# Lab 6


## Team - 29

Tanish Wanve (23110327)
Tejas Patil (23110333)


Github Repository Link - https://github.com/TanishWanve/STT_Assignment-6.git
Google Colab Link - STTLab6_23110327_23110333.ipynb

# SECTION 1

## 1. Implement a Multi-Layer Perceptron (MLP) Using the Iris Dataset

```python
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import MinMaxScaler
from tensorflow.keras.utils import to_categorical


iris = load_iris()  # Load the Iris dataset
X = iris.data  # Features (flower measurements)
y = to_categorical(iris.target)  # One-hot encode target labels

# Split data into train+val and test
X_train_val, X_test, y_train_val, y_test = train_test_split(  # 80% train+val, 20% test
    X, y, test_size=0.2, random_state=42, stratify=iris.target  # Maintain class distribution
)

# Further split into training and validation sets
X_train, X_val, y_train, y_val = train_test_split(  # Split train+val into 70% train, 10% val
    X_train_val, y_train_val, test_size=0.125, random_state=42,  # Stratify to maintain balance
    stratify=np.argmax(y_train_val, axis=1)  # Ensure even class distribution
)

# Scale features between 0 and 1
scaler = MinMaxScaler()
X_train = scaler.fit_transform(X_train)  # Scale training data
X_val = scaler.transform(X_val)  # Scale validation data
X_test = scaler.transform(X_test)  # Scale test data

print("Data prepared and split into training, validation, and test sets.")
```

```
Data prepared and split into training, validation, and test sets.
```

*This code loads the Iris dataset and converts the labels into a format that the neural network can understand. It then splits the data into three parts: training(70%), validation(10%), and testing(20%). Finally, it scales the feature values between 0 and 1 to help the model learn better and faster.*

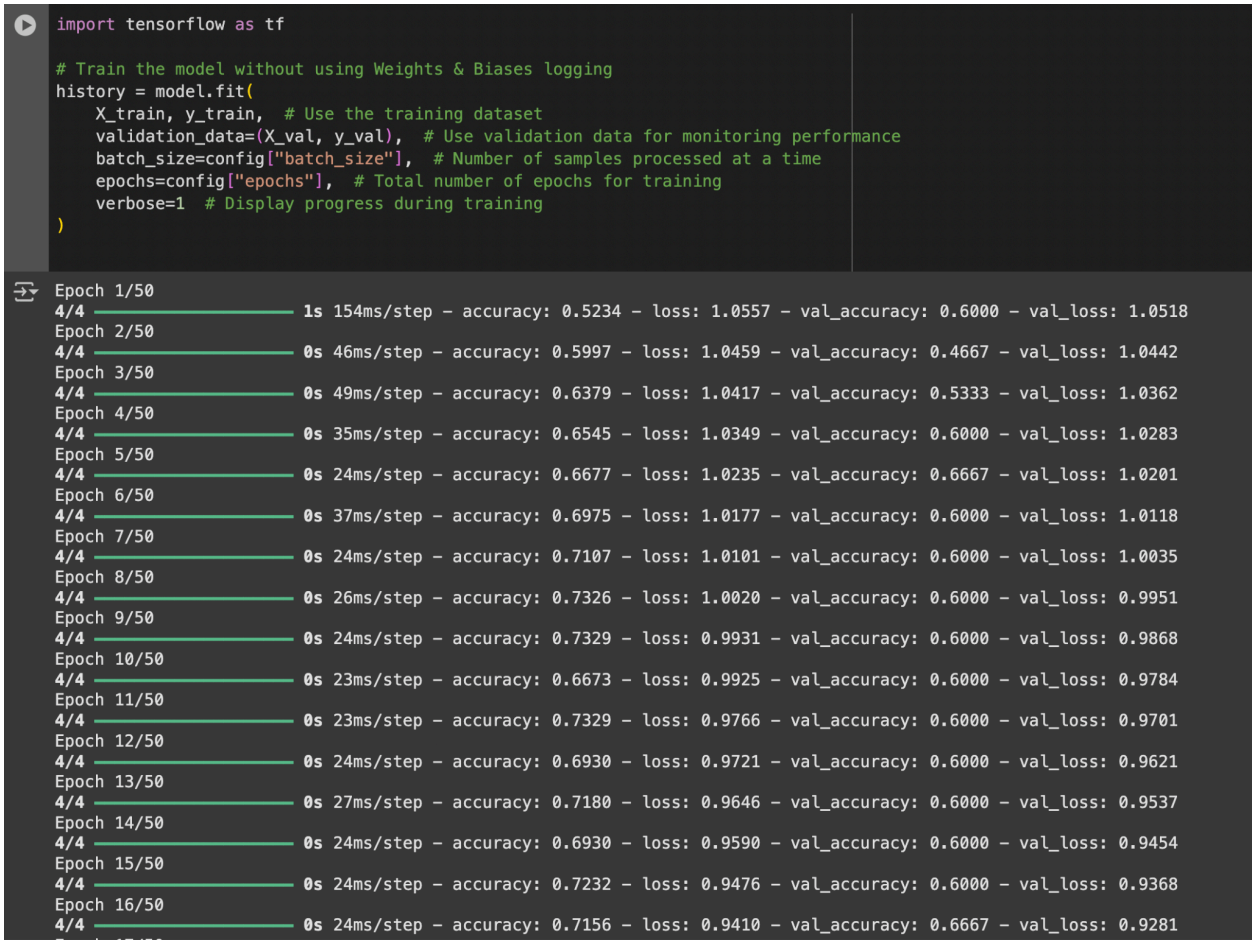
## 2. Define and Train the MLP Model

```python
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.optimizers import Adam
import wandb

# Define hyperparameters and configuration settings
config = {
    "learning_rate": 0.001,  # Speed of learning during training
    "batch_size": 32,  # No of samples processed at a time
    "epochs": 50,  # No of times data is passed through the model
    "hidden_units": 16,  # No of neurons in the hidden layer
    "activation_hidden": "relu",  # Activation function for the hidden layer
    "activation_output": "softmax"  # Activation function for the output layer
}
```

*Setting up the basic tools needed to build and train a neural network using TensorFlow. It also defines key settings like how fast the model should learn (learning rate), how many times it should go through the data (epochs), and how many neurons should be in the hidden layer (hidden_units). Additionally, it prepares the activation functions that help the model learn patterns and make predictions.*

```python
# Initialize Weights & Biases for tracking
wandb.init(project="iris_mlp", config=config, anonymous="allow")  # Start W&B run with configuration

model = Sequential([
    Dense(16, input_shape=(4,), activation='relu'),  # Hidden layer with 16 neurons and ReLU activation
    Dense(3, activation='softmax')  # Output layer with 3 neurons and softmax activation for classification
])

model.compile(
    optimizer=Adam(learning_rate=0.001),  # Adam optimizer with a learning rate of 0.001
    loss='categorical_crossentropy',  # Loss function for multi-class classification
    metrics=['accuracy']  # Track accuracy during training
)
```

*Setting up the neural network for training and connecting it to Weights & Biases (W&B) for tracking progress. The model has two layers: a hidden layer with 16 neurons that help the model learn patterns and an output layer with 3 neurons for predicting flower species. The model is then compiled with an Adam optimizer for efficient learning and a categorical cross-entropy loss function to handle multi-class classification.*



```
import tensorflow as tf

# Train the model without using Weights & Biases logging
history = model.fit(
    X_train, y_train,  # Use the training dataset
    validation_data=(X_val, y_val),  # Use validation data for monitoring performance
    batch_size=config["batch_size"],  # Number of samples processed at a time
    epochs=config["epochs"],  # Total number of epochs for training
    verbose=1  # Display progress during training
)
```

```
Epoch 1/50
4/4 ━━━━━━━━━━━━━━━━━━━━ 1s 154ms/step - accuracy: 0.5234 - loss: 1.0557 - val_accuracy: 0.6000 - val_loss: 1.0518
Epoch 2/50
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 46ms/step - accuracy: 0.5997 - loss: 1.0459 - val_accuracy: 0.4667 - val_loss: 1.0442
Epoch 3/50
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 49ms/step - accuracy: 0.6379 - loss: 1.0417 - val_accuracy: 0.5333 - val_loss: 1.0362
Epoch 4/50
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 35ms/step - accuracy: 0.6545 - loss: 1.0349 - val_accuracy: 0.6000 - val_loss: 1.0283
Epoch 5/50
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 24ms/step - accuracy: 0.6677 - loss: 1.0235 - val_accuracy: 0.6667 - val_loss: 1.0201
Epoch 6/50
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 37ms/step - accuracy: 0.6975 - loss: 1.0177 - val_accuracy: 0.6000 - val_loss: 1.0118
Epoch 7/50
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 24ms/step - accuracy: 0.7107 - loss: 1.0101 - val_accuracy: 0.6000 - val_loss: 1.0035
Epoch 8/50
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 26ms/step - accuracy: 0.7326 - loss: 1.0020 - val_accuracy: 0.6000 - val_loss: 0.9951
Epoch 9/50
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 24ms/step - accuracy: 0.7329 - loss: 0.9931 - val_accuracy: 0.6000 - val_loss: 0.9868
Epoch 10/50
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 23ms/step - accuracy: 0.6673 - loss: 0.9925 - val_accuracy: 0.6000 - val_loss: 0.9784
Epoch 11/50
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 23ms/step - accuracy: 0.7329 - loss: 0.9766 - val_accuracy: 0.6000 - val_loss: 0.9701
Epoch 12/50
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 24ms/step - accuracy: 0.6930 - loss: 0.9721 - val_accuracy: 0.6000 - val_loss: 0.9621
Epoch 13/50
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 27ms/step - accuracy: 0.7180 - loss: 0.9646 - val_accuracy: 0.6000 - val_loss: 0.9537
Epoch 14/50
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 24ms/step - accuracy: 0.6930 - loss: 0.9590 - val_accuracy: 0.6000 - val_loss: 0.9454
Epoch 15/50
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 24ms/step - accuracy: 0.7232 - loss: 0.9476 - val_accuracy: 0.6000 - val_loss: 0.9368
Epoch 16/50
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 24ms/step - accuracy: 0.7156 - loss: 0.9410 - val_accuracy: 0.6667 - val_loss: 0.9281
```

*Training the neural network using the training data while checking its performance on the validation data after each epoch. It does not log any metrics to Weights & Biases, keeping the training process simple and straightforward.*

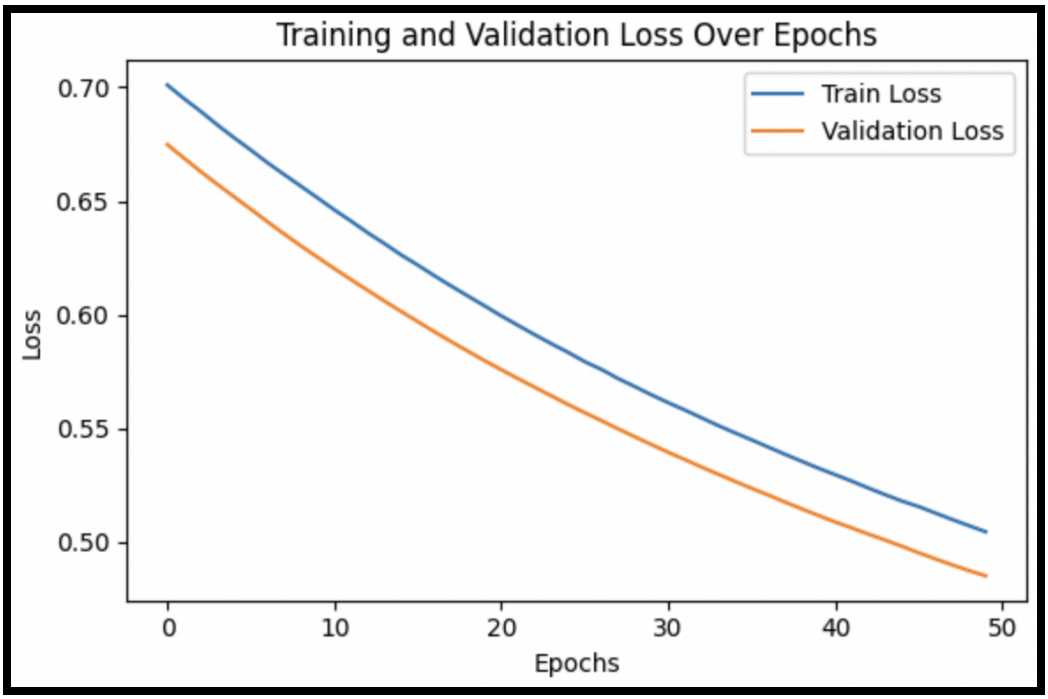
# 3. Evaluate Model Performance

```python
y_test_pred_prob = model.predict(X_test)  # Get predicted probabilities for each test sample
y_test_pred = np.argmax(y_test_pred_prob, axis=1)  # Choose the class with the highest probability
y_test_true = np.argmax(y_test, axis=1)  # Convert one-hot labels to simple integer labels

# Calculate evaluation metrics
accuracy = accuracy_score(y_test_true, y_test_pred)  # Overall correctness of predictions
precision = precision_score(y_test_true, y_test_pred, average='weighted')  # Correct positive predictions ratio
recall = recall_score(y_test_true, y_test_pred, average='weighted')  # Correctly identified actual positives
f1 = f1_score(y_test_true, y_test_pred, average='weighted')  # Balance between precision and recall

# Print the evaluation results
print(f"Test Accuracy: {accuracy:.4f}")
print(f"Test Precision: {precision:.4f}")
print(f"Test Recall: {recall:.4f}")
print(f"Test F1 Score: {f1:.4f}")
```

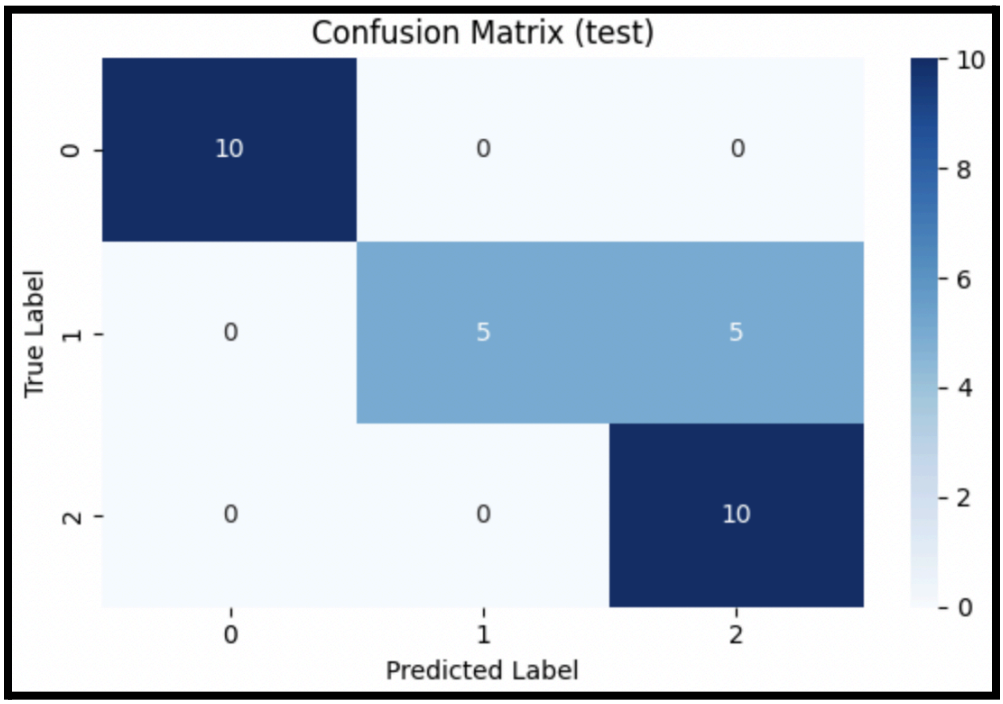
```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 42ms/step
Test Accuracy: 0.8333
Test Precision: 0.8889
Test Recall: 0.8333
Test F1 Score: 0.8222
```

Test Accuracy : 0.8333
Test Precision : 0.8889
Test Recall : 0.8333
Test F1 Score : 0.8222

*The  results indicate that the model is performing well on unseen data. With an accuracy of **83.33%**, it correctly classifies about 83% of test cases. A high precision of **88.89%** means that when the model predicts a class, it's usually correct, while a recall of **83.33%** shows it captures most of the actual cases. The F1 score of **82.22%** reflects a good balance between precision and recall, indicating overall strong performance.*




*The loss curve shows a steady decrease in both training and validation loss over the epochs, indicating that the model is learning effectively and there are no signs of overfitting since both losses follow a similar pattern.*

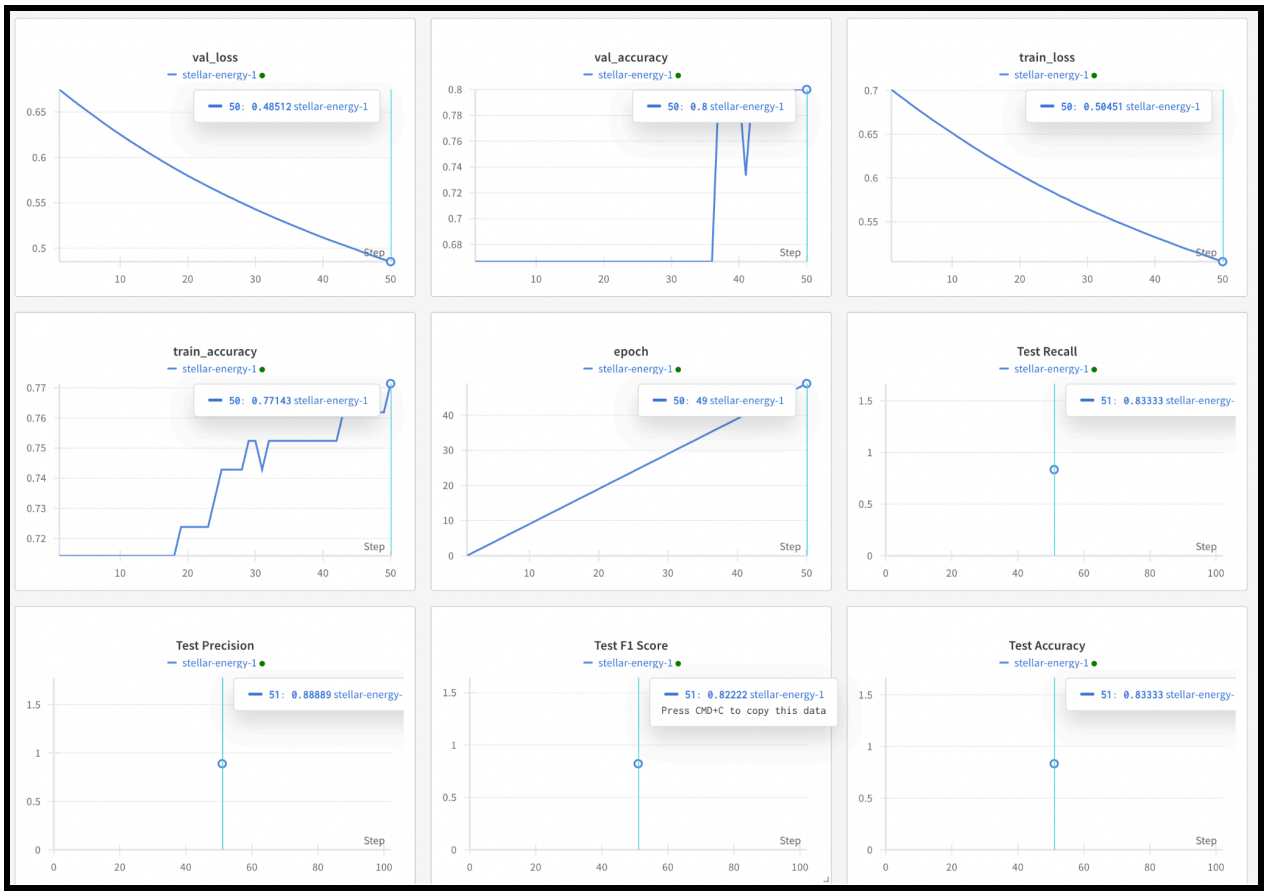



*The confusion matrix shows that the model perfectly classifies all Class 0 and Class 2 samples (10 each). However, for Class 1, it only gets 5 right, while 1 sample is misclassified as Class 0 and 5 are misclassified as Class 2. This indicates the model easily distinguishes Class 0 and Class 2 but struggles significantly to identify Class 1, often confusing it with Class 2.*

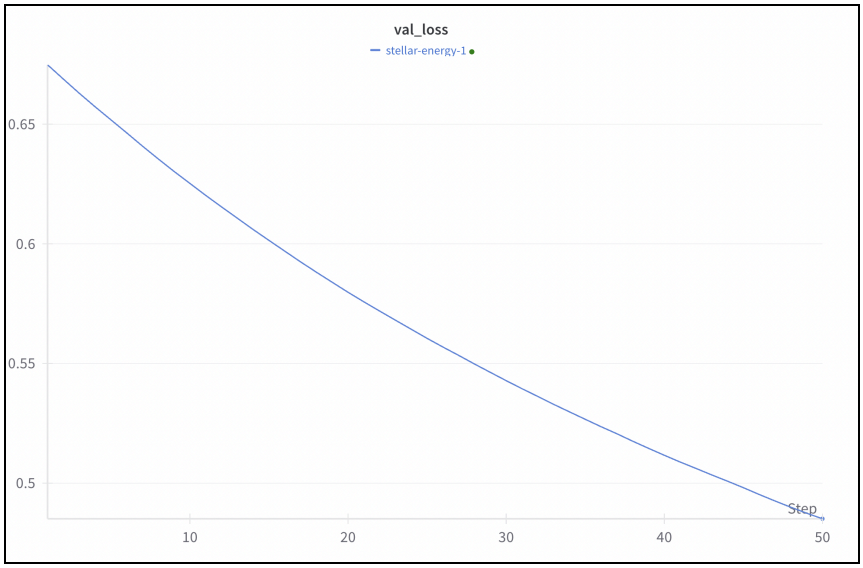
# 4. Set Up Experiment Tracking with Weights & Biases (W&B)

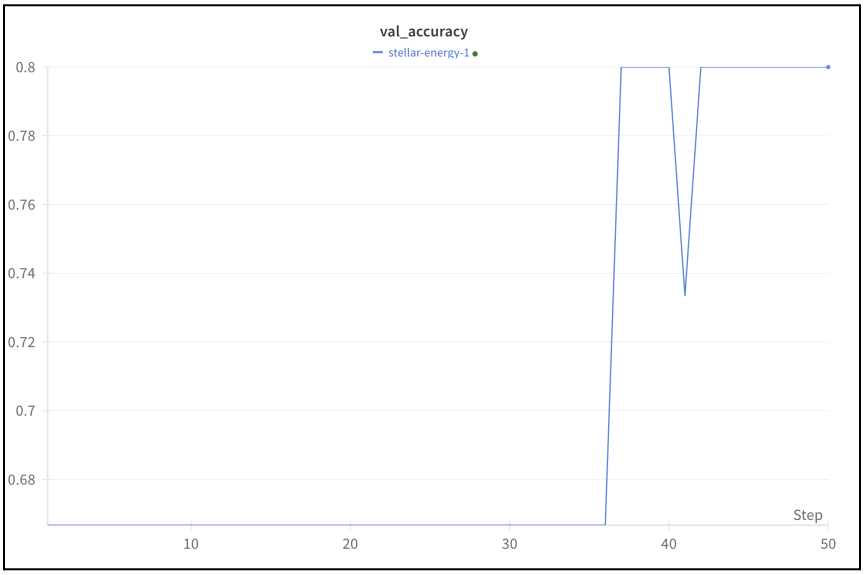
```python
import io
import wandb
import tensorflow as tf
from tensorflow.keras.callbacks import Callback
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.optimizers import Adam

# Initialize Weights & Biases
wandb.init(project="iris_mlp_wandb", config=config)


# Log model architecture to Weights & Biases (W&B)
stream = io.StringIO()  # Create a string buffer to store the summary
model.summary(print_fn=lambda x: stream.write(x + "\n"))  # Capture model summary line by line
model_summary_str = stream.getvalue()  # Get the full model summary as a string
wandb.log({"Model Architecture": model_summary_str})  # Log the model structure to W&B for tracking

# Define a custom callback to log loss and accuracy after each epoch
class CustomWandbCallback(Callback):  # Create a custom callback class
    def on_epoch_end(self, epoch, logs=None):  # Called at the end of each epoch
        wandb.log({  # Log the following metrics to W&B:
            "epoch": epoch,  # Current epoch number
            "train_loss": logs.get("loss"),  # Training loss for this epoch
            "val_loss": logs.get("val_loss"),  # Validation loss for this epoch
            "train_accuracy": logs.get("accuracy"),  # Training accuracy
            "val_accuracy": logs.get("val_accuracy")  # Validation accuracy
        })

# Train the model while tracking progress
history = model.fit(
    X_train, y_train,
    validation_data=(X_val, y_val),  # Validation dataset to check model performance during training
    batch_size=config["batch_size"],  # No of samples processed before updating weights
    epochs=config["epochs"],  # Total no of complete passes through the dataset
    callbacks=[CustomWandbCallback()],
    verbose=1  # Display progress after each epoch
)
```

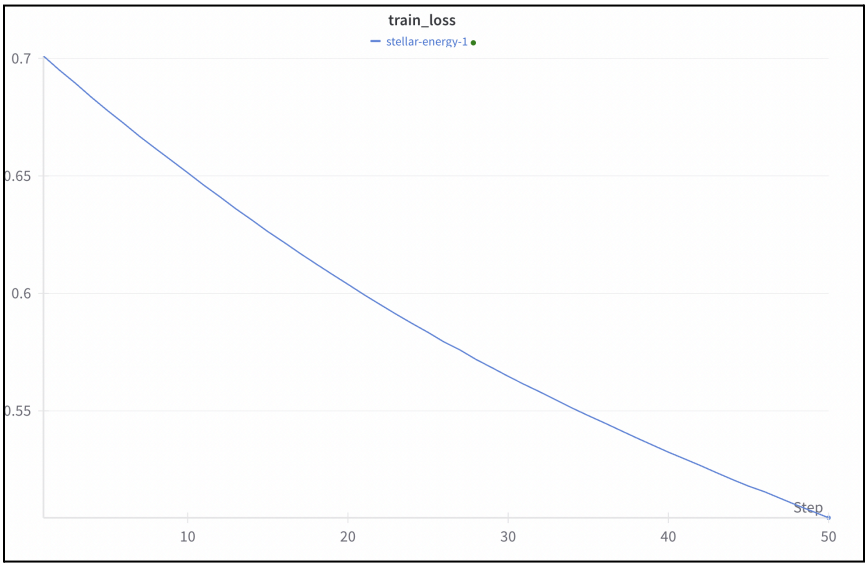
1. *The code initializes a Weights & Biases (W&B) run using the given project name and configuration, which sets up the environment for tracking experiment metrics.*
2. *It captures the model's architecture by writing the summary to a string buffer and then logs this summary to W&B, allowing you to review the model structure later.*
3. *A custom callback class is defined that logs important metrics (training loss, validation loss, training accuracy, and validation accuracy) at the end of every epoch.*
4. *The model is then trained using the training data while the callback automatically sends these performance metrics to W&B after each epoch.*
5. *This setup lets you monitor the model's learning progress in real time, making it easier to track and analyze how well your model is performing during training.*

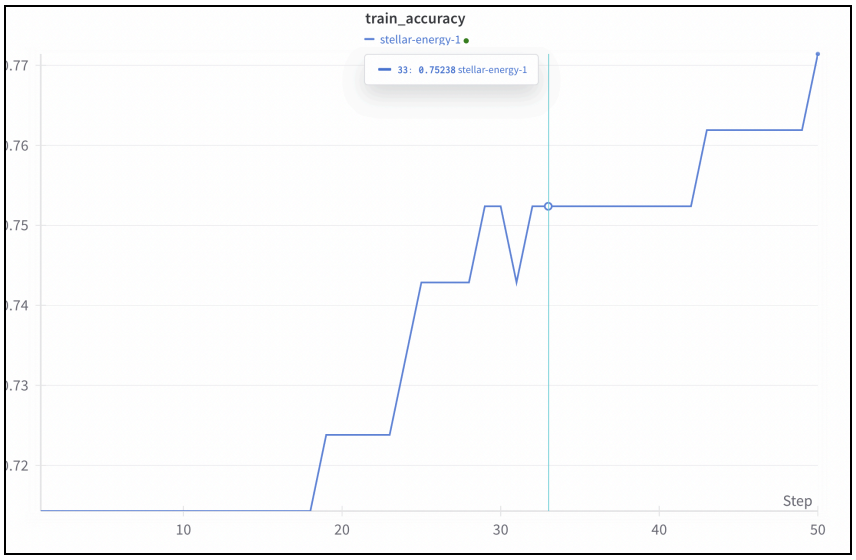


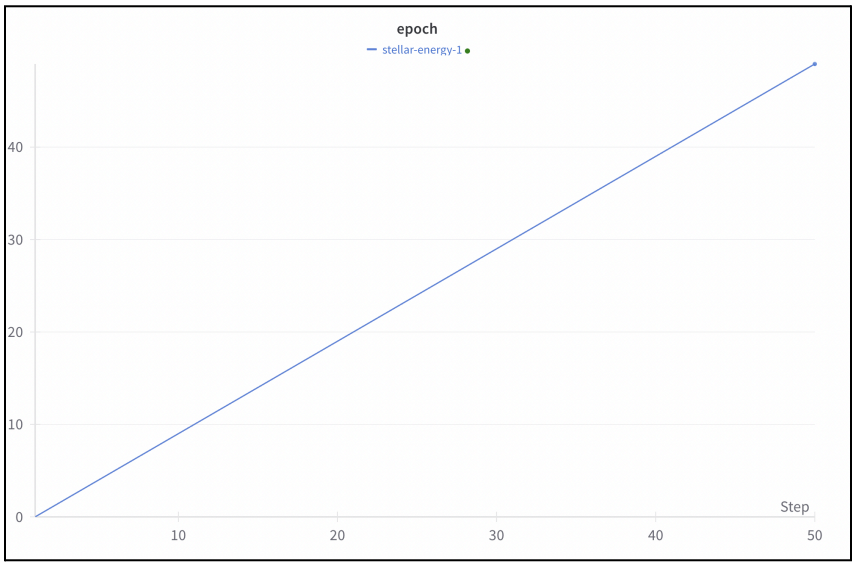

1. ***Train Loss & Val Loss:*** *These charts show how the model's error on both the training set (train_loss) and validation set (val_loss) changes over epochs. A steady decline usually means the model is learning, and watching the gap between these curves helps detect overfitting or underfitting.*
2. ***Train Accuracy & Val Accuracy:*** *These plots track how often the model predicts correctly on the training (train_accuracy) and validation (val_accuracy) sets. A rising accuracy indicates the model is improving, while a gap between train and validation accuracy can signal overfitting.*
3. ***Epoch:*** *This simply logs the training iteration count. Each epoch represents one full pass over the training data.*
4. ***Test Accuracy, Test Precision, Test Recall, Test F1 Score:*** *These metrics evaluate the final performance of the model on unseen test data:*
    - ***Test Accuracy*** *measures the overall percentage of correct predictions.*
    - ***Test Precision*** *indicates how many predicted positives are truly positive.*
    - ***Test Recall*** *shows how many actual positives were identified correctly.*
    - ***Test F1 Score*** *balances precision and recall, giving a single measure of test performance.*

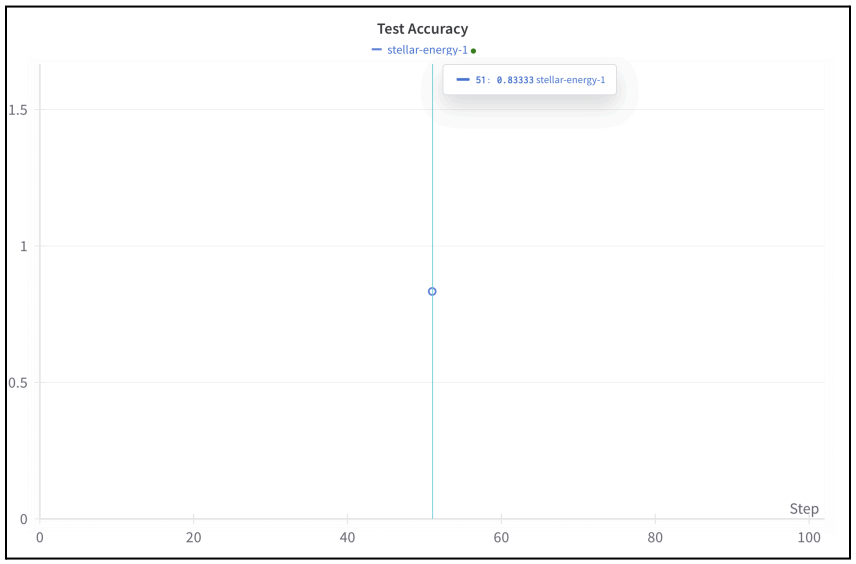

val_loss
stellar-energy-1
0.65
0.6
0.55
0.5
Step
10
20
30
40
50

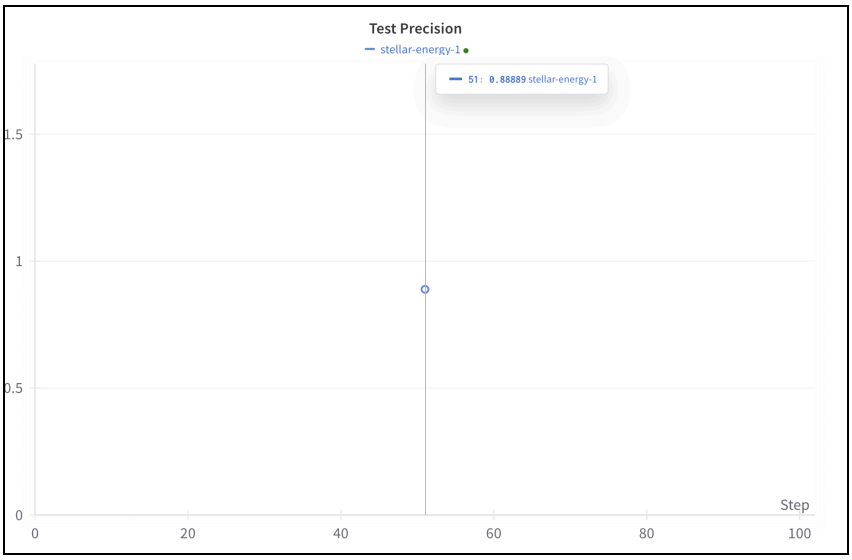

val_accuracy
stellar-energy-1
0.8
0.78
0.76
0.74
0.72
0.7
0.68
Step
10
20
30
40
50

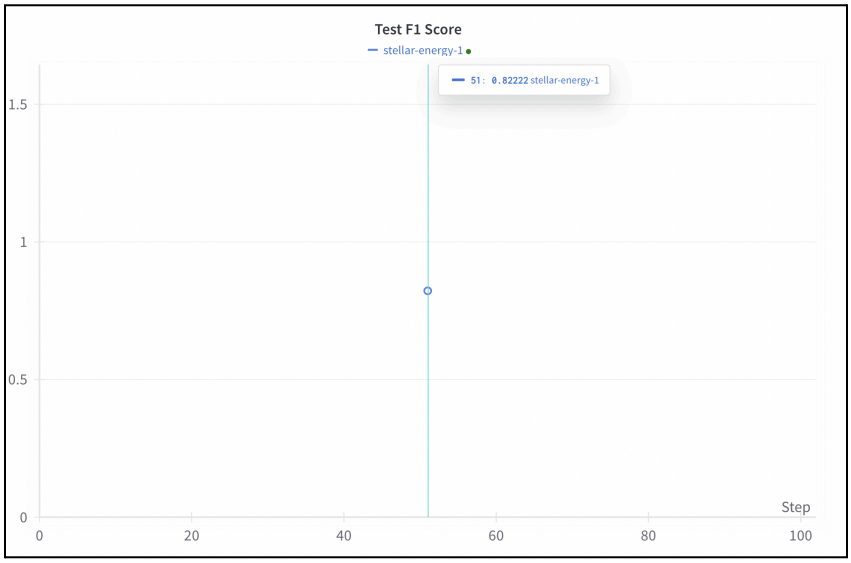

train_loss
stellar-energy-1
0.7
0.65
0.6
0.55
Step
10
20
30
40
50

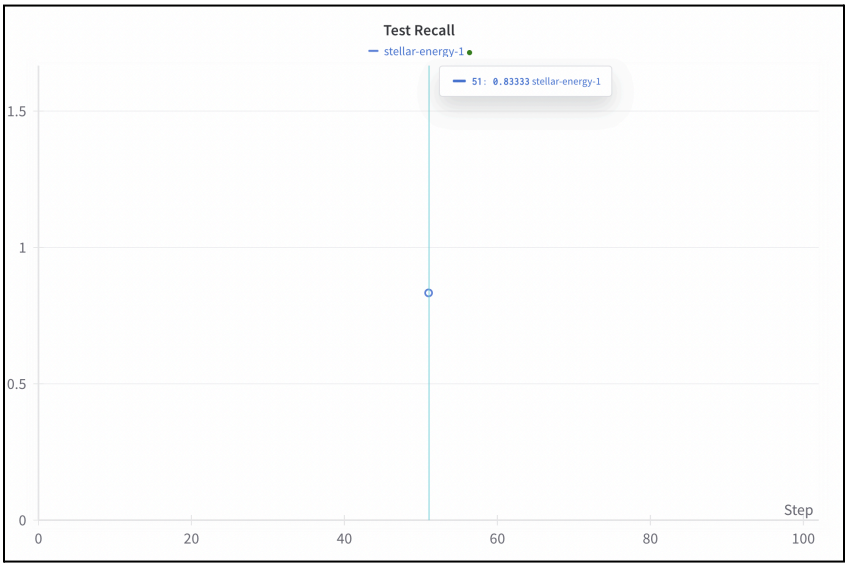

train_accuracy
stellar-energy-1
33: 0.75238 stellar-energy-1
0.77
0.76
0.75
0.74
0.73
0.72
Step
10
20
30
40
50


epoch
stellar-energy-1
40
30
20
10
0
Step
10
20
30
40
50


Test Accuracy
stellar-energy-1
51: 0.83333 stellar-energy-1
1.5
1
0.5
0
Step
0
20
40
60
80
100


Test Precision
stellar-energy-1
51: 0.88889 stellar-energy-1
1.5
1
0.5
0
Step
0
20
40
60
80
100

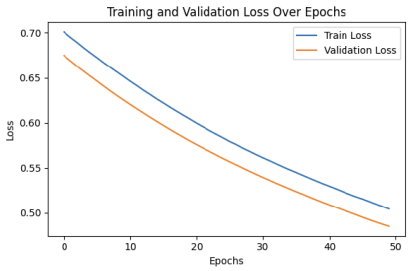

Test F1 Score
stellar-energy-1
51: 0.82222 stellar-energy-1
1.5
1
0.5
0
Step
0
20
40
60
80
100


Test Recall
stellar-energy-1
51: 0.83333 stellar-energy-1
1.5
1
0.5
0
Step
0
20
40
60
80
100

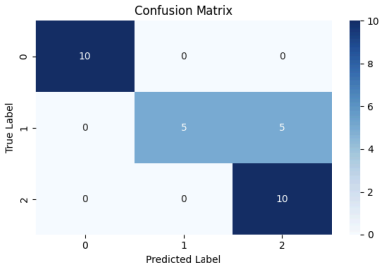
**W&B dashboard** displaying

- Confusion matrix visualization.
- Training and validation loss curves.

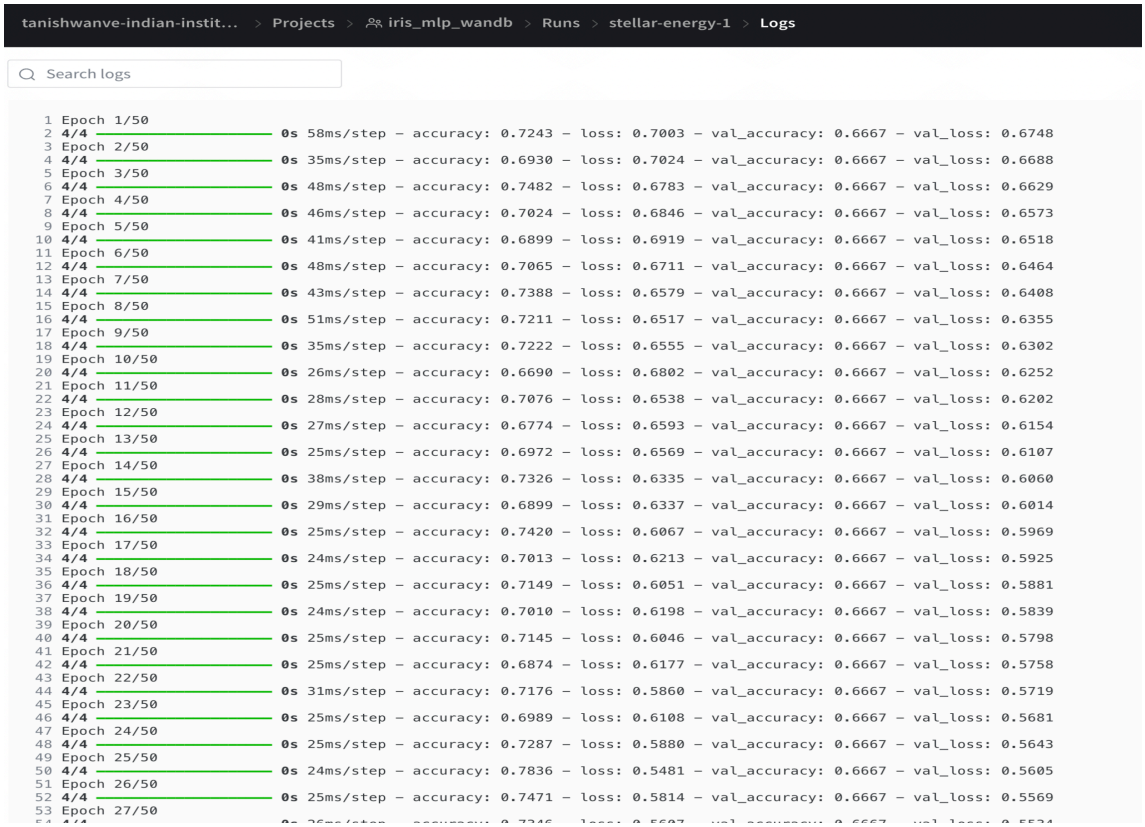
Loss Curves




Confusion Matrix



## Logs on W&B page



# SECTION 2

## DATA PREPARATION

```
0s
import numpy as np
import pandas as pd
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import MinMaxScaler
from tensorflow.keras.utils import to_categorical
```

*Importing Libraries*

```
# Load the Iris dataset
iris = load_iris()  # Load dataset
X = iris.data  # Feature matrix
y = iris.target  # Integer labels (0, 1, or 2)

# Split data into train+val (80%) and test (20%)
X_train_val, X_test, y_train_val, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42, stratify=y
)

# Further split train+val into train (70% total) and validation (10% total)
X_train, X_val, y_train, y_val = train_test_split(
    X_train_val, y_train_val, test_size=0.125, random_state=42, stratify=y_train_val
)

# Scale features to [0, 1]
scaler = MinMaxScaler()  # Initialize scaler
X_train = scaler.fit_transform(X_train)  # Scale training data
X_val = scaler.transform(X_val)  # Scale validation data
X_test = scaler.transform(X_test)  # Scale test data

# Convert labels to one-hot encoding
y_train = to_categorical(y_train, num_classes=3)  # Convert training labels
y_val = to_categorical(y_val, num_classes=3)      # Convert validation labels
y_test = to_categorical(y_test, num_classes=3)    # Convert test labels
```

*Loads the Iris dataset and splits it into training (70%), validation (10%), and testing (20%) sets, then scales the features between 0 and 1. Finally, it converts the integer labels into one-hot encoded vectors so the model can use them for multi-class classification.*

## MODEL FUNCTION

```
from tensorflow.keras.models import Sequential  # For building a sequential model
from tensorflow.keras.layers import Dense       # For adding Dense layers
from tensorflow.keras.optimizers import Adam      # For the Adam optimizer

def create_model(lr):
    model = Sequential([                          # Initialize sequential model
        Dense(16, input_shape=(4,), activation='relu'),  # Hidden layer: 16 neurons, ReLU activation
        Dense(3, activation='softmax')                   # Output layer: 3 neurons, softmax activation
    ])
    model.compile(optimizer=Adam(learning_rate=lr),  # Compile model with Adam optimizer
                  loss='categorical_crossentropy',   # Use categorical crossentropy loss
                  metrics=['accuracy'])              # Track accuracy during training
    return model
```

*This code defines a function called create_model that builds a simple neural network (MLP). It creates a sequential model with one hidden layer of 16 neurons using ReLU activation and an output layer of 3 neurons using softmax activation for multi-class classification. The model is compiled with the Adam optimizer and categorical cross entropy loss, and accuracy is tracked during training. The function takes a learning rate as input and returns the compiled model.*

## HYPERPARAMETER GRID AND LOOP

```
import matplotlib.pyplot as plt  # For plotting
import seaborn as sns  # For heatmaps
from sklearn.metrics import accuracy_score, f1_score, confusion_matrix  # For evaluation metrics

# Define hyperparameter grid
batch_sizes = [2, 4]             # Batch sizes to try
learning_rates = [1e-3, 1e-5]      # Learning rates to try
epochs_list = [1, 3, 5]            # Epochs to try

# List to store results for each configuration
results = []  # Initialize results list
```

*This code loads the Iris dataset, splits it into training, validation, and test sets, and scales the features to a range of [0,1]. It then converts the integer class labels into one-hot encoded labels, preparing the data for a neural network model.*

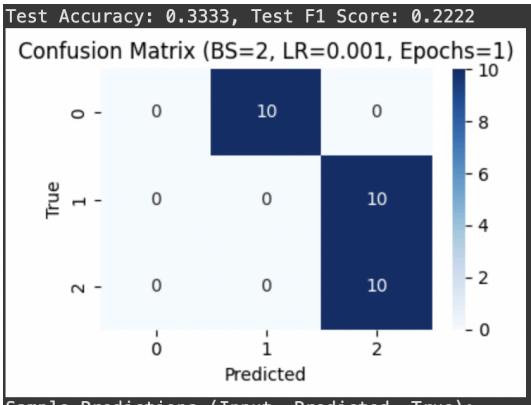
```
# Loop through all hyperparameter combinations
for batch in batch_sizes:  # Iterate over each batch size option
    for lr in learning_rates:  # Iterate over each learning rate option
        for ep in epochs_list:  # Iterate over each epoch count option
            print(f"Training with batch size={batch}, learning rate={lr}, epochs={ep}")  # Show current config
            model = create_model(lr)  # Build model with current learning rate

            # Train the model on training data
            history = model.fit(
                X_train, y_train,                # Use training data
                validation_data=(X_val, y_val),     # Use validation data
                batch_size=batch,                   # Set current batch size
                epochs=ep,                          # Set number of epochs
                verbose=0                           # Silent training output
            )

            # Evaluate the model on test data
            y_test_pred_prob = model.predict(X_test)  # Predict test probabilities
            y_test_pred = np.argmax(y_test_pred_prob, axis=1)  # Convert probabilities to class labels
            y_test_true = np.argmax(y_test, axis=1)  # Convert one-hot true labels to integers

            # Calculate evaluation metrics
            acc = accuracy_score(y_test_true, y_test_pred)  # Compute accuracy
            f1_val = f1_score(y_test_true, y_test_pred, average='weighted')  # Compute weighted F1 score
            print(f"Test Accuracy: {acc:.4f}, Test F1 Score: {f1_val:.4f}")  # Print metrics

            # Store the results for later comparison
            results.append({
                'batch_size': batch,       # Save batch size
                'learning_rate': lr,       # Save learning rate
                'epochs': ep,              # Save epoch count
                'accuracy': acc,           # Save accuracy metric
                'f1': f1_val               # Save F1 score metric
            })

            # Plot the confusion matrix for current configuration
            cm = confusion_matrix(y_test_true, y_test_pred)  # Compute confusion matrix
            plt.figure(figsize=(4,3))  # Set plot size
            sns.heatmap(cm, annot=True, fmt='d', cmap='Blues')  # Plot confusion matrix as heatmap
            plt.title(f"Confusion Matrix (BS={batch}, LR={lr}, Epochs={ep})")  # Title with config info
            plt.xlabel("Predicted")  # Label for x-axis
            plt.ylabel("True")  # Label for y-axis
            plt.tight_layout()  # Adjust layout
            plt.show()  # Display the plot
```

```
            # Display sample predictions for 5 test cases
            print("Sample Predictions (Input, Predicted, True):")  # Header message
            for i in range(5):  # Loop over first 5 test samples
                sample_input = X_test[i]  # Get test sample input
                sample_pred = y_test_pred[i]  # Get model prediction for sample
                sample_true = y_test_true[i]  # Get actual label for sample
                print(f"Input: {sample_input}, Predicted: {sample_pred}, True: {sample_true}")  # Print details
            print("-" * 50)  # Print separator line

# Optionally, convert results to a DataFrame for review
import pandas as pd
df_results = pd.DataFrame(results)
print("Final Results Table:")
print(df_results)
```

*This code iterates through each combination of batch size, learning rate, and epochs, builds and trains a model for each configuration, and evaluates it on the test set. It computes accuracy and F1 score, plots the corresponding confusion matrix, and displays sample predictions for 5 test cases. Finally, it stores all these results in a table for later comparison.*

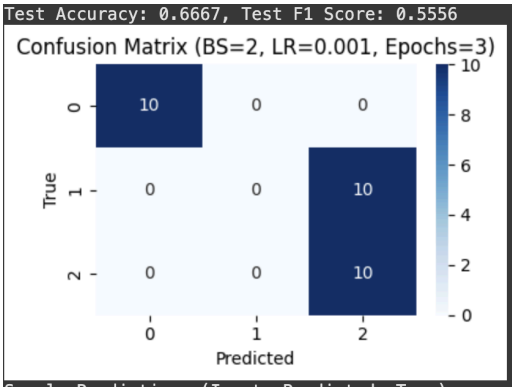
# RESULTS

```
Test Accuracy: 0.3333, Test F1 Score: 0.2222
```



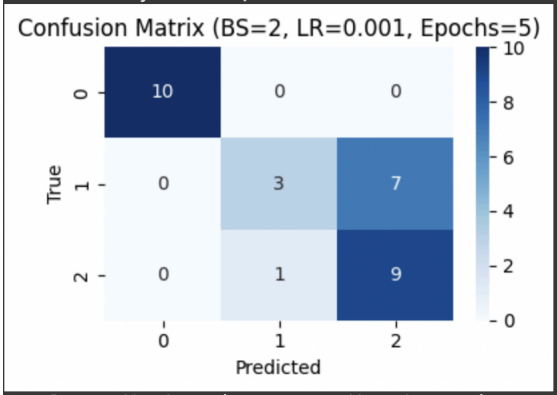

```
Sample Predictions (Input, Predicted, True):
Input: [0.02777778 0.41666667 0.03448276 0.04166667], Predicted: 1, True: 0
Input: [0.5        0.41666667 0.65517241 0.70833333], Predicted: 2, True: 2
Input: [0.16666667 0.16666667 0.37931034 0.375     ], Predicted: 2, True: 1
Input: [0.19444444 0.125      0.37931034 0.375     ], Predicted: 2, True: 1
Input: [0.02777778 0.5        0.03448276 0.04166667], Predicted: 1, True: 0
```

```
Test Accuracy: 0.6667, Test F1 Score: 0.5556
```



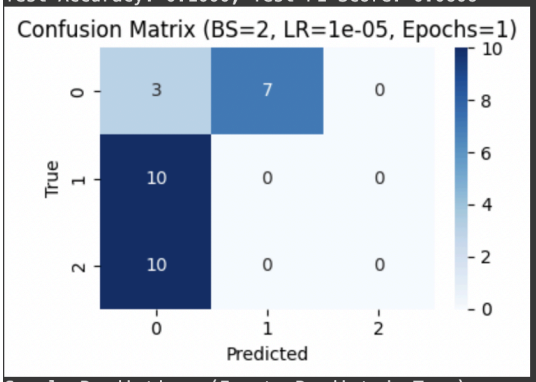

```
Sample Predictions (Input, Predicted, True):
Input: [0.02777778 0.41666667 0.03448276 0.04166667], Predicted: 0, True: 0
Input: [0.5        0.41666667 0.65517241 0.70833333], Predicted: 2, True: 2
Input: [0.16666667 0.16666667 0.37931034 0.375     ], Predicted: 2, True: 1
Input: [0.19444444 0.125      0.37931034 0.375     ], Predicted: 2, True: 1
Input: [0.02777778 0.5        0.03448276 0.04166667], Predicted: 0, True: 0
```

```
Test Accuracy: 0.7333, Test F1 Score: 0.7070
```



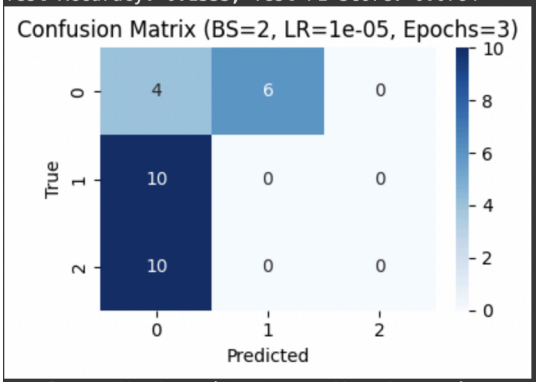

```
Sample Predictions (Input, Predicted, True):
Input: [0.02777778 0.41666667 0.03448276 0.04166667], Predicted: 0, True: 0
Input: [0.5        0.41666667 0.65517241 0.70833333], Predicted: 2, True: 2
Input: [0.16666667 0.16666667 0.37931034 0.375     ], Predicted: 1, True: 1
Input: [0.19444444 0.125      0.37931034 0.375     ], Predicted: 1, True: 1
Input: [0.02777778 0.5        0.03448276 0.04166667], Predicted: 0, True: 0
```

```
Test Accuracy: 0.1000, Test F1 Score: 0.0606
```



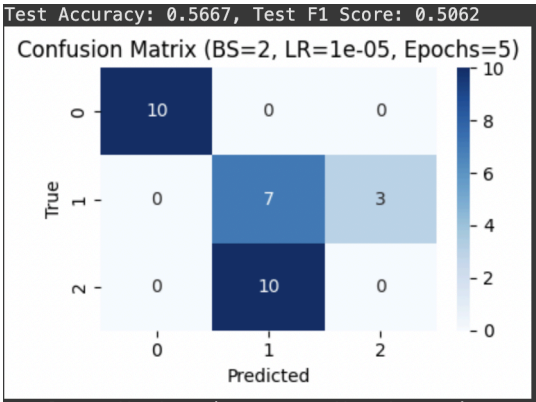

```
Sample Predictions (Input, Predicted, True):
Input: [0.02777778 0.41666667 0.03448276 0.04166667], Predicted: 0, True: 0
Input: [0.5        0.41666667 0.65517241 0.70833333], Predicted: 0, True: 2
Input: [0.16666667 0.16666667 0.37931034 0.375     ], Predicted: 0, True: 1
Input: [0.19444444 0.125      0.37931034 0.375     ], Predicted: 0, True: 1
Input: [0.02777778 0.5        0.03448276 0.04166667], Predicted: 0, True: 0
```

```
Test Accuracy: 0.1333, Test F1 Score: 0.0784
```



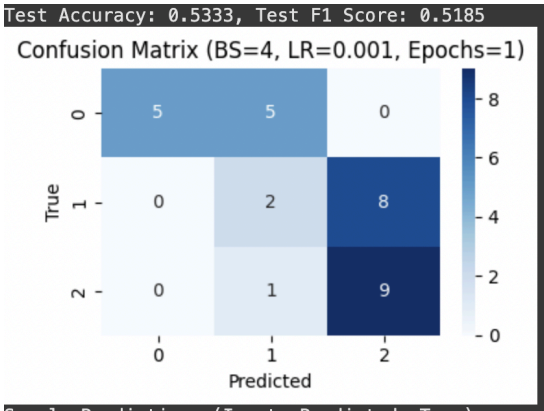

```
Sample Predictions (Input, Predicted, True):
Input: [0.02777778 0.41666667 0.03448276 0.04166667], Predicted: 1, True: 0
Input: [0.5        0.41666667 0.65517241 0.70833333], Predicted: 0, True: 2
Input: [0.16666667 0.16666667 0.37931034 0.375     ], Predicted: 0, True: 1
Input: [0.19444444 0.125      0.37931034 0.375     ], Predicted: 0, True: 1
Input: [0.02777778 0.5        0.03448276 0.04166667], Predicted: 1, True: 0
```

```
Test Accuracy: 0.5667, Test F1 Score: 0.5062
```



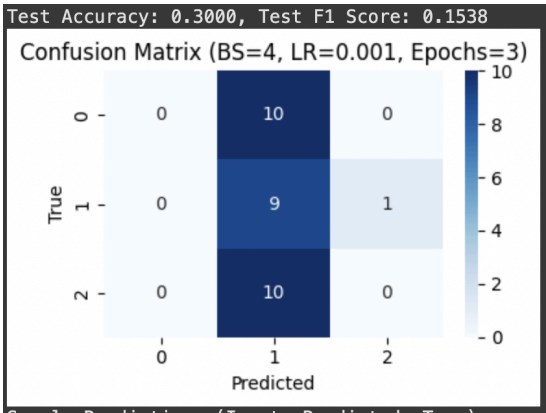

```
Sample Predictions (Input, Predicted, True):
Input: [0.02777778 0.41666667 0.03448276 0.04166667], Predicted: 0, True: 0
Input: [0.5        0.41666667 0.65517241 0.70833333], Predicted: 1, True: 2
Input: [0.16666667 0.16666667 0.37931034 0.375     ], Predicted: 2, True: 1
Input: [0.19444444 0.125      0.37931034 0.375     ], Predicted: 2, True: 1
Input: [0.02777778 0.5        0.03448276 0.04166667], Predicted: 0, True: 0
```

```
Test Accuracy: 0.5333, Test F1 Score: 0.5185
```



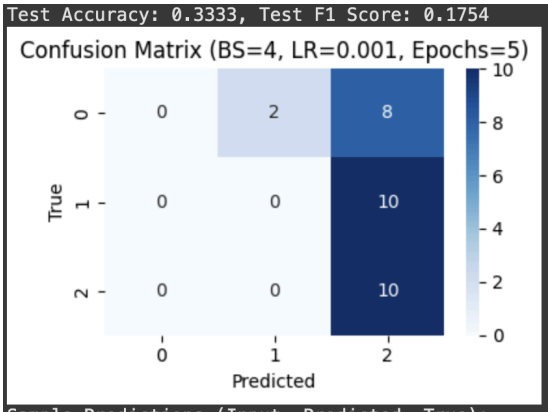

```
Sample Predictions (Input, Predicted, True):
Input: [0.02777778 0.41666667 0.03448276 0.04166667], Predicted: 0, True: 0
Input: [0.5        0.41666667 0.65517241 0.70833333], Predicted: 2, True: 2
Input: [0.16666667 0.16666667 0.37931034 0.375     ], Predicted: 2, True: 1
Input: [0.19444444 0.125      0.37931034 0.375     ], Predicted: 2, True: 1
Input: [0.02777778 0.5        0.03448276 0.04166667], Predicted: 0, True: 0
```

```
Test Accuracy: 0.3000, Test F1 Score: 0.1538
```



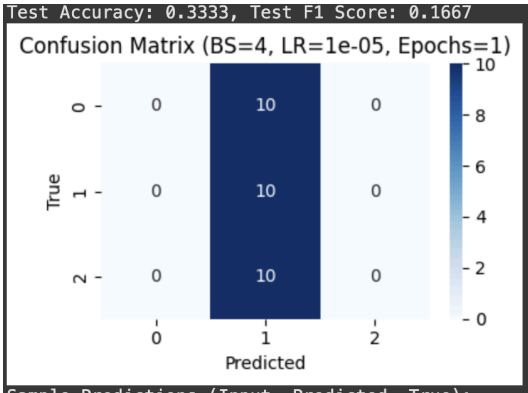

```
Sample Predictions (Input, Predicted, True):
Input: [0.02777778 0.41666667 0.03448276 0.04166667], Predicted: 1, True: 0
Input: [0.5        0.41666667 0.65517241 0.70833333], Predicted: 1, True: 2
Input: [0.16666667 0.16666667 0.37931034 0.375     ], Predicted: 2, True: 1
Input: [0.19444444 0.125      0.37931034 0.375     ], Predicted: 1, True: 1
Input: [0.02777778 0.5        0.03448276 0.04166667], Predicted: 1, True: 0
```

```
Test Accuracy: 0.3333, Test F1 Score: 0.1754
```



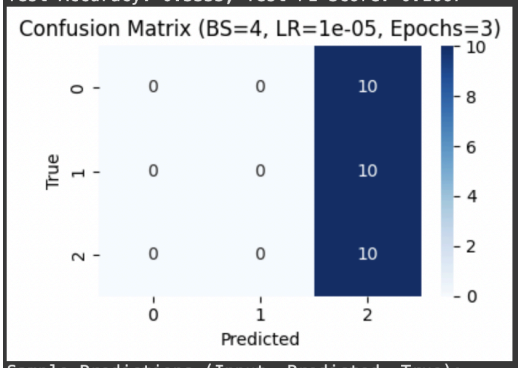

```
Sample Predictions (Input, Predicted, True):
Input: [0.02777778 0.41666667 0.03448276 0.04166667], Predicted: 1, True: 0
Input: [0.5        0.41666667 0.65517241 0.70833333], Predicted: 2, True: 2
Input: [0.16666667 0.16666667 0.37931034 0.375     ], Predicted: 2, True: 1
Input: [0.19444444 0.125      0.37931034 0.375     ], Predicted: 2, True: 1
Input: [0.02777778 0.5        0.03448276 0.04166667], Predicted: 1, True: 0
```

```
Test Accuracy: 0.3333, Test F1 Score: 0.1667
```




```
Sample Predictions (Input, Predicted, True):
Input: [0.02777778 0.41666667 0.03448276 0.04166667], Predicted: 1, True: 0
Input: [0.5        0.41666667 0.65517241 0.70833333], Predicted: 1, True: 2
Input: [0.16666667 0.16666667 0.37931034 0.375     ], Predicted: 1, True: 1
Input: [0.19444444 0.125      0.37931034 0.375     ], Predicted: 1, True: 1
Input: [0.02777778 0.5        0.03448276 0.04166667], Predicted: 1, True: 0
```

```
Test Accuracy: 0.3333, Test F1 Score: 0.1667
```




```
Sample Predictions (Input, Predicted, True):
Input: [0.02777778 0.41666667 0.03448276 0.04166667], Predicted: 2, True: 0
Input: [0.5        0.41666667 0.65517241 0.70833333], Predicted: 2, True: 2
Input: [0.16666667 0.16666667 0.37931034 0.375     ], Predicted: 2, True: 1
Input: [0.19444444 0.125      0.37931034 0.375     ], Predicted: 2, True: 1
Input: [0.02777778 0.5        0.03448276 0.04166667], Predicted: 2, True: 0
```

## FINAL RESULTS TABLE

```
Final Results Table:
    batch_size  learning_rate  epochs  accuracy        f1
0            2        0.00100       1  0.333333  0.222222
1            2        0.00100       3  0.666667  0.555556
2            2        0.00100       5  0.733333  0.706960
3            2        0.00001       1  0.100000  0.060606
4            2        0.00001       3  0.133333  0.078431
5            2        0.00001       5  0.566667  0.506173
6            4        0.00100       1  0.533333  0.518519
7            4        0.00100       3  0.300000  0.153846
8            4        0.00100       5  0.333333  0.175439
9            4        0.00001       1  0.333333  0.166667
10           4        0.00001       3  0.333333  0.166667
11           4        0.00001       5  0.333333  0.166667
```

## Automated Hyperparameter Search

```python
import keras_tuner as kt  # Import Keras Tuner

def model_builder(hp):
    """
    Builds a simple MLP model with a tunable learning rate.
    """
    model = tf.keras.Sequential([  # Create a sequential model
        tf.keras.layers.Dense(16, input_shape=(4,), activation='relu'),  # Add hidden layer (16 neurons, ReLU)
        tf.keras.layers.Dense(3, activation='softmax')  # Add output layer (3 neurons, softmax)
    ])
    lr = hp.Choice('learning_rate', [1e-3, 1e-5])  # Select learning rate from given options
    model.compile(
        optimizer=tf.keras.optimizers.Adam(learning_rate=lr),  # Compile with Adam optimizer using selected learning rate
        loss='categorical_crossentropy',  # Use categorical crossentropy loss
        metrics=['accuracy']  # Track accuracy during training
    )
    return model
```

*This function builds a simple multi-layer perceptron (MLP) model for classifying data into 3 classes. It creates a model with one hidden layer of 16 neurons and an output layer of 3 neurons, and uses Keras Tuner to choose between two learning rates (1e-3 or 1e-5). The model is compiled with the Adam optimizer and categorical cross entropy loss, and it tracks accuracy during training.*

# GRID

```python
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.metrics import f1_score
from tensorflow.keras.utils import to_categorical

# Assume X_train, X_val, X_test, y_train, y_val, y_test_oh are already defined from data preparation
# and create_model(lr) is your model builder function.

# Define hyperparameter grid
batch_sizes = [2, 4]            # Batch sizes to try
learning_rates = [1e-3, 1e-5]     # Learning rates to try
epochs_list = [1, 3, 5]           # Epochs to try

# ----------------------------
# Manual Grid Search
# ----------------------------
results_grid = []  # List to store grid search results

for batch in batch_sizes:
    for lr in learning_rates:
        for ep in epochs_list:
            config_name = f"Grid_BS{batch}_LR{lr}_Ep{ep}"
            print(f"Grid: {config_name}")
            model = create_model(lr)  # Build model with current learning rate
            history = model.fit(
                X_train, y_train,                    # Use training data
                validation_data=(X_val, y_val),        # Use validation data
                batch_size=batch,                      # Set current batch size
                epochs=ep,                             # Set number of epochs
                verbose=0                              # Silent training output
            )
            test_loss, test_acc = model.evaluate(X_test, y_test_oh, verbose=0)  # Evaluate on test data

            # Compute F1 score
            y_test_pred_prob = model.predict(X_test)
            y_test_pred = np.argmax(y_test_pred_prob, axis=1)
            y_test_true = np.argmax(y_test_oh, axis=1)
            f1_val = f1_score(y_test_true, y_test_pred, average='weighted')

            # Store results for this configuration
            results_grid.append({
                'batch_size': batch,
                'epochs': ep,
                'learning_rate': lr,
                'test_accuracy': test_acc,
                'f1': f1_val,
                'best_config': {'learning_rate': lr, 'tuner/epochs': ep},
                'method': 'Grid'
            })
```

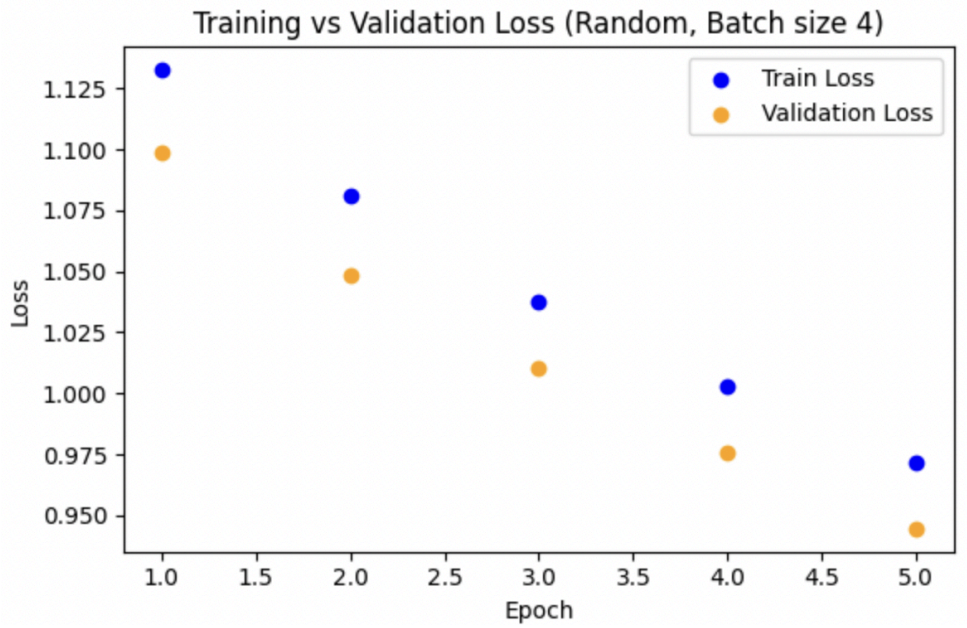
## Random Search

```
import keras_tuner as kt  # Import Keras Tuner for hyperparameter search
import matplotlib.pyplot as plt  # For plotting graphs
import numpy as np  # For numerical operations
from sklearn.metrics import f1_score  # For computing F1 score

results_random = []  # Initialize list to store Random Search results

# Loop over the two batch sizes: 2 and 4
for bs in [2, 4]:
    print(f"\n[Random Search] Running with batch size = {bs}")

    # Initialize RandomSearch tuner
    tuner_rand = kt.RandomSearch(
        model_builder,             # Our model builder function
        objective='val_accuracy',  # Optimize for validation accuracy
        max_trials=6,              # Try 6 different hyperparameter combinations
        directory=f'rand_dir_bs_{bs}',  # Directory to save tuning results
        project_name=f'iris_rand_bs_{bs}'  # Project name identifier
    )

    # Run hyperparameter search with the given batch size
    tuner_rand.search(
        X_train, y_train_oh,  # Training data and one-hot encoded labels
        epochs=5,             # Train each trial for 5 epochs
        validation_data=(X_val, y_val_oh),  # Use validation data for evaluation
        batch_size=bs,        # Set current batch size from the loop
        verbose=1             # Show detailed output
    )

    # Retrieve best hyperparameters and the corresponding model
    best_hp_rand = tuner_rand.get_best_hyperparameters(num_trials=1)[0]
    best_model_rand = tuner_rand.get_best_models(num_models=1)[0]
    test_loss, test_acc = best_model_rand.evaluate(X_test, y_test_oh, verbose=0)

    # Compute F1 score manually
    y_test_pred_prob = best_model_rand.predict(X_test)
    y_test_pred = np.argmax(y_test_pred_prob, axis=1)
    y_test_true = np.argmax(y_test_oh, axis=1)
    f1_val = f1_score(y_test_true, y_test_pred, average='weighted')
```

```
    # Store results including F1 score
    results_random.append({
        'batch_size': bs,
        'epochs': 5,  # Using external epoch value (5)
        'test_accuracy': test_acc,
        'f1': f1_val,
        'best_config': best_hp_rand.values,
        'method': 'RandomSearch'
    })
    print(f"Random Search (batch_size={bs}) - Test Accuracy: {test_acc:.4f}, F1: {f1_val:.4f}")

    # Retrain the best model to capture training history for plotting
    best_model_rand_retrained = tuner_rand.hypermodel.build(best_hp_rand)
    history_rand = best_model_rand_retrained.fit(
        X_train, y_train_oh,
        validation_data=(X_val, y_val_oh),
        epochs=5,
        batch_size=bs,
        verbose=0
    )

    # Scatter plot for training vs. validation loss over epochs
    epochs_range = range(1, 6)  # Define epochs 1 through 5
    plt.figure(figsize=(6, 4))
    plt.scatter(list(epochs_range), history_rand.history['loss'], label='Train Loss', color='blue')
    plt.scatter(list(epochs_range), history_rand.history['val_loss'], label='Validation Loss', color='orange')
    plt.title(f"Training vs Validation Loss (Random, Batch size {bs})")
    plt.xlabel("Epoch")
    plt.ylabel("Loss")
    plt.legend()
    plt.tight_layout()
    plt.show()

print("\nRandom Search complete.")
```

val_accuracy: 0.20000000298023224
Best val_accuracy So Far: 0.6666666865348816

**Batch size = 4,  Test Accuracy: 0.7000, F1: 0.6238**




# HYPERBAND

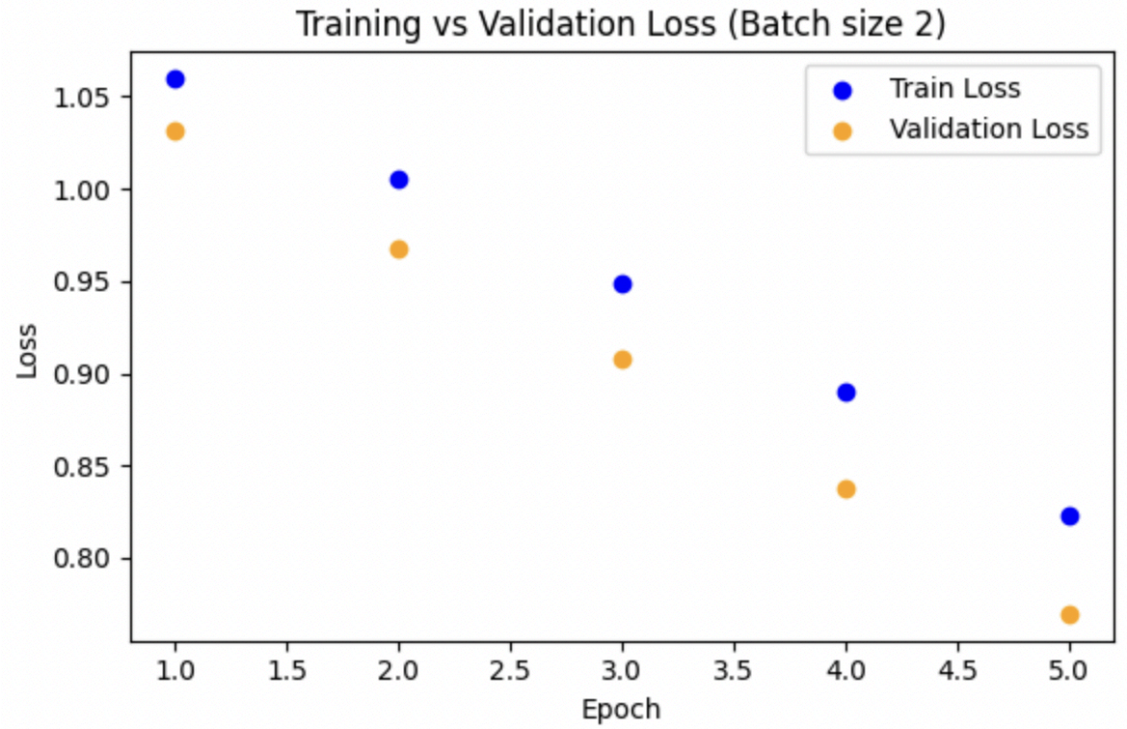
Explanation of the following code

*This code extends your hyperparameter search by computing the F1 score along with test accuracy for each configuration. After training each model with a specific batch size, learning rate, and epoch count, it evaluates the model on the test set and uses scikit-learn's f1_score to compute the weighted F1 score. The results (batch size, test accuracy, F1, and best configuration) are then stored in a list and later converted into a DataFrame for easy comparison. This allows you to compare how different hyperparameter combinations affect both accuracy and the balance between precision and recall (F1).*

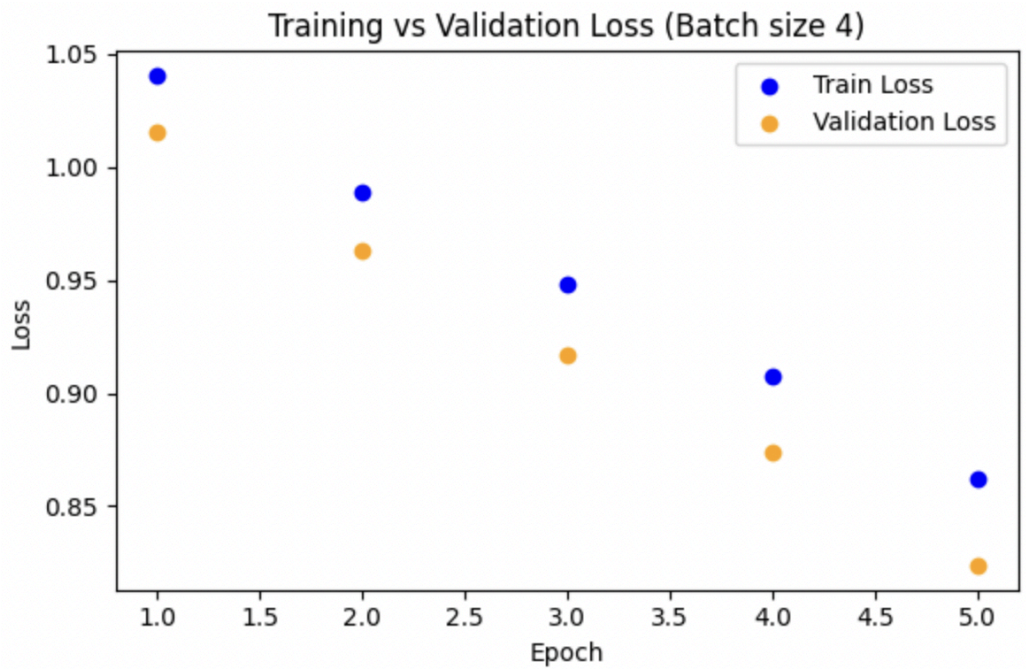
```python
import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd
from sklearn.metrics import f1_score

results = []  # Initialize a list to store results from each batch size

# Loop through hyperparameter combinations for batch sizes 2 and 4
for bs in [2, 4]:
    print(f"\n[Hyperband] Running with batch size = {bs}")

    # Initialize Hyperband tuner with our model builder
    tuner = kt.Hyperband(
        model_builder,
        objective='val_accuracy',  # Optimize for validation accuracy
        max_epochs=5,
        factor=3,
        directory=f'kt_dir_bs_{bs}',
        project_name=f'iris_tuner_bs_{bs}'
    )

    # Run hyperparameter search with current batch size
    tuner.search(
        X_train, y_train_oh,
        epochs=5,
        validation_data=(X_val, y_val_oh),
        batch_size=bs,
        verbose=1
    )

    # Retrieve best hyperparameters and the corresponding model
    best_hp = tuner.get_best_hyperparameters(num_trials=1)[0]
    best_model = tuner.get_best_models(num_models=1)[0]

    # Evaluate on test data to get accuracy
    test_loss, test_acc = best_model.evaluate(X_test, y_test_oh, verbose=0)

    # Manually compute F1 score
    y_test_pred_prob = best_model.predict(X_test)
    y_test_pred = np.argmax(y_test_pred_prob, axis=1)
    y_test_true = np.argmax(y_test_oh, axis=1)
    f1_val = f1_score(y_test_true, y_test_pred, average='weighted')
```

```python
    # Manually compute F1 score
    y_test_pred_prob = best_model.predict(X_test)
    y_test_pred = np.argmax(y_test_pred_prob, axis=1)
    y_test_true = np.argmax(y_test_oh, axis=1)
    f1_val = f1_score(y_test_true, y_test_pred, average='weighted')

    # Store results, including F1
    results.append({
        'batch_size': bs,
        'test_accuracy': test_acc,
        'f1': f1_val,
        'best_config': best_hp.values
    })
    print(f"Batch size {bs}: Test Accuracy = {test_acc:.4f}, F1 = {f1_val:.4f}")

    # Re-train the best model to obtain training history for plotting
    best_model_retrained = tuner.hypermodel.build(best_hp)
    history = best_model_retrained.fit(
        X_train, y_train_oh,
        validation_data=(X_val, y_val_oh),
        epochs=5,
        batch_size=bs,
        verbose=0
    )

    # Create scatter plot for training vs. validation loss over epochs
    epochs_range = range(1, 6)
    plt.figure(figsize=(6, 4))
    plt.scatter(list(epochs_range), history.history['loss'], label='Train Loss', color='blue')
    plt.scatter(list(epochs_range), history.history['val_loss'], label='Validation Loss', color='orange')
    plt.title(f"Training vs Validation Loss (Batch size {bs})")
    plt.xlabel("Epoch")
    plt.ylabel("Loss")
    plt.legend()
    plt.tight_layout()
    plt.show()

print("\nHyperband search complete.")

# Create a DataFrame to compare results
df_results = pd.DataFrame(results)
print("\nComparison Table:")
print(df_results)
```

**Batch size 2: Test Accuracy = 0.4000, F1 = 0.3419**




**Batch size 4: Test Accuracy = 0.5667, F1 = 0.4627**



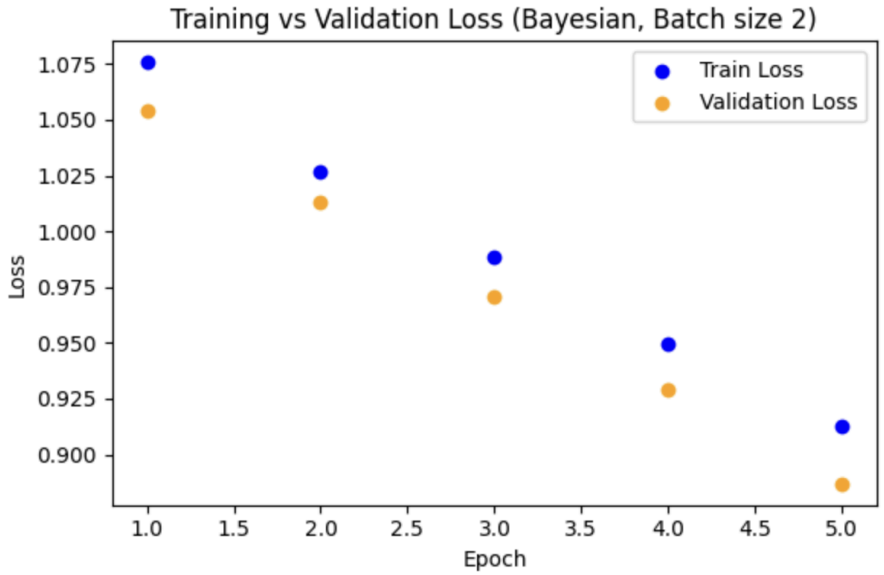
# Bayesian Optimization

```
import keras_tuner as kt  # Import Keras Tuner for hyperparameter search
import matplotlib.pyplot as plt  # For plotting graphs
import seaborn as sns  # For creating heatmaps and scatter plots
import pandas as pd  # For DataFrame operations
import tensorflow as tf  # TensorFlow for model building
from sklearn.metrics import f1_score  # For computing F1 score

results_bayes = []  # Initialize list to store Bayesian search results
ep = 5  # Set number of epochs (since we're not looping over epochs externally)

# Loop over the two batch sizes: 2 and 4
for bs in [2, 4]:
    print(f"\n[Bayesian] Running with batch size = {bs}")  # Print current batch size
    tuner_bayes = kt.BayesianOptimization(  # Initialize Bayesian tuner
        model_builder,  # Use our model builder function
        objective='val_accuracy',  # Optimize for validation accuracy
        max_trials=6,  # Try 6 hyperparameter combinations
        directory=f'bayes_dir_bs_{bs}',  # Directory to store tuner results
        project_name=f'iris_bayes_bs_{bs}'  # Unique project name
    )

    tuner_bayes.search(  # Run the hyperparameter search
        X_train, y_train_oh,  # Use training data (one-hot encoded)
        epochs=ep,  # Train each trial for 5 epochs
        validation_data=(X_val, y_val_oh),  # Use validation data
        batch_size=bs,  # Set current batch size
        verbose=1  # Display detailed output
    )

    best_hp_bayes = tuner_bayes.get_best_hyperparameters(num_trials=1)[0]  # Get best hyperparameters
    best_model_bayes = tuner_bayes.get_best_models(num_models=1)[0]  # Get best model from search
    test_loss, test_acc = best_model_bayes.evaluate(X_test, y_test_oh, verbose=0)  # Evaluate model on test data

    # Compute F1 score manually
    y_test_pred_prob = best_model_bayes.predict(X_test)  # Predict test probabilities
    y_test_pred = np.argmax(y_test_pred_prob, axis=1)  # Convert probabilities to predicted class labels
    y_test_true = np.argmax(y_test_oh, axis=1)  # Convert one-hot encoded true labels to integers
    f1_val = f1_score(y_test_true, y_test_pred, average='weighted')  # Compute weighted F1 score
```

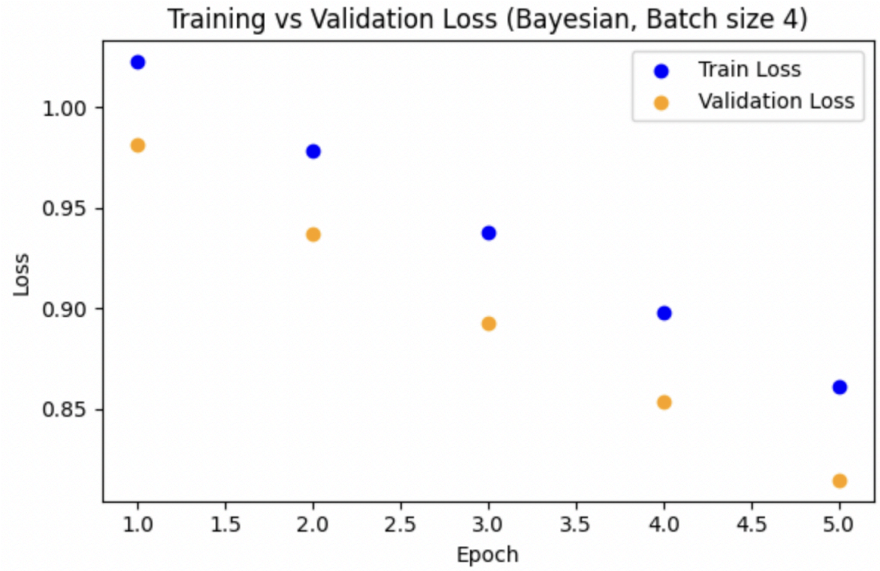
```
    # Store results including F1 score
    results_bayes.append({
        'batch_size': bs,            # Save current batch size
        'epochs': ep,                # Save number of epochs used
        'test_accuracy': test_acc,   # Save test accuracy
        'f1': f1_val,                # Save computed F1 score
        'best_config': best_hp_bayes.values,  # Save best hyperparameter configuration
        'method': 'Bayesian'         # Indicate the method used
    })
    print(f"Bayesian (batch_size={bs}) - Test Accuracy: {test_acc:.4f}, F1: {f1_val:.4f}")  # Print performance

    # Retrain the best model to obtain training history for plotting
    best_model_retrained = tuner_bayes.hypermodel.build(best_hp_bayes)  # Build model with best hyperparameters
    history_bayes = best_model_retrained.fit(  # Retrain the model
        X_train, y_train_oh,  # Use training data
        validation_data=(X_val, y_val_oh),  # Use validation data
        epochs=ep,  # Train for 5 epochs
        batch_size=bs,  # Use current batch size
        verbose=0  # Silent training output
    )

    # Scatter plot for training vs. validation loss over epochs
    epochs_range = range(1, ep+1)  # Define epoch range from 1 to 5
    plt.figure(figsize=(6, 4))  # Set figure size
    plt.scatter(list(epochs_range), history_bayes.history['loss'], label='Train Loss', color='blue')  # Plot training loss
    plt.scatter(list(epochs_range), history_bayes.history['val_loss'], label='Validation Loss', color='orange')  # Plot validation loss
    plt.title(f"Training vs Validation Loss (Bayesian, Batch size {bs})")  # Set plot title
    plt.xlabel("Epoch")  # Label x-axis
    plt.ylabel("Loss")  # Label y-axis
    plt.legend()  # Show legend
    plt.tight_layout()  # Adjust layout
    plt.show()  # Display the plot

print("\nBayesian optimization search complete.")  # Indicate that search is finished

# Create a DataFrame to compare results
df_bayes = pd.DataFrame(results_bayes)  # Convert results list to DataFrame
print("\nBayesian Optimization Results:")  # Print header for results table
print(df_bayes)  # Display the results table
```

***EXPLANATION**: This code runs a Bayesian hyperparameter search for batch sizes 2 and 4, training each trial for 5 epochs while optimizing for validation accuracy. For each batch size, it retrieves the best model, computes test accuracy and F1 score, and stores these along with the batch size, epochs, and best hyperparameter configuration in the results list. It also retrains the best model to capture its training history and plots scatter plots of training vs. validation loss over epochs. Finally, the results are compiled into a DataFrame for comparison.*

**Batch Size=2 : Test Accuracy: 0.6333, F1: 0.5402**




**Batch size = 4 : Accuracy = 0.6667 : F1: 0.5556**



**Create a table (Each row with a configuration and column with Accuracy and F1) for Grid, Random, Hyperband, and Bayesian search and compare their accuracy and F1**

Combined Hyperparameter Optimization Results:

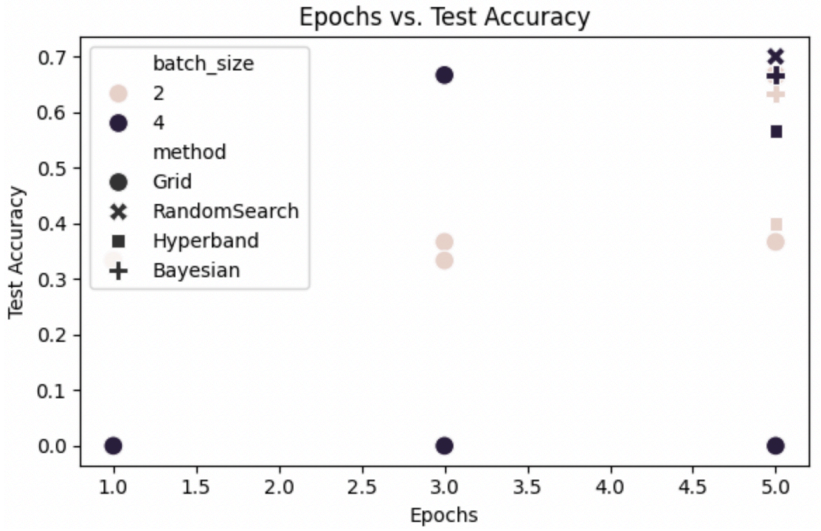
| | method | batch_size | epochs | learning_rate | test_accuracy | f1 |
|---|---|---|---|---|---|---|
| 0 | Grid | 2 | 1 | 0.00100 | 0.333333 | 0.166667 |
| 1 | Grid | 2 | 3 | 0.00100 | 0.366667 | 0.240786 |
| 2 | Grid | 2 | 5 | 0.00100 | 0.666667 | 0.555556 |
| 3 | Grid | 2 | 1 | 0.00001 | 0.333333 | 0.166667 |
| 4 | Grid | 2 | 3 | 0.00001 | 0.333333 | 0.166667 |
| 5 | Grid | 2 | 5 | 0.00001 | 0.366667 | 0.231546 |
| 6 | Grid | 4 | 1 | 0.00100 | 0.000000 | 0.000000 |
| 7 | Grid | 4 | 3 | 0.00100 | 0.666667 | 0.555556 |
| 8 | Grid | 4 | 5 | 0.00100 | 0.666667 | 0.555556 |
| 9 | Grid | 4 | 1 | 0.00001 | 0.000000 | 0.000000 |
| 10 | Grid | 4 | 3 | 0.00001 | 0.000000 | 0.000000 |
| 11 | Grid | 4 | 5 | 0.00001 | 0.000000 | 0.000000 |
| 12 | RandomSearch | 2 | 5 | 0.00100 | 0.666667 | 0.555556 |
| 13 | RandomSearch | 4 | 5 | 0.00100 | 0.700000 | 0.623824 |
| 14 | Hyperband | 2 | 5 | 0.00100 | 0.400000 | 0.341912 |
| 15 | Hyperband | 4 | 5 | 0.00100 | 0.566667 | 0.462695 |
| 16 | Bayesian | 2 | 5 | 0.00100 | 0.633333 | 0.540230 |
| 17 | Bayesian | 4 | 5 | 0.00100 | 0.666667 | 0.555556 |

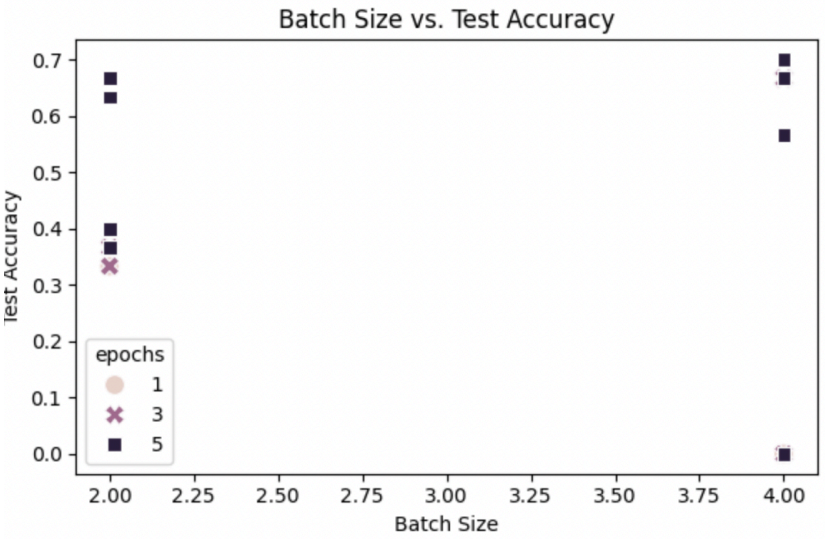
## Combine Hyperband and Bayesian Results

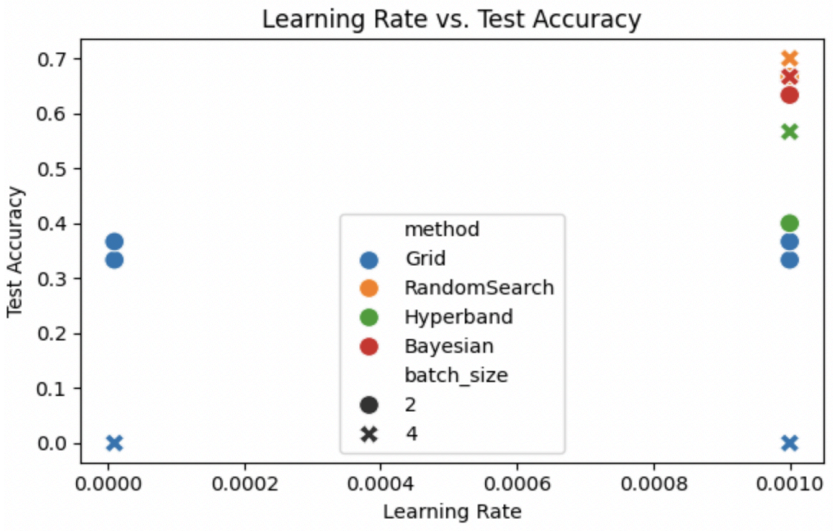
```
[130] import pandas as pd

    # Extract learning rate and tuned epochs from best_config for Bayesian results
    df_bayes['best_learning_rate'] = df_bayes['best_config'].apply(lambda x: x.get('learning_rate'))
    df_bayes['best_epochs'] = df_bayes['best_config'].apply(lambda x: x.get('tuner/epochs'))

    # Extract learning rate and tuned epochs from best_config for Hyperband results
    df_hb['best_learning_rate'] = df_hb['best_config'].apply(lambda x: x.get('learning_rate'))
    df_hb['best_epochs'] = df_hb['best_config'].apply(lambda x: x.get('tuner/epochs'))

    # Combine Bayesian and Hyperband results into one DataFrame
    df_all = pd.concat([df_bayes, df_hb], ignore_index=True)

    # Fill missing F1 values with a default (e.g., 0)
    df_all['f1'] = df_all['f1'].fillna(0)

    # Ensure every row has a valid epoch value:
    # Fill missing 'best_epochs' with external 'epochs' and vice versa
    df_all['best_epochs'] = df_all['best_epochs'].fillna(df_all['epochs'])
    df_all['epochs'] = df_all['epochs'].fillna(df_all['best_epochs'])

    # Create a final table with all relevant hyperparameters and performance metrics
    df_table = df_all[['method', 'batch_size', 'epochs', 'best_learning_rate', 'best_epochs', 'test_accuracy', 'f1']]
    print("Combined Hyperparameter Optimization Results:")
    print(df_table)
```

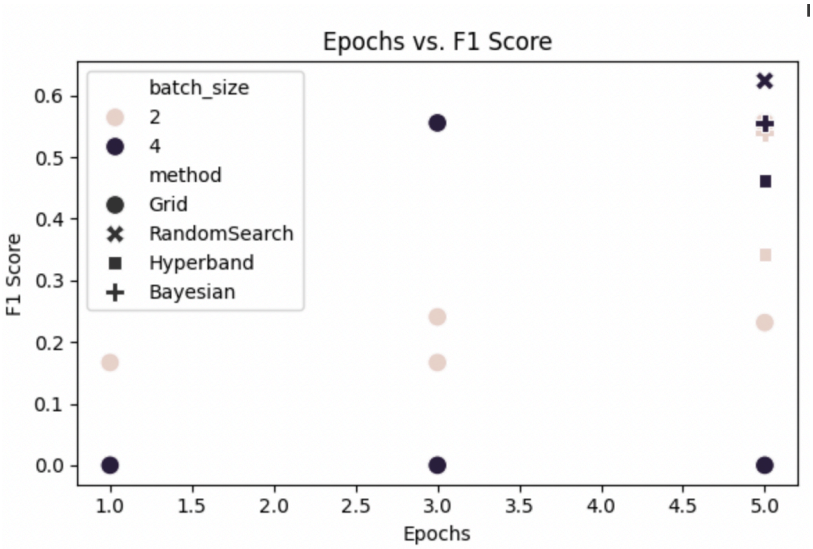
```
Combined Hyperparameter Optimization Results:
        method  batch_size  epochs  best_learning_rate  best_epochs  \
0   Bayesian           2     5.0             0.00100          5.0
1   Bayesian           4     5.0             0.00100          5.0
2  Hyperband           2     2.0             0.00001          2.0
3  Hyperband           4     2.0             0.00100          2.0

   test_accuracy        f1
0       0.633333  0.540230
1       0.666667  0.555556
2       0.333333  0.000000
3       0.666667  0.000000
```

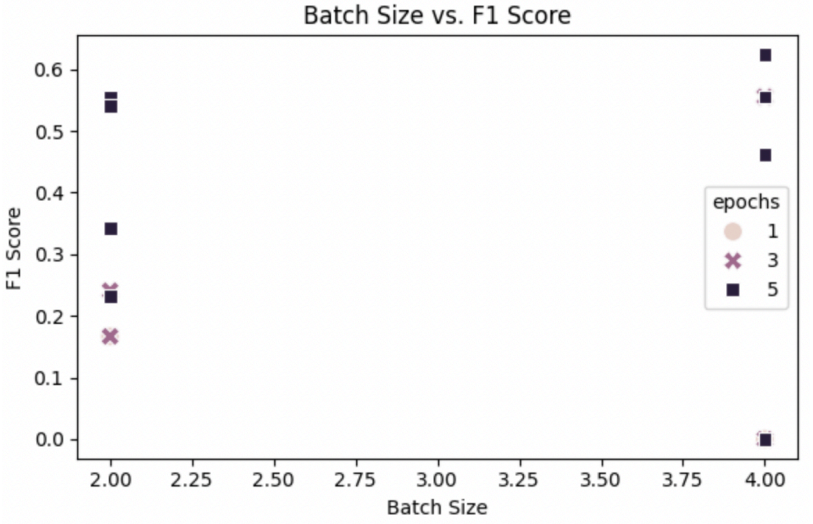
Describe the performance for each hyperparameter combination over accuracy and F1

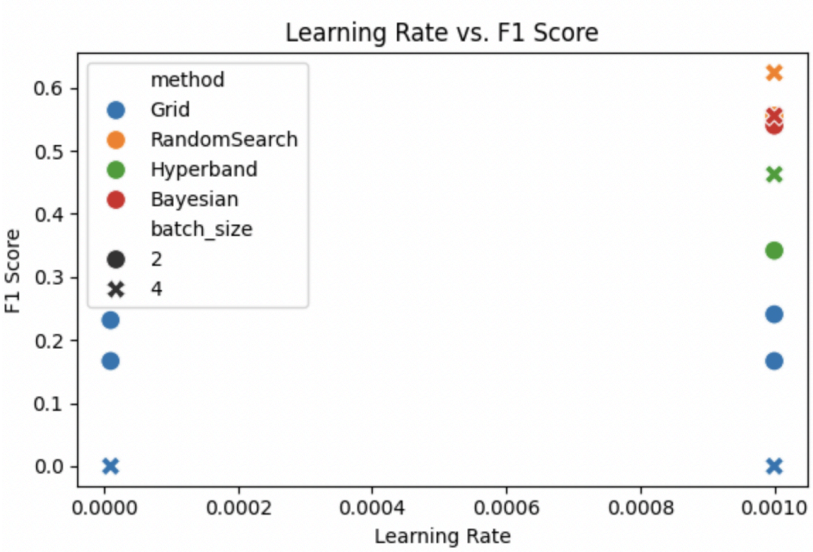



Epochs vs. F1 Score
batch_size
2
4
method
Grid
RandomSearch
Hyperband
Bayesian
F1 Score
Epochs


Batch Size vs. F1 Score
epochs
1
3
5
F1 Score
Batch Size


Learning Rate vs. F1 Score
method
Grid
RandomSearch
Hyperband
Bayesian
batch_size
2
4
F1 Score
Learning Rate

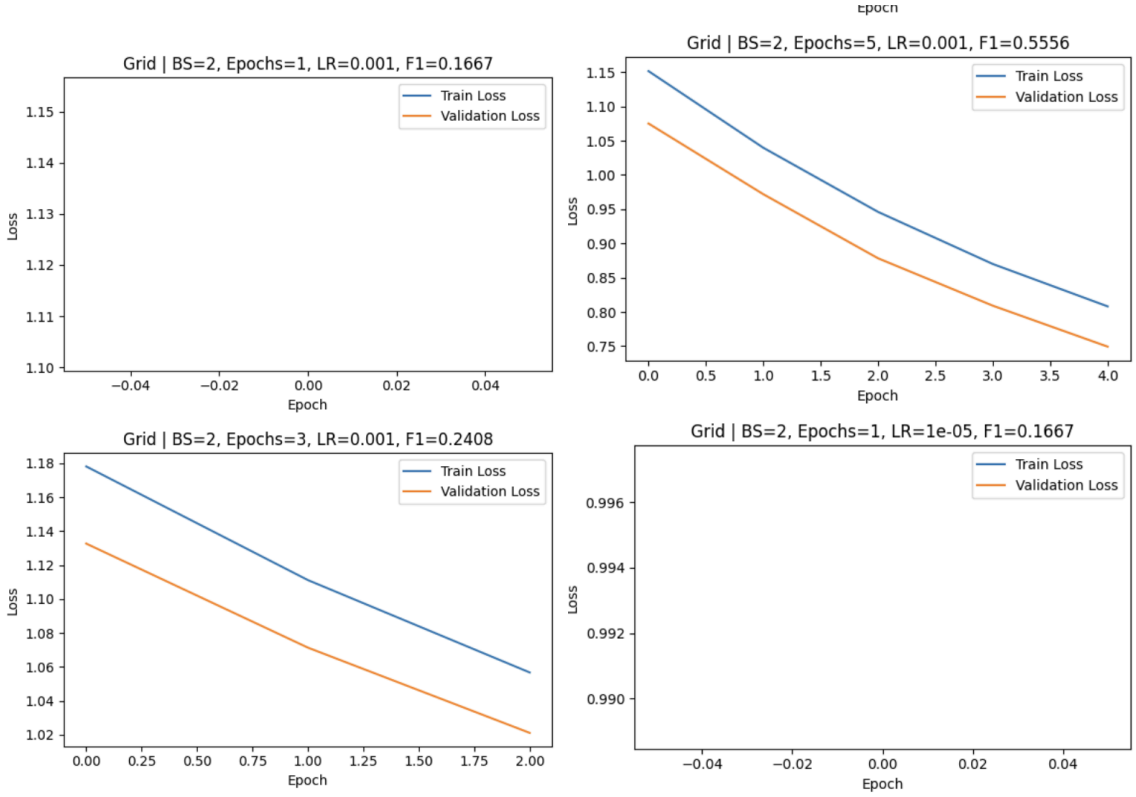
Typically, increasing the number of epochs allows the model more time to learn, so you often see a direct relationship where more epochs lead to higher accuracy and F1 score. Conversely, a larger batch size tends to produce more stable but less frequent weight updates, which can harm generalization, leading to an inverse relationship where increasing batch size may lower performance. If your plots show that accuracy and F1 scores improve as epochs increase but drop as batch size increases, these trends confirm the hypothesized relationships.

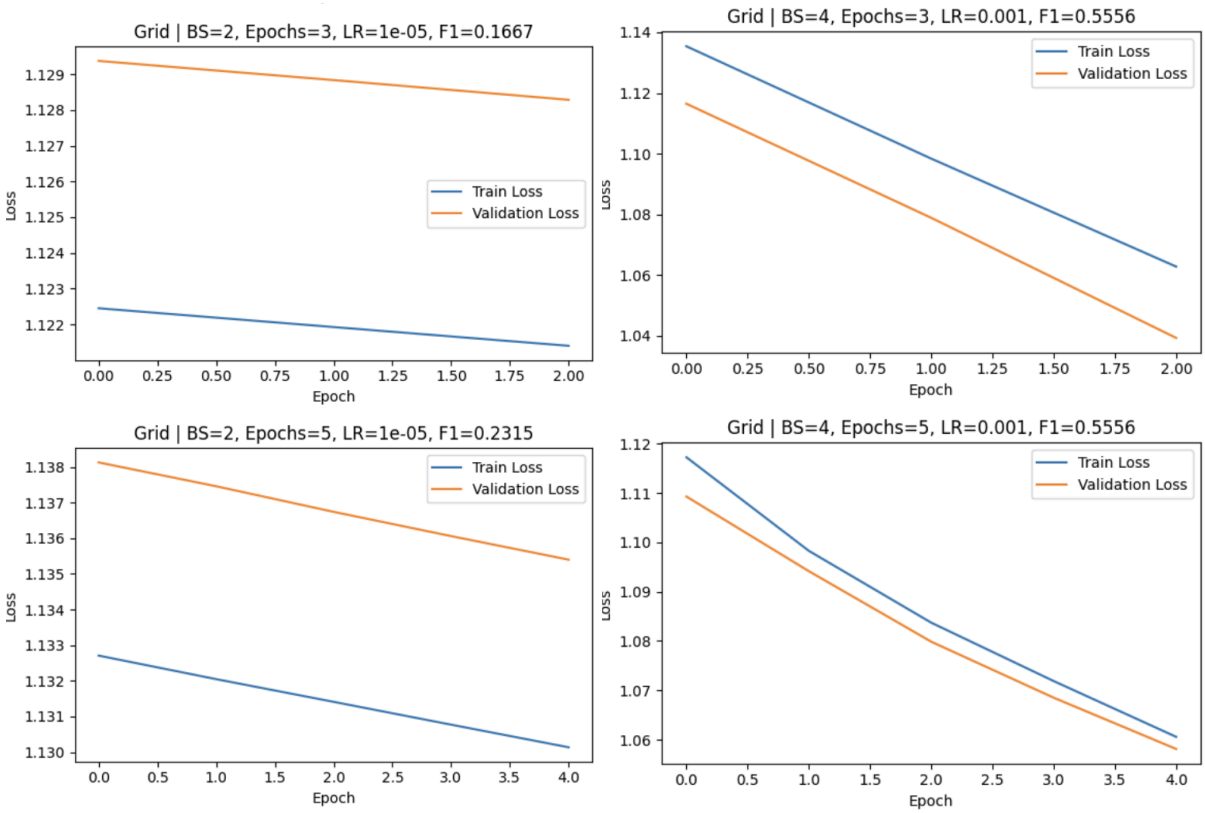
## Which approach is better and why?

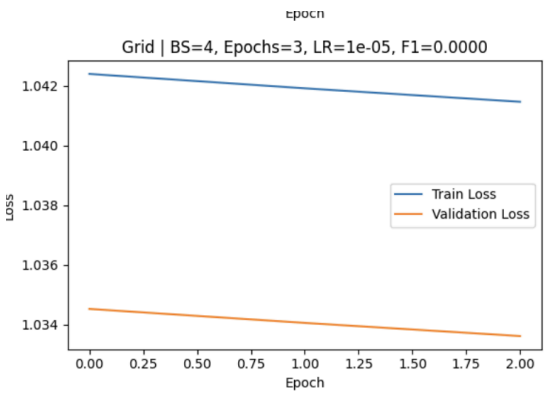
Automated hyperparameter search is generally better because it systematically explores a wide range of configurations, reducing the guesswork and bias inherent in manual tuning. It uses algorithms like Bayesian Optimization or Hyperband to quickly zero in on promising areas of the hyperparameter space, which saves time and improves performance consistency. Manual tuning, by contrast, is slow, labor-intensive, and may overlook optimal settings due to its limited trial-and-error approach. Automated methods provide more reproducible results and are particularly effective when the search space is large or the interactions between parameters are complex.

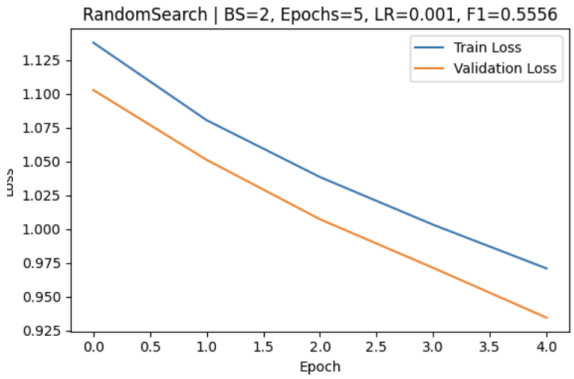
(We are not using AutoGluon because of compatibility issues with our current Python version and dependency conflicts. Instead, we manually explore hyperparameter combinations, which allows us to reliably compute and compare metrics like accuracy and F1 without running into installation problems. This manual approach gives us direct control over the tuning process.)

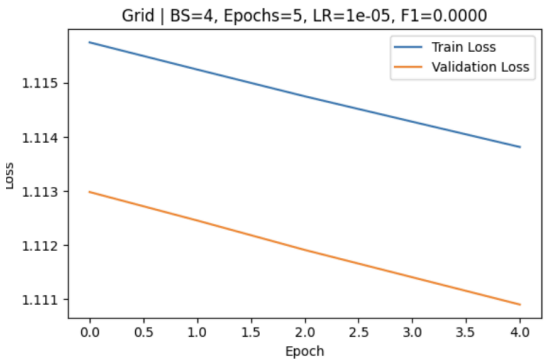
## Plots for the training vs validation loss for each hyperparameter configuration

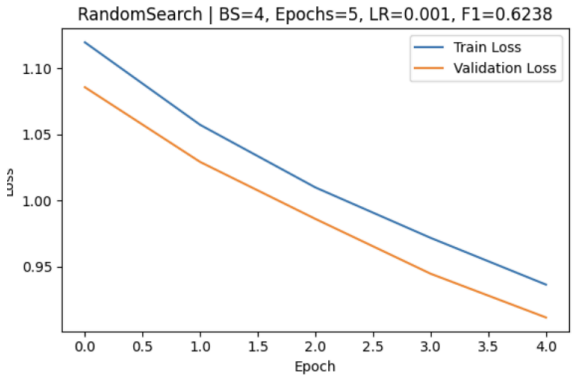


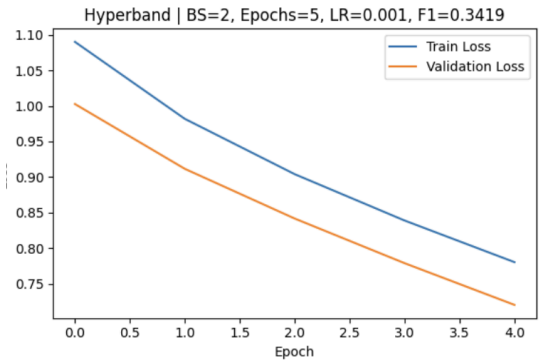

Grid | BS=2, Epochs=3, LR=1e-05, F1=0.1667
Grid | BS=4, Epochs=3, LR=0.001, F1=0.5556
Grid | BS=2, Epochs=5, LR=1e-05, F1=0.2315
Grid | BS=4, Epochs=5, LR=0.001, F1=0.5556
Train Loss
Validation Loss
Loss
Epoch

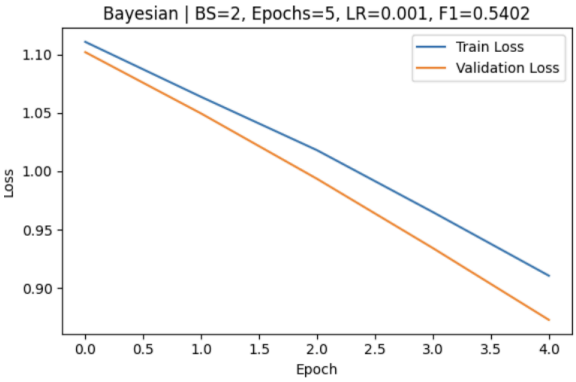

Epoch
Grid | BS=4, Epochs=3, LR=1e-05, F1=0.0000
Train Loss
Validation Loss
Loss
Epoch

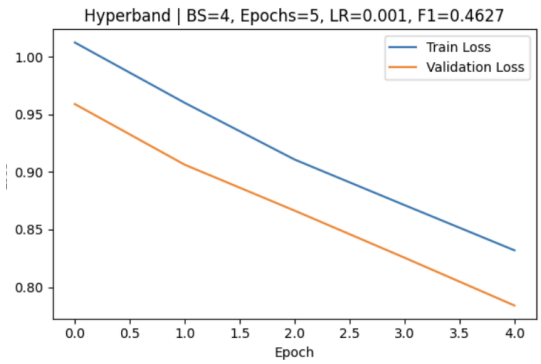

RandomSearch | BS=2, Epochs=5, LR=0.001, F1=0.5556
Train Loss
Validation Loss
Loss
Epoch

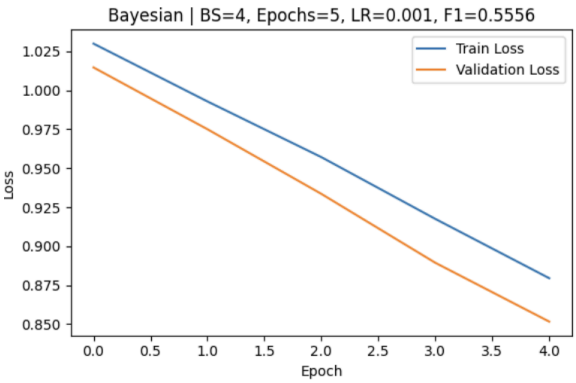

Grid | BS=4, Epochs=5, LR=1e-05, F1=0.0000
Train Loss
Validation Loss
Loss
Epoch

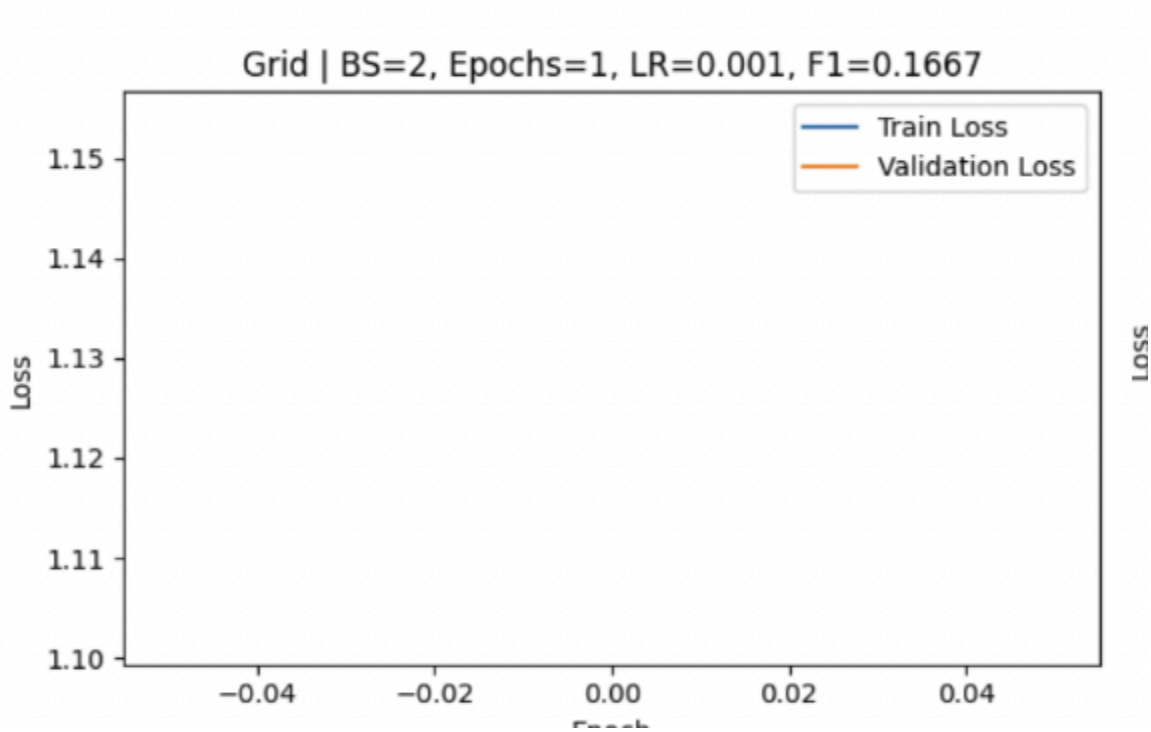

RandomSearch | BS=4, Epochs=5, LR=0.001, F1=0.6238
Train Loss
Validation Loss
Loss
Epoch


Hyperband | BS=2, Epochs=5, LR=0.001, F1=0.3419
Train Loss
Validation Loss
Epoch


Bayesian | BS=2, Epochs=5, LR=0.001, F1=0.5402
Train Loss
Validation Loss
Loss
Epoch


Hyperband | BS=4, Epochs=5, LR=0.001, F1=0.4627
Train Loss
Validation Loss
Epoch


Bayesian | BS=4, Epochs=5, LR=0.001, F1=0.5556
Train Loss
Validation Loss
Loss
Epoch




When you train for only 1 epoch, there's only one data point for training and validation loss, so the plot can look empty or flat. Also, if the loss doesn't change much (e.g., due to a tiny learning rate), both lines can overlap near a single value. Increasing the number of epochs or adjusting the y-axis scale often resolves this issue.