 correct classifications and 72 misclassifications as "Medium", improving compared to the sparse model.
- Medium category: 1949 instances are correctly classified, with 78 misclassified as "Low", which is slightly worse than the sparse crossentropy model.

The categorical crossentropy model performs similarly to the sparse version, with slight variations in misclassification between "Low" and "Medium", but overall robust performance.

- Sparse categorical crossentropy performs slightly better when distinguishing between "Low" and "Medium" categories, as it has fewer misclassifications in both.
- Categorical crossentropy has a minor advantage in the "Low" category but slightly worse in "Medium" predictions compared to sparse categorical crossentropy.

Both models perform similarly well in general, with sparse categorical crossentropy having a slight edge in distinguishing between Low and Medium categories, while categorical crossentropy is marginally better in the "Low" category predictions. These differences are minimal, suggesting that either loss function can be used depending on the specific needs of the task.

Achievements

- Transformed the dataset into a multi-class classification problem by engineering a categorical target variable.
- Built deep learning models with categorical crossentropy and sparse categorical crossentropy as loss functions.
- Tuned hyperparameters to maximize model performance, achieving high validation accuracy.
- Generated and analyzed confusion matrices to understand model performance and identify misclassifications.
- Plotted the network architecture and optimized the classification models to achieve the best accuracy.
- Identified the optimal hyperparameters for both loss functions

Summary of the Project

1. Data preprocessing: Successfully handled both numerical and categorical features, including scaling, encoding, and engineering features for both regression and classification tasks.

2. Model development: Built and optimized multiple neural network models using both PyTorch and Keras frameworks for the regression task, and used Keras for classification.

3. Model evaluation and optimization:

- For regression, evaluated models using MSE and RMSE.
- For classification, evaluated models using validation accuracy, validation loss, and confusion matrices. Performed hyperparameter tuning to identify the best configurations for both tasks.

1. Visualization:

- Plotted training and validation loss curves for both tasks to monitor overfitting.
- Generated confusion matrices for classification models to analyze the classification performance.
- Visualized the network architecture for both regression and classification models.

1. Optimization results:

- Identified the best performing regression model with optimal epochs = 100 and batch size = 32.
- Achieved optimal performance in classification with the same configuration for both loss functions (categorical crossentropy and sparse categorical crossentropy).

Future Work

In the future, we aim to extend our work in the domain of player performance prediction by incorporating **image data** and **video analysis**. The next step in advancing this project would

involve leveraging **Convolutional Neural Networks (CNNs)** for analyzing player images to extract visual features, such as body posture, player positioning, and in-game actions. Additionally, using **Recurrent Neural Networks (RNNs)**, particularly **Long Short-Term Memory (LSTM) networks**, could enable us to analyze video sequences to track player movements, performance trends over time, and behavior during match play.

However, such analysis is computationally intensive and requires significantly more powerful hardware to handle the processing demands of larger datasets. In addition, finding well-structured, high-quality datasets that include both numerical and multimedia data remains a challenge in this domain. Most publicly available datasets focus on either structured data or images and videos, but rarely both. This limitation presents an interesting task for future work, as accessing and working with such comprehensive datasets will allow us to apply advanced neural network techniques for deeper insights into player performance and tactical decision-making.

#References

The following is the table of References for each section of our project.

Section	Name	URL
Domain Knowledge	Player Performance Analysis	https://playerscout.co.uk/player-performance-analysis/
	Performance Analysis in Football	https://www.catapult.com/blog/performance-analysis-in-football#what-is-it
	Performance Analysis in Football	https://analyisport.com/performance-analysis-in-football/
Similar Case Studies	Neural Networks in Sports Analytics	https://medium.com/@neha.raut20/neural-networks-in-sports-analytics-4e991970925c#:~:text=Player%20Performance%20Analysis,performance%20in%20an%20upcoming%20game
	Research Paper on Sports Analytics	https://arxiv.org/pdf/2108.10125
	Thesis on Player Performance Analysis	https://studenttheses.uu.nl/bitstream/handle/20.500.12932/40763/Goes_652923.pdf?sequence=1&isAllowed=y
	Deep Learning Analysis for Player Performances	https://www.researchgate.net/publication/359479143_A_deep_learning_analysis_for_the_effect_of_individual_player_performances_on_match_results
	Predicting Player Performance Using Neural Networks	https://chriszaire.medium.com/predicting-player-performance-using-neural-networks-f6142784b681
	Transforming Player Recruitment with Artificial Neural Networks	https://medium.com/@marin11amf11/how-can-artificial-neural-networks-transform-player-recruitment-in-football-823e5f9690bf
Pytorch Code	Pytorch Basics Tutorial	https://github.com/yunjey/pytorch-tutorial/blob/master/tutorials/01-basics/pytorch_basics/main.py
	Learn Pytorch Fundamentals	https://www.learnpytorch.io/00_pytorch_fundamentals/
	Pytorch Linear Regression Tutorial	https://www.tutorialspoint.com/pytorch/pytorch_linear_regression.htm
Keras	First Neural Network with Keras	https://machinelearningmastery.com/tutorial-first-neural-network-python-keras/
	Keras Tutorial for Deep Learning	https://elitedatascience.com/keras-tutorial-deep-learning-in-python
	Deep Learning Tutorial with Keras	https://www.datacamp.com/tutorial/deep-learning-python
	Keras Regression Tutorial	https://huseynkishiyev.medium.com/keras-regression-41f62c95f514
Differences	Keras vs Pytorch Guide	https://www.kaggle.com/code/utcarshagrawal/keras-vs-pytorch-a-perfect-guide
	Comparing Keras and Pytorch	https://medium.com/@kanerika/keras-vs-pytorch-which-ml-framework-is-the-best-for-you-b3eb6451dd66
Regularization	Regularization in Neural Networks	https://www.pinecone.io/learn/regularization-in-neural-networks/
	Regularization Explained	https://www.ibm.com/topics/regularization
Cost and Loss functions	Neural Network Basics: Loss and Cost Functions	https://medium.com/artificialis/neural-network-basics-loss-and-cost-functions-9d089e9de5f8
	Cost Function and Loss Function	https://nadeemm.medium.com/cost-function-loss-function-c3cab1ddffa4
Differences	Choosing Between Cross-Entropy and Sparse Cross-Entropy	https://medium.com/@shireenchand/choosing-between-cross-entropy-and-sparse-cross-entropy-the-only-guide-you-need-abea92c84662
	Cross-Entropy vs Sparse Cross-Entropy	https://stats.stackexchange.com/questions/326065/cross-entropy-vs-sparse-cross-entropy-when-to-use-one-over-the-other
Sparse	Understanding Sparse Categorical Cross-Entropy	https://rmoklesur.medium.com/what-you-need-to-know-about-sparse-categorical-cross-entropy-9f07497e3a6f
	Sparse Categorical Cross Entropy - Keras Documentation	https://keras.io/api/losses/probabilistic_losses/#sparsecategoricalcrossentropy-class
	Using Sparse Categorical Cross-Entropy in Keras	https://github.com/christianversloot/machine-learning-articles/blob/main/how-to-use-sparse-categorical-crossentropy-in-keras.md
Binary Cross	Binary Cross Entropy - Keras Documentation	https://keras.io/api/losses/probabilistic_losses/#binarycrossentropy-class
Categorical	Categorical Cross Entropy - Keras Documentation	https://keras.io/api/losses/probabilistic_losses/#categoricalcrossentropy-class
Epoch and Batch Size	Understanding Epoch in Machine Learning	https://www.geeksforgeeks.org/epoch-in-machine-learning/
	Difference Between a Batch and an Epoch	https://machinelearningmastery.com/difference-between-a-batch-and-an-epoch/
Hyperparameter Tuning	Hyperparameter Tuning Explained	https://aws.amazon.com/what-is/hyperparameter-tuning/#:~:text=Hyperparameters%20directly%20control%20model%20structure,values%20is%20crucial%20for%20success.
Tensorboard	Using Tensorboard in Google Colab	https://www.geeksforgeeks.org/how-to-use-tensorboard-in-google-colab/
	Tensorboard Graphs	https://www.tensorflow.org/tensorboard/graphs
		https://colab.research.google.com/github/tensorflow/tensorb

Thank you!

We express our sincere gratitude to the data providers on Kaggle. Most importantly, we extend our heartfelt thanks to our professors for inspiring and fostering the curiosity that drove us to explore this project.