SELF-AWARE DEEP LEARNING (SAL) TUTORIAL

Paolo Dell'Aversana (June 2024)

****

The following Python notebook is an example of implementing a Self-Aware Learning (SAL) model using TensorFlow and Keras. The SAL approach is designed to dynamically adjust its hyperparameters and architecture based on performance during training, aiming to improve model accuracy and robustness without any external (human) intervention. This process is crucial in deep learning, where finding optimal hyper-parameters configurations and network architecture/workflow can be challenging and time-consuming.

The script begins by importing necessary libraries for data visualization/pre-processing. It includes pandas and numpy libraries for data handling, scikit-learn for preprocessing and data splitting, and TensorFlow/Keras for building and training the neural network models.

First, we explore the data set, to see how it is done, looking at its features and instances.

The core component of the notebook is the SALModel class. This class is responsible for creating, training, and evolving the neural network architecture. It takes the number of input and output dimensions as parameters and defines methods for creating a model, evaluating it, and updating hyperparameters based on performance. The model is built using Keras' Sequential API, with configurable hidden layers, neurons per layer, dropout rates, and learning rates. In this simplified tutorial, we considered just these hyper-parameters, but it should be clear that a much larger number of hyper-parameters can be optimized through the same self-learning and self-aware (SAL) approach. During training, the model's performance is monitored, and an early stopping mechanism is employed to prevent overfitting by halting training if the validation loss does not improve for a specified number of epochs.

We remark that, more in general, the update-hyperparameters method within the SALModel class adjusts the model's architecture and hyperparameters dynamically. If the model's performance does not reach good values with respect to "reference deep learning models", it increases the architecture complexity (for instance, by adding more layers and neurons), reduces the dropout rate to retain more information during training, and slightly decreases the learning rate to fine-tune the weights. These are just few among the many hyperparameters that can be updated by the SAL model.

The following script(s) also includes a read-and-classify function, which reads an Excel file containing the dataset, preprocesses the data, splits it into training and test sets, and performs classification using both a "reference-standard neural network" (without any self-reflection mechanisms) and the SAL model. The data is scaled, and labels are encoded to ensure compatibility with the neural network. The function then initializes and trains a reference-standard model for baseline comparison, followed by the SAL model which adapts its structure and hyperparameters during training.

After training, the notebook-script compares the performance of the standard and SAL models by plotting training and validation losses. It also displays a scatter plot to visualize the final classification results. The scatter plot highlights how well the two classes are separated, which is a key indicator of the model's effectiveness.

In summary, this notebook-script demonstrates a self-aware deep learning approach (SAL) where the model can adaptively improve itself during training. This adaptive capability is essential in real-world applications where optimal model configurations are not always known in advance and need to be discovered through iterative learning and self-reflection/self-learning mechanisms.

The following code(s) represents just an illustrative tutorial. It can be (and should be) upgraded and adapted by the user who is interested in obtaining better performances from his/her neural networks. An example of format for data file (in this case it is just an Excel file including simulated "synthetic data" with some features for distinguishing between benign and malign cells) is provided. Of course, the user can prepare his/her own input data properly in the same format for testing the scripts below.

****

FIRST PART: READING DATA FILE

```
import pandas as pd

def read_excel_file(file_path):
    # Read the Excel file
    try:
        df = pd.read_excel(file_path)
    except FileNotFoundError:
        print("File not found. Please provide a valid file path.")
        return

    # Display the first 10 rows of the data
    print("First 10 rows of the data:")
    print(df.head(10))
```

```
    # Extract header names
    header = list(df.columns)

    # Assuming the first column contains the class name
    class_name = header[0]

    # Features will be the rest of the columns
    features = header[1:]

    print("\nClass Name:", class_name)
    print("Features:", features)

    # Perform classification or any other operations using the extracted data

# Prompt the user to enter the file name
file_path = input("Enter the Excel file name (including extension): ")

# Call the function to read the Excel file
read_excel_file(file_path)
```

****

The following is an example of input data visualization, to provide a guide for the user who wants to input his/her own data:

```
Enter the Excel file name (including extension): synth.xlsx
First 10 rows of the data:

     type  feature1  feature2  feature3  feature4  feature5  feature6  \
0  malign     39.49     19.88    268.30    1902.5   1.61840   1.77760
1  malign     38.07     35.27    277.40    2851.5   1.58474   1.57864
2  malign     42.19     42.75    274.50    2513.5   1.60960   1.65990
3  malign     23.92     37.88    141.08     727.6   1.64250   1.78390
4  malign     40.79     30.84    247.60    2762.5   1.60030   1.63280
5  malign     22.95     33.20    162.07     919.6   1.62780   1.67000
6  malign     36.75     41.48    234.10    2129.5   1.59463   1.60900
7  malign     26.21     40.33    184.70    1214.4   1.61890   1.66450
8  malign     28.50     41.32    168.00     939.3   1.62730   1.69320
9  malign     22.96     48.54    161.47    1009.4   1.61860   1.73960

   feature7  feature8  feature9  ...  feature21  feature22  feature23  \
0   1.80010   1.64710    1.7419  ...      49.88      31.83     378.10
1   1.58690   1.57017    1.6812  ...      49.49      50.91     288.30
2   1.69740   1.62790    1.7069  ...      49.07      51.03     314.00
3   1.74140   1.60520    1.7597  ...      30.41      55.00     182.37
4   1.69800   1.60430    1.6809  ...      49.04      34.17     297.70
5   1.65780   1.58089    1.7087  ...      28.97      44.25     214.90
6   1.61270   1.57400    1.6794  ...      49.38      58.16     304.70
7   1.59366   1.55985    1.7196  ...      32.56      53.64     227.10
8   1.68590   1.59353    1.7350  ...      31.99      60.23     194.70
9   1.72730   1.58543    1.7030  ...      30.59      82.18     211.15

   feature24  feature25  feature26  feature27  feature28  feature29  feature30
0     4363.5     1.6622     2.1656     2.2119     1.7654     1.9601    1.61890
1     4272.5     1.6238     1.6866     1.7416     1.6860     1.7750    1.58902
2     3466.5     1.6444     1.9245     1.9504     1.7430     1.8613    1.58758
3     1176.2     1.7098     2.3663     2.1869     1.7575     2.1638    1.67300
4     3395.5     1.6374     1.7050     1.9000     1.6625     1.7364    1.57678
5     1348.1     1.6791     2.0249     2.0355     1.6741     1.8985    1.62440
```

```
6     2998.5     1.6442     1.7576     1.8784     1.6932     1.8063    1.58368
7     1736.5     1.6654     1.8682     1.7678     1.6556     1.8196    1.61510
8     1523.8     1.6703     2.0401     2.0390     1.7060     1.9378    1.60720
9     1396.9     1.6853     2.5580     2.6050     1.7210     1.9366    1.70750

[10 rows x 31 columns]

Class Name: type
Features: ['feature1', 'feature2', 'feature3', 'feature4', 'feature5',
'feature6', 'feature7', 'feature8', 'feature9', 'feature10', 'feature11',
'feature12', 'feature13', 'feature14', 'feature15', 'feature16', 'feature17',
'feature18', 'feature19', 'feature20', 'feature21', 'feature22', 'feature23',
'feature24', 'feature25', 'feature26', 'feature27', 'feature28', 'feature29',
'feature30']
```

****

SECOND PART: DATA ANALYSIS, SAL MODEL DEFINITION AND DATA CLASSIFICATION

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder
from sklearn.preprocessing import StandardScaler
import tensorflow as tf
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.callbacks import EarlyStopping

# Define the SAL model class
class SALModel:
    def __init__(self, input_dim, output_dim):
        self.input_dim = input_dim
        self.output_dim = output_dim

    def create_model(self, num_hidden_layers, num_neurons_per_layer, dropout_rate, learning_rate):
        model = Sequential()
        model.add(Dense(num_neurons_per_layer, activation='relu', input_shape=(self.input_dim,)))
        model.add(tf.keras.layers.Dropout(dropout_rate))
        for _ in range(num_hidden_layers):
            model.add(Dense(num_neurons_per_layer, activation='relu'))
            model.add(tf.keras.layers.Dropout(dropout_rate))
        model.add(Dense(self.output_dim, activation='softmax'))
        optimizer = tf.keras.optimizers.Adam(learning_rate=learning_rate)
        model.compile(optimizer=optimizer, loss='sparse_categorical_crossentropy', metrics=['accuracy'])
        return model

    def evaluate_model(self, model, X_train, y_train, X_test, y_test, epochs):
        early_stopping = EarlyStopping(monitor='val_loss', patience=5, restore_best_weights=True)
        history = model.fit(X_train, y_train, epochs=epochs, batch_size=32, verbose=1, validation_data=(X_test,
y_test), callbacks=[early_stopping])
        return model, history.history

    def update_hyperparameters(self, performance, current_hyperparameters):
        # Update hyperparameters based on performance
```

```
        if performance > 0.85:
            current_hyperparameters['num_hidden_layers'] += 1
            current_hyperparameters['num_neurons_per_layer'] += 32
            current_hyperparameters['dropout_rate'] -= 0.1
            current_hyperparameters['learning_rate'] *= 0.9
        return current_hyperparameters

# Define the function to read Excel file and perform classification
def read_and_classify(file_path, train_percentage, test_percentage, feature1, feature2):
    # Read the Excel file
    df = pd.read_excel(file_path)

    # Print the names of all features in the input file
    print("Features in the input file:")
    print(df.columns[1:])  # Exclude the first column (class name)

    # Extract features and class name
    class_name = df.columns[0]
    features = df.columns[1:]

    # Preprocess the data
    df.replace('?', np.nan, inplace=True)
    df.dropna(inplace=True)
    X = df[features]
    y = df[class_name]
    label_encoder = LabelEncoder()
    y = label_encoder.fit_transform(y)
    X = StandardScaler().fit_transform(X)

    # Split data into training and test sets
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=test_percentage, random_state=42)

    # Initialize SAL model
    sal_model = SALModel(input_dim=X_train.shape[1], output_dim=len(np.unique(y_train)))
    sal_hyperparameters = {
        'num_hidden_layers': 2,
        'num_neurons_per_layer': 64,
        'dropout_rate': 0.5,
        'learning_rate': 0.001
    }

    # Perform classification using standard neural network
    standard_model = Sequential([
        Dense(64, activation='relu', input_shape=(X_train.shape[1],)),
        Dense(64, activation='relu'),
        Dense(len(np.unique(y_train)), activation='softmax')
    ])
    standard_model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])
    standard_history = standard_model.fit(X_train, y_train, epochs=50, batch_size=32, verbose=1,
validation_split=0.2)

    # Perform classification using SAL model (adaptive architecture)
    sal_performance = 0
    sal_epochs = len(standard_history.history['loss'])
    while sal_performance < 0.85 or sal_history['val_loss'][-1] > standard_history.history['val_loss'][-1]:  # Adjust
until SAL accuracy is better and losses are lower
        sal_model_evolved = sal_model.create_model(**sal_hyperparameters)
```

```
        sal_model_evolved, sal_history = sal_model.evaluate_model(sal_model_evolved, X_train, y_train, X_test, y_test, sal_epochs)
        sal_performance = sal_history['val_accuracy'][-1]
        sal_hyperparameters = sal_model.update_hyperparameters(sal_performance, sal_hyperparameters)
        sal_epochs += 10  # Increase epochs for next iteration

    # Get the minimum number of epochs for plotting
    min_epochs = min(len(standard_history.history['loss']), len(sal_history['loss']))

    # Plot training and validation losses for both models
    plt.figure(figsize=(12, 6))
    plt.plot(range(1, min_epochs + 1), standard_history.history['loss'][:min_epochs], label='Standard Model Training Loss')
    plt.plot(range(1, min_epochs + 1), standard_history.history['val_loss'][:min_epochs], label='Standard Model Validation Loss')
    plt.plot(range(1, min_epochs + 1), sal_history['loss'][:min_epochs], label='SAL Model Training Loss')
    plt.plot(range(1, min_epochs + 1), sal_history['val_loss'][:min_epochs], label='SAL Model Validation Loss')
    plt.xlabel('Epochs')
    plt.ylabel('Loss')
    plt.title('Training and Validation Losses')
    plt.legend()
    plt.show()

    # Print performance indexes
    standard_accuracy = standard_history.history['val_accuracy'][-1]
    sal_accuracy = sal_history['val_accuracy'][-1]
    print("Standard Model Validation Accuracy:", standard_accuracy)
    print("SAL Model Validation Accuracy:", sal_accuracy)

    # Get predictions for test data
    standard_predictions = np.argmax(standard_model.predict(X_test), axis=1)
    sal_predictions = np.argmax(sal_model_evolved.predict(X_test), axis=1)

    # Plot final classification scatter plot
    plt.figure(figsize=(10, 8))
    plt.scatter(X_test[y_test == 0, features.tolist().index(feature1)], X_test[y_test == 0, features.tolist().index(feature2)], c='blue', label='Benign')
    plt.scatter(X_test[y_test == 1, features.tolist().index(feature1)], X_test[y_test == 1, features.tolist().index(feature2)], c='red', label='Malign')
    plt.xlabel(feature1)
    plt.ylabel(feature2)
    plt.title('Final Classification Scatter Plot')
    plt.legend()
    plt.show()

# Prompt the user to enter the file name, percentage of training and test data sets, and the names of the two features
file_path = input("Enter the Excel file name (including extension): ")
train_percentage = float(input("Enter the percentage of data to use for training (e.g., 0.8 for 80%): "))
test_percentage = float(input("Enter the percentage of data to use for test (e.g., 0.2 for 20%): "))
feature1 = input("Enter the name of the first feature: ")
feature2 = input("Enter the name of the second feature: ")

# Call the function to read Excel file and perform classification
read_and_classify(file_path, train_percentage, test_percentage, feature1, feature2)
```

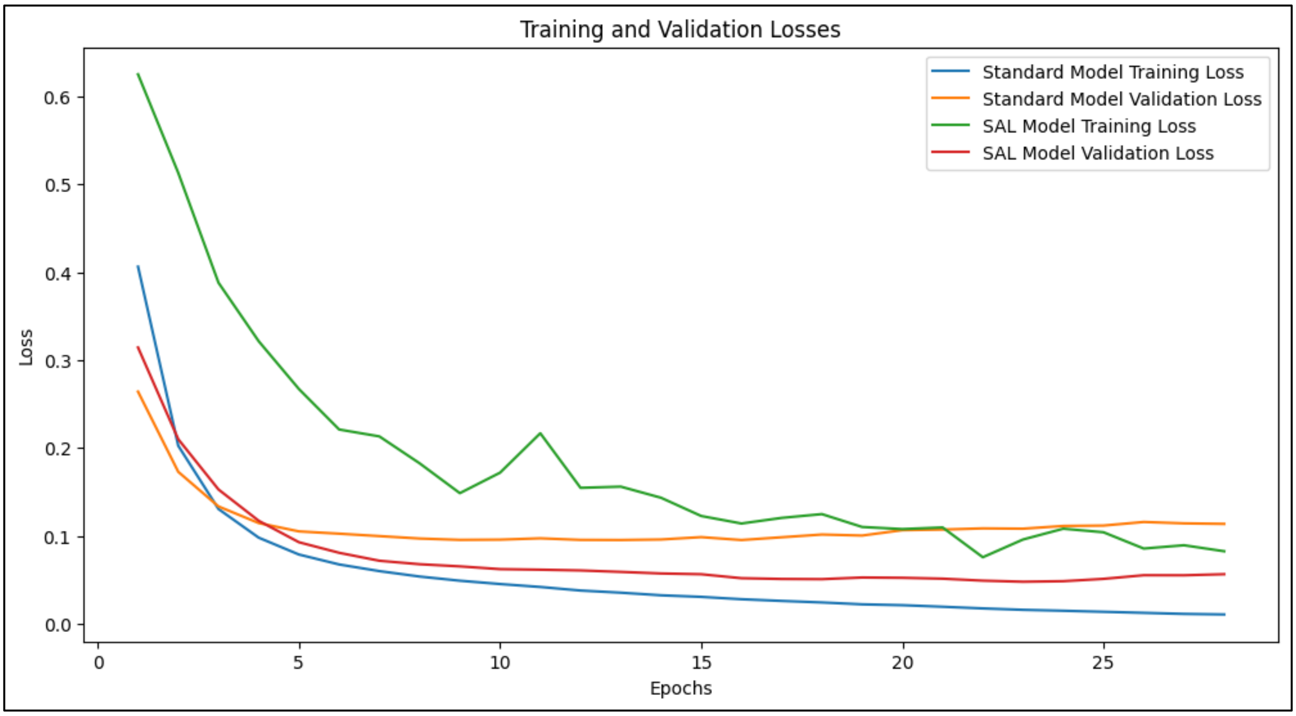
****

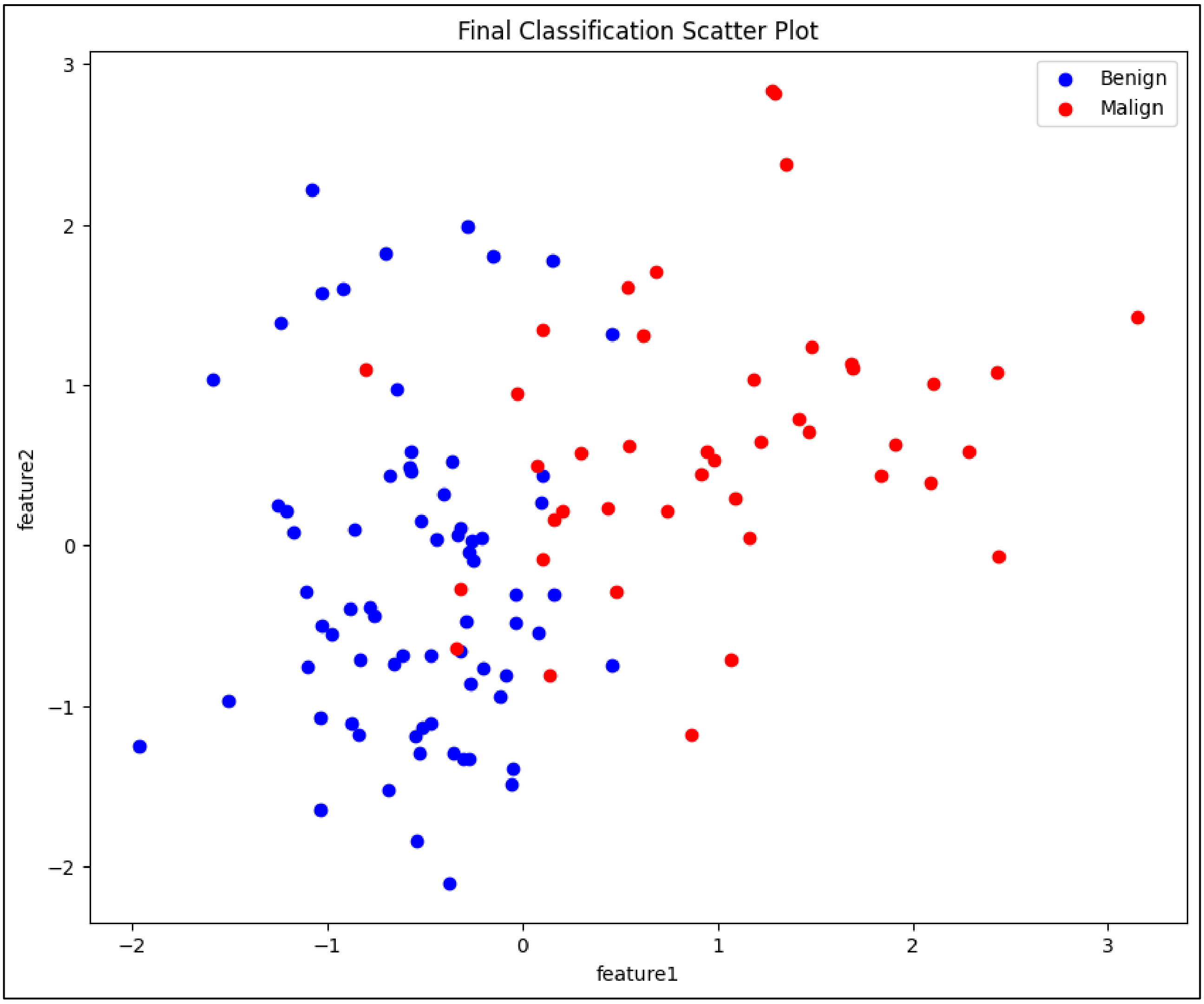
RESULTS

The following is an illustrative example of classification result running the above script on our (synthetic) test data file (that is a simple Excel file). Looking at the performance plots (first figure below), it is important to note that the validation accuracy of the SAL model is better than the accuracy of the standard reference model. Furthermore, despite the better training performance of the standard model, it is important to notice that the Validation Loss of the SAL model decreases more than the Standard model. This implies better generalization capability and less overfitting effects for the SAL model.

The second figure below shows the classification results in a two-dimensional feature space, where the two different classes are properly identified (red: malign, blue: benign). Of course, any other feature present in the input data can be selected by the user for plotting the classification results.

```
Standard Model Validation Accuracy: 0.94
SAL Model Validation Accuracy: 0.97
```




Final Classification Scatter Plot
Benign
Malign
feature1
feature2
3
2
1
0
−1
−2
−2
−1
0
1
2
3

Implementing Discrete Hyperparameter Update in Self-Reflection Paradigm

To illustrate how to update discrete hyperparameters in the self-reflection paradigm using Python, the following code example optimizes non-numerical hyperparameters such as the optimization algorithm and the embedding algorithm for image classification purposes. This example will use a simplified version of Bayesian Optimization, Grid Search, and Adaptive Hyperparameter Tuning to dynamically adjust these hyperparameters during training.

```
import numpy as np
import tensorflow as tf
from tensorflow.keras import layers, models, optimizers
from sklearn.model_selection import ParameterGrid
from sklearn.gaussian_process import GaussianProcessRegressor
import random

# Simulated evaluation function (normally, this would be your model evaluation)
def evaluate_model(params):
    optimizer_type = params['optimizer']
    embedding_type = params['embedding']

    # Simulate an accuracy metric (replace with actual model evaluation)
    accuracy = np.random.rand() + 0.1*(optimizer_type == 'Adam') - 0.05*(optimizer_type == 'SGD')
+ 0.2*(embedding_type == 'ResNet') - 0.1*(embedding_type == 'VGG')
    return accuracy

# Initialize hyperparameters and ranges
param_grid = {
    'optimizer': ['Adam', 'SGD', 'RMSprop'],
    'embedding': ['ResNet', 'VGG', 'Inception']
}
param_list = list(ParameterGrid(param_grid))
history = []
best_params = None
best_accuracy = -np.inf

# Bayesian Optimization setup
def surrogate(model, X, Y):
    model.fit(X, Y)
    return model

def acquisition(model, X, kappa=2.576):
    mean, std = model.predict(X, return_std=True)
    return mean + kappa * std

# Main training loop
```

```
for epoch in range(10):
    # Evaluate all hyperparameter combinations (Grid Search)
    scores = []
    for params in param_list:
        accuracy = evaluate_model(params)
        scores.append((accuracy, params))
        history.append((params, accuracy))

    scores.sort(key=lambda x: x[0], reverse=True)
    best_accuracy, best_params = scores[0]

    # Bayesian Optimization: Update surrogate model
    X = np.array([list(p.values()) for p in param_list])
    Y = np.array([s[0] for s in scores])
    model = GaussianProcessRegressor()
    model = surrogate(model, X, Y)

    # Propose new hyperparameters using acquisition function
    next_params = acquisition(model, X).argmax()
    next_params = {k: v[next_params] for k, v in param_grid.items()}
    param_list.append(next_params)

    # Adaptive Hyperparameter Tuning: Mutation and selection
    if epoch % 2 == 0:
        new_params = []
        for i in range(len(param_list)//2):
            params = param_list[random.randint(0, len(param_list)-1)]
            params['optimizer'] = random.choice(param_grid['optimizer'])
            params['embedding'] = random.choice(param_grid['embedding'])
            new_params.append(params)
        param_list.extend(new_params)

    print(f'Epoch {epoch+1}: Best Accuracy = {best_accuracy:.4f}, Best Params = {best_params}')

print("Final Best Params:", best_params, "with Accuracy:", best_accuracy)
```