Repository Link: https://github.com/shreyamali01/CS203_Lab06

Section 1: MLP Model Implementation and Experiment Tracking

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

Loading the **Iris dataset** using sklearn.datasets.load_iris

Importing Required Libraries

```
import numpy as np
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler, OneHotEncoder
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score, confusion_matrix
import matplotlib.pyplot as plt
import seaborn as sns
import wandb
```

Loading IRIS Dataset

```
iris = load_iris()
X = iris.data   #extracting features
y = iris.target  #extracting labels

#printing raw labels
print(f"Raw Labels: \n {y[:10]}")
```

```
Raw Labels:
 [0 0 0 0 0 0 0 0 0 0]
```

Extracting features and labels, ensuring labels are one-hot encoded.

Loading IRIS Dataset

```
iris = load_iris()
X = iris.data   #extracting features
y = iris.target  #extracting labels

#printing raw labels
print(f"Raw Labels: \n {y[:10]}")
```

```
Raw Labels:
 [0 0 0 0 0 0 0 0 0 0]
```

Applying One-Hot Encoding

```
#reshaping labels for one-hot encoding
y = y.reshape(-1, 1)

#one-hot encoding the labels
encoder = OneHotEncoder(sparse_output=False)
y_onehot = encoder.fit_transform(y)
```

Split the dataset into: **Training set** 70%, **Validation set:** 10%, **Testing set:** 20%.

Splitting the dataset in 70% Training Set, 20% Testing Set, and 10% Validation Set

```
#first split is 70% training and 30% test+validation set
X_train, X_temp, y_train, y_temp = train_test_split(X, y_onehot, test_size=0.3, random_state=42, stratify=y)

#second split is splitting 30% test+validation into 20% test and 10% validation set
X_test, X_val, y_test, y_val = train_test_split(X_temp, y_temp, test_size=0.333, random_state=42, stratify=y_temp)
```

Dataset Statistics

```
print("Original dataset shape:", X.shape, y_onehot.shape)
print("Training set shape:", X_train.shape, y_train.shape)
print("Validation set shape:", X_val.shape, y_val.shape)
print("Testing set shape:", X_test.shape, y_test.shape)
```

```
Original dataset shape: (150, 4) (150, 3)
Training set shape: (105, 4) (105, 3)
Validation set shape: (15, 4) (15, 3)
Testing set shape: (30, 4) (30, 3)
```

Normalizing feature values to [0,1] using standard scaling.

```
#normalizing feature values to [0, 1] using standard scaling
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_val = scaler.transform(X_val)
X_test = scaler.transform(X_test)
```

2. Define and Train the MLP Model

Constructing a Multi-Layer Perceptron (MLP) model with Input layer: 4 neurons (for 4 features), Hidden layer: 16 neurons with ReLU activation and Output layer: 3 neurons (for each class) with softmax activation. Training using Loss function: Categorical cross-entropy, Optimizer: Adam, Learning rate: `0.001`, Batch size: `32`, Epochs: `50`. Tracking and storing both training and validation loss during training.

Step 2 : Defining and Training the Model

```
#defining the MLP model
model = Sequential([
    Dense(16, activation='relu', input_shape=(4,)),     #hidden layer
    Dense(3, activation='softmax')                      #output layer
])

#compiling the model
model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=0.001),
              loss='categorical_crossentropy',
              metrics=['accuracy'])

#training the model
history = model.fit(X_train, y_train,
                    validation_data=(X_val, y_val),
                    batch_size=32,
                    epochs=50,
                    verbose=1)
```

3. Evaluate Model Performance

Computing and storing Accuracy, Precision, Recall, F1-Score, Confusion matrix using the **test set**.

Step 3 : Evaluate Model Performance

```
#evaluating the model on the test set
y_pred = model.predict(X_test)
y_pred_classes = np.argmax(y_pred, axis=1)
y_test_classes = np.argmax(y_test, axis=1)
```

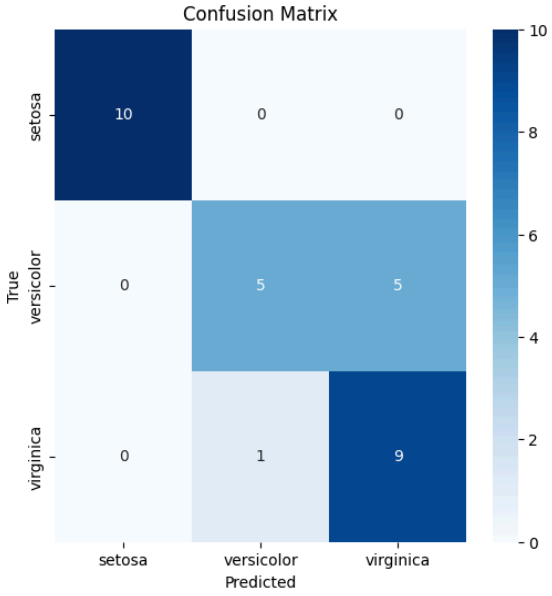
```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 34ms/step
```

```
#computing the metrics
accuracy = accuracy_score(y_test_classes, y_pred_classes)
precision = precision_score(y_test_classes, y_pred_classes, average='weighted')
recall = recall_score(y_test_classes, y_pred_classes, average='weighted')
f1 = f1_score(y_test_classes, y_pred_classes, average='weighted')
conf_matrix = confusion_matrix(y_test_classes, y_pred_classes)
```

Print Metrics

```
print(f"Accuracy: {accuracy}")
print(f"Precision: {precision}")
print(f"Recall: {recall}")
print(f"F1-score: {f1}")
```

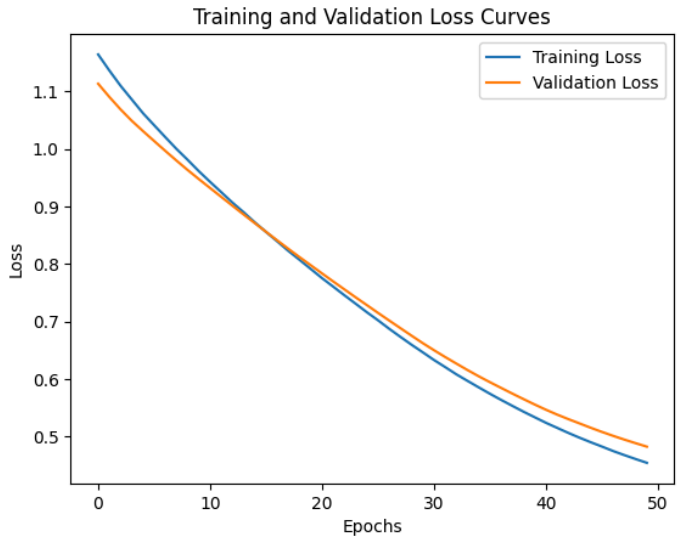
```
Accuracy: 0.8
Precision: 0.8253968253968256
Recall: 0.8
F1-score: 0.7916666666666666
```



Plotting training and validation loss curves over epochs using Matplotlib.

Plotting train and validation loss curves

```
plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.title('Training and Validation Loss Curves')
plt.legend()
plt.show()
```



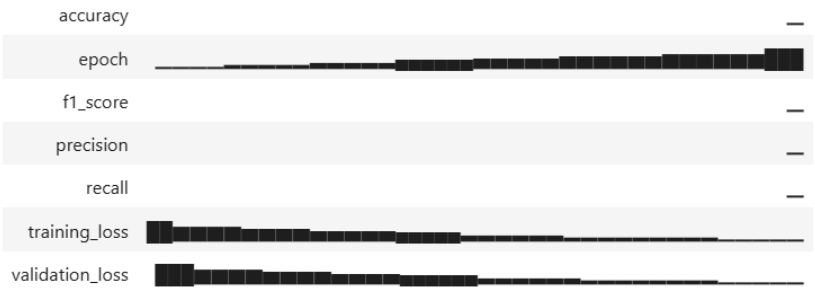
4. **Set Up Experiment Tracking with Weights & Biases (W&B)**

Logging the Model architecture, Hyperparameters, Training and validation loss per epoch, Final evaluation metrics, Confusion matrix and loss curve visualizations.

```
#initializing W&B
wandb.init(project="cs203-lab6", config={
    "learning_rate": 0.001,
    "batch_size": 32,
    "epochs": 50,
    "architecture": "MLP",
    "dataset": "Iris"
})

#logging model architecture and hyperparameters
wandb.config.update({
    "layers": [4, 16, 3],
    "activations": ["relu", "softmax"]
})

#logging metrics
wandb.log({
    "accuracy": accuracy,
    "precision": precision,
    "recall": recall,
    "f1_score": f1
})
```

```
#logging confusion matrix
wandb.log({"confusion_matrix": wandb.plot.confusion_matrix(
    y_true=y_test_classes,
    preds=y_pred_classes,
    class_names=iris.target_names
)})

#logging loss curves
for epoch in range(len(history.history['loss'])):
    wandb.log({
        "epoch": epoch,
        "training_loss": history.history['loss'][epoch],
        "validation_loss": history.history['val_loss'][epoch]
    })

#finish W&B run
wandb.finish()
```

Run history:



Run summary:

accuracy	0.8
epoch	49
f1_score	0.79167
precision	0.8254
recall	0.8
training_loss	0.45456
validation_loss	0.48264

Screenshots of W&B Dashboard

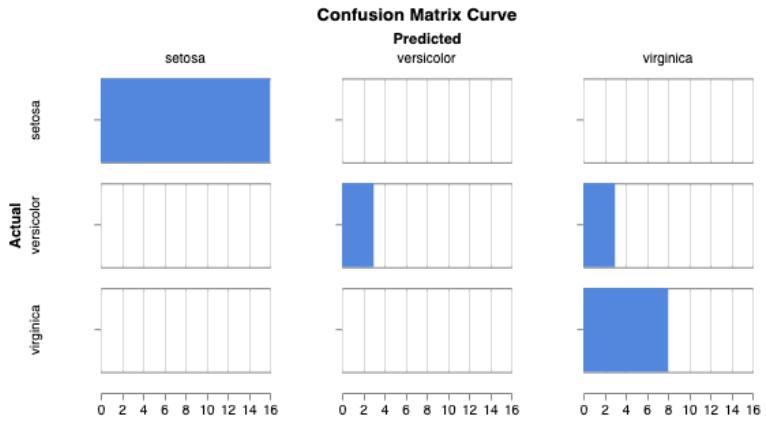
1. Model Architecture and hyperparameters

Key	Value
Run Config	
activations	
0	"relu"
1	"softmax"
architecture	"MLP"
batch_size	32
dataset	"Iris"
epochs	50
layers	
0	4
1	16
2	3
learning_rate	0.001

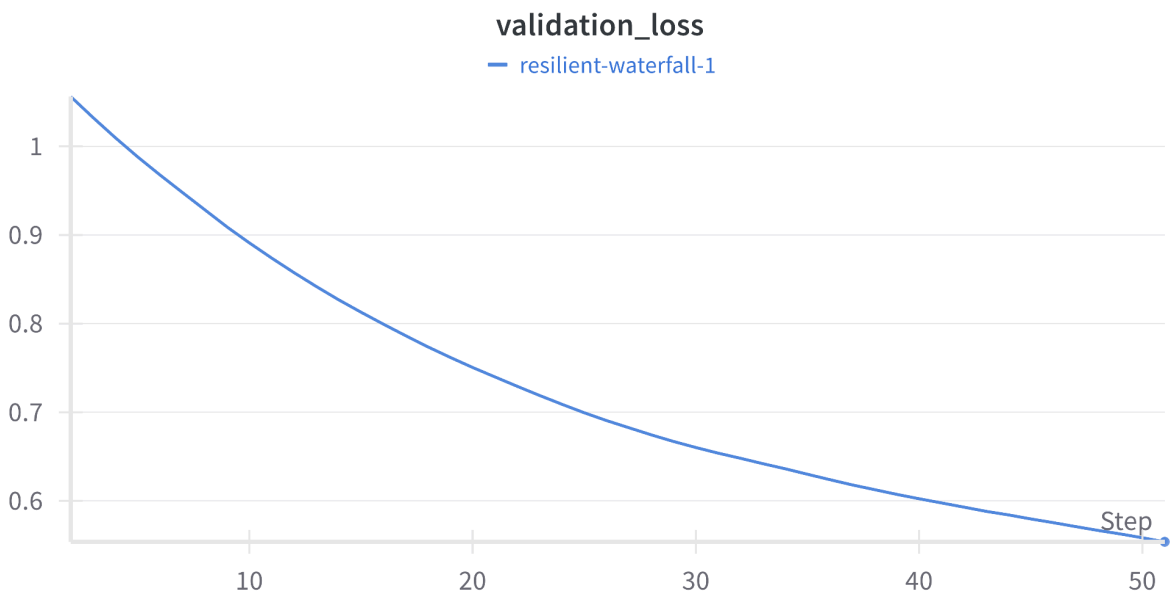
2. Logged Metrics and Final Evaluation Metrics

accuracy	0.9
f1_score	0.8912280701754386
precision	0.9272727272727274
recall	0.9

3. Confusion Matrix



4. Training and validation loss curves



Section 2

Task 1 : Hyperparameter Optimization

1. **Hyperparameter Optimization**

Training the model on the batch size of [2 & 4], learning rate [1e-3 and 1e-5], and epochs [1, 3, and 5].

Function to train the model with manual hyperparameters

```python
def train_model(train_data, batch_size, lr, epochs):
    predictor = TabularPredictor(
        label='target',
        verbosity=4,
        problem_type='multiclass',
        eval_metric='accuracy'
    )
    predictor.fit(
        train_data,
        hyperparameters={
            'NN_TORCH': {
                'num_epochs': epochs,
                'learning_rate': lr,
                'batch_size': batch_size,
                'hidden_size': 4,
            }
        },
        time_limit=600,
        presets='good_quality_faster_inference_only_refit',
        ag_args_fit={'num_gpus': 0}
    )

    return predictor
```

```python
#defining hyperparameter grid
batch_sizes = [2, 4]
learning_rates = [1e-3, 1e-5]
epochs = [1, 3, 5]

#empty lists to store accuracy and F1-score
accuracies = []
f1_scores = []

#iterating over all the hyperparameters
for batch_size, lr, epoch in product(batch_sizes, learning_rates, epochs):
            print(f"training parameters : \n batch_size={batch_size}, lr={lr}, epochs={epoch}")

            #training the model
            model = train_model(train_data, batch_size, lr, epoch)

            #making predictions on test data
            predictions = model.predict(test_data)

            #computing and appending accuracy and F1-score
            accuracy = accuracy_score(y_test, predictions)
            f1 = f1_score(y_test, predictions, average='weighted')
            print(f"accuracy: {accuracy:.4f}")
            print(f"f1 score: {f1:.4f}")
            accuracies.append(accuracy)
            f1_scores.append(f1)
```

```python

            #plotting confusion matrix
            cm = confusion_matrix(y_test, predictions)
            plt.figure(figsize=(6, 6))
            sns.heatmap(cm, annot=True, fmt='d', cmap='coolwarm', linewidths=1.5, linecolor='black',
                        xticklabels=iris.target_names, yticklabels=iris.target_names, cbar=True, annot_kws={"size": 14})

            plt.xlabel('predicted label', fontsize=12, fontweight='bold')
            plt.ylabel('true label', fontsize=12, fontweight='bold')
            plt.title(f'confusion matrix for : \n batch_size={batch_size}, lr={lr}, epochs={epoch}',fontsize=14, fontweight='bold')
            plt.show()
```

Results like confusion matrix, accuracy and f1 score can be found in the code output
Showing the input, predicted and truth values of 5 random samples

Printing 5 random samples from the test set with their : Input features, predicted labels and true labels

```python
import random

#selecting 5 random indices from the test set
sample_indices = random.sample(range(len(test_data)), 5)

print("\n=== Sample Predictions ===\n")
for i in sample_indices:
    #printing the input features
    print(f"Input: {test_data.iloc[i]}")
    print(f"Predicted: {predictions[i]}")
    print(f"Truth: {y_test[i]}\n")
```

```
=== Sample Predictions ===

Input: sepal length    0.305556
sepal width     0.416667
petal length    0.593220
petal width     0.583333
Name: 9, dtype: float64
Predicted: 0.0
Truth: [1]

Input: sepal length    0.027778
sepal width     0.500000
petal length    0.050847
petal width     0.041667
Name: 4, dtype: float64
Predicted: 0.0
Truth: [0]

Input: sepal length    0.583333
sepal width     0.333333
petal length    0.779661
petal width     0.875000
Name: 24, dtype: float64
Predicted: 0.0
Truth: [2]
```

```
Input: sepal length    0.194444
sepal width     0.125000
petal length    0.389831
petal width     0.375000
Name: 3, dtype: float64
Predicted: 0.0
Truth: [1]

Input: sepal length    0.472222
sepal width     0.416667
petal length    0.644068
petal width     0.708333
Name: 19, dtype: float64
Predicted: 0.0
Truth: [2]
```

2. **Automated Hyperparameter Search**

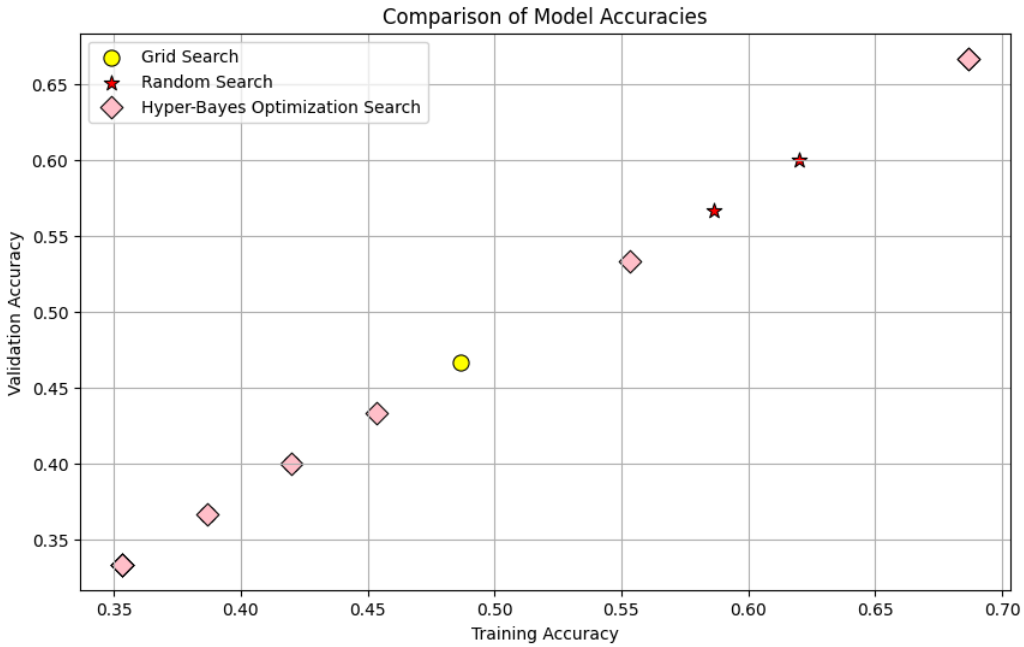
Results of using Grid Search over the parameters defined above, Random Search, and Hyperband + Bayesian Optimization hyperparameter to search for the hyperparameters defined in Task 1.

Search Method	Accuracy	F1 Score
Grid Search	0.5333	0.4461
Random Search	0.6667	0.5556
Hyperband-Bayesian	0.6667	0.5556

The other results can be found in the python notebook

Performing hyperparameter optimization using AutoGluon

Plotting the scatter plot for training vs validation loss.



Observed relation (direct or inverse) between the hyperparameters and their impact on the performance is given below.

Relationship of HyperParameters and their Performance

- Epochs : Directly proportional to the performance. Increasing number of epochs, help the model to learn and generalize more. As a result, accuracy and F1 score increases.
- Batch Size : Inversly proportional to the performance. Smaller batch-sizes help the model to learn better and generalize better. Whereas, larger batch-size makes the training easier and the model doesn't learn to generalize.
- Learning Rate : The learning doesn't have a linear relationship with the performance of the model. A low learning rate may lead to slow convergence whereas high learning rate can cause the model to diverge.

The result and performance observed for each hyperparameter combination over accuracy and F1 for different search methods is given below.

Grid Seach

	Model Name	Batch Size	Num Epochs	Learning Rate	Train Accuracy	Validation Accuracy	F1 Score
0	NeuralNetTorch/2fd60_00000	2	1	0.001	0.466667	0.466667	0.3757
1	NeuralNetTorch/2fd60_00001	4	5	1e-05	0.333333	0.333333	0.1667
2	NeuralNetTorch/2fd60_00002	4	3	1e-05	0.533333	0.533333	0.4461
3	NeuralNetTorch/2fd60_00003	4	5	0.001	0.533333	0.533333	0.4373
4	NeuralNetTorch/2fd60_00004	2	1	1e-05	0.333333	0.333333	0.1667
5	NeuralNetTorch/2fd60_00005	4	5	0.001	0.633333	0.633333	0.5244
6	NeuralNetTorch/2fd60_00006	4	5	1e-05	0.333333	0.333333	0.2143
7	NeuralNetTorch/2fd60_00007	2	5	1e-05	0.333333	0.333333	0.2143
8	NeuralNetTorch/2fd60_00008	4	1	0.001	0.333333	0.333333	0.1667
9	NeuralNetTorch/2fd60_00009	4	3	0.001	0.4	0.4	0.2794
10	NeuralNetTorch/2fd60_00010	4	1	0.001	0.366667	0.366667	0.2704
11	NeuralNetTorch/2fd60_00011	2	1	0.001	0.333333	0.333333	0.1667

- Here the combination of batch size = 4, epochs = 5 and learning rate = 0.01 provides best performance in terms of accuracy and F1 score.
- We can observe that the learning rate of 1e-3 tends to perform better than the learning rate of 1e-5.
- Also, the models performed for 1 and 3 perform less well as compared to those trained on epoch 5.
- A general observation is that batch-size = 4 performs better than batch-size = 2.

Random Search

	Model Name	Batch Size	Num Epochs	Learning Rate	Train Accuracy	Validation Accuracy	F1-Score
0	NeuralNetTorch/819d8_00000	2	1	1e-05	0.333333	0.333333	0.1667
1	NeuralNetTorch/819d8_00001	2	10	0.000930003	0.666667	0.666667	0.5473
2	NeuralNetTorch/819d8_00002	3	7	2.99065e-05	0.333333	0.333333	0.1667
3	NeuralNetTorch/819d8_00003	3	1	0.000402137	0.333333	0.333333	0.1667
4	NeuralNetTorch/819d8_00004	3	7	0.000678685	0.633333	0.633333	0.5635
5	NeuralNetTorch/819d8_00005	4	9	0.000480119	0.366667	0.366667	0.231
6	NeuralNetTorch/819d8_00006	3	10	0.000969272	0.9	0.9	0.8982
7	NeuralNetTorch/819d8_00007	3	6	0.000920288	0.7	0.7	0.6238
8	NeuralNetTorch/819d8_00008	4	4	0.00057341	0.333333	0.333333	0.1667
9	NeuralNetTorch/819d8_00009	3	4	0.000238435	0.566667	0.566667	0.4765
10	NeuralNetTorch/819d8_00010	2	9	0.000795761	0.666667	0.666667	0.5894
11	NeuralNetTorch/819d8_00011	3	9	0.000934872	0.466667	0.466667	0.3671

- The best combination here is with batch-size = 2, epochs = 10 and learning rate = 0.0009.
- Better performing configurations are with batch size 3, and epochs 7 or 9.
- The configurations with poor performance are batch-size 2 with epochs ranging from 1 to 4.

Hyperband+Bayesian Optimzer

	Model Name	Batch Size	Num Epochs	Learning Rate	Train Accuracy	Validation Accuracy	F1-Score
0	NeuralNetTorch/981633a0	2	1	1e-05	0.333333	0.333333	0.1667
1	NeuralNetTorch/e24c65b3	3	6	0.000364508	0.333333	0.333333	0.2143
2	NeuralNetTorch/fac43766	2	8	0.00072908	0.333333	0.333333	0.1667
3	NeuralNetTorch/b491f49c	4	4	0.000674812	0.666667	0.666667	0.5556
4	NeuralNetTorch/1d420573	3	7	0.000210726	0.533333	0.533333	0.4461
5	NeuralNetTorch/153fd061	2	9	0.000856874	0.333333	0.333333	0.1667
6	NeuralNetTorch/124b77a6	3	9	0.000346506	0.4	0.4	0.2827
7	NeuralNetTorch/4156371c	3	3	0.000443321	0.333333	0.333333	0.2143
8	NeuralNetTorch/2f4b0709	3	7	0.000559445	0.666667	0.666667	0.5473
9	NeuralNetTorch/db12d5f4	2	4	0.000754307	0.366667	0.366667	0.2845
10	NeuralNetTorch/4e0af195	3	1	0.00058899	0.333333	0.333333	0.1667
11	NeuralNetTorch/110c1baa	3	5	0.000221785	0.433333	0.433333	0.329

- The best combination of hyperparameters is model with configurations : batch-size = 4 and epoch = 4.
- Notable observation here is that low learning rates (around 0.0002 to 0.0007) show better performance as compared to that of higher learning rates.
- The model with poor performance is with configurations batch-size = 2 and lower epochs in the range (1,4)

Compare manual tuning vs. automated search

Which approach is better and why?

- Manual Tuning means iterating over different hyperparameters based on intuition. It is time consuming and may sometimes lead to missing of optimal configurations.
- Automated Search usually relies on exploring broader range of possible hyperparameter configurations. It is more reliable and gives faster results.

Rest of the results and observations can be found in the python notebook.