



SEMESTER 2 SESSION 2023/2024





SECB4313 BIOINFORMATICS MODELING AND SIMULATION

ASSIGNMENT 3

LECTURER : DR AZURAH BTE SAMAH

SUBMITTED BY :

GROUP 4

NAME	MATRIC NO
 <p data-bbox="363 548 812 583">ERICA DESIRAE MAURITIUS</p>	<p data-bbox="1208 411 1373 447">A20EC0032</p>
 <p data-bbox="328 856 846 892">INDRADEVI A/P VIKNESHWARAN</p>	<p data-bbox="1208 737 1373 772">A20EC0050</p>
 <p data-bbox="336 1199 837 1234">NUR HAZNIRAH BINTI HAZMAN</p>	<p data-bbox="1208 1062 1373 1098">A20EC0114</p>
 <p data-bbox="253 1516 922 1551">SAYANG ELYIANA AMIERA BINTI HELMEY</p>	<p data-bbox="1208 1390 1373 1425">A20EC0143</p>

1. The selected FOUR hyper parameters and its corresponding values. Summary on combination of hyper parameters that generate the most improved result? (copy result from previous assignment).

Table 1 Summary of hyperparameters and the results

No.	Activation Function	Optimizer	Learning Rate	Batch Size	Test Loss	Test Accuracy	Precision	Recall	F1-Score
1.	LeakyReLU	RMSprop	0.001	32	0.13	0.84	0.80	0.86	0.83
2.	LeakyReLU	RMSprop	0.001	64	0.12	0.84	0.80	0.86	0.83
3.	LeakyReLU	RMSprop	0.0001	32	0.17	0.81	0.72	0.93	0.81
4.	LeakyReLU	RMSprop	0.0001	64	0.19	0.81	0.75	0.86	0.80
5.	LeakyReLU	Adamax	0.001	32	0.12	0.84	0.80	0.86	0.83
6.	LeakyReLU	Adamax	0.001	64	0.12	0.87	0.81	0.93	0.87
7.	LeakyReLU	Adamax	0.0001	32	0.20	0.77	0.73	0.79	0.76
8.	LeakyReLU	Adamax	0.0001	64	0.24	0.55	0.00	0.00	0.00
9.	ReLU	RMSprop	0.001	32	0.12	0.84	0.80	0.86	0.83
10.	ReLU	RMSprop	0.001	64	0.12	0.87	0.81	0.93	0.87

11.	ReLU	RMSprop	0.0001	32	0.22	0.77	0.82	0.64	0.72
12.	ReLU	RMSprop	0.0001	64	0.21	0.74	0.71	0.71	0.71
13.	ReLU	Adamax	0.001	32	0.12	0.84	0.80	0.86	0.83
14.	ReLU	Adamax	0.001	64	0.13	0.87	0.81	0.93	0.87
15.	ReLU	Adamax	0.0001	32	0.24	0.55	0.00	0.00	0.00
16.	ReLU	Adamax	0.0001	64	0.24	0.55	0.00	0.00	0.00

Notes:

- Highlights in yellow indicate the best hyperparameter combination.

The initial model, which had an F1-score of 0.79, an accuracy of 0.81, a precision of 0.79, and a recall of 0.79, underwent enhancement through hyperparameter tuning using sixteen distinct combinations, involving adjustments to the activation function, optimizer, learning rate and batch size, were examined. Experiments 6 and 10 demonstrated notable improvements, reaching an accuracy of 0.87, precision of 0.81, recall of 0.93, and an F1-score of 0.87. Experiment 6 made use of a LeakyReLU activation function, Adamax optimizer, a learning rate of 0.001, and 64 batch sizes. Experiment 10 utilized a ReLU activation function, RMSprop optimizer, a learning rate of 0.001, and a batch size of 64. The new configurations resulted in enhancements to the original model's metrics, with improvements of 0.06, 0.02, 0.14, and 0.08, respectively. On the other hand, Experiments 8, 15, and 16 yielded lower performance, with an accuracy of 0.55 and an F1-score of 0.00, highlighting the diverse effects of hyperparameter selections. This analysis highlights the significance of hyperparameter tuning in optimizing machine learning models, providing their robustness, reliability, and accuracy in making predictions.

- For each Grid Search and Random Search, screenshot the report (generated at the end of each run) that shows the best score and hyperparameter configuration that achieved the best performance.

```
# Summarize results
print("Best: %f using %s" % (grid_result.best_score_, grid_result.best_params_))
means = grid_result.cv_results_['mean_test_score']
stds = grid_result.cv_results_['std_test_score']
params = grid_result.cv_results_['params']
for mean, stdev, param in zip(means, stds, params):
    print("%f (%f) with: %r" % (mean, stdev, param))

Best: 0.552702 using {'activation': 'relu', 'batch_size': 64, 'learning_rate': 0.0001, 'optimizer': 'adamax'}
0.543569 (0.036376) with: {'activation': 'relu', 'batch_size': 32, 'learning_rate': 0.001, 'optimizer': 'adamax'}
0.543569 (0.036376) with: {'activation': 'relu', 'batch_size': 32, 'learning_rate': 0.001, 'optimizer': 'rmsprop'}
0.543569 (0.036376) with: {'activation': 'relu', 'batch_size': 32, 'learning_rate': 0.0001, 'optimizer': 'adamax'}
0.538940 (0.037198) with: {'activation': 'relu', 'batch_size': 32, 'learning_rate': 0.0001, 'optimizer': 'rmsprop'}
0.543569 (0.036376) with: {'activation': 'relu', 'batch_size': 64, 'learning_rate': 0.001, 'optimizer': 'adamax'}
0.543569 (0.036376) with: {'activation': 'relu', 'batch_size': 64, 'learning_rate': 0.001, 'optimizer': 'rmsprop'}
0.552702 (0.048172) with: {'activation': 'relu', 'batch_size': 64, 'learning_rate': 0.0001, 'optimizer': 'adamax'}
0.515791 (0.054517) with: {'activation': 'relu', 'batch_size': 64, 'learning_rate': 0.0001, 'optimizer': 'rmsprop'}
0.543569 (0.036376) with: {'activation': <keras.layers.activation.leaky_relu.LeakyReLU object at 0x7adc07ecd8b0>, 'batch_size': 32, 'learning_rate': 0.001, 'optimizer': 'adamax'}
0.543569 (0.036376) with: {'activation': <keras.layers.activation.leaky_relu.LeakyReLU object at 0x7adc07ecd8b0>, 'batch_size': 32, 'learning_rate': 0.001, 'optimizer': 'rmsprop'}
0.543569 (0.036376) with: {'activation': <keras.layers.activation.leaky_relu.LeakyReLU object at 0x7adc07ecd8b0>, 'batch_size': 32, 'learning_rate': 0.0001, 'optimizer': 'adamax'}
0.543569 (0.036376) with: {'activation': <keras.layers.activation.leaky_relu.LeakyReLU object at 0x7adc07ecd8b0>, 'batch_size': 32, 'learning_rate': 0.0001, 'optimizer': 'rmsprop'}
0.543569 (0.036376) with: {'activation': <keras.layers.activation.leaky_relu.LeakyReLU object at 0x7adc07ecd8b0>, 'batch_size': 64, 'learning_rate': 0.001, 'optimizer': 'adamax'}
0.539803 (0.030816) with: {'activation': <keras.layers.activation.leaky_relu.LeakyReLU object at 0x7adc07ecd8b0>, 'batch_size': 64, 'learning_rate': 0.001, 'optimizer': 'rmsprop'}
0.543569 (0.036376) with: {'activation': <keras.layers.activation.leaky_relu.LeakyReLU object at 0x7adc07ecd8b0>, 'batch_size': 64, 'learning_rate': 0.0001, 'optimizer': 'adamax'}
0.543569 (0.036376) with: {'activation': <keras.layers.activation.leaky_relu.LeakyReLU object at 0x7adc07ecd8b0>, 'batch_size': 64, 'learning_rate': 0.0001, 'optimizer': 'rmsprop'}
```

Figure 1 The outcome of hyperparameter tuning using Grid Search

Figure 1 displays the outcome of hyperparameter tuning using grid search. The purpose of grid search is to identify the best combination of hyperparameters for model performance. The best performance was achieved with activation='relu', batch_size=64, learning_rate=0.0001, and optimizer='rmsprop'. Some hyperparameters, such as LeakyReLU, appear to have been examined but did not perform as well as the optimal configuration, as shown by lower mean scores. In this case, the best combination resulted in a score of 0.580670.

```
# Summarize results
print("Best: %f using %s" % (random_search.best_score_, random_search.best_params_))
means = random_search.cv_results_['mean_test_score']
stds = random_search.cv_results_['std_test_score']
params = random_search.cv_results_['params']
for mean, stdev, param in zip(means, stds, params):
    print("%f (%f) with: %r" % (mean, stdev, param))

Best: 0.557458 using {'optimizer': 'adamax', 'learning_rate': 0.001, 'batch_size': 64, 'activation': 'relu'}
0.484209 (0.054517) with: {'optimizer': 'rmsprop', 'learning_rate': 0.0001, 'batch_size': 32, 'activation': <keras.layers.activation.leaky_relu.LeakyReLU object at 0x7ba0f8e5f670>}
0.520358 (0.060686) with: {'optimizer': 'rmsprop', 'learning_rate': 0.0001, 'batch_size': 32, 'activation': 'relu'}
0.543569 (0.036376) with: {'optimizer': 'rmsprop', 'learning_rate': 0.001, 'batch_size': 64, 'activation': 'relu'}
0.538940 (0.037198) with: {'optimizer': 'adamax', 'learning_rate': 0.0001, 'batch_size': 64, 'activation': 'relu'}
0.552638 (0.055961) with: {'optimizer': 'rmsprop', 'learning_rate': 0.001, 'batch_size': 32, 'activation': 'relu'}
0.557458 (0.040696) with: {'optimizer': 'adamax', 'learning_rate': 0.001, 'batch_size': 64, 'activation': 'relu'}
0.543569 (0.036376) with: {'optimizer': 'adamax', 'learning_rate': 0.001, 'batch_size': 32, 'activation': 'relu'}
0.543569 (0.036376) with: {'optimizer': 'adamax', 'learning_rate': 0.0001, 'batch_size': 32, 'activation': <keras.layers.activation.leaky_relu.LeakyReLU object at 0x7ba0f8e5f670>}
0.543569 (0.036376) with: {'optimizer': 'adamax', 'learning_rate': 0.0001, 'batch_size': 64, 'activation': <keras.layers.activation.leaky_relu.LeakyReLU object at 0x7ba0f8e5f670>}
0.543569 (0.036376) with: {'optimizer': 'adamax', 'learning_rate': 0.0001, 'batch_size': 32, 'activation': 'relu'}
```

Figure 2 The outcome of hyperparameter tuning using Random Search

Figure 2 shows the outcome of hyperparameter tuning using grid search. Random search, like grid search, seeks to identify the optimum hyperparameters by sampling randomly from the parameter space rather than exhaustively examining all potential combinations. The best performance was achieved with optimizer='adamax', learning_rate=0.001, batch_size=64, and activation='relu', resulting in a score of 0.557458. Random search is a more efficient method of searching the hyperparameter space since it randomly samples possibilities, which is especially advantageous when the parameter space is huge. In this situation, the optimal combination yielded a score of 0.557458.

3. Discuss the results obtained in (2) and (3). Make comparisons in terms of

a) Effort To Get The Results

Research was done and sample code was discovered to assist in the implementation of Grid Search and Random Search in an initial study. The coding and methods for determining the ideal set of hyperparameters were explained by two primary sources. Grid Search ensures thoroughness by exhaustively evaluating every possible combination of parameters, but it also takes more time and computational power. Random Search, on the other hand, allows control over the computational budget by specifying the number of iterations and is quicker to set up and execute. It also samples from the parameter space. While Random Search frequently finds good parameter combinations with less effort but requires more careful analysis, Grid Search offers a comprehensive view of each parameter's impact, making results easier to interpret.

b) Computational Time.

The computational time taken for Grid Search is 43.16 seconds and for Random Search is 37.84 seconds. Grid Search requires more time compared to Random Search due to its exhaustive evaluation of all possible parameter combinations, resulting in a higher likelihood of finding the optimal parameters but also incurring greater computational costs. The time complexity of Grid Search grows exponentially as the

number of parameters and level of detail of the search increase. On the other hand, Random Search is quicker because it only assesses a portion of the parameter space. Its computational time is directly related to the number of iterations, which makes it more adaptable and easier to handle.

```
# Calculate the elapsed time
elapsed_time = end_time - start_time
print(f"Grid search took {elapsed_time:.2f} seconds")
```

Grid search took 43.16 seconds

Figure 3 Computational time taken for Grid Search

```
# Calculate the elapsed time
elapsed_time = end_time - start_time
print(f"Random search took {elapsed_time:.2f} seconds")
```

Random search took 37.84 seconds

Figure 4 Computational time taken for Random Search

4. Highlight why hyper parameter optimization/tuning is vital in order to enhance your model's performance?

Based on this experiment, hyperparameter optimization is vital in order to enhance model performance because it can improve metrics in terms of accuracy, precision, recall, and F1-score. For example, the optimization improved the model accuracy from 0.81 to 0.87 and its F1-score from 0.79 to 0.87 by using LeakyReLU function and Adamax optimizer. By using ReLU function and RMSProp optimizer, the model's performance can also be improved, achieving the same results. This shows that the combinations leverage the efficient gradient propagation, at the same time enhance the model's stability and convergence.

Moreover, both Grid and Random Search help identify the best hyperparameters efficiently. Grid Search is thorough but computationally expensive, while Random Search is faster and more efficient with large parameter spaces.

Appendix

Grid Search

```
!pip install keras==2.9
!pip install tensorflow==2.9
!pip install patchify      #To install and import other mentioned libraries in code
!pip install segmentation_models

#Import all library needed
import numpy as np
import pandas as pd
import tensorflow as tf
import sklearn
from sklearn.model_selection import train_test_split
from keras.models import Sequential
from keras.layers import Dense, BatchNormalization, Dropout, LeakyReLU
from tensorflow.keras.callbacks import EarlyStopping, ReduceLROnPlateau,
ModelCheckpoint
from sklearn.model_selection import GridSearchCV, RandomizedSearchCV
from tensorflow.keras.wrappers.scikit_learn import KerasClassifier
import matplotlib.pyplot as plt
import os
import time
from sklearn.datasets import load_iris

#Confusion Matrix Visualization
import seaborn as sns
from sklearn.metrics import confusion_matrix, classification_report

# Set the data path for data set and model location
dataset_dir = "/content/gdrive/MyDrive/Semester 8/Bio
Modeling&Simulation/Dataset/Heart Disease/"
model_loc = "/content/gdrive/MyDrive/Semester 8/Bio Modeling&Simulation/Dataset/"

print(os.listdir(dataset_dir))
data = pd.read_csv(dataset_dir + 'heart.csv')

catagorialList = ['sex','cp','fbs','restecg','exang','ca','thal']
for item in catagorialList:
    data[item] = data[item].astype('object') #casting to object

data.head() #display first 5 rows

data = pd.get_dummies(data, drop_first=True)

y = data['target'].values #take target value
y = y.reshape(y.shape[0],1) #reshape y array into 1 dimension
x = data.drop(['target'],axis=1)#drop 'target' in x data

print("Shape of x:", x.shape)
print("Shape of y:", y.shape)

y.size
```



```

# Create a simple dataset
data = pd.DataFrame({'A': [10, 20, 30], 'B': [100, 200, 300], 'C': [1000, 2000, 3000]})
print('Original dataset:')
print(data)

# Normalize data (range 0 - 1)
minx = np.min(data)
maxx = np.max(data)
data_norm = (data - minx) / (maxx - minx)
print('\nNormalized dataset:')
print(data_norm)

#Normalize data (range 0 - 1)
minx = np.min(x)
maxx = np.max(x)
x = (x - minx) / (maxx - minx)
x.head()

#Split the dataset into train and test (train 70%, val 20% and test 10%).
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.1,
random_state=42)
# re-create train and validation set
x_train, x_val, y_train, y_val = train_test_split(x_train, y_train, test_size=0.2,
random_state=42)
# train 70%, validation 20%, test 10%
print(x_train.shape)
print(x_val.shape)
print(x_test.shape)

def create_model(activation='softmax', optimizer='adam', learning_rate=0.01):
    model = Sequential()
    model.add(Dense(64, input_dim=21, activation=activation))
    model.add(Dense(32, activation=activation))
    model.add(Dense(1, activation='sigmoid'))

    if optimizer == 'adam':
        optimizer = tf.keras.optimizers.Adam(learning_rate=learning_rate)
    elif optimizer == 'rmsprop':
        optimizer = tf.keras.optimizers.RMSprop(learning_rate=learning_rate,
rho=0.9, momentum=0.0, epsilon=1e-07, centered=False)

    model.compile(loss='mse', optimizer=optimizer, metrics=['acc'])
    return model

# Create KerasClassifier
keras_model = KerasClassifier(build_fn=create_model, epochs=10, verbose=0)

# Define the grid of hyperparameters to search
param_grid = {
    'activation': ['relu', LeakyReLU()],
    'optimizer': ['adamax', 'rmsprop'],
    'batch_size': [32, 64],
    'learning_rate': [0.001, 0.0001]
}

# Load dataset
iris = load_iris()
x, y = iris.data, iris.target

# Create the GridSearchCV object

```

```

grid = GridSearchCV(estimator=keras_model, param_grid=param_grid, n_jobs=-1, cv=3,
scoring='accuracy')

# Perform grid search
grid_result = grid.fit(x_train, y_train)

# Print the best parameters
print("Best parameters found: ", grid_result.best_params_)

# Summarize results
print("Best: %f using %s" % (grid_result.best_score_, grid_result.best_params_))
means = grid_result.cv_results_['mean_test_score']
stds = grid_result.cv_results_['std_test_score']
params = grid_result.cv_results_['params']
for mean, stdev, param in zip(means, stds, params):
    print("%f (%f) with: %r" % (mean, stdev, param))

# Get best model from grid search
best_model = grid_result.best_estimator_

# Make predictions on the test set
y_pred = best_model.predict(x_test)

# Generate confusion matrix
conf_matrix = confusion_matrix(y_test, y_pred)

# Plot confusion matrix
plt.figure(figsize=(8, 6))
sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues', cbar=False)
plt.xlabel('Predicted Label')
plt.ylabel('True Label')
plt.title('Confusion Matrix')
plt.show()

# Extract TP, FP, TN, FN values from confusion matrix
TP = conf_matrix[1, 1]
FP = conf_matrix[0, 1]
TN = conf_matrix[0, 0]
FN = conf_matrix[1, 0]

print("True Positives (TP):", TP)
print("False Positives (FP):", FP)
print("True Negatives (TN):", TN)
print("False Negatives (FN):", FN)

# Measure the time
start_time = time.time()
grid_result.fit(x_train, y_train)
end_time = time.time()

# Calculate the elapsed time
elapsed_time = end_time - start_time
print(f"Grid search took {elapsed_time:.2f} seconds")

```

Random Search

```
!pip install keras==2.9
!pip install tensorflow==2.9
!pip install patchify      #To install and import other mentioned libraries in code
!pip install segmentation_models

#Import all library needed
import numpy as np
import pandas as pd
import tensorflow as tf
import sklearn
from sklearn.model_selection import train_test_split
from keras.models import Sequential
from keras.layers import Dense, BatchNormalization, Dropout, LeakyReLU
from tensorflow.keras.callbacks import EarlyStopping, ReduceLROnPlateau, ModelCheckpoint
from sklearn.model_selection import GridSearchCV, RandomizedSearchCV
from tensorflow.keras.wrappers.scikit_learn import KerasClassifier
import matplotlib.pyplot as plt
import os
import time
from sklearn.datasets import load_iris

#Confusion Matrix Visualization
import seaborn as sns
from sklearn.metrics import confusion_matrix, classification_report

# Set the data path for data set and model location
dataset_dir = "/content/gdrive/MyDrive/Semester 8/Bio
Modeling&Simulation/Dataset/Heart Disease/"
model_loc = "/content/gdrive/MyDrive/Semester 8/Bio Modeling&Simulation/Dataset/"

print(os.listdir(dataset_dir))
data = pd.read_csv(dataset_dir + 'heart.csv')

catagorialList = ['sex','cp','fbs','restecg','exang','ca','thal']
for item in catagorialList:
    data[item] = data[item].astype('object') #casting to object

data.head() #display first 5 rows

data = pd.get_dummies(data, drop_first=True)

y = data['target'].values #take target value
y = y.reshape(y.shape[0],1) #reshape y array into 1 dimension
x = data.drop(['target'],axis=1)#drop 'target' in x data

print("Shape of x:", x.shape)
print("Shape of y:", y.shape)

y.size

# Create a simple dataset
data = pd.DataFrame({'A': [10, 20, 30], 'B': [100, 200, 300], 'C': [1000, 2000, 3000]})
print('Original dataset:')
print(data)

# Normalize data (range 0 - 1)
```

```

minx = np.min(data)
maxx = np.max(data)
data_norm = (data - minx) / (maxx - minx)
print('\nNormalized dataset:')
print(data_norm)

#Normalize data (range 0 - 1)
minx = np.min(x)
maxx = np.max(x)
x = (x - minx) / (maxx - minx)
x.head()

#Split the dataset into train and test (train 70%, val 20% and test 10%).
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.1,
random_state=42)
# re-create train and validation set
x_train, x_val, y_train, y_val = train_test_split(x_train, y_train, test_size=0.2,
random_state=42)
# train 70%, validation 20%, test 10%
print(x_train.shape)
print(x_val.shape)
print(x_test.shape)

def create_model(activation='softmax', optimizer='adam', learning_rate=0.01):
    model = Sequential()
    model.add(Dense(64, input_dim=21, activation=activation))
    model.add(Dense(32, activation=activation))
    model.add(Dense(1, activation='sigmoid'))

    if optimizer == 'adam':
        optimizer = tf.keras.optimizers.Adam(learning_rate=learning_rate)
    elif optimizer == 'rmsprop':
        optimizer = tf.keras.optimizers.RMSprop(learning_rate=learning_rate,
rho=0.9, momentum=0.0, epsilon=1e-07, centered=False)

    model.compile(loss='mse', optimizer=optimizer, metrics=['acc'])
    return model

# Create KerasClassifier
keras_model = KerasClassifier(build_fn=create_model, epochs=10, verbose=0)

# Define the parameter grid
param_dist = {
    'activation': ['relu', LeakyReLU()],
    'optimizer': ['adamax', 'rmsprop'],
    'batch_size': [32, 64],
    'learning_rate': [0.001, 0.0001]
}

# Load dataset
iris = load_iris()
x, y = iris.data, iris.target

# Perform random search
random_search = RandomizedSearchCV(estimator=keras_model,
param_distributions=param_dist, n_iter=10, cv=3)

# Perform grid search
random_search.fit(x_train, y_train)

# Print the best parameters
print("Best parameters found: ", random_search.best_params_)

```

```

# Summarize results
print("Best: %f using %s" % (random_search.best_score_, random_search.best_params_))
means = random_search.cv_results_['mean_test_score']
stds = random_search.cv_results_['std_test_score']
params = random_search.cv_results_['params']
for mean, stdev, param in zip(means, stds, params):
    print("%f (%f) with: %r" % (mean, stdev, param))

# Get best model from grid search
best_model = random_search.best_estimator_

# Make predictions on the test set
y_pred = best_model.predict(x_test)

# Generate confusion matrix
conf_matrix = confusion_matrix(y_test, y_pred)

# Plot confusion matrix
plt.figure(figsize=(8, 6))
sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues', cbar=False)
plt.xlabel('Predicted Label')
plt.ylabel('True Label')
plt.title('Confusion Matrix')
plt.show()

# Extract TP, FP, TN, FN values from confusion matrix
TP = conf_matrix[1, 1]
FP = conf_matrix[0, 1]
TN = conf_matrix[0, 0]
FN = conf_matrix[1, 0]

print("True Positives (TP):", TP)
print("False Positives (FP):", FP)
print("True Negatives (TN):", TN)
print("False Negatives (FN):", FN)

# Measure the time
start_time = time.time()
random_search.fit(x_train, y_train)
end_time = time.time()

# Calculate the elapsed time
elapsed_time = end_time - start_time
print(f"Random search took {elapsed_time:.2f} seconds")

```

Reference

1. Brownlee, J. (2016, August 8). *How to Grid Search Hyperparameters for Deep Learning Models in Python With Keras*. Machine Learning Mastery.
<https://machinelearningmastery.com/grid-search-hyperparameters-deep-learning-models-python-keras/>
2. Brownlee, J. (2020, September 13). *Hyperparameter Optimization With Random Search and Grid Search*. Machine Learning Mastery.
<https://machinelearningmastery.com/hyperparameter-optimization-with-random-search-and-grid-search/>